UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

UM GERADOR DE PROGRAMAS PARA SISTEMAS DE REGRAS DE PRODUÇÃO VISANDO À EFICIÊNCIA NA EXECUÇÃO

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA

ALVARO DANIEL ARIONI PALADINO

FLORIANÓPOLIS, MARÇO DE 1991

UM GERADOR DE PROGRAMAS PARA SISTEMAS DE REGRAS DE PRODUÇÃO VISANDO A EFICIÊNCIA NA EXECUÇÃO

ALVARO DANIEL ARIONI PALADINO

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA A OBTENÇÃO DO TÍTULO DE

MESTRE EM ENGENHARIA ELÉTRICA

ESPECIALIDADE **ENGENHARIA ELÉTRICA**, ÁREA DE CONCENTRAÇÃO **SISTEMAS DE CONTROLE**, E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO

Prof. HERVÉ ERIC GARNOUSSETT, Dr.
Orientador

Prof. JOÃO PEDRO ASSUMPÇÃO BASTOS, Dr. D'Etat
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

BANCA EXAMINADORA

Prof. AUGUSTO HUMBERTO BRUCIAPAGLIA, Dr.
Co-orientador

Prof. JEAN MARIE FARINES, Dr. Ing.

Prof. LUIZ FERNANDO JACINTHO MAIA, M.

Prof. GUILHERME BITTENCOURT, Dr.

À Gabriela

Agradecimentos

Aos professores Hervé Eric Garnousset e Augusto Humberto Bruciapaglia pela orientação, aos membros da Banca Examinadora e aos demais professores e colegas do LCMI pela amizade e convívio neste período. Também agradeço à CAPES e ao CNPQ pelo auxílio financeiro em forma de bolsa.

Sumário

Lista de Figuras

Resumo

Esta dissertação apresenta o SPP, um sistema de produção proposicional desenvolvido com características adequadas para sua utilização em aplicações de tempo real, embora possa ser igualmente usado para outro tipo de aplicações.

O SPP é implementado como um gerador de programa, tem uma sintaxe lisp-like e uma boa capacidade matemática. Requer a declaração explícita de atributos e funções, está baseado na lógica proposicional e a sua máquina de inferência trabalha sem backtracking, com raciocínio não monotônico, em encadeamento para frente e atendendo às estratégias clássicas de resolução de conflitos.

Para conseguir uma boa eficiência de execução, a linguagem C foi escolhida como base do desenvolvimento, sendo usada tanto para o sistema especialista gerado como para o programa gerador. Por outro lado, uma interface ampla e flexível entre SPP e a linguagem C insere o sistema num contexto de programação híbrida (mistura de procedural e não procedural), necessário para tornar compatíveis o tempo real e os sistemas especialistas.

Abstract

The SPP propositional production system is introduced in this dissertation. It can be used in real time as well as traditional artificial intelligence applications.

The SPP, which is based upon the propositional logic works as a "program generator", having a lisp-like input syntax and a strong mathematical capacity. The declaration of attributes and functions must be explicit. The inference engine operates with forward chaining, with non-monotonic reasoning and without backtracking. It follows the traditional conflict resolution strategies.

In order to obtain portability and efficiency in the execution, the system was developed using the C programming language. The expert systems produced with the SPP generator are also written in C. On the other hand, a wide and flexible interface with C inserts the system in a hybrid programming context (miscegenation of procedural and non-procedural programming). In that way, an expert system can be easily included into a real time environment.

Capítulo 1

Introdução

O uso de técnicas de inteligência artificial em informática tem se mostrado particularmente adequado na resolução de problemas onde uma abordagem através de algoritmos se torna extremamente difícil ou até impossível. Em particular os sistemas especialistas usam o conhecimento de especialistas humanos para a resolução, sobre um domínio restrito e bem definido, de problemas nas mais diversas áreas. Para uma descrição mais detalhada dos sistemas especialistas o leitor pode ver por exemplo [Nil80,FG87,HWL83,HK85,Wat86].

Entre as linguagens criadas para o desenvolvimento de sistemas especialistas, as baseadas em regras de produção têm tido maior sucesso. Estas linguagens, denominadas **sistemas de produção**, constituem um paradigma de representação centrado num conjunto não ordenado de "regras" na forma de pares (condição, ação). A sua popularidade se deve sobretudo aos seguintes aspectos:

- As regras de produção possuem uma grande expressividade.
- São um modelo natural e de fácil compreensão.
- Cada regra representa uma porção específica do conhecimento, facilitando o desenvolvimento.
- Se tem uma forte separação entre o conhecimento (expressado nas regras) e a forma de utilizá-lo (controle).

Porém, para conseguir todas essas vantagens, os sistemas de produção geralmente têm que pagar o preço de serem mais lentos e normalmente repousar sobre outros sistemas de grande consumo de recursos de máquina como LISP ou PROLOG. Isto tem dificultado a sua aplicação em sistemas de tempo real, onde a resposta do sistema a uma solicitação externa deve ser fornecida num tempo limitado.

O objeto desta dissertação é o desenvolvimento do SPP, um sistema de produção que visa contornar essas desvantagens e se tornar viável para formar parte de aplicações de tempo real[LCS*88]. Para isto, o SPP traduz o sistema de regras (declarativo) num módulo escrito na linguagem C (procedural), levando em conta características particulares da

entrada, como por exemplo condições do tipo "*atributo* = *constante*", para tornar esse módulo C o mais eficiente que for possível.

Como antecedente deste trabalho se deve mencionar o sistema de produção SP0, desenvolvido no LCMI sobre LISP interpretado para dar suporte à primeira versão do projeto de um controlador PID auto-ajustável[Pag89]. A possibilidade da substituição do sistema SP0 pelo SPP na segunda versão do controlador PID auto-ajustável, em desenvolvimento no LCMI, foi um motivador importante deste trabalho. No entanto SPP está pensado como um sistema de produção de utilização geral e pode ser usado em quaisquer outras aplicações, sejam estas com restrições de tempo ou não.
A especificação do sistema SPP inclui os seguintes pontos:

- O sistema deve aceitar como entrada uma linguagem baseada no paradigma das regras de produção, e fornecer na saída a tradução para a linguagem convencional desse programa de regras.
- Se deve conseguir do programa convencional gerado na saída a maior eficiência de execução que for possível. A ênfase do trabalho está colocada neste ponto.
- O sistema deve possuir mecanismos que facilitem a comunicação e interface do módulo gerado com tarefas escritas em linguagem convencional, dando lugar ao conceito de programação híbrida.
- A sintaxe da linguagem de regras deve adotar de preferência as estruturas fundamentais das linguagens LISP e SP0.
- O sistema de produção deve estar sustentado na lógica proposicional.
- A sua máquina de inferência deve trabalhar sem backtracking, com raciocínio não monotônico, em encadeamento para frente e atendendo às estratégias clássicas de resolução de conflitos.
- A linguagem definida deve possuir ainda uma razoável capacidade matemática construída internamente, e a declaração dos atributos e funções deve ser explícita.

As principais etapas cumpridas foram:

1. Especificação da linguagem de representação do conhecimento.
2. Especificação da forma padrão do módulo C gerado.
3. Desenvolvimento do motor de inferência inserido dentro do módulo C gerado.
4. Desenvolvimento de um algoritmo de avaliação de premissas em tempo de compilação.
5. Implementação de um compilador para fazer a tradução.

Estes assuntos são apresentados no presente trabalho da seguinte forma:

O capítulo 2 objetiva definir os conceitos fundamentais envolvidos na problemática do tempo real e dos sistemas de produção, assim como realizar uma primeira análise das características de um sistema de produção compatível com os requisitos do tempo real.

No capítulo 3 são analisadas as diferentes alternativas disponíveis para a construção do sistema, apresentando a justificativa de cada uma das decisões tomadas.

No capítulo 4 se estabelecem os objetivos de nível mais baixo, assim como uma metodologia de desenvolvimento. São detalhadas as principais características desejadas para o sistema e é formulada a abordagem escolhida para desenvolvê-lo.

No capítulo 5 se faz uma descrição da linguagem definida, apresentando em particular o funcionamento dos diferentes operadores e funções que possui. Por outro lado, num enfoque mais genérico, é analisada a lógica de funcionamento da máquina de inferência, indicando sua influência na forma de escrever as regras de produção.

O capítulo 6 trata aspectos relativos à implementação do sistema, analisando primeiro os algoritmos e conceitos utilizados pelo compilador, e posteriormente o algoritmo e as estruturas de dados utilizados pela aplicação gerada.

No capítulo 7 é analisada a complexidade do algoritmo de execução, e o seu comportamento é avaliado com alguns exemplos práticos. Também se estudam aspectos da aplicação do sistema SPP ao projeto de um controlador PID auto-ajustável.

Finalmente, o capítulo 8 apresenta as conclusões e perspectivas finais sobre o trabalho desenvolvido.

Capítulo 2

O tempo real em sistemas de produção

Os sistemas de produção têm se mostrado ferramentas particularmente adequadas na resolução de vários problemas de decisão quando estes não permitem uma formalização algorítmica simples, sendo numerosas as aplicações onde têm sido utilizados substituindo ou complementando especialistas humanos. Os sistemas de tempo real, por outro lado, também têm sido usados amplamente sobretudo na indústria onde controlam processos cada vez mais complexos. Porém, a incapacidade de resolver alguns problemas de controle por métodos clássicos [Pag89] faz que o uso dos sistemas de produção para controlar processos de tempo real, se torne uma alternativa pragmática e desejável.

O tempo de execução geralmente elevado dos sistemas de produção, tem impedido no entanto o seu maior aproveitamento em universos cuja dinâmica impõe uma forte restrição no tempo de resposta. Este capítulo objetiva definir os conceitos fundamentais envolvidos na problemática do tempo real e dos sistemas de produção, assim como realizar uma primeira análise das características de um sistema de produção compatível com os requisitos do tempo real.

2.1 A problemática do tempo real

Sendo o objetivo do trabalho o desenvolvimento de um sistema que visa aplicações em tempo real, o primeiro passo deve ser definir o que se entende por *tempo real*. Em conseqüência se define [LCS*88]:

Sistema informático de tempo real Sistema para o qual existe um tempo pré-fixado (imposto) ao final do qual deve produzir uma resposta ao seu ambiente.

Embora muitos sistemas populares como os de reserva de passagens de avião sejam chamados de *tempo real*, isto tecnicamente não é correto, pois eles não fornecem a resposta num tempo imposto. Este tipo de sistemas, onde se supõe que uma pessoa fica esperando uma resposta *rápida*, mas que só será fornecida num tempo variável, seriam mais adequadamente chamados sistemas em linha.

Os sistemas em tempo real devem ainda ter um conjunto de propriedades relacionadas com os seguintes conceitos [LCS*88]:

Tempo de resposta O sistema deve ser capaz de responder no instante em que a resposta é necessária, de acordo com a restrição temporal presente. Assim para um dado evento de entrada e um estado arbitrário do sistema, este deve ser capaz de produzir uma resposta aceitável no tempo necessário.

Correção A conduta do sistema deve ser mantida dentro de limites aceitáveis ao longo do tempo; isto corresponde à geração de uma resposta aceitável sem violar as restrições de tempo impostas pelo ambiente.

Confiabilidade É necessário fixar um limite inferior para o tempo médio entre falhas (MTBF: Mean Time Between Fails).

Disponibilidade É necessário igualmente fixar um limite superior para o MTTR (Mean Time To Repair), ou seja estabelecer um tempo máximo que o sistema possa ficar indisponível.

A definição do tempo real é bastante exigente no que se refere ao tempo imposto, pois mesmo utilizando uma abordagem algorítmica seria difícil saber se em todos os casos se obedece ao tempo imposto. Com uma linguagem baseada em regras de produção , onde o controle é não determinista, seria em conseqüência muito mais difícil. De qualquer forma, o caminho para verificar se a resposta pode ser obtida num tempo imposto, passa pela decomposição da tarefa em operações elementares cuja duração máxima possa ser estimada. E é neste sentido que uma abordagem algorítmica leva vantagem sobre uma baseada em regras para aplicações em tempo real. Porém, a "proceduralização" do controle de um sistema de regras por meio da tradução numa linguagem convencional, reduziria evidentemente os problemas.

Por outro lado, quanto menor o tempo de execução de uma aplicação, maior a probabilidade da resposta ser dada num tempo imposto, mesmo que não seja possível estimar exatamente esse tempo de resposta. Seria o caso, por exemplo, de um controlador digital onde o período de amostragem para fazer o controle seja da ordem de milissegundos. Para não alterar a função de controle é bom que o controle seja acionado antes de tomar a próxima amostra. Portanto, se experimentalmente o sistema der a resposta na mesma ordem de tempo (milissegundos), será muito difícil determinar se a aplicação atende aos requisitos de tempo imposto. No entanto, se nessas provas o tempo de resposta ficar na ordem de micro-segundos ou até centenas de micro-segundos, mesmo sem ter garantias teóricas sobre o tempo de resposta, haverá uma grande probabilidade de que o tempo imposto seja atingido na maioria das vezes.

Em função disto, para construir um sistema de produção *visando* aplicações em tempo real, deve-se dar ênfase nos seguintes aspectos fundamentais:

- O tempo de execução da aplicação deve possuir um limite superior.

- Embora seja difícil, deve ser possível estimar o tempo máximo das operações necessárias para obter a resposta requerida.

Estas características são desejáveis, mas não é possível quantificá-las. Deve ficar claro então que não se pretende criar um sistema de produção que funcione para qualquer aplicação de tempo real, nem um que garanta sempre uma resposta a respeito de sua aplicabilidade. De fato, o sistema de produção desenvolvido que será apresentado nos próximos capítulos não possui nenhum mecanismo especial relacionado com a variável *tempo* [LCS*88].

O que se pretende é criar um sistema de produção que, pela proceduralização do controle, e pela otimização do tempo de execução, possa ser usado com segurança para uma quantidade importante aplicações com restrição de tempo. É por causa disto que foi escolhido como título do trabalho "Um gerador de programas para sistemas de regras de produção *visando* a eficiência na execução".

2.2 Características gerais dos sistemas de produção

O modelo *sistema de produção*, introduzido por Post (1943), revelou-se como um poderoso formalismo de programação devido sobretudo à forma declarativa de expressão do conhecimento e ao caráter associativo de sua manipulação. Embora não exista uma análise formal única para os Sistemas de Produção, se tentará apresentar aqui um sumário do que se encontra classicamente.

2.2.1 A arquitetura dos sistemas de produção

Um sistema de produção se compõe basicamente de três elementos:

Memória de trabalho Uma base de dados cujos elementos descrevem o estado do sistema no processo de resolução.

Base de regras Um conjunto de *regras de produção* contendo o conhecimento operacional sobre o problema.

Motor de inferência Um interpretador responsável pelo controle do sistema.

O poder de expressão e generalidade dos sistemas de produção é devida principalmente às seguintes características:

1. O acesso às regras é feito de forma associativa, ou seja em função do estado do problema.

2. A execução de um sistema de produção normalmente é *não determinista* pois existem fases de escolha abertas a diversas alternativas.

Então, em oposição à seqüência fixa de operações definida nos algoritmos (em programação convencional), nos sistemas de produção o controle é dirigido pelos dados.

A memória de trabalho contem a qualquer momento o estado corrente do sistema, seus elementos são normalmente chamados de *objetos*. Eles podem representar instâncias de objetos físicos relacionados com o problema ou elementos conceituais como metas a serem atingidas em passos intermediários. Os objetos da memória de trabalho condicionam o disparo das regras e podem ser criados, modificados ou destruídos por elas.

A base de regras é formada por um conjunto não ordenado de regras de produção. Elas têm uma parte condição (ou antecedente) que deve ser satisfeita pelo estado corrente do sistema (memória de trabalho), e uma parte ação (ou conseqüente) indicando as alterações a serem aplicadas no estado corrente no momento do disparo da regra. A parte conseqüente pode conter também ações atuando sobre o conjunto de regras como criar, apagar ou modificar regras [CM86,Win84], ações permitindo alterar o comportamento do motor de inferência, e interface de entrada/saída. Às regras podem também ser associados atributos especiais como prioridades, ou fatores de certeza.

A principal tarefa do motor de inferência é decidir qual a próxima regra a ser disparada. Ele pode ser visto como uma máquina de estado finita com quatro estados [BFKM86] como ilustrado na seguinte figura:

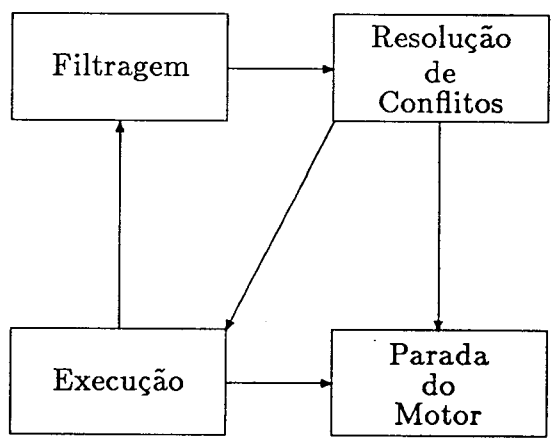


Figura 2.1: Diagrama de estados do Motor de inferência

A descrição de cada estado é a seguinte:

- Na etapa de filtragem, são determinadas as regras satisfeitas pelo estado corrente do sistema. Cada regra instanciada, junto com um objeto que a instancia, passam a constituir um elemento do *Conjunto de conflitos*. Como uma regra pode ser instanciada por vários conjuntos de objetos, ela pode aparecer várias vezes no conjunto de conflitos.

- Na etapa de resolução de conflitos se escolhe um par regra-objeto do conjunto de conflitos para disparar. Essa seleção da regra a disparar é um problema aberto, portanto esta fase é não determinista.

- Na fase de execução são aplicadas as ações presentes na parte de conseqüente da regra escolhida para o disparo.

- A parada do sistema pode ocorrer por uma ação de controle, quando se está executando a conclusão de uma regra, ou quando o conjunto de conflitos fica vazio.

Conceitualmente simples, este modelo envolve no entanto problemas complexos, como o da ineficiência inerente à etapa de filtragem [Sob87,GD84] e o não determinismo da etapa de resolução de conflitos.

2.2.2 A estrutura de controle

A estrutura de controle está intimamente relacionada com o não determinismo da resolução de conflitos. Os seguintes aspectos são fundamentais:

Regime de funcionamento Define a atitude a tomar quando o conjunto de conflitos estiver vazio. Em regime irrevogável o motor de inferência simplesmente para. Em regime por tentativas (com *backtracking*) o motor de inferência tenta voltar num estado anterior e reconsiderar a escolha que tinha sido feita, disparando outra regra [Lau84,HWL83].

Critérios de resolução de conflitos Como já foi dito, a escolha da regra a disparar dentro do conjunto de conflitos é um problema aberto, e portanto existe um grande número de critérios utilizados. Alguns deles são [BF86,BFKM86,Kae89]:

- A escolha da primeira regra.
- A regra de maior prioridade.
- A regra mais específica.
- A instância da regra que utiliza os objetos modificados mais recentemente.
- A escolha de uma regra que ainda não tenha sido instanciada.
- O disparo de todas as regras pseudo paralelamente, como no sistema MYCIN [BS85].
- O uso de metaregras para escolher a regra a disparar [Dav80].

A definição adequada da estratégia de resolução de conflitos é fundamental para encontrar a solução de uma forma rápida e eficiente. A melhor estratégia, bem como a melhor escolha do regime de funcionamento depende principalmente da natureza do problema.

2.2.3 Raciocínio monotônico

Quando o sistema usa um raciocínio monotônico [Tur86], nenhum fato estabelecido pode ser retirado da memória de trabalho. Portanto, nos sistemas monotônicos, quando uma

regra se torna válida permanece válida durante o restante da execução do sistema [Nil80]. Isto evidentemente simplifica o controle do sistema.

Nos sistemas especialistas no entanto, que envolvem um conjunto de conhecimentos muito complexos, é interessante reconsiderar fatos da memória de trabalho. Os sistemas que o permitem, por oposição aos anteriores, são chamados de não monotônicos. Este tipo de raciocínio complica o controle do sistema, porém fornece uma maior flexibilidade.

2.2.4 Direção do encadeamento das regras

As regras são encadeadas para frente (*forward chaining*) quando elas são aplicadas a partir do estado corrente do sistema, descrito pela memória de trabalho, até produzir um estado que represente a solução do problema ou até chegar à conclusão de que não é possível achar uma solução.

As regras são aplicadas para trás (*backward chaining*) quando são usadas para decompor o problema inicial em sub-problemas mais simples, até encontrar um conjunto de problemas equivalentes ao original cuja veracidade possa ser facilmente verificada a partir dos dados iniciais [Nil71,FG87].

Os sistemas com encadeamento para frente são mais adequados para problemas onde se tem vários estados finais aceitáveis, enquanto o encadeamento para trás é mais adequado nos sistemas de diagnóstico onde, para chegar a um único objetivo, se tem um amplo conjunto de fatos iniciais.

2.2.5 Diferenças com a programação clássica

As diferenças entre a programação convencional e a programação usando o modelo de sistemas de produção podem ser sumariadas na seguinte tabela [Wat86,BF86]:

Programação clássica	Sistemas de produção
Informações principalmente numéricas.	Informações simbólicas.
O controle e os dados estão misturados.	O controle e os dados estão separados, o controle na máquina de inferência e os dados na base de conhecimento (memória de regras + memória de trabalho).
O fluxo de controle está determinado por um algoritmo pré-definido.	Se tem um comportamento não determinista.
Se trabalha com um conhecimento exato.	Se pode ter uma representação do conhecimento incerto.
A supressão ou acréscimo de dados é geralmente trabalhosa.	A modificação dos dados é simples e fácil.

Essas diferentes características levam a situações onde é melhor usar o modelo de regras e outras onde o modelo clássico é mais vantajoso [BF86,BFKM86].

As vantagens dos sistemas de produção estão relacionadas com a maior modularidade (pois uma regra é uma porção de conhecimento independente do resto), a facilidade de aprendizado do modelo devido à sua uniformidade, e a naturalidade do modelo em si pois os conhecimentos dos especialistas podem ser expressados de forma natural por regras de produção.

A desvantagem do modelo de produção reside basicamente na ineficiência de execução, e no fato que, sendo o controle determinado pelo sistema, acaba sendo pouco transparente. Isto pode dificultar a detecção de problemas como *loops* ou *deadlocks*. A ineficiência de execução é conseqüência também do controle dirigido pelo sistema. Entre uma operação e a seguinte, existe um *overhead* importante devido ao fato que o sistema tem que escolher de forma automática a operação seguinte. Nos sistemas clássicos, onde o usuário dá explicitamente a ordem de execução das operações, esse *overhead* não existe.

2.3 Como conciliar os dois conceitos do ponto de vista prático

Qualquer alternativa para tornar compatíveis o conceito de tempo real e a flexibilidade dos sistemas de produção passará necessariamente por uma solução de compromisso. Isto significa que terá que deixar de lado algumas características não fundamentais dos sistemas de produção, para concentrar os esforços nos aspectos mais importantes, ou por outro lado aceitar tempos de resposta muito maiores.

Como a ênfase do trabalho é de construir um sistema de produção que forneça a resposta num tempo aceitável, será interessante analisar aqui quais são as características do modelo baseado em regras que se pode abrir mão sem por isso desvirtuá-lo demais. Porém, essa flexibilização sozinha não será suficiente para melhorar o desempenho de forma apreciável. Um eficiente algoritmo de compilação do conhecimento será desenvolvido com base nesse modelo flexibilizado.

Como já foi dito, a ferramenta desenvolvida neste trabalho não possui mais mecanismos especializados para aplicações de tempo real do que uma linguagem convencional como C. No entanto, consegue eliminar a maior parte dos aspectos dos sistemas de produção que podem dificultar a sua utilização nesse tipo de aplicações. As regras de produção são transformadas num módulo escrito na linguagem C (procedural) que fornece o mesmo resultado que se obteria da execução não determinista das regras.

Esse módulo C está pensado para rodar como uma tarefa "inteligente" de um sistema maior (possivelmente de tempo real), podendo atuar tanto como uma simples função a ser chamada, ou até como um processo concorrente com o resto do sistema. Isto vai depender de uma escolha de programação (seqüencial ou concorrente) e da natureza do sistema operacional, pois todos os mecanismos de controle de processos disponíveis em C estarão disponíveis também no módulo gerado através de uma interface flexível com C.

Em conseqüência não serão tratados aqui aspectos teóricos da problemática do tempo real, como sincronismo ou assincronismo [Mil83], nem será feita uma análise mais detalhada do problema do tempo de resposta finito [LCS*88]. No lugar disso se estudarão aspectos práticos relativos à construção de um sistema de produção de desempenho aceitável quando utilizado como módulo inteligente de um sistema de tempo real.

Eliminação do manuseio dinâmico da memória

A estrutura interna dos sistemas de produção dispõe tradicionalmente de métodos para criar e apagar objetos de forma dinâmica na memória. Isto se torna evidente para os sistemas baseados na lógica de primeira ordem [MCS], onde uma classe de objetos pode ser instanciada um número ilimitado de vezes. Cada nova instanciação representa na prática a criação de um novo objeto de forma dinâmica.

Os sistemas baseados na lógica proposicional, onde não há instanciação de objetos, também não dispensam esse mecanismo já que geralmente possuem comandos para criar e apagar atributos. No entanto, como normalmente se conhecem previamente todos os atributos que virão a ser criados, é possível, dispensar a criação dinâmica de objetos. Isto não implica que a linguagem não irá ter comandos para criar ou apagar atributos, mas simplesmente que, durante a execução, a criação e o apagado de atributos será simulada pois cada atributo terá seu espaço de memória reservado durante toda a execução.

Evidentemente , o modelo de sistemas de produção não é afetado por este modo de tratar a criação e destruição de atributos. Por outro lado, isso permite eliminar um *overhead* importante que sem dúvida contribuirá a melhorar o desempenho do sistema. A melhoria obtida pela eliminação do manuseio dinâmico dos atributos pode ser ainda maior se a linguagem tiver declarações dos tipos dos atributos, pois assim cada atributo será tratado de acordo com seu tipo particular e em conseqüência só seriam gastos os recursos de máquina estritamente necessários para cada caso.

A flexibilidade necessária

Geralmente os sistemas de produção possuem uma importante flexibilidade que é muito útil, sobretudo na etapa de desenvolvimento da aplicação. São coisas como por exemplo:

- A possibilidade de criar novas regras de forma dinâmica, um esforço em direção a sistemas que aprendam sozinhos.
- A possibilidade de *desfazer* inferências de forma interativa para explorar outros caminhos, uma ferramenta pensada especialmente para facilitar o desenvolvimento.
- A possibilidade de escolher entre encadeamento para frente (Forward chaining) ou encadeamento para atrás (Backward chaining).
- A implementação de um mecanismo de backtracking, que garanta (se o grafo de estados for finito [Nil80]) que todas as alternativas possíveis sejam exploradas, mas

obriga o sistema a lembrar de todos os estados onde poderá reconsiderar o caminho escolhido.

Um sistema de produção sem ferramentas como as mencionadas acima seria evidentemente menos geral e menos poderoso, porém ainda manteria as suas características fundamentais e a probabilidade de conseguir uma implementação razoavelmente eficiente seria maior. Então, na procura de uma solução de compromisso, se podem descartar alguns dispositivos que por serem extremamente gerais e poderosos acabariam consumindo recursos exagerados.

A solução interpretada

Tradicionalmente as abordagens interpretadas são tidas como vantajosas em termos do desenvolvimento de programas. Isto é ainda mais válido para as linguagens baseadas em regras de produção, onde como já vimos, o enfoque é não procedural. Por outro lado, as abordagens interpretadas são ineficientes e inadequadas para aplicações onde existem requisitos sobre o tempo de resposta. Por causa disso, às vezes até são produzidos tanto compiladores como interpretadores para a mesma linguagem.

Em última instância, a maior vantagem das abordagens interpretadas sobre as compiladas é que resulta num ciclo de modificação e testes mais curto. Então visando um sistema de produção *eficiente* em termos de execução, se poderia adotar uma alternativa compilada. Contudo, a solução interpretada tem sido a mais usada em inteligência artificial devido sobretudo à linguagem LISP [Cha85], que é amplamente usada nas implementações. Em conseqüência uma alternativa que nos parece interessante é de desenvolver um compilador cuja entrada pudesse ser também interpretada.

2.4 Conclusão

Neste capítulo foi abordada a problemática do tempo real, definindo conceitos importantes como o tempo imposto para o sistema gerar uma resposta. Posteriormente se fez uma descrição sumária do que se entende por sistema de produção, destacando especialmente a abordagem declarativa e não procedural deste tipo de linguagens e como isto incide na ineficiência de execução deste tipo de sistemas. Finalmente se fez uma primeira análise do que deveria ser levado em consideração na procura de uma solução de compromisso que implementasse um sistema de produção compatível com as restrições do tempo real.

A criação e destruição de objetos de forma dinâmica na memória, a adoção de uma solução interpretada, e uma certa dose de flexibilidade são alguns dos pontos que poderiam ser deixados num segundo plano sem por isso desvirtuar o modelo clássico de sistema da produção.

Capítulo 3

As escolhas fundamentais

Para atingir o objetivo de um sistema de produção compatível com aplicações em tempo real, será necessário um determinado número de hipóteses e restrições que diminuam a complexidade do problema e o situem num marco de referência adequado para a classe de problemas que tentará resolver. Esse conjunto de postulados representará em conseqüência um ponto de partida na direção de um sistema com as características desejadas.

No presente capítulo serão analisadas diferentes alternativas possíveis, apresentando a justificativa das decisões tomadas. Cada hipótese feita aqui implicará numa escolha de alto nível, que servirá como ponto de referência e guia na realização do trabalho.

3.1 Gerador de programa ou interpretador

O modelo de sistemas de produção, apresentado no capítulo anterior, poderia ser implementado tanto por um *interpretador* de regras, como por um *gerador* de programa. Na segunda opção um compilador traduziria o programa de regras de produção para uma linguagem procedural que realizasse uma função equivalente.

Embora a maioria dos sistemas especialistas baseados num interpretador disponham de um processo mais ou menos complexo de *compilação* de regras de produção, isto sempre implica na representação interna das regras por estruturas de dados específicas. Em conseqüência, neste tipo de sistemas o processo chamado de "compilação" é somente um jeito de fazer com que o motor de inferência perca menos tempo nas etapas de filtragem e resolução de conflitos.

A compilação de regras de produção, por um sistema *gerador de programa*, seria um processo bem mais abrangente pois além da utilização de uma estrutura de dados específica tanto para as regras como para os objetos que constituem o sistema, a máquina de inferência tradicional seria também substituída por um conjunto de instruções e estruturas de dados particulares distribuídas dentro do código procedural gerado pelo sistema. A possibilidade de mexer, no processo de geração, tanto com as estruturas de dados quanto com o código que será executado com essas estruturas, dá uma maior liberdade para encontrar as formas mais eficientes de implementação. Se o gerador de programa fornecesse como

saída um código executável, ele seria diretamente um compilador. Em face às dificuldades de implementação isto geralmente não será o caso – pelo menos nas primeiras etapas de pesquisa. Em conseqüência, sendo a saída um programa escrito em alguma linguagem convencional como C ou PASCAL, a função desse gerador de programa seria, numa forma mais conveniente, descrita como a de um pré-compilador.

Na opção do interpretador toda a base de regras vai ser transformada numa estrutura de dados. Neste caso se teria então a flexibilidade de poder mexer, praticamente sem restrições, com as regras de produção durante a execução do sistema. Com a possibilidade de acrescentar ou modificar regras em função da execução de outras regras, se poderiam modelar funções complexas, dando lugar até a um processo de aprendizado. Em contrapartida, para aplicações em tempo real onde o sistema especialista normalmente deverá rodar como uma tarefa de um sistema maior, parece difícil conseguir o entrosamento da aplicação com um interpretador de regras complexo.

O gerador de programa, em comparação com o interpretador, fornece um sistema especialista menos flexível pois a parte das regras que for traduzida para código será inalterável durante a execução do sistema. Isto, no entanto, resulta numa vantagem quando se pensa em aplicações de tempo real, pois o código gerado será bem mais simples que um complexo interpretador de regras, sendo em conseqüência mais fácil o seu entrosamento com outras tarefas de um sistema em tempo real.

3.2 A ordem da lógica que sustenta o sistema

A lógica proposicional (também chamada lógica de ordem 0 devido à ausência de variáveis) é a base dos sistemas de produção de ordem 0 (ou proposicionais), que não admitem variáveis e funcionam com a restrição da existência de uma quantidade finita de objetos na memória de trabalho.

Na lógica de primeira ordem [Men79], no entanto, uma mesma condição pode ser testada com diferentes objetos, representados por variáveis. Nos sistemas de produção de primeira ordem [For81,Kae89], baseados nesta lógica, a possibilidade de testar as condições das regras entre diferentes objetos, aumenta a expressividade e flexibilidade da linguagem de representação do conhecimento. Na maioria destes sistemas as variáveis são consideradas quantificadas universalmente.

Alguns poucos sistemas de produção, como SNARK-2 [Via84], admitem o uso de variáveis também para os predicados, e são chamados em conseqüência como sistemas de ordem 2.

Os sistemas de ordem 1, e com maior motivo os de ordem 2, requerem um algoritmo de filtragem muito complexo para encontrar todas as instâncias possíveis de uma regra a cada ciclo de inferência. De fato este é um problema aberto, onde existem muitas abordagens [For82,Gha81,GK88]. Porém, trata-se de um problema de uma complexidade intrínseca muito grande, onde os avanços não podem ir além de um determinado limite.

A grande complexidade intrínseca dos algoritmos que sustentam os sistemas de ordem

1 ou maior faz com que geralmente se torne difícil a sua aplicação nos sistemas em tempo real, onde as restrições de tempo de execução são com certeza importantes. Então, em primeira instância a alternativa proposicional seria a mais adequada. Deve-se notar no entanto, que sistemas "simulando" primeira ou segunda ordem não seriam incompatíveis com o tempo real. Por exemplo, um sistema com a restrição de uma quantidade máxima (maior do que 1) de objetos a serem considerados poderia ter uma implementação muito eficiente. Um sistema deste tipo teria somente a aparência de um de primeira ordem, pois poderia ser reduzido a um sistema proposicional transformando cada condição com variável num conjunto de condições simples associadas a cada objeto considerado.

3.3 Características gerais da linguagem SPP

A linguagem SPP (Sistema de Produção Proposicional), desenvolvida neste trabalho, foi pensada com um conjunto de características que visam tornar compatíveis as aplicações de tempo real e os sistemas especialistas. Muitas destas características já têm sido discutidas anteriormente, porém serão igualmente sumariadas na presente seção. Essas *opções de alto nível*, adotadas para a implementação do SPP, são as seguintes:

- O SPP é implementado como um gerador de programa. As razões de um melhor tempo de execução, junto com um entrosamento mais simples com outras tarefas de uma aplicação em tempo real, foram as que mais pesaram na decisão. De qualquer forma as duas opções não são exclusivas entre si. Poderia ser também implementado um interpretador que aceitasse como entrada a sintaxe do SPP. Isto, de fato, aumentaria a flexibilidade de desenvolvimento sem diminuir em nada a capacidade de tempo real, pois seria possível desenvolver com o interpretador e, a seguir, obter o produto final por geração de um programa na linguagem convencional (C neste caso).

- A sintaxe da linguagem SPP está baseada nas estruturas fundamentais da linguagem LISP, ficando assim uma linguagem *lisp-like*. Em particular todas as funções e operadores usam a notação pré-fixada do LISP. Esta escolha obedece aos seguintes motivos:

  1. A tradição do LISP em inteligência artificial é muito grande, e resulta portanto natural construir uma linguagem desse tipo com uma sintaxe lisp-like.
  2. Essa sintaxe facilitaria grandemente a eventual implementação de um ambiente de desenvolvimento em LISP com funções de edição, interpretação, depuração, compilação, etc. Nesta hipótese o SPP seria utilizado só para a geração do código *runtime* em C.

- Um procedimento de *backtracking*, como já foi visto, criaria um "*overhead*" indesejável para aplicações com restrição de tempo. Porém, pior do que isso é o fato de

que uma tal estratégia não permitiria garantir um tempo de resposta limitado, indispensável para aplicações em tempo real. Portanto o SPP adota, como OPS5 [For81] e outros sistemas especialistas, o critério de nunca reavaliar as decisões tomadas. Dessa forma, o processo de inferência termina cada vez que o conjunto de conflitos fica vazio. Nestes sistemas sem backtracking, (se também são não monotônicos) a estratégia de resolução de conflitos passa a ter uma importância fundamental.

- A máquina de inferência do SPP funciona em encadeamento para frente. Como não é possível criar um sistema de produção com um algoritmo de compilação eficiente que funcione tanto para frente como para trás, o melhor é tentar otimizar uma das duas alternativas. A escolha foi realizada em função da maior aplicabilidade do encadeamento para frente à classe de problemas que se pretendia resolver.

- O sistema SPP utiliza um raciocínio não monotônico. Esta escolha foi feita visando a maior flexibilidade do sistema. Porém, sistemas de produção monotônicos com uma grande eficiência de execução (como KHEOPS [GP88]) têm sido desenvolvidos com sucesso.

- O SPP, como seu próprio nome indica, é um sistema baseado na lógica proposicional. Na seção anterior foram analisadas em profundidade as razões desta opção.

- O tipo dos atributos e funções externas deve ser declarado de forma explícita nos fontes SPP. Isto vai contribuir para a eficiência de execução e para a otimização da utilização da memória.

- Mesmo com algumas restrições secundárias, a intenção é que o SPP funcione realmente como um sistema de produção. Em conseqüência, todas as estratégias clássicas de resolução de conflitos [For81], como a prioridade, a refração, a recentidade e a especificidade, são implementadas.

- As aplicações de tempo real freqüentemente devem realizar cálculos complexos para determinar, por exemplo, ações de controle. O sistema SPP está dotado de uma boa capacidade matemática construída internamente, para auxiliar o programador nesse sentido.

Tudo o que acaba de ser sumariado, representa as características globais da linguagem de produção desenvolvida. Para encontrar um maior detalhe, o leitor deve se dirigir ao Capítulo 5.

3.4 Conclusão

No presente capítulo foram delineadas as principais características do sistema de produção SPP. É implementado como um gerador de programa, tem uma sintaxe lisp-like e uma boa capacidade matemática. Requer a declaração explícita de atributos e funções, está

baseado na lógica proposicional, e a sua máquina de inferência trabalha sem backtracking, com raciocínio não monotônico, em encadeamento para frente e atendendo às estratégias clássicas de resolução de conflitos.

Por trás de cada opção adotada, sempre se teve presente o objetivo de entrosar o modelo de sistemas de produção com as exigências do tempo real. As restrições que esta abordagem impõe ao modelo de regras são secundárias e não degradam o modelo.

Capítulo 4

Metodologia de desenvolvimento

As hipóteses feitas até agora dão um marco suficiente para se estabelecer objetivos de nível mais baixo, assim como uma metodologia de desenvolvimento. No presente capítulo são detalhadas as principais características desejadas para o sistema, e é formulada a abordagem escolhida para desenvolvê-lo.

A escolha das ferramentas de computação a serem utilizadas no processo de desenvolvimento é outro ponto importante que será analisado a seguir.

4.1 Os objetivos propostos

Um sistema com as características especificadas até agora pode eventualmente tomar diversas formas. As opções de implementação são muitas e a escolha entre elas dependerá sobretudo dos objetivos que se pretenda atingir. Neste trabalho, o objetivo de nível mais alto é sem dúvida obter um sistema de produção que possa ser usado como uma tarefa "inteligente" de um sistema em tempo real. Ele , no entanto, deve ser dividido em objetivos mais específicos de modo a dar uma forma definitiva ao sistema.

Por motivos de padronização e generalidade, assim como em função da sua reconhecida eficiência de execução, a linguagem C foi escolhida como suporte para o programa gerado pelo sistema SPP. No marco desta decisão, os objetivos que se tentou atingir podem ser sumariados assim:

- Todo programa escrito na linguagem de regras do SPP deverá ser transformado num conjunto de rotinas em C que quando chamadas por outra tarefa C simularão o trabalho que faria a máquina de inferência do sistema de produção.

- A comunicação de dados entre a tarefa C gerada e o restante do sistema será facilitado com a possibilidade da chamada dos dois lados. Isto é, uma comunicação pode ser iniciada pelo programa principal, chamando ao sistema especialista gerado, ou também pelo sistema especialista, requerendo por exemplo dados do programa principal como conseqüência da execução da parte conclusão de alguma regra.

- O programa C gerado deverá fazer uso dos operadores do C em todos os casos em que isto for possível, de modo a conseguir uma melhor eficiência de execução.

- Um sistema de tempo real que utilize o conceito de regras de produção será necessariamente uma aplicação híbrida – parte dele serão regras de produção e outra parte será linguagem convencional. Neste sentido se torna interessante que a maior quantidade possível de recursos da linguagem C estejam disponíveis na linguagem de regras. Isto inclui desde a possibilidade de chamar diretamente qualquer rotina escrita em C, até a capacidade de usar, mesmo que indiretamente, os mais complexos tipos de dados que podem ser definidos em C.

- Sempre que for possível, o programa C gerado usará estruturas de apontadores para evitar a procura seqüencial. Em particular, as estruturas de dados assim como o código gerado, deverão estar organizados de tal forma que a resolução de conflitos (de acordo com as estratégias especificadas no capítulo anterior) não leve a nenhum *overhead.* Ou seja que após a filtragem, as regras devem ficar ordenadas no conjunto de conflitos de acordo com essas estratégias.

- A ordem das tarefas, na etapa de filtragem, deverá ser orientada para realizar somente os testes estritamente necessários para a determinação do novo conjunto de conflitos. Isto significa, por exemplo, que quando uma regra tiver uma premissa falsa, as outras não serão avaliadas.

Da observação das metas especificadas se conclui que o programa C gerado será um software complexo, em face sobretudo à generalidade que se pretende obter e a sua importante dependência de estruturas encadeadas para conseguir uma maior eficiência de execução.

4.2 A definição do sistema

O desenvolvimento do sistema SPP foi baseado numa metodologia estruturada, implementando os diversos módulos separadamente. Uma visão de alto nível da arquitetura do sistema pode ser apreciada na figura 4.1.

O programa principal analisa as diferentes opções de compilação e divide o trabalho entre os outros módulos de nível inferior. Basicamente, a transformação da linguagem de regras no programa C é realizada por etapas. Numa primeira etapa, a informação contida nas regras de produção é armazenada num conjunto de tabelas internas (ver capítulo 6). Posteriormente, a informação dessas tabelas é ainda compilada para resolver entre outras coisas as referências cruzadas. Finalmente, o conteúdo das tabelas é usado para construir as funções e estruturas C do programa gerado.

O módulo analisador léxico, identifica os diferentes *tokens* presentes na linguagem de regras e os passa para o analisador sintático. Este, por sua vez analisa a sintaxe da entrada, colocando por separado nas tabelas já mencionadas, os diferentes pedaços de informação

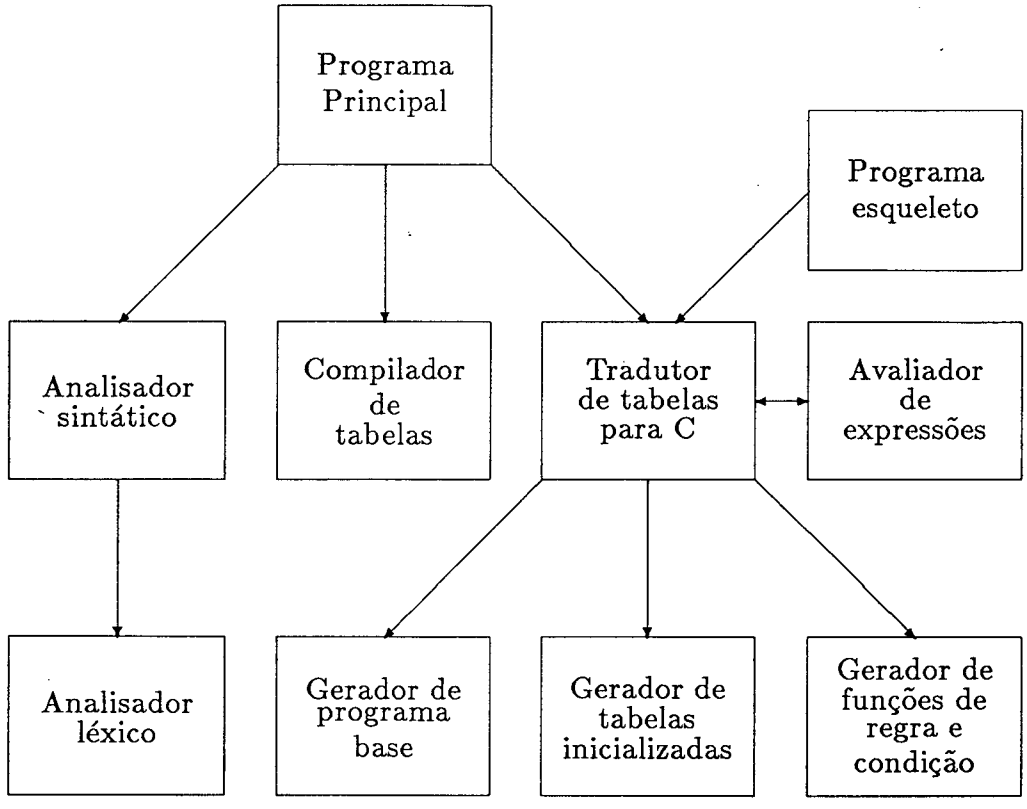


Figura 4.1: Diagrama dos módulos do sistema SPP

identificados. Para realizar estes módulos foram usadas as ferramentas de computação LEX e YACC , junto com a linguagem C.

O módulo compilador de tabelas é necessário pois a ordem em que se encontra a informação na entrada não permite resolver todas as referências numa passada só. Nesse sentido o SPP trabalha como um compilador de duas passadas [AU72,Wir76].

Embora a maioria dos erros de sintaxe sejam detectados pelos módulos de análise léxica e sintática, uns poucos (referências incorretas) só serão identificados no módulo compilador de tabelas. De qualquer forma, quando houver erros de sintaxe, o módulo tradutor de tabelas para linguagem C não será executado. Isto acontece pois as tabelas internas estão pensadas de uma forma que a sua informação só será consistente quando não houver nenhum tipo de erro.

O módulo tradutor de tabelas para linguagem C utiliza um "programa esqueleto" e um módulo avaliador de expressões para cumprir sua tarefa. A partir do programa esqueleto, as informações necessárias para construir um programa base são compiladas e posteriormente passadas para o módulo gerador respectivo. O módulo de avaliação de expressões, que também está escrito em LEX e YACC, é utilizado para pré-compilar a parte de condição das regras sempre que for possível. Isto é uma das maiores contribuições do algoritmo desenvolvido neste trabalho, e visa o objetivo de aumentar a eficiência de execução do programa gerado.

Finalmente, os módulos de geração de tabelas inicializadas e de geração de funções de

regra e condição fornecem o resto do código necessário para completar a construção do programa.

As características do programa gerado

O detalhamento da estrutura do programa C gerado pelo SPP será tratado na seção 6.2. Porém, de forma genérica, se devem destacar os pontos seguintes:

- Os dados específicos associados a cada regra são armazenados numa estrutura de dados de tipo `struct`
- Também para cada regra se tem uma função contendo o código da parte conclusão, e o código necessário para determinar o novo conjunto de conflitos após o disparo desta.
- Uma função é gerada para avaliar cada condição que aparece na parte esquerda das regras.
- Um conjunto de rotinas básicas, entre elas as de entrada e saída padrão, completam a estrutura do programa gerado.

Embora o sistema especialista gerado seja acedido desde C como uma função de nome "`inf_engine()`", de fato não existe uma máquina de inferência interpretativa no sentido tradicional. O trabalho que ela faria se encontra distribuído entre as funções de regra e a estrutura de dados.

4.3 As ferramentas de implementação

Na primeira seção deste capítulo foi justificada a escolha da linguagem de programação C para o programa gerado. Ela se baseou sobretudo na eficiência de execução da linguagem, embora também fosse levada em conta a sua portabilidade sobre outras máquinas. Na escolha da linguagem C também como ferramenta de implementação outros motivos foram importantes:

1. A facilidades que oferece o C para manusear estruturas de dados complexas.
2. A linguagem C possui uma importante biblioteca de funções externas.
3. Para trabalhar em conjunto com C está disponível o gerador de analisadores léxicos LEX, e o compilador de compiladores YACC.

O terceiro ponto foi, de fato, o mais importante, pois sem ferramentas como LEX e YACC a tarefa de implementar um software como SPP se tornaria muito complexa e trabalhosa. Em conseqüência, o conjunto C-YACC-LEX foi a ferramenta de implementação usada para desenvolver o sistema SPP.

A ferramenta LEX

LEX [LS78] ajuda a escrever programas cujo fluxo de controle é dirigido por instâncias de expressões regulares, e é adequado para segmentar a entrada preparando-a para uma rotina de *parsing*.

O código fonte do LEX é uma tabela de expressões regulares e fragmentos de programa correspondentes. Essa tabela é transformada num programa que divide a entrada de acordo com os padrões estabelecidos nas expressões. Cada vez que um padrão é reconhecido o fragmento de programa correspondente é executado. O reconhecimento das expressões é realizado por um autômato finito gerado por LEX.

Os analisadores léxicos escritos com LEX aceitam especificações ambíguas e escolhem o padrão mais comprido em cada caso. Isto faz com que às vezes um substancial *lookahead* seja necessário.

O programa gerado por LEX pode ser código fonte em C ou em outras linguagens. Para o presente trabalho, a escolha do C é evidente.

YACC, o compilador de compiladores

YACC [Joh78] fornece uma ferramenta geral para descrever a entrada para um programa de computador. O usuário de YACC especifica a estrutura da sua entrada em forma de regras gramáticas, junto com código a ser invocado toda vez que uma estrutura é reconhecida.

Como saída YACC produz uma sub-rotina, usualmente em linguagem C, que implementa uma máquina de estado finita para realizar o *parsing* da entrada. Essa sub-rotina, no entanto, não lê a entrada diretamente senão através de uma outra rotina de nome `yylex()` fornecida pelo usuário, ou pelo gerador LEX.

LEX e YACC podem ser usados separadamente, porém é particularmente simples a sua utilização em conjunto. Os programas LEX reconhecem somente expressões regulares; YACC escreve compiladores que aceitam uma ampla classe de gramáticas livres de contexto, mas requer um analisador léxico de baixo nível para fornecer os *tokens* de entrada.

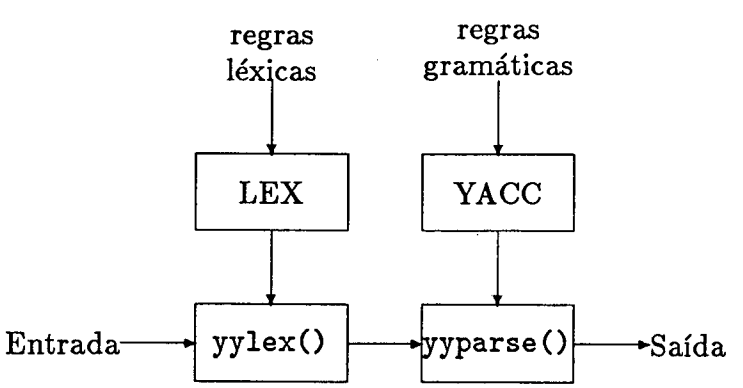


Figura 4.2: LEX em conjunto com YACC

Em conseqüência, a combinação entre LEX e YACC é muitas vezes apropriada. O fluxo de controle nesse caso pode ser apreciado na figura 4.2.

A saída fornecida pela função `yyparse()` depende do código que o usuário especificou junto com as regras gramáticas. No caso de uma linguagem interpretada, por exemplo, essa saída pode nem existir. No caso do sistema SPP essa saída é o conjunto de tabelas internas que já têm sido mencionadas.

4.4 Conclusão

A linguagem C foi escolhida como base do desenvolvimento, sendo usada tanto para o sistema especialista gerado como para o programa gerador. Se espera que essa escolha dê uma boa eficiência de execução à aplicação gerada. Por outro lado, uma interface ampla e flexível entre SPP e a linguagem C coloca o sistema mais perto do conceito de programação híbrida (mistura de procedural e não procedural), necessário para tornar compatíveis os sistemas especialistas e as aplicações com restrição de tempo. Parte da programação do SPP foi realizada também com as ferramentas de geração de compiladores YACC e LEX, que estão intimamente ligadas com o C.

A modularização do sistema foi realizada de modo que primeiramente o fonte SPP é lido, analisado e os diferentes pedaços de informação identificados são armazenados numa serie de tabelas internas. Posteriormente essas tabelas são compiladas e utilizadas para gerar o sistema especialista.

Capítulo 5

Aspectos específicos da linguagem do gerador

No presente capítulo far-se-á uma descrição da linguagem definida, apresentando em particular o funcionamento dos diferentes operadores e funções que possui. Por outro lado, num enfoque mais genérico, será analisada a lógica de funcionamento da máquina de inferência, indicando sua influência na forma de escrever as regras de produção.

Para uma descrição detalhada da sintaxe da linguagem o leitor deve dirigir-se ao Apêndice A, onde encontrará o BNF da linguagem (Bakus-Naur form).

5.1 Descrição básica

Um programa na linguagem SPP consistirá sempre de uma lista de *declarações* de atributos e funções externas, seguida de uma lista de *regras de produção*. As declarações acabam com a aparição da primeira regra.
As declarações de funções externas e atributos terão a seguinte forma:

$$(< tipo > \left\{ \begin{array}{l} \hat{} < atributo > \\ \hat{} < atributo >=< valor\ inicial > \\ < função > () \end{array} \right\} \star)$$

Onde <*tipo*> representa uma das palavras terminais seguintes:

- INT
- LONG
- SYMBOL
- FLOAT
- DOUBLE
- COMPLEX
- LIST
- REGn[1]

Todos os atributos e funções externas usados nas regras de produção devem ser declarados.

[1] n pode ser qualquer número positivo

O SPP gera um erro de sintaxe quando não for o caso. Um programa SPP poderia ter, por exemplo, as seguintes declarações:

```
(float ^alfa func() ^beta = 23.34)
(complex ^gamma = (12 24))
(reg45 ^delta ^sigma)
```

O atributo `^alfa` se declara como `float` mas sem valor inicial, enquanto `^beta` tem valor inicial igual a 23,34. Já `func` é uma função que retorna um `float`, e `^gamma` é um atributo complexo com valor inicial $12 + 24j$. Os atributos `^delta` e `^sigma` são simplesmente registros de 45 caracteres.
Por outro lado, a sintaxe das regras de produção será a seguinte:

(RULE [<*nome*>] [<*prioridade*>] IF <*condição*> $\star$ THEN
<*conclusão*> +)

O nome e a prioridade são parâmetros opcionais. O nome somente é utilizado quando a regra é referenciada na parte conclusão de outra regra. Se a prioridade é omitida, o sistema adota uma prioridade *por default*. Finalmente, a regra contem zero ou mais condições seguidas de uma ou mais conclusões.

As condições e as conclusões são estruturas *lisp-like* complexas. A notação das expressões é pré-fixada e diversas alternativas estão disponíveis. A forma geral das condições é a seguinte:

(<*predicado*> <*operando*> $\star$)
ou ((BETWN[2]<*operando*> <*operando*>) <*operando*>)
ou ((MEMBER <*operando*>) <*operando*>)
ou (EXIST ^<*atributo*>)
ou (FIRED <*regra*>)

Onde o <*predicado*> pode ser:

AND	OR	NOT
EQ	NE	LT
LE	GT	GE
<*função*>[3]		

Como já foi dito, todas as condições devem ser verdadeiras para que a regra seja válida. Os *operandos* podem ser atributos, constantes, ou expressões LISP com algumas restrições. BETWN é um predicado particular usado para testar se um operando (o terceiro) se encontra entre os outros dois. MEMBER avalia se o segundo operando (que deve ser de tipo SYMBOL) é um elemento da lista indicada pelo primeiro operando. A existência de um atributo se testa com o predicado EXIST, e se uma regra já disparou com FIRED. Note-se que a tentativa de testar a existência de um atributo não declarado daria num erro

[2]Em função do software disponível, a palavra BETWEEN ficou muito comprida para ser implementada
[3]Alguma função especial definida em C

de compilação. Considera-se que um atributo passa a existir somente quando lhe é dado algum valor.
O seguinte é um exemplo da sintaxe de uma regra de produção em SPP:

```
(rule reg10 12 if
      (eq 13 ^alfa)
      (exist ^gamma)
      (fired reg09)
      ((betwn 10.0 12.3) ^beta)
then
      (conclude ^alfa (* ^beta 3.1416))
      (set (fun12) ^delta ^sigma)
)
```

Esta regra só será válida se `^alfa` é igual a 13, `^beta` está entre 10,0 e 12,3, o atributo `^gamma` existe e a regra `reg09` já disparou. Na seção 5.4 se fará uma descrição mais detalhada do que pode aparecer na parte de conclusão das regras.

5.2 Os tipos de dados permitidos

Na definição dos tipos de dados que o sistema SPP suporta se tentou chegar a um equilíbrio entre poder de expressão e generalidade. Isto significa que por um lado há os tipos de dados básicos, como inteiro, real, complexo, simbólico, etc, que permitem realizar com pouco esforço diversas operações comuns. O cálculo matemático por exemplo, é simples nesses tipos de dados. Por outro lado se tem também os tipos de dados *definidos pelo usuário*, que permitem lidar com as situações incomuns trazendo assim generalidade ao sistema.

5.2.1 Tipos de dados básicos

Pode-se dividir estes tipos de dados do SPP em três grupos:

1. Tipos de dados que têm equivalente em C [KR78].
2. Tipos de dados que têm um pseudo equivalente em LISP [Cha85,Wer85].
3. Os números complexos.

Os do primeiro grupo são os clássicos tipos `int`, `long`, `float` e `double` do C. Eles são implementados por SPP como variáveis do mesmo tipo em C, portanto a equivalência é total.

Os do segundo grupo são os tipos `list` e `symbol`. A equivalência com LISP não é total pois somente é permitida lista de símbolos, e ainda somente são válidos os operadores CAR e CDR além do predicado MEMBER. Em particular, não é permitido um operador equivalente ao CONS do LISP.

O tipo de dados complexo é uma implementação feita especialmente para o SPP. Além de possibilitar todas as operações básicas com complexos, fornece uma forma de "construir" um operando complexo a partir de dois escalares. Por exemplo se `^alfa` fosse um atributo de tipo `float` armazenando o valor 3,1416, então

```
( ^alfa 55 )
```

seria um operando complexo com o valor $3,1416 + 55,0j$.

5.2.2 Tipo de dados lógico

Na implementação do SPP não se achou necessário incluir em particular um tipo de dados lógico. Esse tipo de dados é considerado como SYMBOL. De fato, qualquer símbolo é considerado como sendo `true` com a exceção do símbolo `nil` que é considerado `false`. Duas razões orientaram particularmente esta decisão:

1. É pouco provável que por causa dessa *fusão*, algum erro de tipagem não seja percebido pelo sistema, pois o tipo SYMBOL é muito simples e não existe mistura entre as funções CAR e CDR onde é usado como símbolo mesmo, e os operadores lógicos onde é usado como tipo lógico.

2. Como em SPP somente são permitidas listas de símbolos, o CAR de qualquer lista será sempre de tipo SYMBOL, e em particular o CAR de uma lista vazia será o símbolo `nil` (que é `false`). Desta forma se consegue conservar, mesmo que parcialmente, a convenção do LISP de que o CAR de uma lista vazia é `false`. Porém, para o LISP a própria lista vazia também é `false` enquanto no SPP a tentativa de usar uma lista como operando lógico daria em erro de sintaxe.

Em conseqüência, os seguintes exemplos serão expressões válidas:

```
(eq simbolo (car (cdr ^lista))
(and aaaa t)
(or bbbb nil)
```

As constantes simbólicas se escrevem simplesmente como uma palavra (`simbolo`, `aaaa` e `bbbb` no exemplo).

Além do símbolo `nil` que tem valor 0, também se encontra pré-definido dentro do sistema o símbolo `t` com valor 1. Aos símbolos definidos pelo usuário são dados valores de 2 em diante.

5.2.3 Tipos definidos pelo usuário

Para o funcionamento correto de um sistema baseado em regras de produção é necessário que todo o *estado* do sistema seja representado na memória de trabalho. De modo que não existe problema em definir qualquer estrutura de dados complexa, desde que ela seja

declarada como parte da memória de trabalho. E é com esse fim que foi desenvolvido o tipo de dados REG*n*, que declara um atributo como sendo alguma estrutura de dados de tamanho igual a *n* bytes.
O sistema SPP fornece um suporte básico como para:

1. Reservar espaço de memória para essa estrutura.

2. Em tempo de execução, monitorar as modificações que a estrutura sofre. (Se verá no próximo capítulo que a modificação de atributos é um elo fundamental no algoritmo usado pelo SPP).

3. Dar *forma* ao tipo de dados através de uma interface C.

Com os elementos próprios da linguagem, pouco poderia ser feito com atributos de tipo REG*n*. No entanto, é possível definir funções escritas em C que testem ou modifiquem esses atributos. Como exemplo suponha-se que se deseja utilizar atributos que representem uma data. Para tal poderia ser definida a seguinte estrutura em C:

```
struct date
    {int dia,mes,ano;};
```

e também a seguinte função `fs01` (na seção 5.3.2 se verá a razão do nome), para testar por exemplo se o ano é maior ou igual a um número determinado:

```
int fs01(data, ano)
struct date data;
int ano;
{if(data.ano >= ano)
    return 1;           /* True */
else
    return 0;           /* False */
}
```

Ainda pode ser definida esta outra função `cdata` para *construir* a data a partir de três números:

```
cdata(dia, mes, ano, i1, i2, data)
int dia, mes, ano;
int i1, i2;             /* i1 e i2 devem ser ignorados */
struct date *data;      /* data e um apontador a estrutura */
{data->dia = dia;
data->mes = mes;
data->ano = ano;
}
```

Este código C é suficiente como para escrever regras em SPP com atributos de tipo "data". Como a estrutura ocupa 6 caracteres em C, os atributos devem ser definidos de tipo reg6 para SPP:

```
(reg6 ^data1 ^data2)
```

Então a seguinte regra armazenará em ^data2 o 31 de dezembro de 1990 se o ano da ^data1 for superior ou igual a 1950:

```
(rule if
    (fs01 ^data1 1950)
then
    (set (cdata 31 12 1990) ^data2)
)
```

O formato do comando set será estudado na seção 5.4.3. Por enquanto só é necessário saber que SPP o transforma numa chamada à função cdata definida antes em C.

Com isto se tem todos os elementos necessários para trabalhar com qualquer estrutura de dados que se desejar. A única restrição é que ela deve ser de tamanho fixo, ou seja que a memória não pode ser *alocada* de forma dinâmica.

5.2.4 Inicialização de atributos

Seguindo a filosofia do C, todos os atributos declarados como sendo de algum dos tipos básicos (seção 5.2.1) podem ser inicializados por declaração. A sintaxe destas declarações é ilustrada abaixo:

```
(int ^alfa ^beta=55 ^gamma = (+ 14 18))
```

O atributo ^alfa é declarado sem valor inicial, ^beta com valor inicial igual a 55, e ^gamma com valor inicial igual a 32. Como se vê no exemplo, o valor inicial pode resultar da avaliação de qualquer expressão válida da linguagem. Numa expressão de inicialização podem até ser usadas funções *internas* definidas em C, como por exemplo sin() ou cos(). Na seguinte seção dar-se-á a definição do que se entende por função interna.

5.3 Operadores e funções

Da mesma forma que com LISP e C, no SPP as expressões têm um papel fundamental. A sintaxe pré-fixada dá à linguagem um aspecto *lisp-like*, embora seu modo de funcionamento esteja mais perto do C. A sintaxe geral de uma expressão é a seguinte:

$$(< \textit{operador ou função} >< \textit{operando}_1 > \ldots < \textit{operando}_n >)$$

Além desta forma genérica, existem formas particulares de expressões como por exemplo as que envolvem os predicados BETWN e MEMBER (seção 5.1), e outras que serão descritas adiante.

5.3.1 Operadores básicos

No contexto do SPP, se definem os conceitos de *operador* e *função* da seguinte maneira:

Operador representa uma operação onde cada operando pode ter diversos tipos, sendo que o sistema é responsável pela verificação desses tipos.

Função representa uma operação onde cada operando deve ser de um tipo fixo. O uso do tipo correto nos operandos é responsabilidade do usuário.

De acordo com estas definições, o SPP tem os seguintes operadores:

+	Soma	aplicáveis entre operandos de tipos `int`, `long`, `float`, `double` e `complex`
−	Subtração	
*	Produto	
/	Divisão	
NEG	Negativo	
ABS	Valor absoluto	
%	Módulo	aplicável entre operandos de tipos `int` e `long`
CAR	Cabeçalho	aplicáveis a operandos de tipo `list`
CDR	Resto da lista	

A seguir se apresentam exemplos de expressões válidas usando estes operadores:

`(+ 12 (13.4 15.1))`	Soma de um inteiro e um complexo. O resultado será $25,4 + 15,1j$.
`(car (cdr '(um dois)))`	O resultado é o símbolo `dois`.
`( / 3.0 2 )`	Divisão real. O resultado é 1,5.
`( / 3 2 )`	Divisão inteira. O resultado é 1.

Estas outras expressões, no entanto, contém erros de tipagem que o sistema vai detectar:

`( % 12.5 56 )`	Um dos operandos é real.
`( car tres )`	O operando é um símbolo e deveria ser uma lista.
`(+ 13 (car ^l)`	Um dos operandos do + é de tipo `symbol`.

A coerência dos tipos nas funções é responsabilidade do usuário. Por exemplo a função `sqrt` (raiz quadrada) está disponível no SPP para operandos de tipo `double`. Se for usada com um tipo diferente (como inteiro) nenhum erro será detectado pelo compilador, embora o sistema com certeza irá funcionar mal. Evidentemente, este problema do SPP é herdado diretamente da linguagem C.

5.3.2 Funções externas e internas

Pela própria natureza de um sistema de produção, seria absolutamente inaceitável que os atributos fossem modificados no momento de avaliar uma condição. Eles só podem ser alterados pelo disparo de uma regra. Por outro lado é igualmente inaceitável que uma função chamada por alguma condição possua memória estática, pois poderia resultar em dois valores de retorno diferentes para os mesmos dados de entrada.

Por causa disto foi necessário estabelecer uma diferença entre as funções que podem aparecer nas condições ou em expressões de inicialização, que são chamadas de *internas* e portanto não devem ser declaradas, e por outro lado funções que só podem ser usadas na parte de conclusão das regras, chamadas *externas* e que devem ser declaradas.

Existe total liberdade para o usuário definir as suas funções externas na linguagem C, contanto que elas sejam declaradas para que o SPP conheça o tipo de seu valor de retorno. Para as funções internas no entanto, se entendeu que não poderia ser admitida a mesma liberdade, pois o desrespeito às regras mencionadas no primeiro parágrafo levaria com certeza a um mau-funcionamento do sistema. Em qualquer caso, os tipos das funções (internas ou externas) não podem ser nem LIST nem REGn para evitar problemas com as estruturas de dados complexas associadas de um modo ou outro a esses tipos.

A forma mais simples que o usuário tem para se comunicar com a linguagem C é através de uma função externa. Se se quisesse dispor por exemplo de uma função para calcular a raiz cúbica de um número (que não está disponível no SPP), deveria simplesmente programar o algoritmo como uma função C:

```
double cubic ( numero )
double numero;
{
    ...
}
```

e depois declará-la para o SPP da seguinte forma:

```
( double cubic() )
```

Com isto já se pode escrever regras que contenham por exemplo:

```
( conclude ^alfa ( cubic 27.0 ) )
```

que armazena no atributo `^alfa` o valor 3,0. Porém, a tentativa de usar `cubic` numa condição provocaria um erro de compilação, pois ela foi definida como externa.

Por outro lado, funções internas usadas em expressões de inicialização ou pelo mecanismo de pré-compilação (que será apresentado no próximo capítulo), deveriam ser conhecidas não somente pelo sistema especialista gerado mas também pelo programa gerador. Em conseqüência as funções internas ficam ainda divididas em dois grupos:

1. As que são conhecidas tanto pelo SPP como pela aplicação que ele gera, como por exemplo as funções `sqrt` (raiz quadrada), `sin` (seno) ou `log10` (logaritmo em base

10). Estas funções têm a mesma hierarquia que os operadores + ou −, e podem aparecer em todos os lugares onde eles aparecem. De fato o operador + por exemplo, seria equivalente a um conjunto de funções internas como as abaixo relacionadas:

- Soma de dois inteiros.
- Soma de um inteiro com um longo.
- Soma de dois longos.
- Soma de um inteiro com um ponto flutuante.
- Soma de um inteiro com um complexo.
- etc.

A vantagem dos operadores na presente implementação, é que conseguem expressar de uma forma compacta algo que só seria possível implementar usando um número grande de *funções internas.*

Evidentemente, as funções internas deste grupo só podem ser definidas por alguém com acesso ao código fonte do SPP. De qualquer forma, este é um ponto onde o sistema fica *aberto* pois outras funções podem ser adicionadas sem esforço, aumentando o poder de expressão do mesmo.

2. Funções que só precisam ser conhecidas pela aplicação gerada. Um conjunto de nomes de função associados aos seus tipos está *catalogado*[4] no sistema para dar a possibilidade de definir funções internas sem ter necessidade de alterar o código fonte do SPP. Usando um desses nomes, o usuário pode definir em C funções de nível interno.

   O motivo de ter somente um conjunto restrito de nomes para esta finalidade é que assim o sistema fica pelo menos parcialmente protegido contra o uso desses nomes em determinados lugares que poderiam prejudicá-lo, como por exemplo em declarações de inicialização.

   A função `cubic` do exemplo anterior, não podia ser usada em premissas embora evidentemente não possuísse memória estática nem alterasse atributos. Para usá-la numa condição, se poderia utilizar um dos nomes disponíveis para funções internas definidas pelo usuário. Para uma função interna de tipo `double` se tem por exemplo o nome `fd01`. Então, o usuário escreveria:

```
double fd01 ( numero )
double numero;
{   ...
    (igual que a função cubic)
    ...
}
```

[4]Se tem os nomes `fi01` até `fi10` para inteiros, `fl01`, ... para `long`, `fs01`, ... para `symbol`, `ff01`, ... para `float`, `fd01`, ... para `double` e `fc01`, ... para `complex`.

e em conseqüência a regra:

```
( rule if
    ( gt ^alfa ( fd01 ^beta ) )
then
    ( conclude ^alfa ( fd01 27.0 ) )
)
```

estaria correta e significaria que se `^alfa` $> \sqrt[3]{\texttt{\^{}beta}}$ então ao atributo `^alfa` lhe será atribuído o valor 3,0.

No entanto, quando se implementarem funções como esta `fd01` raiz cúbica, além de respeitar as regras mencionadas no começo da seção (não usar memória estática nem alterar atributos), será necessário cuidar também que estas funções não sejam usadas em expressões de inicialização nem pelo mecanismo de pré-compilação. Em particular, toda função com algum parâmetro de tipo REG*n* está garantida de não cair em nenhum desses casos. E é justamente com atributos de tipo REG*n* que estas funções são mais úteis, pois eles não podem ser inicializados e toda a semântica da estrutura que eles representam deve ser definida por meio de funções em C.

As funções internas do primeiro grupo, disponíveis na presente versão do SPP, são a maior parte das rotinas da biblioteca matemática do C (`sin()`, `cos()`, etc.) [Cor84]. Se deve ressaltar que uma vez incluídas (ou melhor *catalogadas*) no código fonte do SPP, essas funções passam a ser de utilização transparente para o usuário. Só com as do segundo grupo é que o usuário deve se preocupar em fornecer a definição.

5.3.3 Expressões particulares

Nesta seção serão tratadas as expressões da linguagem do SPP que diferem da sintaxe genérica definida na seção 5.3. Estas formas são, indo das mais simples às mais complexas, as seguintes:

- Quaisquer atributos, números ou constantes simbólicas são considerados expressões. Para diferençá-los das constantes simbólicas, o nome dos atributos começa sempre pelo símbolo "^".

- Função que fornece a prioridade de uma regra:

  (PRI <*nome da regra*>)

- Expressão de uma lista de símbolos de forma explícita:

  '(<*nome de símbolo*> $\star$)

  como por exemplo:

```
'(um dois tres quatro)
```

onde `um`, `dois`, ..., etc. são constantes simbólicas.

- Construção de complexo:

(<*escalar*> <*escalar*>)

onde esses *escalares* são qualquer expressão de tipos `int`, `long`, `float` ou `double`.

- Finalmente, qualquer condição (ver seção 5.1) pode também ser usada como expressão, desde que seja no âmbito de algum operador lógico.

Contudo, pode-se apreciar que mesmo diferindo da sintaxe genérica apresentada na seção 5.3, estas formas ainda seguem o estilo *lisp-like*.

5.4 Parte conclusão das regras

Na parte conclusão das regras podem aparecer ações diretas sobre a memória de trabalho, ações de comunicação com o *mundo externo* (com outras tarefas, o usuário, etc.), ou ações de controle. Se tratará separadamente cada um destes tipos de ações.

5.4.1 Ações que afetam a memória de trabalho

As ações que afetam diretamente os atributos descritos na memória de trabalho são as mais simples e clássicas, e por outro lado são também as únicas imprescindíveis num sistema baseado em regras.
A sintaxe deste tipo de ações é:

(conclude ^<*atributo*> <*expressão*>)
(del ^<*atributo*>)

A primeira expressão permite atribuir a ^<*atributo*> o resultado da avaliação de <*expressão*>. Deve haver compatibilidade entre os tipos do atributo e da expressão, embora sejam possíveis as conversões de tipos da linguagem C, estendidas também para o tipo `complex`.

A segunda expressão permite apagar um atributo da memória de trabalho. Tem de se notar que o atributo não desaparece fisicamente da memória, mas logicamente. De qualquer forma o valor armazenado está perdido.

A seguinte é uma lista de exemplos destas expressões, onde se supõe que `^alfa` é inteiro, `^beta` é `float`, `^gamma` é complexo e `^delta` é uma lista:

`(conclude ^alfa 3.1416)`	O valor 3 é armazenado no atributo `^alfa`.
`(conclude ^beta (12.3 14))`	O valor 12,3 é armazenado no atributo `^beta`.[5]
`(conclude ^gamma 1)`	O valor $1,0 + 0j$ é armazenado no atributo `^gamma`.
`(conclude ^delta 125)`	Dá erro de sintaxe por incoerência de tipos.
`(del ^delta)`	Apaga logicamente o atributo `^delta`.

Os dois comandos, `conclude` e `del` podem ser usados sobre atributos existentes ou não, sem produzir erro. Após `conclude` o atributo necessariamente existe, e após `del` necessariamente deixa de existir. Não há problema em apagar um atributo inexistente, desde que tenha sido declarado. Quando se analisou o predicado EXIST na seção 5.1, comentou-se a diferença entre um atributo inexistente porém declarado, e um que não foi declarado.

5.4.2 Ações de controle

Num sistema de produção com um número importante de regras fica muito difícil estabelecer o controle do sistema (ou seja a ordem de encadeamento das regras). Para contornar esse problema os sistemas especialistas clássicos têm introduzido comandos chamados *de controle* que orientam o sistema na etapa de resolução de conflitos. Tradicionalmente existe uma variedade muito grande de ações desse tipo. No SPP foram implementadas duas delas:

(`pri` <*nome da regra*> <*expressão*>)
(`stop`)

A primeira ação muda a prioridade da regra especificada para o valor estabelecido na expressão. Qualquer expressão é válida, desde que seja de tipo `int` ou `long`. Existem no SPP 16 níveis de prioridade disponíveis.

A ação `stop` faz com que a máquina de inferência pare após a execução da regra. É importante salientar que as ações escritas depois dela serão igualmente executadas.

5.4.3 Ações de comunicação

Da mesma forma que com as ações de controle, aqui se procura fornecer um conjunto mínimo de comandos suficientemente expressivo. Por essa razão foram implementados um comando de *entrada* (`set`), e um comando de *saída* (`print`) com a sintaxe a seguinte:

[5] Na atribuição entre um complexo e um escalar se escolheu usar a parte real no lugar do módulo pois este já está disponível na primitiva ABS.

(set ([<*função*><*expressão*> *]) ^ <*atributo*> +)
(print ([<*função*><*expressão*> *])[<*expressão*> *])

Omitindo a *função* entre parênteses, a comunicação é feita através de uma interface padrão fornecida pelo SPP. Com o comando `set` um conjunto de um ou mais atributos é dado pelo usuário a partir do teclado, e com `print`, zero ou mais expressões são avaliadas e imprimidas na tela. Se uma expressão é um atributo, o nome dele também é mostrado.

Se por exemplo, na parte conclusão de uma regra se quisesse ler do teclado os atributos `^alfa` e `^beta`, deveria ser usado o comando:

```
( set () ^alfa ^beta )
```

A execução da aplicação prosseguiria só após o usuário ter digitado o valor dos dois atributos. Já a execução do comando:

```
( print () ( + 24 8 ) ( / 3.0 2 ) ^alfa )
```

na parte conclusão de uma regra faria com que saíssem na tela:

```
32
1.5
alfa=144
```

supondo que o valor armazenado no atributo `^alfa` fosse 144.

Se se especifica uma *função* entre parênteses, significa que o usuário irá fornecer uma outra interface particular utilizando a linguagem C. Neste caso o sistema SPP constrói uma chamada a essa função seguindo estes critérios:

- A chamada é construída com informações que possibilitem ao usuário implementar a função admitindo um número variável de parâmetros, facilidade oferecida por C. Para isto, SPP coloca em ordem na lista de parâmetros:

  1. Uma lista com um número fixo de parâmetros (que não pode variar entre chamadas) especificados entre parênteses logo em seguida da *função*. Tanto em `set` como em `print`, esta lista, que é opcional, só pode ser usada para transferir informação do sistema especialista para a função.
  2. Um conjunto de dados indicando a *quantidade* e o *tipo* dos parâmetros que vêm a seguir.
  3. A lista (variável entre chamadas) de parâmetros especificada de acordo com os dados do ponto anterior. Para `print` esta é a lista das zero ou mais expressões da direita, enquanto para `set` é a lista dos endereços dos *um ou mais* atributos da direita. Deve-se lembrar que uma função C só pode modificar uma variável definida fora dela quando lhe é fornecido o endereço desta variável.

  Deve ressaltar-se que fazendo uso principalmente da parte fixa (ponto 1), ou principalmente da parte variável (pontos 2 e 3), se pode obter uma função mais particular ou mais genérica.

- O tipo da função não interessa pois ela é chamada como um procedimento no programa gerado, e em conseqüência também não precisa ser declarado.

Tabelas com os nomes dos atributos, das regras, etc. são acessíveis desde a função de interface através de um comando `extern`.

O seguinte exemplo ilustrará a forma em que SPP realiza a geração da chamada a uma função C a partir de um comando `set`. Para a conclusão:

```
(set (funset (* 2 ^alfa)) ^beta ^gamma)
```

o SPP criaria a seguinte chamada em linguagem C:

```
funset( 2*alfa, 44, 51, -1, &beta, &gamma);
```

onde 44 seria o índice do atributo `^beta` e 51 o do atributo `^gamma`. O símbolo "&" na frente de `beta` e `gamma` significa "endereço de". Os parâmetros de chamada são colocados pela linguagem C num "stack" de modo que `funset` pode recuperá-los em ordem, de esquerda a direita. Assim, após `2*alfa` (que é a parte fixa da chamada), o 44 indica que o endereço de `^beta` está no começo da parte variável da chamada, o 51 indica que lhe segue o endereço de `^gamma`, e o -1 indica que não há mais endereços de atributos após o de `^gamma`. Após encontrar o -1, `funset` reconhece que só tem que recuperar mais dois parâmetros do stack: os endereços de `^beta` e de `^gamma`.

Uma função implementada usando esse mecanismo de quantidade variável de parâmetros, fornece uma grande flexibilidade. Seguindo com o exemplo anterior, `funset` poderia também ser usada nestes outros comandos `set` do mesmo programa fonte de regras:

```
(set (funset 234) ^gamma)
(set (funset (/ ^gamma 23)) ^alfa ^beta ^gamma)
```

que gerariam por sua vez as seguintes chamadas:

```
funset( 234, 51, -1, &gamma);
funset( gamma/23, 102[a], 44, 51, -1, &alfa, &beta, &gamma);
```

[a] 102 seria o índice de `^alfa`

A forma como SPP gera uma chamada a partir de um comando `print` é muito similar. Porém deve-se lembrar que a lista (variável entre chamadas) de parâmetros do comando `print` admite além de simples atributos, expressões quaisquer. Então, no lugar onde colocava o índice do atributo, para uma expressão SPP coloca um índice negativo fornecendo o tipo da expressão de acordo com a seguinte tabela:

-2	INT
-3	LONG
-4	SYMBOL
-5	FLOAT
-6	DOUBLE
-7	COMPLEX
-8	LIST
-9	REG1
-10	REG2
⋮	etc.

Então por exemplo o comando:

```
(print (funp) (* 12.3 14) ^alfa 21)
```

seria transformado na chamada:

```
funp( -6, 102, -2, -1, 12.3*14, alfa, 21)
```

onde o -6 indica que a expressão 12.3*14 é de tipo `double` e o -2 indica que 21 é de tipo `int`.

Modificação de atributos de tipo REG*n*

Como não pode haver funções de tipo REG*n*, nem há operadores com esse tipo, o uso da ação `conclude` fica restrito nesse caso à simples cópia. Para realizar operações e armazenar dados particulares em atributos REG*n* é necessário usar o comando `set` com uma "função de interface" definida pelo usuário, como no exemplo da seção 5.2.3.

Em conseqüência, a semântica dos atributos de tipo REG*n* se define em C com funções *internas* para testar o seu estado, e com funções `set` para atribuir-lhes valores.

5.5 Funcionamento da máquina de inferência

Do ponto de vista do modelo de sistemas de produção, o motor de inferência do sistema SPP trabalha sem backtracking, com raciocínio não monotônico e em encadeamento para frente. Em conseqüência se diferença de outros sistemas similares somente na forma de admitir uma regra no conjunto de conflitos, e em como se escolhe depois uma das regras do conjunto de conflitos para ser disparada.

Nas próximas seções serão tratados esses aspectos do SPP, tomando em conta a forma como eles condicionam a programação de regras por parte do usuário.

5.5.1 Condições para que uma regra ingresse no conjunto de conflitos

Num primeiro momento pareceria que, de acordo com o modelo de sistemas de produção, a avaliação verdadeira de todas as condições seja suficiente para habilitar uma regra a disparar. Porém, ainda poderia acontecer que na parte conclusão se referenciasse algum atributo inexistente. Se por exemplo se tivesse `^alfa` igual a 5, `^beta` igual a 6, e `^gamma` não existisse então a seguinte regra:

```
(rule if
      (eq ^alfa 5)
      (eq ^beta 6)
then
      (conclude ^alfa ^gamma))
```

poderia ser considerada como estando habilitada para disparar, pois as duas premissas são verdadeiras. A conclusão, no entanto, não poderia ser executada. Há várias formas de solucionar este tipo de problemas:

1. Fazer com que o analisador sintático denuncie um erro de compilação. Esta alternativa no entanto, pode levar a considerar sistemas de regras perfeitamente corretos como contendo erros, pois uma regra como a apresentada nunca seria permitida, embora o programador soubesse, pelo contexto, que toda vez que ela fosse disparar o atributo `^gamma` iria existir.

2. Deixar a responsabilidade ao programador, gerando um *runtime error* só quando realmente o problema acontecer.

3. Considerar que uma regra como a do exemplo, onde as condições são verdadeiras mas um atributo da conclusão —no caso o `^gamma`— não existe, na verdade não está habilitada para disparar.

O sistema SPP adota essa última solução, e portanto quando uma regra fica habilitada para disparar é certo que nenhum erro pode acontecer.

De fato, após a análise sintática SPP insere dentre as condições da regra uma expressão `( exist ^<`*atributo*`> )` por cada atributo cuja existência é necessária para a avaliação da parte conclusão. Assim a regra do nosso exemplo seria equivalente a:

```
(rule if
      (eq ^alfa 5)
      (eq ^beta 6)
      (exist ^gamma)
then
      (conclude ^alfa ^gamma))
```

Resumindo, para que uma regra possa disparar se deve cumprir:

1. Todas as premissas escritas na regra devem ser válidas (isto se refere às premissas que o usuário coloca efetivamente, não considerando os (`exist` ^*<atributo>*) que o sistema insere internamente).

2. Todos os atributos que aparecem em alguma expressão da parte conclusão devem existir (salvo se estão exclusivamente numa expressão (`exist ...`), que justamente testa a existência).

Estas condições são necessárias para que a regra se encontre no conjunto de conflitos, mas devido ao **princípio de reflexividade** (uma regra não dispara duas vezes com os mesmos dados) não são suficientes. De fato, em função deste princípio as regras são retiradas do conjunto de conflitos após o disparo, e em conseqüência se poderia ter regras cumprindo as condições de cima mas fora deste.

O momento em que se considera uma regra para ingressar no conjunto de conflitos, é determinado assim:

> *Uma regra é testada para ser adicionada ao conjunto de conflitos só quando se modifica algum elemento das expressões que aparecem nas premissas, incluindo os* (`exist ...`) *que o analisador sintático insere.*

Em função deste método de trabalho do SPP, as seguintes situações podem acontecer:

- A mudança somente da prioridade de uma regra não fará com que ela seja testada para ingressar no conjunto de conflitos.

- Atributos e expressões (`pri` *<regra>*) na parte conclusão provocarão comportamentos bem diferentes no sistema. Considere-se por exemplo que num determinado ponto da execução do sistema especialista estas duas regras estão fora do conjunto de conflitos:

```
(rule reg1 if                          (rule reg2 if
    t                                      t
then                                   then
    (conclude ^alfa ^beta))                (conclude ^alfa (pri reg3)))
```

  Se ao atributo `^beta` lhe fosse atribuído algum valor, a premissa `(exist ^beta)` (invisível) inserida pelo sistema, teria um dos seus elementos alterados e passaria a valer *true*. Então, segundo a convenção especificada a regra `reg1` deveria ser considerada para ingressar no conjunto de conflitos. Como a outra premissa também é verdadeira (`t = true`), ela seria imediatamente adicionada ao conjunto de conflitos. No entanto, a modificação da prioridade de `reg3` não faria com que `reg2` ingressasse no conjunto de conflitos pois esta regra não tem nenhuma premissa adicional inserida. De fato, como a única premissa de `reg2` (`t`) é uma constante, `reg2` é uma regra que só pode disparar uma vez no contexto do SPP.

É importante ter presente o conteúdo desta seção para entender em todos os casos o comportamento de uma aplicação gerada por SPP.

5.5.2 Resolução de conflitos

A etapa de resolução de conflitos é onde normalmente se encontram mais diferenças entre distintos sistemas de produção. As estratégias de controle escolhidas para serem implementadas têm uma influência muito grande na forma como os sistemas especialistas conseguem resolver os problemas.

No caso do sistema SPP, se adotou um conjunto de critérios simples para a resolução dos conflitos, seguem-se em ordem as seguintes etapas, passando ao ponto seguinte só se o anterior não foi suficiente para resolver os conflitos:

1. A regra de maior prioridade é escolhida.

2. A regra que entrou mais recentemente no conjunto de conflitos é escolhida (estratégia de recentidade ou *recency* em inglês).

3. A regra mais específica é escolhida. No SPP isto se avalia pela quantidade de premissas que a regra possui (maior número de premissas implica numa regra mais específica).

4. A regra escrita antes no programa é escolhida. Este é o último critério a ser usado e na realidade diminui a característica não determinista do sistema. Porém, pode ser útil se se quiser analisar a fundo o tempo de execução de uma aplicação de tempo real, pois para cada conjunto de dados se pode dizer exatamente como o sistema irá evoluir.

Estes critérios, juntamente com as ações de controle estudadas na seção 5.4.2 constituem um conjunto mínimo, porém suficiente como para lidar com um grande número de problemas. Em particular, o fato de poder modificar a prioridade de uma regra durante a execução do programa, dá um poder de controle bastante grande pois a prioridade é o primeiro critério de resolução de conflitos a ser utilizado.

5.6 Conclusão

O sistema SPP suporta tipos de dados básicos ou definidos pelo usuário. Os primeiros tomam como ponto de referência as linguagens C e LISP, enquanto os segundos são programados usando funções C.

Todos os operadores matemáticos comuns estão implementados, e ainda se tem a possibilidade de definir funções *internas* em C com a mesma hierarquia destes operadores. Funções C para serem usadas nos comandos `set` e `print` e funções externas completam o conjunto de opções da interface com C.

Uma estratégia de resolução de conflitos clássica (incluindo o critério de prioridade das regras), junto com comandos de controle que podem alterar essa prioridade, constituem os principais elementos de controle do sistema.

Capítulo 6

Conceitos e algoritmos desenvolvidos

O algoritmo de compilação do conhecimento desenvolvido para o sistema SPP, tenta realizar a maior quantidade possível de tarefas durante a fase de geração do sistema especialista, deixando para o tempo de execução somente o imprescindível. Por causa disso, tem-se na realidade dois algoritmos diferentes, um usado para gerar o sistema e outro utilizado na execução.

Neste capítulo tratar-se-ão primeiramente os algoritmos e conceitos relacionados com a geração do sistema especialista. Se mostrará como se ordena a informação fornecida pelas regras, antecipando todos os cálculos que sejam possíveis para depois poupar tempo de execução. Finalmente se analisará o algoritmo e as estruturas de dados utilizados pela aplicação gerada.

6.1 Estruturas de dados e conceitos usados na geração

A determinação do novo conjunto de conflitos, após cada disparo de regra, é um processo geralmente custoso em tempo de execução. O algoritmo desenvolvido para a etapa de geração visa reduzir a um mínimo esse tempo, determinando , para cada regra do sistema especialista, o subconjunto das regras que poderiam entrar ou sair do conjunto de conflitos se ela disparasse.

O algoritmo desenvolvido se sustenta em estruturas de dados e conceitos que serão apresentados a seguir. Basicamente as estruturas de dados utilizadas são tabelas onde alguns dos seus elementos são listas [Wir76] cujos nós apontam a entradas de outras tabelas. As tabelas vão se preenchendo a medida que as regras vão sendo analisadas, e as listas de apontadores fornecem os caminhos de procura necessários para colocar a informação em ordem no programa gerado.

6.1.1 As tabelas do gerador

O algoritmo do gerador se baseia principalmente nas tabelas chamadas de *atributos*, de *condições* e de *regras*; embora existam outras tabelas complementares.

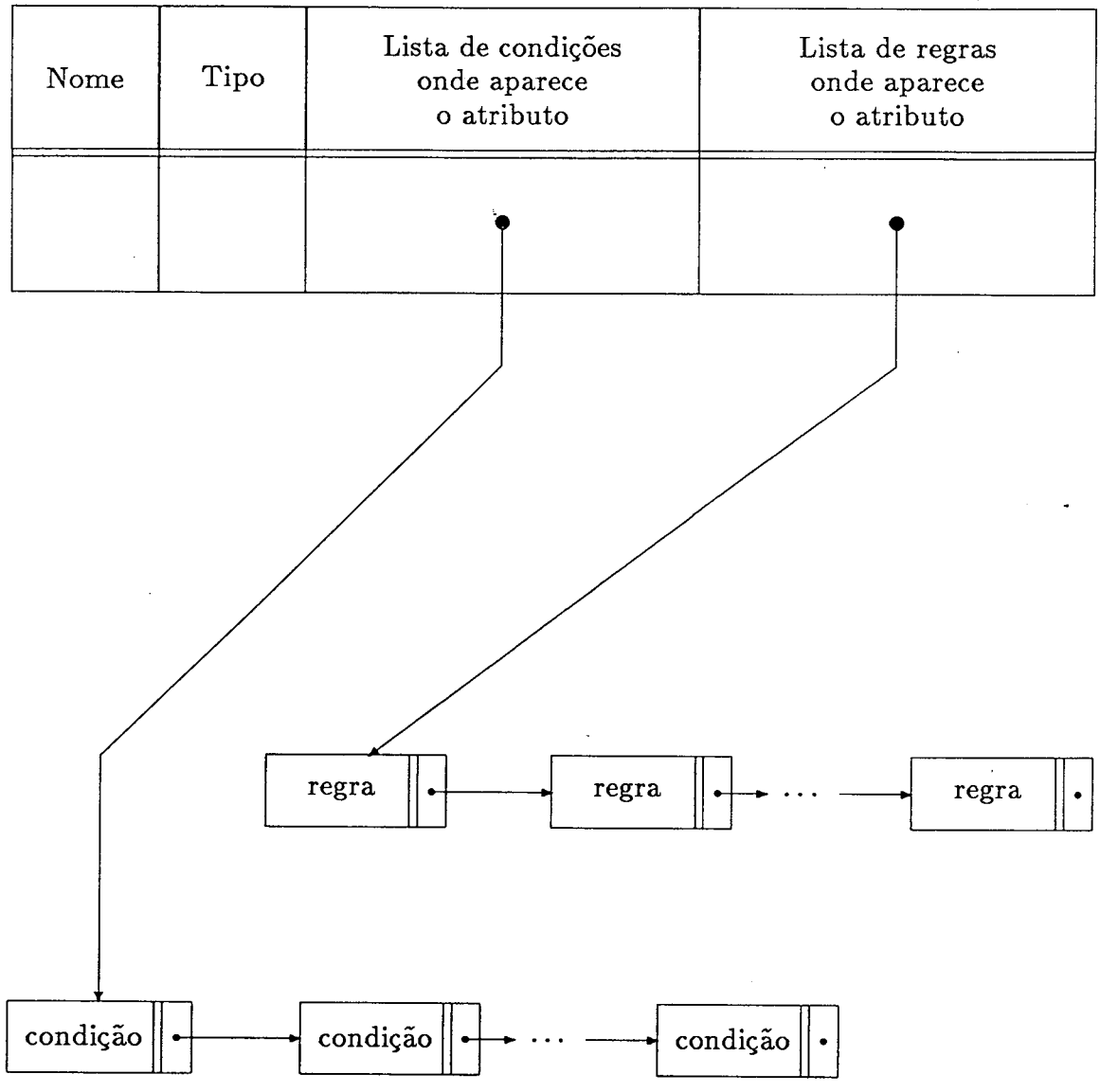


Figura 6.1: Tabela de atributos

Na tabela de atributos se armazena, além do nome e o tipo, a lista de condições e a lista de regras onde aparece o atributo. Nessas listas o algoritmo impede que hajam elementos repetidos. Em particular, a lista de regras onde aparece o atributo representa o conjunto de regras que precisam da existência do atributo para poder ser habilitadas, e em conseqüência não estão incluídas regras onde o atributo só aparece em:

```
(conclude ^<atributo>...)
(del ^<atributo>)
(set (...) ...^<atributo>...)
```

Na figura 6.1 se pode apreciar um diagrama esquemático desta tabela.

A tabela de condições possui uma "linha" para cada condição diferente que o algoritmo identifica. Os dados contidos são basicamente o texto traduzido para a linguagem C da

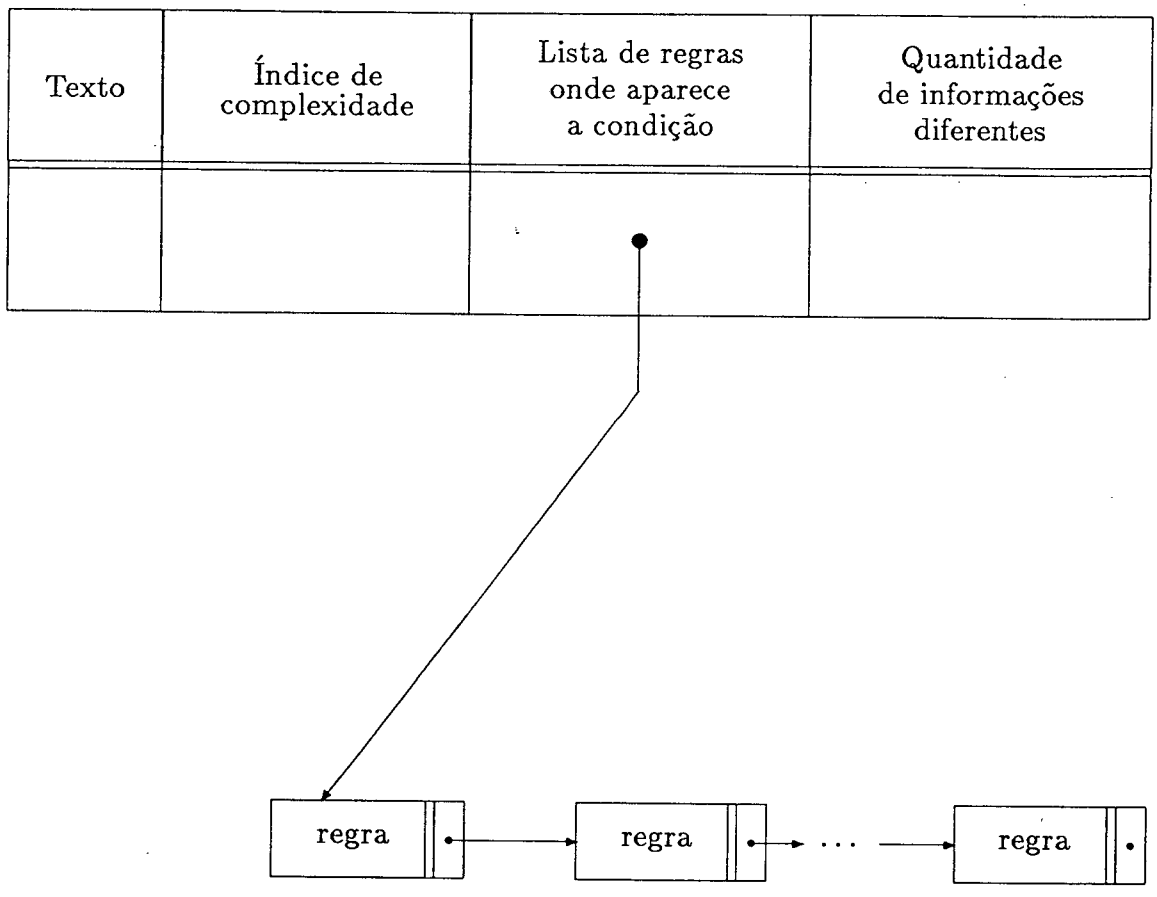


Figura 6.2: Tabela de condições

condição, um índice de complexidade, a lista de regras onde aparece a condição, e a quantidade de informações diferentes envolvidas na condição. Este último dado é usado só para fins de pré-avaliar a premissa em tempo de compilação, coisa que se estudará na seção 6.1.4. O índice de complexidade é um número obtido em forma heurística que tenta avaliar o grau de complexidade da condição. Se verá mais em detalhe na próxima seção. Na figura 6.2 se pode apreciar um diagrama esquemático desta tabela.

A tabela de regras é talvez a estrutura de dados mais complexa do sistema SPP. Cada "linha" da tabela contém uma série de dados relativos a cada regra de produção do programa fonte. As informações de maior relevância são as seguintes:

- O nome da regra se foi especificado (lembrar que era um parâmetro opcional)
- Um vetor contendo o índice dentro da tabela de condições, das premissas da regra. No sistema SPP, o número de máximo de condições por regra está limitado a 16 para poder armazenar nos 16 bits de uma variável de tipo `int` o estado verdadeiro ou falso de cada condição. Portanto este vetor é de tamanho fixo igual a 16.
- O texto, na linguagem C, da parte de conclusão da regra.
- A prioridade inicial da regra, se foi especificada. Se não foi o sistema adota uma

prioridade por *default* igual a 7.

- O índice de especificidade da regra. Já tinha sido dito na seção 5.5.2 que este índice representa a quantidade de premissas que a regra possui.

- A quantidade de atributos diferentes cuja existência é necessária para que a regra possa ser habilitada.

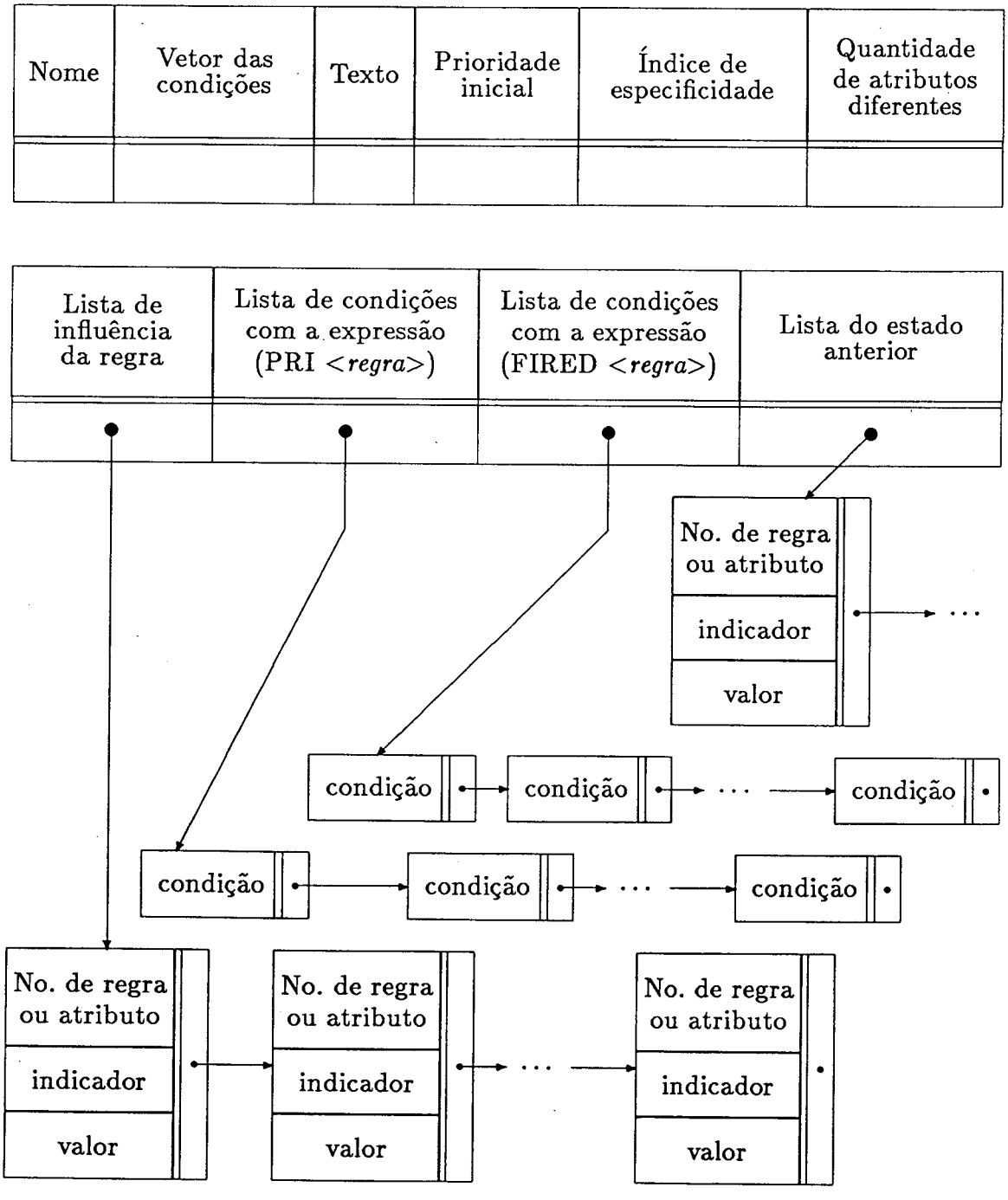


Figura 6.3: Tabela de regras

- A *lista de influência* da regra. É uma lista com todos os atributos ou regras que se vem afetados pelo disparo desta.

- A lista de condições onde aparece a expressão (PRI <*regra*>).

- A lista de condições onde aparece a expressão (FIRED <*regra*>).

- A lista do *estado anterior.* É uma lista com todas as informações que se podem deduzir do estado em que o sistema se encontrava antes do disparo da regra, só pelo fato dela ter disparado. Se por exemplo disparasse a regra:

```
(rule if
      (eq ^alfa 5)
      (eq 10 (pri avalia))
then
      (print ...)
)
```

  então necessariamente antes do disparo, o atributo `^alfa` valia 5 e a prioridade da regra de nome `avalia` era 10. Esta lista é usada para a pré-avaliação das condições que se estudará na seção 6.1.4.

Na figura 6.3 se pode apreciar um diagrama esquemático desta tabela.

Para completar o conjunto de estruturas de dados usadas pelo algoritmo de geração, se devem mencionar ainda a tabelas de símbolos, a tabela de funções e a matriz associada ao tipo LIST que será estudada na seção 6.2.1.

6.1.2 As listas de influência e do estado anterior

Como se pode observar na figura 6.3, as listas de influência e do estado anterior são estruturas de dados mais complexas que as outras listas onde cada nó contém somente um índice de regra, de atributo ou de condição. Por outro lado, embora sejam de estrutura similar, a informação armazenada numa e outra lista tem um significado diferente.

A lista do estado anterior, como já foi dito, contem informações sobre atributos e regras que necessariamente deveriam ser verdadeiras para que a regra disparasse. Essas informações representam uma parte *conhecida* do estado anterior, por isso o nome da lista. Cada nó inclui os seguintes dados:

Índice de regra ou atributo Como já se estudou, as informações conhecidas do estado do sistema anterior ao disparo da regra, podem envolver atributos ou outras regras. Segundo o valor do campo **indicador** este índice representará um atributo ou uma regra.

Indicador Pode representar as seguintes situações relacionadas com o dado anterior:

1. O índice é de um atributo que necessariamente existia no momento do disparo da regra.
2. O índice é de um atributo que devia ser igual a um valor conhecido quando a regra disparou. Evidentemente esta situação implica também a do ponto anterior e em conseqüência leva precedência sobre ela. Neste caso ainda, esse valor conhecido é armazenado no campo **valor**.
3. O índice é de uma outra regra cuja prioridade devia ser conhecida no momento de disparo da regra, e essa prioridade conhecida está armazenada no campo **valor**.

Valor Segundo a quantidade armazenada no **indicador**, este campo pode representar o valor conhecido de um atributo ou a prioridade conhecida de uma regra.

Nesta lista, nem atributo nem regra podem estar repetidos, em conseqüência o número de nós com o valor do **indicador** segundo os pontos 1. ou 2., deve igualar à quantidade de atributos diferentes cuja existência é necessária para a habilitação da regra (campo da tabela de regras).

A lista de influência, por outro lado, além das alterações que o disparo da regra provoca no "estado" do sistema, inclui também a informação do "estado anterior" que segue sendo válida após o disparo. Os dados contidos em cada nó desta lista são os seguintes:

Índice de regra ou atributo Como já foi estudado, a parte de conclusão de uma regra por um lado pode alterar atributos por meio dos comandos `conclude`, `set` ou `del`, e por outro lado pode alterar outras regras (no caso só a prioridade delas) por meio do comando `pri`. Cada regra ou atributo afetados pelo disparo estão representados por um nó nesta lista de influência. Este índice, como já se explicou, também pode representar alguma regra ou atributo com informações conhecidas que não foram afetadas pelo disparo.

Indicador Pode representar as seguintes situações relacionadas com o dado anterior:

1. O índice indica um atributo que necessariamente existe pelo fato da regra ter disparado. Embora não se conheça o valor armazenado nele, se sabe que não foi alterado pelo disparo da regra.
2. O índice indica um atributo cujo valor conhecido não foi alterado pelo disparo. Esse valor, conhecido simplesmente pelo contexto da própria regra, se armazena no campo **valor**.
3. O índice indica um atributo afetado por um comando `conclude` ou `set` na conclusão da regra. O valor armazenado porém, não é conhecido.
4. O índice indica um atributo alterado por um comando:

   (`conclude` ^*<atributo>* *<expressão constante>*)

   A avaliação dessa expressão constante se armazena no campo **valor**.

5. O índice indica um atributo apagado por um comando `del`.
6. O índice representa a prioridade de alguma regra, conhecida pelo contexto da regra que disparou e que não foi afetada por dito disparo. Essa prioridade se armazena no campo **valor**.
7. O índice representa uma regra cuja prioridade foi modificada por um comando `pri` para um valor que não é conhecido.
8. O índice representa uma regra afetada por um comando:

   (`pri` <*regra*> <*expressão constante*>)

   A avaliação dessa expressão constante se armazena no campo **valor**.

Valor Segundo a quantidade armazenada no **indicador**, este campo pode representar o valor conhecido de um atributo ou a prioridade conhecida de uma regra.

Nesta lista de influência, também não são permitidos dois nós referenciando o mesmo atributo ou a mesma regra.

Do que tem sido exposto nesta seção se pode deduzir facilmente que em relação aos atributos e regras mencionados nas listas vale:

$$\textit{lista do estado anterior} \subseteq \textit{lista de influência}$$

pois todo atributo ou regra que apareça na primeira necessariamente também aparecerá na segunda, embora talvez com um indicador diferente.

6.1.3 Complexidade das condições

Quando o algoritmo de execução precisa determinar se uma regra está ou não habilitada para disparar, ele começa a avaliar as premissas até encontrar uma falsa. Se todas são verdadeiras então a regra está habilitada para disparar; a primeira falsa é suficiente para deduzir que a regra deve ficar fora do conjunto de conflitos.

Em conseqüência, para obter uma melhor eficiência de execução é importante que as premissas menos complexas sejam as primeiras a serem avaliadas. O índice de complexidade da tabela de condições está pensado justamente para estabelecer uma ordem entre as premissas da regra em função da sua complexidade.

A heurística desenvolvida para o cálculo deste índice inicia definindo os índices de complexidade para elementos simples, e depois em função destes elementos se define o índice de complexidade para expressões mais complexas. A convenção adotada é a seguinte:

1. O índice de complexidade de um atributo vale zero.
2. O índice de complexidade de uma constante ou uma expressão constante também é zero.

3. A cada operador se associa um número representando a sua *complexidade*, tentando que esta seja proporcional ao tempo de execução da operação. Assim por exemplo a *complexidade* do operador * deve ser várias vezes superior à do operador +.

4. Da mesma forma que com os operadores, também se atribui uma *complexidade* a cada predicado do sistema.

5. O tempo de execução das funções internas (que são as únicas que podem aparecer nas condições) evidentemente dependerá da implementação da função. No entanto, como o sistema conhece apenas o tipo do valor de retorno destas funções, a solução foi estabelecer para todas elas uma *complexidade* fixa e bastante maior que a *complexidade* de qualquer operador ou predicado.

Em função destas definições, o cálculo do índice de complexidade de uma expressão se realiza da seguinte forma:

$$\text{Abreviando} \begin{cases} \text{ic} & = \textit{Índice de complexidade} \\ \text{cmplx} & = \textit{Complexidade} \end{cases}$$

$$\text{ic}(\textit{expressão}) = \text{cmplx}(\left\{ \begin{array}{l} \textit{operador} \\ \textit{predicado} \\ \textit{função} \end{array} \right\}) + \sum_{i=1}^{n} \text{ic}(\textit{operando}_i)$$

Se deve ressaltar ainda que esta fórmula heurística somente se aplica quando pelo menos um dos operandos não é constante. Se todos os operandos fossem constantes a expressão seria avaliada pelo gerador e substituída por um valor constante, sendo zero em conseqüência o seu índice de complexidade.

Expressões complexas, incluindo até funções internas, que pudessem ser avaliadas em tempo de geração (de compilação) como por exemplo:

```
(+ 59.23 (/ (* 25 (sqrt 4.24)) (+ 7 (cos 3.1234))))
```

representariam um gasto de tempo importante para o programa gerador, mas não para o algoritmo de execução que veria somente um valor constante. Por isso o índice de complexidade, que pretende avaliar a complexidade de *execução* das expressões seria zero nesses casos.

6.1.4 Pré-compilação das condições

O algoritmo de geração em todos os casos tenta realizar a maior quantidade de cálculos possíveis durante a compilação, para assim atenuar o trabalho da aplicação em tempo de execução. Esse trabalho é feito basicamente em duas fases:

1. Numa primeira fase o algoritmo avalia todas as expressões cujos elementos de uma forma ou de outra conhece em função do contexto da regra, e armazena essa informação nas listas de influência e do estado anterior que foram analisadas na seção

6.1.2. Para que uma expressão possa ser avaliada não é necessário que todos os elementos sejam constantes. Se por exemplo se tivesse ^alfa inteiro e ^beta e ^gamma reais na seguinte regra:

```
(rule a_regra if
      (eq ^alfa 25)
      (eq ^beta ^alfa)
then
      (conclude ^gamma (sqrt ^beta))
)
```

o algoritmo deduziria que após o disparo o valor de ^gamma necessariamente deve ser igual a 5.0 (a função sqrt —raiz quadrada— da biblioteca do C está declarada como função interna do SPP).

2. Numa segunda fase o algoritmo utiliza a informação levantada e calculada na primeira fase para realizar a pré-compilação das condições propriamente dita. Para isso, ele se coloca na hipótese de que a regra considerada disparou e utiliza toda a informação sobre valor de atributos e prioridade de regras armazenada nas listas de influência e do estado anterior para calcular o valor (verdadeiro ou falso) da maior quantidade de condições possíveis. Esse valor calculado para as condições só será válido no estado do sistema posterior ao disparo da regra considerada.

Para a questão de avaliar as condições o algoritmo utiliza especialmente um dado armazenado na tabela de condições: a quantidade de informações diferentes envolvidas na condição. Este conceito tenta representar a quantidade de "dados" que se deve possuir sobre a premissa para poder avaliá-la. Por exemplo, considerando as seguintes condições:

```
(gt ^alfa (+ ^gamma 13.4))
(or (eq (pri aaaa) 10) (or ^boole (exist ^delta)))
```

Na primeira condição a quantidade de informações diferentes necessárias para avaliar é 2, ou seja:

1. O valor do atributo ^alfa.
2. O valor do atributo ^gamma.

Já na segunda condição do exemplo, essa quantidade é 3, pois é necessário conhecer:

1. A prioridade da regra de nome aaaa.
2. O valor do atributo ^boole (que deve ser de tipo SYMBOL).
3. Se o atributo ^delta existe ou não.

Para completar o exemplo, se podem analisar estas duas condições na hipótese do disparo da regra a_regra, colocada no começo da seção. É claro que com a segunda condição não se pode fazer nada pois nenhuma das três informações necessárias se deduzem do contexto dessa regra. No entanto, os dois dados necessários para avaliar a primeira condição, como já foi estudado, podem ser deduzidos pelo algoritmo; eles são:

$$\begin{aligned} \texttt{\^{}alfa} &= 25 \\ \texttt{\^{}gamma} &= 5.0 \end{aligned}$$

Então como:

$$25 > 5.0 + 13.4$$

resulta que após o disparo da regra a_regra, essa primeira condição sempre vai ser verdadeira.

Repetindo um processo como o deste exemplo para todas as regras e todas as condições do sistema, o algoritmo de geração consegue reduzir bastante a carga de trabalho do algoritmo de execução, melhorando em conseqüência a eficiência de execução da aplicação gerada. É claro por outro lado que o ganho de eficiência dependerá também do próprio sistema de regras, pois este mecanismo de pré-compilação só poderá ser disparado para uma determinada percentagem dos casos.

Até a forma de escrita das regras influirá na entrada do mecanismo de pré-compilação, pois ele se baseia muito no predicado EQ usado a nível de condição. Se por exemplo a regra a_regra tivesse sido escrita desta outra forma:

```
(rule a_regra if
     (and (eq ^alfa 25)
          (eq ^beta ^alfa))
then
     (conclude ^gamma (sqrt ^beta))
)
```

o algoritmo não teria conseguido deduzir nenhuma informação do contexto, salvo a necessária existência dos atributos ^alfa e ^beta. Acontece que o SPP não está programado para tirar informação de expressões de um nível de complexidade maior como:

```
(and (eq ^alfa 25)
     (eq ^beta ^alfa))
```

Evidentemente, tudo pode ser programado com maior ou menor esforço, porém a opção no SPP foi a de realizar um algoritmo simples o suficiente para que pudesse ser programado num tempo limitado, e complexo o suficiente para abranger um número importante de situações comuns. Fica claro que a forma mais "natural" de escrever a regra é a primeira.

6.2 Arquitetura do programa gerado

O SPP não gera um programa completo, mas todos os elementos para constituir um módulo "inteligente" de algum sistema maior. O mínimo que se necessita acrescentar para conseguir um programa executável é um *programa principal* como este:

```
main()
    {
    inf_engine();
    }
```

Os elementos gerados pelo SPP são os seguintes:

- Para cada atributo definido no programa de regras são criadas duas variáveis, uma contendo o valor do atributo e a outra indicando se o atributo existe. Por exemplo, uma definição como a seguinte no programa de regras:

  ```
  ( double ^alfa = 25 )
  ```

  provocará a geração das seguintes declarações no programa C:

  ```
  double alfa=25.0;
  int _e_alfa=1;
  ```

  onde a variável `_e_alfa = 1` indica que o atributo `^alfa` existe.

- Uma "função de condição" será criada para cada condição diferente que apareça no sistema de regras. Estas funções de condição têm como principal objetivo a avaliação da condição em tempo de execução, assim como a atualização de acordo com o resultado verdadeiro ou falso, de determinadas variáveis *bitwise* que armazenam o valor da condição para cada regra que a possui.

- Associada a cada regra é gerada também uma "função de regra" que executa toda vez que a regra é disparada. Ela possui duas coisas fundamentais:

  1. O texto traduzido para a linguagem C da parte de conclusão da regra.
  2. Instruções para analisar todas as regras que podem entrar ou sair do conjunto de conflitos por causa do disparo.

  Durante a execução da função de regra, além de ser calculado o novo *estado* do sistema, o conjunto de conflitos é atualizado de acordo com esse novo estado.

- Além da função de regra, também se tem uma estrutura de dados associada a cada regra identificada. Ela contém as seguintes informações:

- Um vetor com um máximo de 16 funções de condição que representam a parte de premissas da regra.
- Para cada condição da regra, uma variável *bitwise* indica ainda:
  1. Que o valor da condição não se conhece.
  2. Que o valor se conhece e é verdadeiro.
  3. Que o valor se conhece e é falso.
- Um apontador à função de regra associada.
- A prioridade da regra.
- Se a regra está contida no conjunto de conflitos.
- A marca de tempo da última vez que disparou.
- A quantidade de atributos mencionados na regra e que ainda não existem. A regra só é considerada para entrar no conjunto de conflitos quando esta conta é zero.

Estas estruturas de dados podem ainda estar arranjadas em listas duplamente encadeadas se a regra pertence ao conjunto de conflitos.

A seguir serão analisadas em particular as partes desta arquitetura cuja implementação foi mais complexa em função da procura da maior eficiência de execução possível.

6.2.1 Restrições aos tipos LIST e SYMBOL

Como já foi dito, a equivalência destes tipos com LISP é parcial. As restrições foram impostas visando uma implementação mais eficiente. Assim, internamente os símbolos são transformados em números inteiros e as listas em vetores de números inteiros.

Como o uso de um operador de tipo CONS foi descartado, já na compilação das regras é possível conhecer todas as listas que virão a existir durante a execução do sistema especialista. É que com essa restrição, uma lista só pode ser criada de forma explícita, como no seguinte exemplo:

```
'(um dois tres quatro)                                    (6.1)
```

onde `um`, `dois`, ..., etc. são constantes simbólicas. Por outro lado, usando operadores como CAR e CDR, a partir dessa lista só se podem obter sub-listas como as seguintes:

```
(dois tres quatro)
(tres quatro)
(quatro)
()
```

Em conseqüência, o SPP implementa as listas usando uma matriz de duas dimensões. A cada fila da matriz corresponde uma lista criada de forma explícita como em 6.1, enquanto cada coluna indica o primeiro elemento da sub-lista. Desta forma, é possível para o SPP representar internamente os atributos de tipo lista como um par de números inteiros. Então, se a lista do exemplo fosse a número 57 se teria:

```
(57,0)  (um dois tres quatro)
(57,1)  (dois tres quatro)
(57,2)  (tres quatro)
(57,3)  (quatro)
(57,4)  ()
```

Lembrando que no SPP os símbolos são representados por números inteiros, fica claro que a estrutura básica desta implementação é uma matriz inteira, cujo código C é gerado com inicializadores e cujo conteúdo não é modificado durante a execução do sistema especialista.

6.2.2 Construção da função de regra

Como já foi indicado, na lógica do sistema é a função de regra que fica encarregada de realizar todas as alterações necessárias (especialmente no conjunto de conflitos) para deixar o sistema coerente com o novo estado após o disparo.

O algoritmo de geração do SPP utiliza primeiro a informação armazenada nas tabelas de condições, regras e atributos para determinar a totalidade das condições que seriam afetadas pelo disparo da regra. Para isto, segue estes três caminhos:

1. A partir dos atributos da lista de influência, navega pela lista de condições da tabela de atributos.

2. A partir das regras da lista de influência, navega pela lista de condições com a expressão (`pri` <*regra*>) da tabela de regras.

3. Ainda acrescenta diretamente todas as premissas da lista de condições com a expressão (`fired` <*regra*>).

Na figura 6.3 se podem apreciar as listas de influência, de condições com a expressão (`pri` <*regra*>) e de condições com a expressão (`fired` <*regra*>).

Numa segunda etapa, o SPP utiliza a informação sobre existência e valores de atributos, e prioridade de regras armazenada nas listas de influência e do estado anterior, para fazer a avaliação da maior quantidade de condições afetadas pelo disparo que for possível. Este processo foi analisado na seção 6.1.4.

Finalmente, numa terceira etapa o SPP determina a totalidade das regras que seriam afetadas pelo disparo. Para isto segue dois caminhos:

1. A partir das condições que seriam afetadas pelo disparo, navega pela "lista de regras onde aparece a condição" da tabela de condições.

2. A partir dos atributos da lista de influência, navega pela "lista de regras onde aparece o atributo" da tabela de atributos.

Uma vez determinadas as regras que seriam afetadas pelo disparo, o SPP classifica estas regras em diferentes grupos segundo o valor das premissas que ele pôde avaliar. Para ilustrar esta classificação com um exemplo, suponha-se que se está analisando o disparo desta regra:

```
(rule a_regra if
     (eq ^alfa 25)
     (eq ^beta ^alfa)
then
     (conclude ^gamma (sqrt ^beta))
     (conclude ^beta 200)
)
```

Após a classificação, as regras ficam divididas nestes quatro grupos:

1. Regras que após o disparo devem ser consideradas para ingressar no conjunto de conflitos. Porém, antes do disparo podiam ser válidas ou não. Seria o caso por exemplo da seguinte regra:

```
(rule if
     (ge ^gamma ^delta)
then
     (conclude ^gamma 128)
)
```

   A alteração do atributo ^gamma na conclusão de a_regra faz com que esta outra deva ser considerada para ingressar no conjunto de conflitos.

2. Regras que após o disparo devem ser consideradas para ingressar no conjunto de conflitos, mas que por causa de alguma premissa falsa se tem certeza de que não eram válidas antes do disparo. Seria o caso por exemplo da seguinte regra:

```
(rule if
     (ge ^gamma ^delta)
     (eq ^beta 200)
then
     (conclude ^gamma 128)
)
```

   Antes do disparo de a_regra necessariamente devia ser ^beta = 25, e em conseqüência esta outra não podia ser válida. Porém, após o disparo ^beta passaria a valer 200 e a regra deveria ser considerada para ingressar no conjunto de conflitos.

3. Regras que, se estavam no conjunto de conflitos, devem ser removidas pois em função do disparo alguma das premissas foi pré-avaliada falsa ou algum dos atributos mencionados foi apagado. Seria o caso por exemplo da seguinte regra:

```
(rule if
      (ge ^gamma 100)
then
      (conclude ^gamma 128)
)
```

Caso a_regra disparasse, o atributo ^gamma passaria a valer 5,0 , e em conseqüência a premissa desta regra ficaria falsa.

4. Regras com condições avaliadas para falso tanto antes como após o disparo. Não devem ser consideradas para entrar nem para sair do conjunto de conflitos, embora os valores de algumas premissas devam ser atualizados. Seria o caso por exemplo da seguinte regra:

```
(rule if
      (eq ^beta 200)
      (ge ^gamma 100)
then
      (conclude ^gamma 128)
)
```

A primeira premissa era falsa antes do disparo. Após o disparo ela passa a ser verdadeira mas a regra ainda deve ficar fora do conjunto de conflitos pois a segunda condição passa a ser falsa.

Após estas tarefas prévias, o sistema está em condições de proceder à geração da função de regra propriamente dita, que é feita na seguinte ordem:

1. Se gera o código para tirar do conjunto de conflitos todas as regras afetadas pertencentes aos grupos 1. e 3. (as dos outros dois grupos já estavam fora do conjunto de conflitos)

2. Se coloca o texto da conclusão da regra considerada.

3. Se gera código para atualizar o valor das variáveis *bitwise* associadas às condições afetadas, para cada uma das regras afetadas. Para isto se utiliza o resultado da pré-avaliação das condições em tempo de compilação.

4. Para cada uma das regras afetadas pertencentes aos grupos 1. e 2. se gera código para avaliar em *tempo de execução* as premissas que não puderam ser avaliadas em tempo de compilação e, em função do resultado dessa avaliação, adicionar ou não a regra afetada ao conjunto de conflitos.

   Este código gerado se ordena segundo a especificidade das regras, assim para a mesma prioridade e recentidade as regras no conjunto de conflitos ficarão ordenadas segundo a sua especificidade. Isto facilita a tarefa de resolução de conflitos segundo tinha sido especificada na seção 5.5.2.

Esta forma de gerar a função de regra prioriza a eficiência de execução sobre a otimização da quantidade de memória usada. O fato de colocar explicitamente na função de regra todas as regras que podem ser afetadas pelo disparo pode levar a funções de tamanho muito grande, embora por outro lado o tempo de execução será evidentemente melhorado.

6.2.3 Pré-compilação do estado inicial do sistema

No começo da execução do sistema, os atributos que foram inicializados estarão armazenando esse valor inicial, enquanto o resto se considera que não existe. Também as regras começarão com a prioridade inicial especificada, ou com uma prioridade por *default* que o sistema estabelece.

Em conseqüência todo o estado inicial do sistema é conhecido pelo compilador, e em função disto as condições de todas as regras serão avaliadas[1] para determinar já na compilação, o conjunto de conflitos inicial. Por exemplo, com esta declaração:

```
(int ^alfa ^beta=55 ^gamma = (+ 14 18))
```

as seguintes premissas serão inicialmente verdadeiras:

```
(not (exist ^alfa))
((betwn 10 100) ^beta)
(eq 32 ^gamma)
(gt ^beta ^gamma)
```

Com todas as premissas avaliadas para verdadeiro ou falso, o SPP determina as regras inicialmente válidas. Com isto, o código das estruturas de dados associadas às regras que estarão no conjunto de conflitos no começo da execução, se gera já inicializado para pertencer às listas duplamente encadeadas que implementam o conjunto de conflitos.

Em conseqüência, quando o módulo gerado por SPP for carregado na memória, já estará pronto para rodar e não se perderá tempo de execução em realizar tarefas de inicialização.

6.3 O algoritmo de execução

Durante a execução, o conjunto de conflitos é implementado por 16 listas duplamente encadeadas associadas a cada um dos 16 níveis de prioridade permitidos. Dentro de cada lista, que funciona com uma disciplina LIFO[2] para facilitar a estratégia de recentidade na resolução de conflitos, as regras estão ordenadas assim:

- Por ordem decrescente de recentidade.
- Para a mesma recentidade, por ordem decrescente de especificidade.

[1] As condições com atributos inexistentes se consideram falsas

[2] Last In First Out

- Para as mesmas recentidade e especificidade, por índice (crescente) de regra.

Se deve destacar que esta ordem é mantida naturalmente pelo algoritmo, sem a ajuda de nenhum tipo de *sort*, por causa da forma como são criadas as funções de regra.

Esta forma de implementar o conjunto de conflitos facilita enormemente a tarefa de resolução de conflitos, que fica reduzida a escolher a primeira regra da lista de maior prioridade não vazia. O resumo do algoritmo de execução pode ser apreciado no diagrama

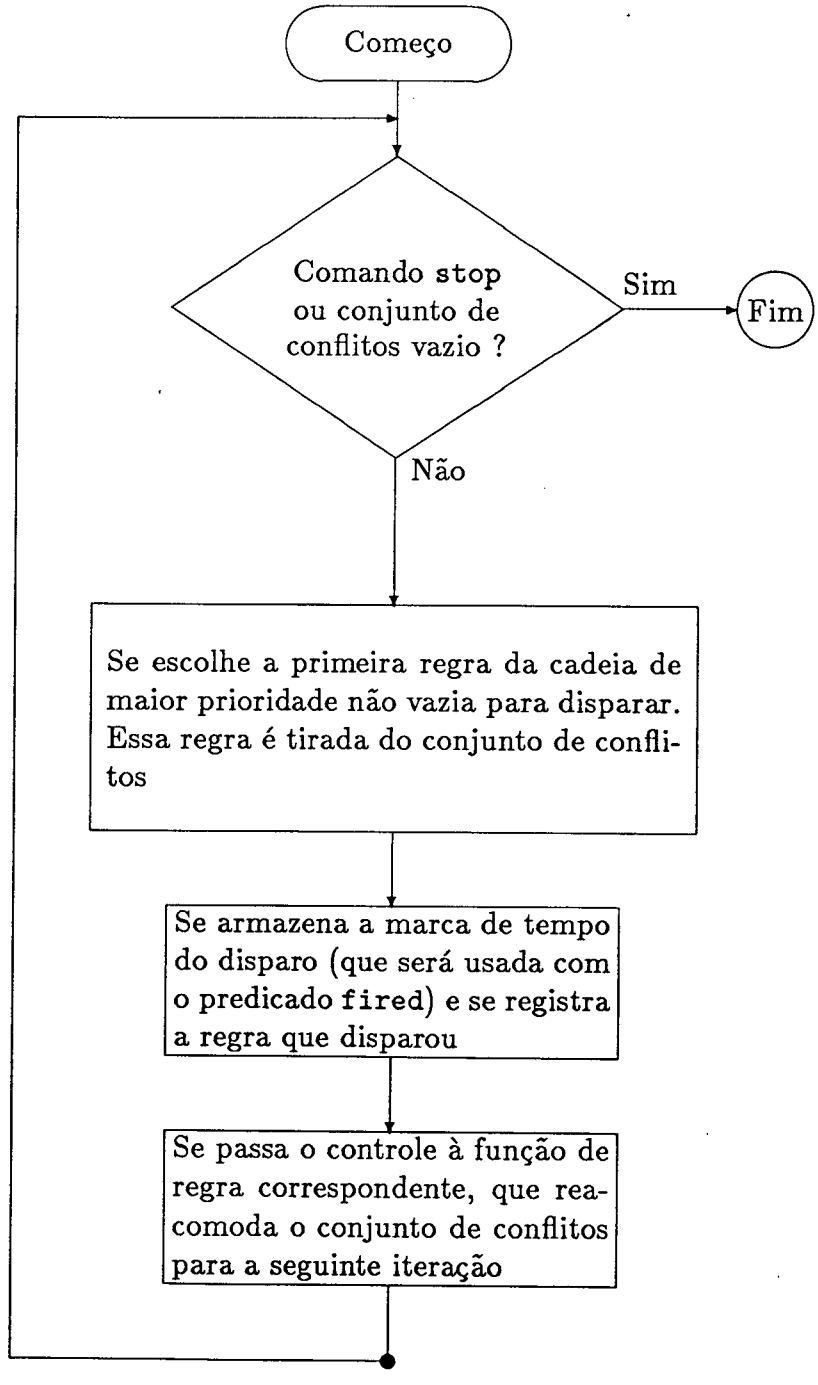


Figura 6.4: Diagrama de blocos do algoritmo de execução

de blocos da figura 6.4.

A simplicidade do algoritmo se deve ao fato de que a resolução dos problemas mais complexos foi deslocada segundo este esquema:

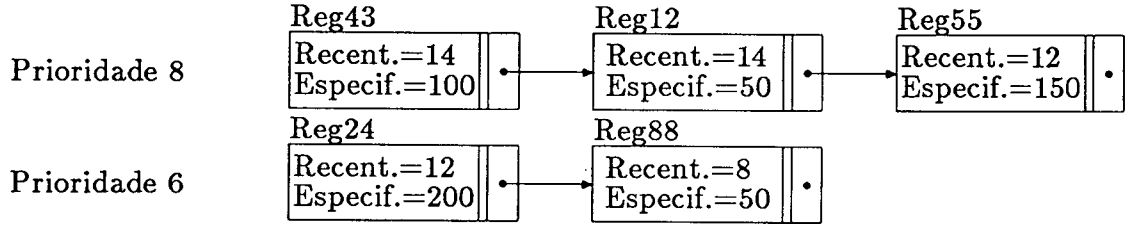
- O problema da determinação do novo conjunto de conflitos é resolvido dentro de cada função de regra.
- A resolução dos conflitos está implícita na estrutura de listas duplamente encadeadas que implementam o conjunto de conflitos.

Desta forma, durante a execução não são realizadas operações desnecessárias, e ainda o acesso aos diferentes objetos (atributos, regras, etc.) sempre é feito em forma direta.

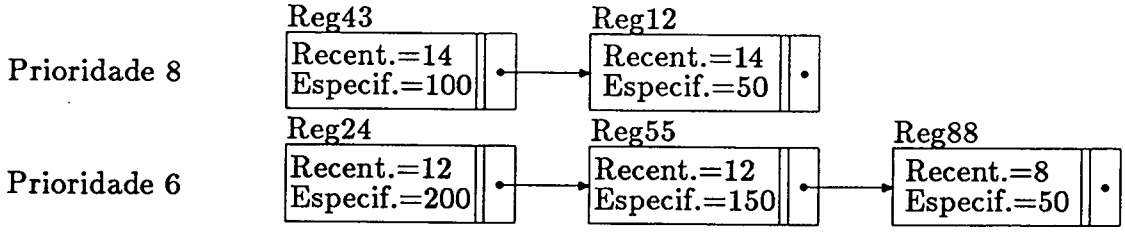
6.3.1 Mudança de prioridade durante a execução

Como já foi especificado, o sistema permite a mudança da prioridade de uma regra durante a execução. Em função da implementação do conjunto de conflitos, isto pode representar um problema se a regra cuja prioridade é alterada está válida.

O código gerado para o comando que muda a prioridade garante que a regra será colocada na nova lista duplamente encadeada de acordo com a mesma recentidade e especificidade que tinha na lista associada à prioridade anterior. Por exemplo, se um conjunto de conflitos estivesse com esta estrutura:



e a prioridade da regra Reg55 fosse mudada para 6, então a nova estrutura do conjunto de conflitos seria a seguinte:



Desta forma a regra ficaria posicionada na estrutura do conjunto de conflitos no mesmo lugar em que estaria se tivesse tido a nova prioridade desde o momento em que ficou válida.

Tem que notar que se além da prioridade tivesse sido alterado também o valor de alguma premissa, mas ainda fosse para a regra permanecer no conjunto de conflitos, então a recentidade não seria mais a mesma e em conseqüência a regra ficaria num outro lugar da estrutura.

6.4 Conclusão

O *parser* do SPP faz uma análise sintática enquanto cria tabelas de regras, condições e atributos com a informação levantada das regras. Algumas das entradas dessas tabelas são listas que fornecem os caminhos de procura necessários para determinar exátamente as conseqüências do disparo de cadá regra no estado do sistema.

Dependendo das aplicações, um poderoso mecanismo de pré-avaliação das condições permite fazer durante a compilação uma importante quantidade de cálculos que de outra forma deveriam ser feitos em tempo de execução.

A arquitetura do módulo gerado inclui variáveis que implementam os atributos, "funções de condição", "funções de regra", uma estrutura de dados associada a cada regra e uma parte fixa independente da aplicação (programa "esqueleto").

O algoritmo de execução é muito simples pois as regras válidas são determinadas diretamente dentro de cada "função de regra" (que é chamada toda vez que a regra dispara), enquanto a resolução dos conflitos está implícita numa estrutura de 16 listas duplamente encadeadas funcionando em modo LIFO.

Capítulo 7

Avaliação de desempenho

O desempenho satisfatório do módulo gerado é a parte crítica deste trabalho, embora também seja importante o comportamento do programa gerador para atingir tempos aceitáveis e evitar o esgotamento da memória durante a compilação. Em conseqüência, todos os esforços foram dirigidos no sentido de conseguir o melhor tempo de execução possível.

No presente capítulo serão tecidos comentários sobre a complexidade do algoritmo de execução [Wir76], e será avaliado o seu comportamento com alguns exemplos práticos. Finalmente, se estudarão aspectos da aplicação do sistema SPP ao projeto do controlador PID auto-ajustável, desenvolvido no LCMI[1] da UFSC.

7.1 Análise do algoritmo de execução

A quantidade de regras disparadas durante a execução de uma aplicação depende exclusivamente do próprio programa de regras. Em particular, para uma aplicação de tempo real o número máximo de disparos deverá estar limitado, embora o SPP possa ser usado também para aplicações onde isso não seja necessário.

Em conseqüência, o desempenho do algoritmo de execução é função exclusiva da complexidade dos processos de determinação das regras válidas e de resolução de conflitos após cada disparo. Como já foi explicado no capítulo anterior, a abordagem deste trabalho se baseia no aproveitamento de determinadas características do programa fonte de regras de produção para reduzir a um mínimo as operações necessárias para realizar cada inferência durante a execução.

Considerando-se um programa de n regras e o fato de o número de condições por regra estar limitado a 16 no SPP, conclui-se que apenas o número de regras deve ser levado em conta numa análise preliminar de ordem de complexidade. Nesta perspectiva o pior que poderia acontecer é que todas as n regras precisassem ser examinadas em cada ciclo de inferência, como no exemplo simples, de três regras, abaixo indicado:

[1] Laboratório de Controle e Microinformática

```
(rule if
    (gt ^alfa 25)
then
    (conclude ^alfa (/ ^alfa 2))
)

(rule if
    (lt ^alfa 15)
then
    (conclude ^alfa (* ^alfa 2))
)

(rule if
    ((betwn 15 25) ^alfa)
then
    (conclude ^alfa (+ ^alfa 1))
)
```

Após o disparo de uma destas regras, qualquer uma das três pode ficar válida, e então todas as três devem ser examinadas, conseqüentemente a ordem de complexidade é n, que corresponde com o pior caso.

Evidentemente este é um caso extremo, pensado especialmente como exemplo, pois o número médio de regras examinadas em cada ciclo de inferência (como se verá com exemplos na próxima seção) geralmente é bastante menor do que n. De qualquer forma, o exemplo serve para verificar que a ordem de complexidade do algoritmo de execução (onde deve ser levado em conta o pior caso) é n.

Neste sentido, se poderia estabelecer um paralelo com o algoritmo "quicksort" [Wir76], considerado o melhor método de ordenação conhecido. Considerando-se todas as possíveis permutações da entrada como igualmente prováveis, a complexidade média do quicksort é de ordem $n \log n$. Porém, essa ordem de complexidade é sensível à forma como está "desordenado" o *array* de entrada e no caso mais desfavorável cai para n^2.

No entanto, uma análise probabilista como a que se faz para o caso do quicksort seria muito difícil de realizar com o algoritmo de execução, principalmente porque deveriam ser feitas hipóteses envolvendo:

- O número médio de atributos modificados em cada regra.
- O número médio de condições onde cada atributo aparece.
- O número médio de regras onde se repete uma condição.
- etc.

De qualquer forma, uma análise deste tipo está além dos objetivos do presente trabalho.

7.2 Avaliação prática

Nesta seção será analisada a forma de resolver dois problemas simples colocados como exemplo, usando SPP. O primeiro deles é um problema clássico de inteligência artificial: o do macaco e das bananas. O outro é a resolução de uma equação de segundo grau, problema especialmente adequado para aproveitar as características matemáticas do SPP.

Problema do macaco e das bananas

Se trata de modelar uma situação onde um macaco está dentro de um quarto e se tem um cacho de bananas pendurado em algum ponto do teto. O macaco dispõe ainda de uma caixa onde ele poderá subir para pegar as bananas [For81]. As posições iniciais do macaco, da caixa e das bananas é aleatória, e o problema consiste em determinar todos os passos que o macaco deverá dar para pegar as bananas.
Os atributos utilizados na resolução do problema com SPP são:

- `^pos_macaco`, `^pos_caixa` e `^pos_banana` representam as coordenadas das posições que vão ocupando no quarto o macaco, a caixa e as bananas. Aproveitando a capacidade de SPP de trabalhar com números complexos, essas coordenadas se expressam como atributos de tipo `complex`.

- `^sobre_caixa`, `^caixa_na_mao` e `^com_banana` são variáveis que atuam como *booleanas* indicando respectivamente se o macaco está em cima da caixa, se está com a caixa na mão e se está com as bananas. Para o SPP são atributos de tipo `symbol` e no contexto do programa podem valer somente `sim` ou `nao`.

- `^gr` é um atributo usado exclusivamente para o controle. Com ele se dividem as regras em dois grupos logicamente diferentes:

  1. O primeiro grupo é formado pelas regras `r0` até `r6`, e se caracteriza pela premissa `(eq ^gr 1)`. Este grupo inclui todas as regras necessárias para adquirir do teclado os dados do problema (as posições iniciais do macaco, a caixa e as bananas, etc).
  2. O segundo grupo é formado pelo resto das regras e se caracteriza pela premissa `(eq ^gr 2)`. São as regras deste grupo que resolvem o problema, determinando especificamente os passos que o macaco deverá dar para alcançar as bananas.

  Tem de se notar que se os dados iniciais fossem passados por exemplo por outro programa, somente as regras do segundo grupo seriam necessárias.

O programa de regras para resolver o problema é apresentado a seguir:

```
(complex ^pos_macaco ^pos_caixa ^pos_banana)
(symbol ^sobre_caixa=nao ^caixa_na_mao=nao ^com_banana=nao )
(int ^gr =1)
```

```
(rule r0
    if
        t
        (eq ^gr 1)
    then
        (set () ^pos_macaco ^pos_caixa ^pos_banana)
)

(rule r1
    if
        t
        (eq ^gr 1)
        (eq ^pos_macaco ^pos_caixa)
    then
        (set () ^sobre_caixa)
)

(rule r2
    if
        (eq ^gr 1)
        (eq ^pos_macaco ^pos_caixa)
        (eq ^sobre_caixa nao)
    then
        (set () ^caixa_na_mao)
)

(rule r3
    if
        (eq ^gr 1)
        (eq ^sobre_caixa sim)
    then
        (concl ^caixa_na_mao nao)
)

(rule r4
    if
        (eq ^gr 1)
        (eq ^pos_macaco ^pos_banana)
        (eq ^caixa_na_mao nao)
    then
        (set () ^com_banana)
)
```

```
(rule r5
    if
        (eq ^gr 1)
        (eq ^com_banana sim)
    then
        (print () macaco_ja_com_banana)
)

(rule r6 5
    if
        (eq ^gr 1)
    then
        (concl ^gr 2)
)

(rule r7 10
    if
        (eq ^gr 2)
        (eq ^com_banana sim)
    then
        (stop)
)

(rule r8
    if
        (eq ^gr 2)
        (ne ^pos_macaco ^pos_caixa)
    then
        (print () macaco_vai_ate_caixa)
        (concl ^pos_macaco ^pos_caixa)
)

(rule r9
    if
        (eq ^gr 2)
        (eq ^pos_macaco ^pos_caixa)
        (ne ^pos_caixa ^pos_banana)
        (eq ^caixa_na_mao nao)
        (eq ^sobre_caixa nao)
    then
        (print () macaco_pega_caixa)
        (concl ^caixa_na_mao sim)
)
```

```
(rule r10
    if
        (eq ^gr 2)
        (eq ^pos_macaco ^pos_caixa)
        (ne ^pos_caixa ^pos_banana)
        (eq ^sobre_caixa sim)
    then
        (print () macaco_desce_caixa)
        (concl ^sobre_caixa nao)
)

(rule r11
    if
        (eq ^gr 2)
        (eq ^pos_macaco ^pos_caixa)
        (ne ^pos_caixa ^pos_banana)
        (eq ^caixa_na_mao sim)
    then
        (print () macaco_move_caixa)
        (concl ^pos_macaco ^pos_banana)
        (concl ^pos_caixa ^pos_banana)
)

(rule r12
    if
        (eq ^gr 2)
        (eq ^pos_macaco ^pos_caixa)
        (eq ^pos_caixa ^pos_banana)
        (eq ^caixa_na_mao sim)
    then
        (print () macaco_solta_caixa)
        (concl ^caixa_na_mao nao)
)

(rule r13
    if
        (eq ^gr 2)
        (eq ^pos_macaco ^pos_caixa)
        (eq ^pos_caixa ^pos_banana)
        (eq ^caixa_na_mao nao)
        (eq ^sobre_caixa nao)
    then
        (print () macaco_sobe_caixa)
```

```
        (concl ^sobre_caixa sim)
)

(rule r14
    if
        (eq ^gr 2)
        (eq ^pos_macaco ^pos_caixa)
        (eq ^pos_caixa ^pos_banana)
        (eq ^sobre_caixa sim)
    then
        (print () macaco_pega_banana)
        (concl ^com_banana sim)
)
```

Para um programa de n regras, se pode definir o número médio de regras acedidas em cada ciclo de inferência como sendo:

$$\text{Média de regras acedidas} = \frac{\sum_{i=1}^{n} \text{Regras examinadas após disparo da i-ésima}}{n}$$

Então, examinando o programa C gerado pelo SPP a partir destas 15 regras se chega à conclusão de que para o exemplo essa média é de $2,47$ regras examinadas por cada ciclo de inferência.

Resolução da equação de $2^{\underline{o}}$ grau

O programa de regras aqui apresentado resolve uma ou mais equações de segundo grau da forma:

$$ax^2 + bx + c = 0$$

onde os coeficientes a, b e c são fornecidos pelo usuário. O programa considera todos os casos possíveis, como raízes duplas, raízes imaginárias, raiz única (por ser $a = 0$), etc.
Os atributos utilizados são os seguintes:

- `^a`, `^b` e `^c` representam os coeficientes da equação.
- `^delta` armazena o valor do discriminante da equação, ou seja $b^2 - 4ac$.
- `^raiz` é onde se armazena o resultado quando se tem uma raiz única.
- Em `^raiz1` e `^raiz2` se armazena o resultado quando há duas raízes reais.
- Em `^raiz_c1` e `^raiz_c2` se armazena o resultado quando há duas raízes complexas.
- `^etapa` é um atributo usado para controle. Ele pode valer:

  `aquisicao` na primeira regra onde se lêem os dados do teclado.

`calculo` nas regras onde se resolve a equação.

`fim` para indicar que o cálculo das raízes da equação já terminou e o programa deve executar a regra r7 que pergunta ao usuário se tem outra equação para resolver.

- `^outra_equacao`, que trabalha como um atributo *booleano* e fará o programa terminar quando assumir o valor nao.

As regras de produção do programa que resolve equações de segundo grau são as seguintes:

```
(symbol ^etapa=aquisicao ^outra_equacao)
(double ^a ^b ^c ^delta ^raiz ^raiz1 ^raiz2)
(complex ^raiz_c1 ^raiz_c2)

(rule r0
    if
        (eq ^etapa aquisicao)
    then
        (set () ^a ^b ^c)
        (concl ^etapa calculo)
)

(rule r1
    if
        (eq ^etapa calculo)
        (eq ^a 0)
        (ne ^b 0)
    then
        (concl ^raiz (neg (/ ^c ^b)))
        (print () equacao_de_1er_grau ^raiz)
        (concl ^etapa fim)
)

(rule r2
    if
        (eq ^etapa calculo)
        (eq ^a 0)
        (eq ^b 0)
    then
        (print () nao_temos_equacao )
        (concl ^etapa fim)
)

(rule r3
    if
```

```
        (eq ^etapa calculo)
    then
        (concl ^delta (- (* ^b ^b) (* 4 (* ^a ^c))))
)

(rule r4
    if
        (eq ^delta 0)
    then
        (concl ^raiz1 (neg (/ ^b (* 2 ^a))))
        (concl ^raiz2 ^raiz1)
        (print () raiz_dupla ^raiz1 ^raiz2)
        (del ^delta)
        (concl ^etapa fim)
)

(rule r5
    if
        (lt ^delta 0)
    then
        (concl ^delta (sqrt (neg ^delta)))
        (concl ^raiz_c1 (/ ((neg ^b) ^delta) (* 2 ^a)))
        (concl ^raiz_c2 (/ ((neg ^b) (neg ^delta)) (* 2 ^a)))
        (print () raizes_complexas ^raiz_c1 ^raiz_c2)
        (del ^delta)
        (concl ^etapa fim)
)

(rule r6
    if
        (gt ^delta 0)
    then
        (concl ^delta (sqrt ^delta))
        (concl ^raiz1 (/(- ^delta ^b) (* 2 ^a)))
        (concl ^raiz2 (/(- (neg ^delta) ^b) (* 2 ^a)))
        (print () raizes_diferentes ^raiz1 ^raiz2)
        (del ^delta)
        (concl ^etapa fim)
)

(rule r7
    if
        (eq ^etapa fim)
```

```
    then
        (set () ^outra_equacao)
        (concl ^etapa aquisicao)
)

(rule r8 10
    if
        (eq ^outra_equacao nao)
    then
        (print () tchau)
        (stop)
)
```

Tem de se notar que embora a regra r7 volte a armazenar aquisicao no atributo ^etapa, se ^outra_equacao assumir o valor nao, então a seguinte regra a disparar será r8 pois é de maior prioridade.

Examinando o programa C gerado pelo SPP a partir destas 8 regras, se chega à conclusão de que a "média de regras acedidas" em cada ciclo de inferência, como definido no exemplo anterior, é de 1,78.

7.3 Aplicação ao controlador PID auto-ajustável

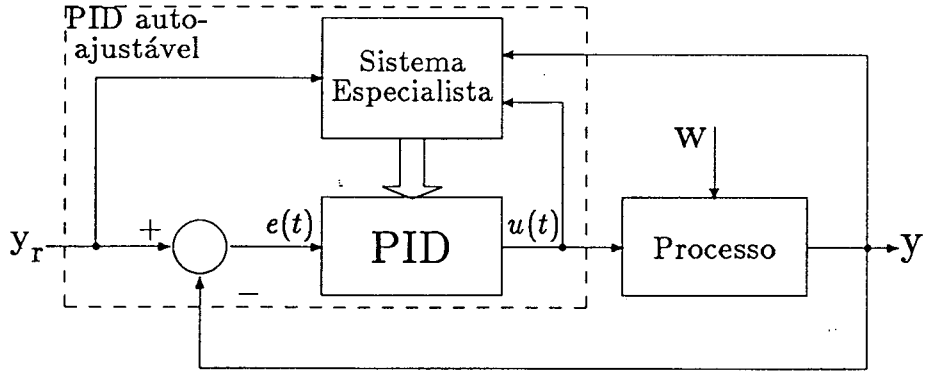
O controlador de tipo PID[2] constitui o núcleo de grande parte dos *softwares* desenvolvidos para controle digital direto de processos industriais e é atualmente o mecanismo básico dos sistemas de controle distribuídos comerciais.

Porém, é comum observar muitas malhas de controle mal ajustadas devido por exemplo a:

- Características inerentes dos processos sob controle como não linearidades.
- Mudanças dos parâmetros dos processos.
- Interação com outros processos.
- Ajustes inadequados realizados por pessoal não especializado.
- etc.

Estes motivos justificam o interesse existente no desenvolvimento de controladores de tipo PID com a capacidade de se auto-ajustar inicialmente e se adaptar automaticamente perante variações no comportamento do processo controlado. A abordagem empregada pelo projeto desenvolvido no LCMI [Pag89] foi a de realizar o ajuste do controlador baseando-se num sistema especialista.

[2]Proporcional + Integral + Derivativo



y: variável controlada (saída do processo)
y_r: sinal de referência do sistema de controle
w: perturbação de carga

Figura 7.1: Controlador PID auto-ajustável

Inicialmente esse sistema especialista foi especificado na linguagem de suporte SP0 [GK88], desenvolvida também no LCMI e implementada como um interpretador de regras na linguagem LISP. O processo por sua vez foi representado por um simulador, o que deu lugar a uma versão acadêmica do projeto. Para desenvolver o controlador como aplicação prática porém, uma das primeiras etapas a desenvolver consistiria em implementar uma versão compilada do sistema especialista. Nesse sentido, se optou pela alternativa de reescrever as regras na linguagem SPP, trabalho que está sendo realizado atualmente. Por outro lado, essa tarefa está sendo facilitada pela semelhança (intencional) das sintaxes de SP0 e SPP.

O diagrama de blocos da figura 7.1 ilustra como o sistema especialista é integrado a um controlador PID clássico, representado pela função de controle:

$$u(t) = K_c[e(t) + \frac{1}{T_i}\int e(t)\,dt + T_d\frac{de(t)}{dt}]$$

sendo:

$e(t)$: erro de seguimento (entrada do controlador)

$u(t)$: sinal de controle

K_c, T_i, T_d: parâmetros de ajuste do controlador

K_c: ganho proporcional

T_i: tempo de integração ("reset time")

T_d: tempo derivativo ("rate time")

Uma importante característica do controlador é o aproveitamento das próprias perturbações exógenas do sistema (mudanças de referência e perturbações de carga, ambas pertencentes à classe dos sinais constantes) como geradoras do ciclo de ajuste do controlador.

Como se observa na figura, o sistema especialista é apenas uma parte do conjunto que forma o controlador PID auto-ajustável, e será implementado como um módulo de software gerado pelo SPP a partir do programa de regras. A cada ciclo de ajuste, cujo objetivo é de se obter os novos valores dos parâmetros do PID, o sistema principal chamará o módulo *especialista* para realizar essa função.

Usando SPP, o sistema principal passa o controle ao sistema especialista executando o comando C:

```
inf_engine();
```

que na realidade é um comando geral e independente de se tratar do PID ou qualquer outra aplicação. Porém, essa chamada não inclui parâmetros. Como já se viu nos capítulos anteriores, para passar informação a um módulo gerado com SPP é necessário usar uma função `set`, que de fato será a "interface de entrada".

Em conseqüência, todos os parâmetros de interesse são definidos dentro do SPP como atributos. Após a chamada, uma regra inicial executa um comando `(set (dados) ...)`, que chama à função C de nome `dados` passando-lhe os endereços dos parâmetros que for necessário fornecer ao módulo especialista. Essa função, que é específica ao projeto do controlador PID auto-ajustável, tem o seguinte formato:

```
dados(i0, i1, ..., in, soin, suin, so2, su2, ...)
int i0, i1, ..., in;   /* Estas variaveis sao ignoradas pois */
                       /* a funcao dados nao esta programada */
                       /* para numero variavel de parametros */
float *soin, *suin, *so2, *su2, ...;
{
    ...
soin=valor;
suin=valor;
so2=valor;
su2=valor;
    ...
}
```

Na seção 5.4.3 o leitor pode ver a forma geral das funções `set` e também o significado das variáveis `i0, i1,...,in`. Os *valores* que esta função `dados` armazena nos atributos, podem ser pegos de variáveis globais pertencentes ao sistema principal, ou também adquiridos diretamente pela própria função.

Para retornar os valores dos novos parâmetros do PID ao programa principal, os atributos `^Kc`, `^Ti` e `^Td` são acessíveis através do comando:

```
extern float kc,ti,td; [3]
```

Porém, essas variáveis não devem ser alteradas dentro do programa principal.

Controle por contexto

O sistema especialista desenvolvido para a aplicação está formado por mais de cem regras. Portanto fica muito difícil estabelecer a ordem de encadeamento das regras entre si. O controle por contexto [BS85] permite reduzir a memória de regras que a máquina de inferência deve considerar, a um sub-conjunto específico vinculado a um contexto particular que neste caso está definido principalmente pela classificação em padrões de formas de ondas. No entanto existem também três sub-conjuntos de regras que se utilizam justamente para selecionar um desses padrões.

Em conseqüência, os sub-conjuntos de regras que possui o controlador PID auto-ajustável são:

- Três sub-conjuntos usados para reconhecer e classificar a forma de onda da variável controlada, determinando um sub-conjunto associado a um padrão específico. São identificados com os símbolos `classifica1`, `classifica2` e `verifica`.

- Os subconjuntos associados a cada padrão, e que determinam em cada caso os novos valores dos parâmetros do PID. São identificados com os símbolos `grupo1`, `grupo2`, etc. .

Aproveitando a capacidade de pré-compilação das condições do SPP, um atributo de nome `^nivel` é utilizado para determinar cada sub-conjunto. Por exemplo o sub-conjunto de regras associado ao segundo padrão estaria caracterizado pela utilização da premissa:

```
(eq ^nivel grupo2)
```

em todas elas. Deve-se notar que o fato de utilizar o atributo `^nivel` em todas as regras do programa, de forma alguma sobrecarrega o sistema.

Um esquema da evolução possível do sistema entre os diferentes sub-conjuntos (ou grupos) de regras pode ser observado na figura 7.2.

Um "metaconhecimento"[MHL*87] do sistema é utilizado para determinar em cada caso o subconjunto de regras que se deve ativar a seguir. Esse metaconhecimento se expressa com regras que modificam o atributo `^nivel`, e que atuam como "metaregras". De acordo com as características do problema, se considera que só haverá mudança do grupo de regras quando no presente grupo não haja mais regras para disparar, portanto todas essas metaregras se agrupam num nível de prioridade menor. Um exemplo de metaregra utilizado pelo sistema é obtido na passagem incondicional do grupo `classifica2` para o grupo `verifica`, quando o grupo `classifica2` estiver esgotado (não haja mais regras para disparar):

[3] Note-se que as maiúsculas são trocadas por minúsculas

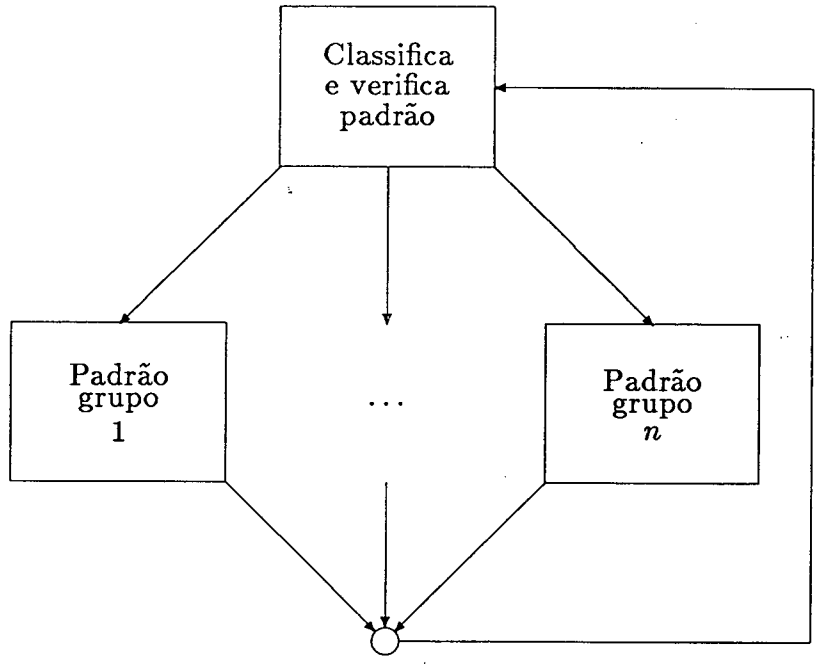


Figura 7.2: Grupos de regras do PID auto-ajustável

```
(RULE class2_1 1      /* Regra de prioridade 1 */
      IF
            (eq ^nivel classifica2)
      THEN
            (conclude ^nivel verifica)
)
```

Deve se notar que as regras comuns (que não modificam `^nivel`) estão todas no nível de prioridade 2. Seguindo com o exemplo, esta outra metaregra seleciona o sub-conjunto `grupo10` se quando não houver mais regras para disparar em `classifica1`, uma determinada condição estiver válida:

```
(RULE class1_0 1      /* Regra de prioridade 1 */
      IF
            (eq ^nivel classifica1)
            (gt ^Soin ^So_lim)
      THEN
            (conclude ^nivel grupo10)
)
```

O nível de prioridade 0 (o menor que admite o sistema SPP) está reservado na presente aplicação para simular um comando "mata-regra". Efetivamente, uma expressão como por exemplo:

```
(pri ver3 0)
```

em alguma conclusão, vai modificar a prioridade da regra `ver3` para zero, impedindo-a de disparar mesmo estando válida pois todas as outras regras do sistema, inclusive aquelas que detectam uma condição para terminar com a execução do sistema estão num nível de prioridade maior, e se tem certeza de que alguma destas "regras de fim" sempre irá disparar.

Conhecimento temporal (memória)

A estratégia global de resolução considera a evolução da resposta do sistema durante os sucessivos ciclos de ajuste. Desta forma, a decisão a ser tomada em cada ciclo depende dos ajustes passados. Para implementar este histórico, se utiliza um registro de deslocamento que armazena os padrões de resposta gerados nos ciclos anteriores.

Este registro é representado na memória de trabalho através do atributo `^registro`, de tipo `reg16`. Uma das funções internas (programadas especialmente para a aplicação) para trabalhar com `^registro` é `belongs`, que indica se um determinado valor está armazenado no registro. Por exemplo:

```
(belongs 8 4 ^registro)
```

indica se o número 4 está numa das primeiras 8 posições do registro.

As regras que detectam o fim do ciclo de ajuste, utilizam uma função `set` especial para armazenar o padrão de resposta no registro de deslocamento.

7.4 Conclusão

O processo de determinação das regras válidas e de resolução de conflitos do algoritmo de execução do sistema SPP, tem uma ordem de complexidade no pior caso igual a n, sendo n o número de regras que compõem o sistema. No entanto na prática o desempenho é bem melhor pois geralmente o número médio de regras examinadas em cada ciclo de inferência é bastante menor do que n.

Isto fica claro com dois exemplos analisados na seção 7.2: o clássico problema do macaco e as bananas, e a resolução da equação de segundo grau. Para um total de 15 e 9 regras em cada exemplo, se tem uma média de $2,47$ e $1,78$ regras examinadas por ciclo respectivamente.

A geração do módulo inteligente do controlador PID auto-ajustável desenvolvido no LCMI, a partir de uma base de regras, é uma aplicação prática importante do sistema SPP e demonstra a sua utilidade para aplicações com restrição de tempo.

Capítulo 8

Conclusão e perspectivas

Apresentou-se nesta dissertação o SPP, um sistema de regras de produção que visa aplicações em tempo real, e cuja especificação e implementação de um protótipo foram objeto deste trabalho.

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

1. Especificação da linguagem de representação do conhecimento.
2. Especificação da forma padrão do módulo C gerado.
3. Desenvolvimento do motor de inferência inserido dentro do módulo C gerado.
4. Desenvolvimento de um algoritmo de avaliação de premissas em tempo de compilação.
5. Implementação de um compilador para fazer a tradução.

O SPP é implementado como um gerador de programa, tem uma sintaxe lisp-like e uma boa capacidade matemática. Requer a declaração explícita de atributos e funções, está baseado na lógica proposicional, e a sua máquina de inferência trabalha sem backtracking, com raciocínio não monotônico, em encadeamento para frente e atendendo às estratégias clássicas de resolução de conflitos.

Uma interface ampla e flexível entre SPP e a linguagem C insere o sistema num contexto de programação híbrida (mistura de procedural e não procedural), necessário para tornar compatíveis o tempo real e os sistemas especialistas.

Deve-se ressaltar que o protótipo do SPP há cinco meses vem sendo utilizado no LCMI no desenvolvimento da segunda versão do projeto do controlador PID auto-ajustável, não tendo até agora mostrado sinais de incorreção ou erros de programação.

Vantagens da solução apresentada

Como se pôde observar nos exemplos e na aplicação do controlador PID auto-ajustável, o protótipo do SPP apresenta um tempo de execução mais do que aceitável. No entanto, o

desempenho deve ser medido em cada caso. Em particular, com o problema do macaco e das bananas, se atingiram aproximadamente umas 5000 inferências por segundo trabalhando num PC AT de 12Mhz.

Embora possua uma interface com C flexível e fácil de utilizar, a solução é suficientemente ampla como para resolver, usando a entrada-saída padrão, uma grande quantidade de problemas sem praticamente nenhuma ajuda da linguagem C. Nesse sentido, também é importante o fato de que todas as estratégias clássicas de resolução de conflitos estejam implementadas, pois isso permite uma abordagem mais natural dos problemas.

Finalmente, com a implementação de prioridades nas regras, e a possibilidade de modificar essas prioridades durante a execução do sistema, se tem uma forma rudimentar de metaregras e comandos de controle.

Problemas da solução apresentada

Do ponto de vista prático, o módulo C gerado pelo SPP pode ocupar um grande espaço de memória. De fato, se um atributo aparecer por exemplo na parte de premissa e de conclusão de todas as regras, há grandes possibilidades de uma explosão combinatória (isto porém não ocorre com atributos como o `^nivel` do controlador PID auto-ajustável, que em todos os casos o sistema pode pré-avaliar). Por causa disso, e em face às dificuldades do compilador C utilizado para trabalhar com fontes de mais de 64K, o sistema está programado para dividir o programa gerado em até nove módulos fonte.

Por outro lado, o desenvolvimento com um gerador de programa como SPP, está prejudicado pois o ciclo de modificação e testes fica muito demorado. Efetivamente, para testar uma alteração o usuário deve dar os seguintes passos:

1. Executar o SPP para gerar um ou mais módulos C.
2. Compilar esses módulos C.
3. *Linkar* os módulos objeto obtidos junto com um programa principal.
4. Executar a nova versão e testar a modificação.

Para evitar este problema deveria ser implementada uma versão interpretada do SPP.

Perspectivas

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

- Na versão atual do SPP todos os atributos são globais. Há casos porém, onde atributos são criados para passar informação exclusivamente entre um conjunto reduzido de regras. Em conseqüência seria interessante implementar um mecanismo para declarar determinados atributos como locais a um grupo de regras. Isto seria mais um passo

na direção do conceito de controle por contexto[1], e evitaria também a possibilidade de interações indevidas entre regras de distintos grupos (por usar o mesmo nome de atributo para funções diferentes).

- Como já foi explicado, esta versão do SPP tem uma tendência a produzir um módulo C muito grande. Há possibilidade no entanto, de implementar um algoritmo alternativo para gerar um programa C menor sacrificando em parte a eficiência de execução. Isto poderia ser colocado como uma opção de compilação para ser usada quando os requisitos de tempo de resposta forem menos críticos.

- Como já foi dito, a implementação de uma versão interpretada do SPP seria de muita ajuda durante o desenvolvimento de sistemas. Aproveitando que a linguagem especificada tem uma sintaxe *lisp-like*, a melhor alternativa seria a implementação desse interpretador de regras em LISP.

- Outras estratégias de controle, e em particular outros comandos de controle poderiam ser especificados para facilitar o trabalho com sistemas de um grande número de regras.

- Para aumentar a sua expressividade, se poderia implementar um pré-compilador da própria linguagem do SPP, ou seja um outro programa que leia um sistema de produção mais elaborado e forneça na saída regras escritas no estilo do SPP. Fazendo isto, se tiraria vantagem da grande preocupação de SPP com a eficiência de execução, embora por outro lado se esteja trabalhando com um sistema de produção com mecanismos de mais alto nível, como metaregras ou até ordem 1 com uma quantidade limitada de instâncias do mesmo objeto.

- Como foi explicado na seção 5.3.1, o sistema SPP não controla se os tipos dos parâmetros de chamada das funções são coerentes com os tipos utilizados na definição. Um mecanismo que fizesse este controle poderia ser implementado tanto para as funções internas (onde o sistema na realidade conhece o tipo de cada parâmetro) como para as funções externas, onde os tipos dos parâmetros deveriam ser declarados junto com a função.

[1] Ver na apresentação do controlador PID auto-ajustável

Apêndice A

Descrição sintática da linguagem SPP

A seguir é apresentada uma descrição sintática da linguagem SPP numa forma BNF[1] [Don84], onde se adotaram as seguintes convenções:

- As palavras escritas em letra maiúscula representam símbolos terminais (*tokens*).
- Os símbolos não terminais são escritos em itálico, prefixados por $<$ e sufixados por $>$.
- Também são utilizados os seguintes meta-símbolos:
  - ::= significa "é definido por".
  - $\star$ implica repetição zero ou mais vezes da expressão quantificada.
  - $+$ implica repetição uma ou mais vezes da expressão quantificada.
  - | indica que qualquer um dos elementos separados pela barra pode aparecer, mas não os dois.
  - [e] englobam termos opcionais.
  - { e } realizam a tradicional função de agrupamento.

Qualquer símbolo especificado na BNF que não estiver nesta lista, deverá aparecer nas regras em forma textual. Notar a diferença entre $+$ e +, e entre $\star$ e *.

Os nomes dos *tokens* que aparecem nesta BNF correspondem exatamente aos símbolos da linguagem. As únicas exceções desta regra são DIGIT que representa um dígito entre 0 e 9, e LETTER que representa uma letra entre a "a" e a "z". Por outro lado, para a linguagem SPP é indiferente o uso de maiúsculas ou minúsculas, tanto para as palavras chave como para qualquer outro nome (não é *case-sensitive*).

[1] Bakus-Naur form

$$
\begin{aligned}
< number > &::= < integer > | < real > \\
< integer > &::= [+|-]\text{DIGIT} + \\
< real > &::= [+|-]\text{DIGIT} + \ .\text{DIGIT} \star \ [\text{E}[+|-]\text{DIGIT}+] \\
< conclusion > &::= (\text{CONCLUDE} < atr >< operand >) \\
&\quad |(\text{DEL} < atr >) \\
&\quad |(\text{PRI} < rule\_name >< operand >) \\
&\quad |(\text{SET}([< function >< operand > \star]) \\
&\quad < atr > +) \\
&\quad |(\text{PRINT}([< function >< operand > \star]) \\
&\quad < operand > \star) \\
&\quad |(\text{STOP})
\end{aligned}
$$

Apêndice B

Manual do usuário

Para compilar um programa fonte de regras de produção utilizando SPP, o usuário deve executar no seu computador o seguinte comando, que segue as convenções adotadas para a BNF no apêndice A:

$$\text{XSPP}[-[\text{DIGIT}][\text{m}]] < \textit{fonte\_regras} >< \textit{módulo\_C} > \star [< \textit{listagem} >]$$

sendo:

<*fonte_regras*> o arquivo contendo o sistema de regras que se quer traduzir para a linguagem C.

< *módulo_C* > o nome de um ou mais arquivos onde o sistema coloca os módulos C gerados. A quantidade destes módulos está indicado pelo DIGIT (qualquer número entre 0 e 9). Se se indica 0, implica que SPP deve fazer exclusivamente a revisão sintática das regras. Se DIGIT é omitido o sistema gera por *default* um único módulo.

< *listagem* > o nome do arquivo onde se imprime o "relatório de compilação", com os erros de compilação que o sistema detectou e outras informações relevantes. Quando este campo for omitido, o relatório será impresso na tela. Se a opção "m" for utilizada, o relatório conterá exclusivamente os erros de sintaxe detectados na compilação.

Para se obter um programa executável, esses módulos C gerados pelo programa XSPP devem ser compilados e *linkados* junto com um programa principal, que pode ser tão simples como:

```
main()
    {
    inf_engine();
    }
```

Porém, o usuário deverá fornecer também o código C de todas as funções:

- declaradas como externas.

- internas, mas do segundo tipo (ver página 32).
- usadas num comando `set`.
- usadas num comando `print`.

Para facilitar a programação, o usuário tem também a possibilidade de ter acesso, desde o seu código C, àlgumas informações internas do sistema especialista gerado pelo SPP:

- Se se tem por exemplo um atributo inteiro de nome `^atri`, o seu valor pode ser acedido usando a declaração:

  ```
  extern int atri;
  ```

  Porém, essa variável não deve ser alterada fora do módulo gerado pelo SPP para evitar um funcionamento incorreto do sistema.

- Usando a declaração:

  ```
  extern char *_simbol[];
  ```

  o usuário tem acesso à tabela de símbolos. Com isso, pode obter o nome de um símbolo a partir do índice interno que o sistema lhe atribui. Isto resulta particularmente útil pois em todas as chamadas a função, com a exceção das funções `print`, o módulo gerado pelo SPP passa os parâmetros simbólicos pelo seu índice interno.

  Por exemplo, no problema do macaco e das bananas, a tabela de símbolos que o SPP gera é:

```
char *_simbol[]={
"nil"
,"t"
,"nao"
,"sim"
,"macaco_ja_com_banana"
,"macaco_vai_ate_caixa"
,"macaco_pega_caixa"
,"macaco_desce_caixa"
,"macaco_move_caixa"
,"macaco_solta_caixa"
,"macaco_sobe_caixa"
,"macaco_pega_banana"
};
```

- Usando a declaração:

```
extern int _busco_lit(char *);
```

o usuário pode usar a função que a partir do nome do símbolo, fornece o índice interno que o sistema lhe atribui.

- Usando a declaração:

```
extern struct _st_at
      {
      char *name;
      int type;
      }atrib[];
```

o usuário pode ter acesso à tabela que contem o nome e o tipo de todos os atributos que o sistema especialista utiliza. A codificação dos tipos é de acordo com esta tabela:

0	INT
1	LONG
2	SYMBOL
3	FLOAT
4	DOUBLE
5	COMPLEX
6	LIST
7	REG1
8	REG2
⋮	etc.

Por exemplo, no problema do macaco e das bananas, a tabela de atributos que o SPP gera é:

```
struct _st_at atrib[7]
={{"caixa_na_mao" , 2}
,{"com_banana" , 2}
,{"gr" , 0}
,{"pos_banana" , 5}
,{"pos_caixa" , 5}
,{"pos_macaco" , 5}
,{"sobre_caixa" , 2}
}
;
```

- Usando as declarações:

```
extern char *_rule_name[];
extern int rule_fired[];
extern unsigned _mt;
```

o usuário tem acesso à tabela dos nomes das regras (_rule_name), e a tabela das regras que dispararam (rule_fired). A variável _mt contem a quantidade de regras que dispararam. Deve-se notar que o sistema internamente atribui a cada regra um índice de zero em diante.

Por exemplo, no problema do macaco e das bananas, a tabela dos nomes das regras que o SPP gera é:

```
char *_rule_name[]={
"r0"
,"r1"
,"r2"
,"r3"
,"r4"
,"r5"
,"r6"
,"r7"
,"r8"
,"r9"
,"r10"
,"r11"
,"r12"
,"r13"
,"r14"
};
```

Tudo isto pode ser usado junto, para imprimir a relação ordenada dos nomes das regras que dispararam numa sessão de trabalho, por exemplo com o seguinte código:

```
printf("\nRegras=");
for(i=0 ; i < _mt % MAX_FIRED ; i++)
      printf("%s,",_rule_name[rule_fired[i]]);
```

onde MAX_FIRED é uma constante que indica o número máximo de regras que o sistema pode armazenar na tabela rule_fired.

Bibliografia

[Aik83] J. Aikins. Prototipal knowledge for expert systems. *Artificial Intelligence*, 20:163–210, 1983.

[AU72] A. Aho and J. Ullman. *The theory of parsing , translation and compiling.* Volume 1 Parsing, Prentice-Hall, 1972.

[AU73] A.Aho and J. Ullman. *The Theory of parsing , translation and compiling.* Volume 2 Compiling, Prentice-Hall, 1973.

[BF86] A. Barr and E. Feigenbaum. *The handbook of artificial intelligence.* Volume 1 and 2, Addison-Wesley, 1986.

[BFKM86] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming expert systems in OPS5.* Addison-Wesley, 1986.

[BL90] T. Beck and R. Lauber. Integration of an expert system into a real-time software system. In *11 th. World Congress on Automatic Control*, pages 158–161, IFAC, 1990.

[Bor87] H. Bordenave. Systèmes experts et diagnostic en temps réel : des problèmes de rapidité. *RGE*, 4:23–26, Avril 1987.

[BS85] B. Buchanan and E. Shortliffe. *Rule based expert systems : the MYCIN experiments of the Stanford heuristic programming project.* Addison-Wesley, 1985.

[BT88] F. Barachini and N. Theuretzbacher. The challange of real-time process control for production systems. In *AAAI 88*, pages 705–709, 1988.

[BV89] B. Bako and R. Valette. Systèmes de compilation de règles et réseaux de petri à objets. In *Congrès International des systèmes experts*, 1989.

[CGF87] M. Casanova, F. Giorgio, and A. Furtado. *Programação em lógica e a linguagem PROLOG.* Edgar Blücher, 1987.

[Cha85] J. Chailloux. *Le LISP de l'INRIA.* Février 1985.

[CM86] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, 1986.

[Coh86] D. Cohen. *Introduction to Computer Theory.* John-Wiley & Sons, 1986.

[Cor84] Microsoft Corporation. *C : Run-Time Library Reference for the MS-DOS Operating System.* 1984.

[CS87] H. Cunha and S.Ribeiro. *Intrdução aos sistemas especialistas.* Livros Técnicos e Científicos, 1987.

[Das85] B. Dasarathy. Timing constraints of real-time systems: constructs for expresssing them , methods of validating them. *IEEE Transactions on Software Engineering*, SE11(1):80–86, 1985.

[Dav80] R. Davis. Meta-rules: reasoning about control. *Artificial Intelligence*, 15:179–122, 1980.

[DK77] R. Davis and J. King. An overview of productions systems. *Machine Intelligence*, 8:300–332, 1977.

[Don84] J. Donovan. *Systems Programming.* McGraw-Hill, 1984.

[FG87] H. Farreny and M. Ghallab. *Eléments d'intelligence artificielle.* P.Hermes Co., 1987.

[For81] C. Forgy. *OPS5 user's manual.* Dept. of Computer Science, Carnegie-Mellon University, 1981.

[For82] C. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[GD84] M. Ghallab and P. Dufresne. Moteurs d'inférence pour systèmes de règles de production : techniques de compilation et d' interprétation. In *Colloque International D 'Intelligence Artificielle*, pages 89–103, Marseille, 1984.

[GFN89] A. Gupta, C. Forgy, and A. Newell. High-speed implementations of rule-based systems. *ACM Transactions on Computer Systems*, 7(2):119–146, May 1989.

[Gha81] M. Ghallab. Decision trees for optimizing pattern-matching algorithms in production systems. In *7 o. IJCAI*, pages 310–312, Vancouver, 1981.

[Gha88] M. Ghallab. Compilation de bases de connaissances. In *Actes des Journées Nationales*, pages 231–253, Toulouse, 1988.

[GK88] H. Garnousset and C. Kaestner. Spl : motor de interferência para sistemas de regras de produção. In *5 o. SBIA*, Natal, 1988.

[GP88] M. Ghallab and H. Philippe. A compiler for real-time knowledge-base systems. In *Internatinal Workshop on Artificial Intelligence for Industrial Applications 1988*, pages 387–393, 1988.

[HK85] P. Harmon and D. King. *Expert systems : artificial intelligence in business.* John Wiley & Sons, 1985.

[HWL83] F. Hayes-Roth, D. Waterman, and D. Lenat. *Building expert systems.* Addison-Wesley, 1983.

[Joh78] S. Johnson. Yacc : yet another compiler-compiler. *Bell Laboratories*, July 1978. Murray Hill, New Jersey.

[Kae89] C. Kaestner. *Contribuição ao Estudo e Desenvolvimento de un Sistema de Regras de Produção.* Master's thesis, DEEL - UFSC, 1989.

[KK86] R. King and F. Karonis. Rule-based systems in the process industry. In *25 th. Conference on Decision and Control*, pages 622–626, Atenas, 1986.

[Kon79] K. Konolige. An inference net compiler for the prospector rule-based consultation system. In *6 o. IJCAI*, pages 487–489, Tokyo, 1979.

[KR78] B. Kernighan and D. Ritchie. *The C programming languaje.* Prentice-Hall, New Jersey, 1978.

[Lau84] J. Laurent. La structure de contrôle dans les systèmes experts. *Technique et Science Informatiques*, 3(3), 1984.

[LCS*88] T. Laffey, P. Cox, J. Schmidt, S. Kao, and J. Read. Real-time knowledge-based systems. *AI Magazine*, 27–45, 1988. Spring.

[LS78] M. Lesk and E. Schmidt. Lex - a lexical analyzer generator. *Bell Laboratories*, 1978. Murray Hill , New Jersey.

[Luc87] C. Lucena. *Inteligência artificial e engenharia de software.* Jorge Zahar, 1987.

[McC78] D. McCracken. *A production system version of the Hearsay-II speech understanding system.* PhD thesis, Carnegie-Mellon University, 1978.

[MCS] F. McCabe, K. Clark, and B. Steel. *PROLOG : Programmer's Reference Manual.* fourth edition.

[Men79] E. Mendelson. *Introduction to mathematical logic.* E. Litton, 1979.

[MHL*87] R. Moore, L. Hawkinson, M. Levin, A. Hoffmann, B. Mattheus, and M. Davis. Experts systems methodologie for real-time process control. In *10 th. World Congress on Automatic Control*, pages 234–281, 1987.

[Mil83] R. Milner. Calculi for synchrony and assynchrony. *Theorical Computer Science*, 25:267–310, 1983.

[NA79] H. Nii and N. Aiello. Age : a knowledge-based program for building knowledge-based programs. In *6 o. IJCAI*, Tokyo, 1979.

[Nil71] N. Nilsson. *Problem solving methods in artificial intelligence.* McGraw-Hill, 1971.

[Nil80] N. Nilsson. *Principes of artificial intelligence.* P. Tioga, 1980.

[Pag89] D. Pagano. *Desenvolvimento de un controlador digital PID auto-ajustável baseado num sistema especialista.* Master's thesis, Univesidade Federal de Santa Catarina, Florianópolis, março 1989.

[Phi88] H. Phillippe. *Le système CLOPS : compilation efficace de bases de connaissance par suppression de jointure.* Technical Report 88032, LAAS, Février 1988.

[Phi90] H. Phillippe. *Algorithmes pour la compilation de bases de connaissances en langage propositionnel et du premier ordre : les systèmes KHEOPS et CLOPS.* PhD thesis, Université Paul Sabatier, Toulouse, 1990.

[Ric83] E. Rich. *Artificial intelligence.* McGraw-Hill, 1983.

[Sob87] R. Sobek. *AND/OR matchs nets : an efficient production system representation.* Technical Report 87120, LAAS, 1987.

[Tur86] R. Turner. *Logiques pour l'intelligence artificielle.* Masson, 1986.

[Vel88a] F. Velasco. *Manual da linguagem OPS5 para computadores compatíveis com IBM-PC (versão 0.3).* INPE, 1988.

[Vel88b] F. Velasco. *Uma introcução à linguagem OPS5.* INPE, 1988.

[Via84] M. Vialatte. *Introduction de métaconnaissance , de gestion d'hypothèses , de logiques d'ordre 0 et 2 dans SNARK.* Technical Report, Institut de Programmation, 1984.

[Wat86] D. Waterman. *A guide to expert systems.* Addison-Wesley, 1986.

[Wer85] H. Wertz. *LISP : une introduction à la programmation.* Masson, 1985.

[WGFC86] M. Wright, M. Green, G. Fiegl, and P. Cross. An expert system for real-time control. *IEEE Software*, 16–24, March 1986.

[WH84] P. Winston and B. Horn. *LISP.* Addison-Wesley, 2 edition, 1984.

[Win84] P. Winston. *Artificial intelligence.* Addison-Wesley, 1984.

[Wir76] N. Wirth. *Algorithms + data structures = programs.* Prentice Hall , Inc. , Englewood Cliffs, New Jersey , USA, 1976.