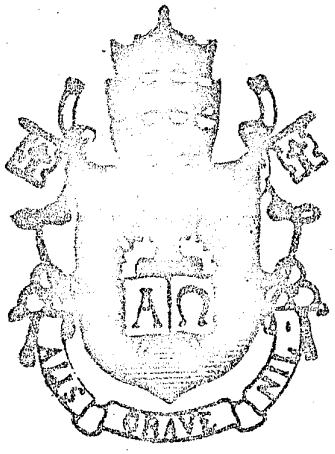

PUC

Série: Monografias em Ciência da Computação,
No. 4/90

O PAPEL DA REPRESENTAÇÃO DO CONHECIMENTO NA CONSTRUÇÃO DE SISTEMAS ESPECIALISTAS

Mariza A.S. Bigonha

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453
RIO DE JANEIRO - BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA
Série: Monografias em Ciência da Computação, No. 04/90
Editor: Paulo A. S. Veloso Abril/1990

O Papel da Representação do Conhecimento na Construção de Sistemas Especialistas[1]

Mariza Andrade da Silva Bigonha

[1]Trabalho apresentado como Qualificação ao Prof. Dr. Carlos José Pereira Lucena

Parcialmente financiado pela CAPES/UFMG.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

Abstract

This text characterizes knowledge representation role in the context of expert systems and the decisions to be taken to select the most adequate representation for a peephole optimization. It presents a general aspects of the most relevant topics in the development of Knowledge-Based Systems, such as, techniques to represent knowledge, the methodology for data aquisition, languages and tools. In addition, it shows, through an example, how the captured knowledge from a code optimization expert can be represented using one commercially available tool.

Keywords: Knowledge Representation, Expert Systems, Tools, Code Optimization

Sinopse

Este texto procura caracterizar o papel da Representação do Conhecimento no contexto de Sistemas Especialistas bem como as decisões que devem ser consideradas para a seleção de uma representação adequada na elaboração de um otimizador de Código Local. É apresentado uma visão geral dos tópicos mais relevantes na construção de Sistemas Especialistas, tais como as técnicas existentes para a representação do conhecimento, metodologias utilizadas para a aquisição de dados, linguagens, ferramentas, além de mostrar, através de um exemplo, como o conhecimento humano capturado de um especialista em otimização de código pode ser representado utilizando uma das ferramentas existentes.

Palavras-chave: Representação do Conhecimento, Sistemas Especialistas, Ferramentas, Otimizadores de Código.

SUMÁRIO

1 Introdução

1.1 Inteligência Artificial

Por volta de 1945, grupos de cientistas americanos e britânicos separadamente trabalhavam no desenvolvimento do que hoje daríamos o nome de computador. Cada grupo queria criar uma máquina eletrônica em que pudessem ser armazenados programas capazes de solucionar complexas operações numéricas. O principal cientista do grupo britânico, Alan Turing, argumentava na época que uma máquina com características tão gerais, uma vez desenvolvida, seria usada para diversos fins. Para Turing, as instruções fundamentais de tal máquina deveriam ser baseada em operadores lógicos, como *AND, OR* e *NOT*. Os cientistas americanos, por sua vez, sabiam que a construção desta máquina seria muito cara, e tendo em mente a construção de uma máquina para somente efetuar cálculos matemáticos, eles se decidiram pelos operadores numéricos, "+", "-", ">", ao invés dos operadores lógicos. Esta decisão, inclusive acatada pelos ingleses resultou em grandes computadores que foram, em sua essência, máquinas calculadoras muito velozes até recentemente.

Apesar do fato de computadores terem sido construídos como processadores numéricos, um pequeno grupo de cientistas continuou a explorar a idéia de computadores manipularem símbolos não-numéricos. Simultaneamente, psicólogos preocupados em solucionar os problemas do homem investiam no desenvolvimento de sistemas que simulassem o comportamento humano. Através dos anos, estes pensamentos foram se solidificando até que estes dois grupos de cientistas formaram uma disciplina como um subgrupo da Ciência da Computação chamada Inteligência Artificial.

Inteligência Artificial segundo Barr e Feigenbaum [BARR 82] é uma parte da Ciência da Computação que diz respeito ao desenvolvimento de sistemas inteligentes, ou seja, sistemas que produzem resultados normalmente associados à inteligência humana, como por exemplo, o entendimento de uma linguagem, a capacidade de aprender, de raciocinar, de solucionar problemas, etc. Entretanto, esta definição é muito geral, pois ela engloba todas as atividades vinculadas à Ciência da Computação, uma vez que é necessário o uso da inteligência para realizar operações simples como uma soma e para calcular o coseno de um ângulo. Poderíamos, então, dizer que todo sistema de computação é, sem dúvida, um sistema inteligente.

Denominaremos Sistemas Convencionais aqueles sistemas que executam funções da inteligência humana que sabemos modelar, e Sistemas Inteligentes aqueles que executam funções que até o momento estamos aprendendo a modelar e que só recentemente temos aprendido a fazê-lo. Neste sentido, os Sistemas Especialistas são inteligentes.

1.2 Sistemas Especialistas e Representação do Conhecimento

Nas últimas décadas encontra-se na literatura descrição e construção de inúmeros sistemas, os quais denomina-se Sistemas Especialistas ou Sistemas Baseados em Conhecimento, [BRACHMAN 83], [BROWNSTON 85], [CARNOTA 88], [HARMON 85], [LUCENA 87], [JACKSON 88]. Percebe-se que não é possível definir com exatidão um conceito tão dinâmico como este. Entretanto, uma definição que parece abrangente é:

> Um Sistema Especialista é um sistema de computação inteligente baseado no conhecimento adquirido de um especialista na resolução de problemas significativos em um domínio específico.

Uma mudança substancial na maneira de construção dos Sistemas Baseados em Conhecimento tem sido percebida desde a década de 1960. Os trabalhos anteriores focalizavam na construção de sistemas inteligentes de propósito geral, dando ênfase a poderosos métodos de inferência que poderiam funcionar eficientemente, mesmo quando o conhecimento de um domínio específico fosse relativamente insuficiente. Hoje os métodos poderosos de raciocínio cederam lugar ao papel do conhecimento específico, detalhado.

A primeira aplicação que obteve sucesso dentro deste contexto, a qual foi denominada Sistema Especialista ou Sistema baseado em Conhecimento, é o programa DENDRAL, elaborado em Stanford. A idéia central é a de que *conhecimento* significa *poder* para o especialista, seja ele um homem ou uma máquina. Especialistas são aqueles que mais sabem sobre fatos e heurísticas no âmbito de um domínio específico. Então a tarefa de construir um sistema baseado em conhecimento é predominantemente aquela de *ensinar* um sistema o suficiente sobre estes fatos e heurísticas para habilitá-lo a executar competentemente em um contexto particular de resolução de um problema. Tal coleção de fatos e heurísticas é normalmente denominada *base de conhecimento.* Os sistemas especialistas ainda dependem de métodos de inferência para efetuarem o raciocínio a partir da base de conhecimento, mas experiências têm mostrado que métodos simples de inferência, tais como *gera e testa, encadeamento regressivo e encadeamento progressivo* são muito eficazes em diferentes escopos de problema, quando acoplados a bases de conhecimento poderosas.

Se esta metodologia remanesce preeminente, então a tarefa de construir base de conhecimento torna-se um fator importante no desenvolvimento de sistemas especialistas. De fato, a maior parte da pesquisa aplicada em Inteligência Artifical na última década foi direcionada para o desenvolvimento de técnicas e ferramentas para a representação do conhecimento. Nós estamos agora na terceira geração destes esforços. A primeira geração foi marcada pelo desenvolvimento de linguagens de Inteligência Artificial como *INTERLISP* e *PROLOG.* A segunda geração assistiu o desenvolvimento de ferramentas de representação do conhecimento em instituições de pesquisa de Inteligência Artificial; Stanford, por exemplo, produziu *EMYCIN, Unit System* e *MRS.* A terceira geração está agora produzindo ferramentas para comercialização, como por exemplo, *KEE* e *S.1.* Cada geração

tem contribuído substancialmente para diminuir a quantidade de tempo necessário para construir sistemas baseados em conhecimento.

Três metodologias básicas, *frames, regras e lógica*, têm emergido para auxiliar a complexa tarefa de armazenar o conhecimento humano em um sistema especialista.

[FIKES 85], [HAYES-ROTH 85] e [GENESERETH 85] descrevem e ilustram cada uma destas metodologias. Fikes e Kehler [FIKES 85] descrevem uma visão da representação do conhecimento centrada em objetos [COX 86], onde todo conhecimento é particionado em estruturas discretas denominados *frames* possuindo propriedades individuais denominados *slots*. *Frames* podem ser usados para representar uma série de conceitos, classes de objetos, ou ainda instâncias individuais ou componentes de objetos. Eles são colocados juntos em uma hierarquia de herança, que provê a transmissão de propriedades comuns entre os *frames* sem a necessidade de uma especificação múltipla de todas as propriedades. Os autores usam a representação de conhecimento de *KEE* e *MANIPULATION TOOL* para ilustrar as características da representação baseada em *frames* para uma variedade de exemplos de domínios. Eles mostram ainda como os sistemas baseados em *frames* podem ser incorporados a uma série de métodos de inferência que são comuns a ambos os Sistemas, Baseados em Lógica e Baseados em Regras.

Hayes-Roth [HAYES-ROTH 85] registra os fatos da história e descreve a implementação de regras de produção como um ambiente para a representação do conhecimento. Em essência, regras de produção usam estruturas *"IF conditions THEN conclusions"* e *"IF conditions THEN actions"* para construir a base de conhecimento. O autor cataloga um grande número de aplicações para as quais esta metodologia tem mostrado ser natural, e, pelo menos, parcialmente bem sucedida na reprodução inteligente do comportamento. O artigo também apresenta algumas ferramentas já disponíveis no mercado que facilitam a construção de bases de conhecimento baseadas em regras. Discute os métodos de inferência, particularmente, os encadeamentos progressivo e regressivo, que têm sido vistos como partes integrantes destas ferramentas. O artigo conclui com uma consideração sobre os futuros melhoramentos e expansões destas ferramentas.

Genesereth e Ginsberg [GENESERETH 85] provêem um tutorial introdutório ao método formal de programação através da descrição de cálculo de predicados. Ao contrário da programação tradicional, a qual enfatiza como a computação deve ser feita, a programação em lógica enfatiza a razão dos objetos e seus comportamentos. O artigo ilustra a facilidade com que adições incrementais podem ser efetuadas a uma base de conhecimento baseada em lógica, bem como as facilidades de inferência automática, através de prova de teorema e explicações que resultam de tais descrições formais. Um exemplo prático do diagnóstico de malfuncionamento de um dispositivo digital é usado para ilustrar como problemas importantes e complexos podem ser representados dentro deste formalismo.

Este trabalho procurará descrever estas três metodologias (*frames*, regras e lógica), as ferramentas disponíveis dentro destas metodologias e o exemplo de uma proposta de um Sistema Baseado em Conhecimento para um Otimizador de Código Local utilizando uma

das representações apresentadas neste texto. Na segunda Seção procurar-se-á descrever as técnicas e metodologia de projeto de Sistemas Especialistas, enfatizando a representação do conhecimento na construção dos mesmos. Além dos aspectos que devem ser considerados na seleção de uma ferramenta para aplicação em um domínio específico. A Seção 3 apresenta algumas ferramentas fazendo uma análise geral dos sistemas disponíveis. A quarta Seção exibe o esboço de desenvolvimento de um pequeno exemplo, com o objetivo de apresentar o uso das técnicas discutidas em seções anteriores. Finalmente, as conclusões e a bibliografia utilizada na elaboração deste texto são apresentadas.

2 Técnicas e Metodologia de Projeto de Sistemas Especialistas

A construção de um sistema especialista é uma atividade nova e como tal, ainda não existe uma metodologia estabelecida. No estado atual da arte, a construção de um sistema baseado em conhecimento é uma atividade essencialmente artesanal. Uma característica do processo de construção é o fato de que a mesma se apresenta como uma atividade profundamente evolutiva, incremental. Constrói-se uma primeira versão, ajusta-se, corrige-se, refina-se e amplia-se até obter um protótipo satisfatório.

O processo de extrair conhecimento de um especialista e codificá-lo em forma de programa são denominados, respectivamente, aquisição e representação do conhecimento, e constituem tarefas fundamentais na construção de um sistema especialista.

Segundo Buchanan [BUCHANAN 83], o desenvolvimento de um sistema baseado em conhecimento está dividido em duas fases principais. Na primeira fase inclui-se a identificação e a conceituação do problema e na segunda fase a formalização, implementação e testes de uma arquitetura apropriada para o sistema, incluindo constantes reformulações de conceitos, re-projeto de representações e refinamento do sistema implementado. A Figura 1 ilustra os estágios de aquisição do conhecimento.

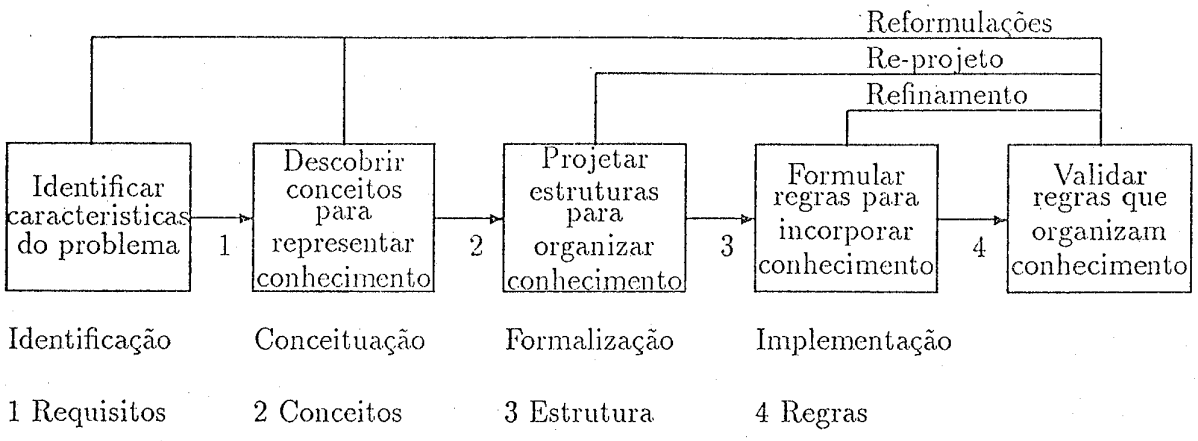

Figura 1: Estágios de aquisição do conhecimento

Identificação: Durante esta fase são selecionados e identificados especialistas para o processo de desenvolvimento do sistema, determina-se os recursos computacionais necessários, as fontes de conhecimento e provê uma definição clara do problema.

Conceituação: A conceituação inclui a descoberta dos conceitos básicos, seus relacionamentos, os mecanismos de controle que se fazem mais adequados, as sub-áreas existentes e as restrições que devem ser consideradas. Esta fase, como a anterior, envolve repetidas interações entre o especialista e o engenheiro do conhecimento e consome um tempo considerável.

Formalização: Inclui o mapeamento dos principais conceitos e relacionamentos em uma representação formal de acordo com alguma linguagem ou ferramenta. O engenheiro do conhecimento deve selecionar uma linguagem e, com a ajuda do especialista, representar os conceitos básicos no contexto da linguagem. Três fatores importantes no processo de formalização são o espaço de busca [RICH 83], o modelo escolhido e as características dos dados.

Implementação: Diz respeito ao mapeamento do conhecimento formalizado da fase anterior em uma forma executável.

Testes: Inclui a avaliação do desempenho do protótipo do programa. Esta avaliação é feita através da execução do programa com um número representativo de testes.

As fases de desenvolvimento de um sistema especialista descritas acima são apenas uma caracterização superficial do complexo processo de aquisição do conhecimento. As fases de formalização e implementação não são tão independentes como parece. Na verdade estão bem relacionadas e falhas na formulação de regras adequadas durante a implementação, podem levar a uma imediata re-formalização. A revisão do sistema resulta das críticas do especialista e de sugestões para o aperfeiçoamento do comportamento do sistema. Uma grande parte do desenvolvimento é consumida na fase de teste a medida que o sistema

evolui. Caso haja a necessidade de correções nesta fase, as mesmas podem resultar em revisões dos resultados de algumas fases anteriores. Isto pode envolver reformulações de regras, re-projeto das estruturas do conhecimento , descoberta de novos conceitos, abandono de outros e, às vezes, a redefinição do objetivo e escopo do problema.

Um outro fator importante na construção de um sistema baseado em conhecimento é a escolha de uma ferramenta adequada ao problema a ser solucionado. Na escolha de uma ferramenta [HAYES-ROTH 83], é importante considerar entre outras características, a sua generalidade. Uma ferramenta deve ser a mais especializada possível a área do problema. Generalidade em excesso implica em mais tempo consumido. Um dos problemas mais difíceis na escolha de uma ferramenta reside no casamento das características do problema com as características da ferramenta.

As características necessárias em uma ferramenta dependem de três fatores:

1. das características do domínio do problema,
2. das características do método adotado para solucionar o problema e
3. das características desejáveis do sistema especialista que vai ser construído.

As características do problema incluem o tamanho do espaço de busca [RICH 83], do formato dos dados e da estrutura do problema.

As características da solução incluem o tipo de pesquisa (*exaustiva, gera e testa, rastreamento para trás*); a representação do conhecimento e o formato do controle (*refinamento de cima para baixo, processamento paralelo de sub-programas*, etc).

As características de um sistema especialista incluem o tipo de usuário (*treinado, inexperiente, cético*) e o método de extensão do sistema (*modificável pelo usuário, modificável pelo construtor ou auto-modificável*).

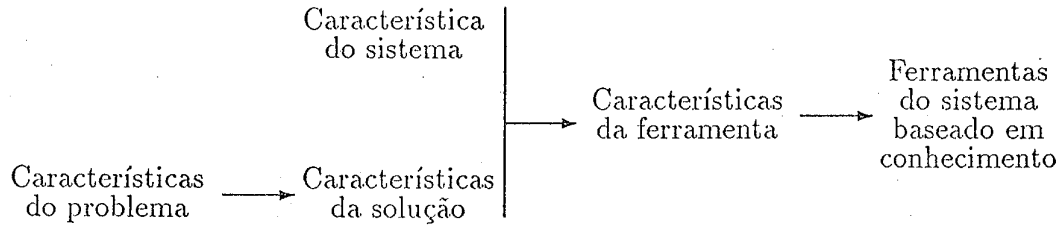

Figura 2: Base para a escolha de uma ferramenta

2.1 A Representação do Conhecimento

Uma observação fundamental provinda de trabalhos em Inteligência Artificial tem sido que *expertise* no domínio de uma tarefa requer um conhecimento substancial sobre aquele domínio. A representação efetiva de conhecimento do domínio é portanto considerada peça fundamental para o sucesso de programas em Inteligência Artificial. O conhecimento do domínio se apresenta tipicamente sob várias formas, incluindo uma definição descritiva dos termos específicos do domínio, como por exemplo, pressão, bomba, fluir; a descrição de objetos individuais do domínio e seus relacionamentos entre si, como por exemplo, "P1 é uma bomba cuja pressão é 230 lb/pol^2"; além de critérios para tomar decisões, como por exemplo, "se a pressão da bomba de alimentação de água exceder 230 lb/pol^2, então feche a entrada da bomba".

Devido a ênfase na representação, os sistemas que usam técnicas de Inteligência Artificial para obter *expertise* são frequentemente referidos como Sistemas Baseados em Conhecimento, Sistemas Especialistas ou simplesmente Sistemas de Conhecimento. A representação do conhecimento, portanto, no campo dos sistemas especialistas diz respeito a maneira sistemática de codificar o que um especialista sabe a respeito de um domínio. Ou seja, ela pode ser descrita como um conjunto de convenções sintáticas e semânticas, através da qual é possível implementar a base de conhecimento. Estas convenções constituem o que se chama linguagem de representação, cujos principais critérios de acesso são a expressividade, o poder heurístico e a conveniência notacional.

Expressão: Permite expressar todo o conhecimento relevante adquirido do especialista para resolver um problema concreto.

Poder heurístico: Significa que além da linguagem possuir uma sintaxe e semântica bem definida ela deve fornecer uma forma eficiente de utilizar o conhecimento expresso para resolver o problema em questão.

Conveniência notacional: Significa que as expressões resultantes devem ser fáceis de escrever e ler, e que deve ser possível entendê-las de forma direta, sem se preocupar como o computador realmente as interpreta.

Experiência tem mostrado que nenhuma das linguagens de representação do conhecimento são por elas mesmo capazes de satisfazer todos estes critérios. As primeiras tentativas de construção de sistemas inteligentes usavam como linguagem o Cálculo de Predicado de Primeira Ordem. Ele era bem aceito devido, basicamente, a dois fatores: primeiro, pelo seu grande poder de expressão; segundo, pela sua semântica bem definida. Entretanto, como a sintaxe da linguagem não provê facilidades adequadas para definir construções mais complexas, uma série de dificuldades aparecem no seu uso. Além disto, a generalidade do Cálculo de Predicados constitui uma barreira no desenvolvimento de facilidades de deduções efetivas para usar o conhecimento nele expresso.

Estas dificuldades contribuíram para o desenvolvimento de outros tipos de representações, tais como *redes semânticas* e várias outras linguagens de representação orientadas a objetos baseados em *frames* que serão vistas na próxima Seção. Atualmente existem várias formas de representar o conhecimento, mas podemos agrupá-las em três classes fundamentais:

- Baseadas em objetos estruturados.
- Baseadas em regras.
- Baseados em lógica.

2.1.1 Representação Baseada em Objetos Estruturados

Nesta categoria se encontram as Redes Semânticas e os *Frames*. O conhecimento nestes dois modelos está organizado em torno de objetos e eventos do universo de aplicação. Ou seja, estes dois modelos provêem ao construtor da base de conhecimento uma maneira eficaz de descrever os tipos dos objetos dentro do domínio que o sistema deve modelar.

Redes Semânticas: Rede Semântica foi pela primeira vez definida por Ross Quillian em seu trabalho "*Semantic Memory*" [ALFONSECA89]. Rede Semântica é um caso especial de grafo dirigido, frequentemente utilizado em sistemas de inteligência artificial como um mecanismo apropriado para representar o conhecimento.

A Rede Semântica é um grafo contendo um certo número de nodos conectados por um certo número de arcos. Os nodos representam os objetos e os arcos representam as relações entre os objetos. Tanto os objetos como os arcos possuem rótulos. Existem vários tipos de relações. Por exemplo, pode existir relações hierárquicas tais como:

Is-a: Indica que um objeto é um subconjunto de outro objeto na rede. Exemplo: Elefante *Is-a* mamífero.

Instance-of: Indica que um objeto é um elemento de outro objeto na rede. Exemplo: Jumbo *Instance-of* elefante.

Prototype-of: Indica que um objeto é um caso especial ou um modelo de outro objeto.

A-part-of: Indica que um objeto é uma parte física de outro objeto. Exemplo: Rodas *A-part-of* carro.

Pode haver ainda outros tipos de relações para definirem propriedades tais como: *cor*, *preço*.

Um conceito importante na operação de uma rede semântica é a herança de relações. Por exemplo, suponha que a relação R é válida para os objetos A e B (nesta ordem), e que a relação S é válida para os objetos B e C. Existe alguma relação entre A e C? É claro que a resposta para esta questão dependerá do que é representado respectivamente, pelas relações R e S.

A transitividade de uma relação é um caso particular da herança de relações, por exemplo, a relação R é válida para os objetos A e B; a relação R é válida para os objetos B e C. R é válida para A e C?

Em um sistema baseado em Redes Semânticas, as decisões sobre a herança de relações e transitividade dependem das relações declaradas. Portanto, estes sistemas normalmente provêem uma maneira de definir as propriedades de uma relação em alguma linguagem de alto nível.

As Redes Semânticas podem ser implementadas usando como estrutura de dados as listas encadeadas. Isto explica o fato de *LISP* ser utilizada como linguagem de programação para sistemas baseados neste tipo de representação.

Frames: Os *frames* foram propostos por *Minsky* em "*A Framework for Representing Knowledge*" com seu trabalho sobre percepção visual e processamento de linguagem natural, como uma estrutura de dados que tenta reproduzir a maneira pela qual os indivíduos mantêm as informações armazenadas em seus cérebros e as usam quando necessário. Como as Redes Semânticas, *frame* é um caso especial de grafo dirigido.

Um sistema *frame* é um grafo no qual os nodos (*frames*) contêm todas as informações disponíveis sobre um dado objeto dividido em uma série de atributos ou *slots*, cada qual com o nome da propriedade do objeto e um ou mais valores. Um *frame* também possui um nome, permitindo assim recuperar as informações contidas nos *slots*. Por exemplo,

$$
\begin{array}{ll}
Frame & Table \\
 & is-a : Furniture \\
 & files : 0,1,2 \\
 & drawers : 0,1 \\
 & legs : integer(default4) \\
 & light : 0,1
\end{array}
$$

Dada a definição do *frame* ela pode ser utilizada para definir novos *frames* que herdarão automaticamente as propriedades do primeiro *frame*. Por exemplo,

Frame $My - desk - Table$
$is - a : Table$
$files : 2$
$drawers : 0$
$light : 1$

No exemplo acima, certas propriedades são repetidas, ou seja, alguns *slots* têm o mesmo nome que o *frame Table* e eles especificam as propriedades do novo *frame* em mais detalhes. Outras propriedades podem ser omitidas, como ocorreu por exemplo, com o número de cadeiras. Neste caso, assume-se o valor *default* 4.

Frames se utilizam de alguns *slots* para gerar uma estrutura hierárquica similar àquela descrita para Redes Semânticas. Propriedades podem ser herdadas dentro desta estrutura.

Os objetos [COX 86] são definidos como pertencentes às classes das quais herdam propriedades, e às vezes, procedimentos. Em alguns sistemas, os objetos podem simultaneamente pertencer a várias classes distintas e herdar propriedades de todas elas. As classes por sua vez também são objetos. Por exemplo, um *frame* pode representar um automóvel, e outro toda uma classe de automóveis. Construções são disponíveis na linguagem de *frames* para organizar os *frames*, que representam classes em uma taxonomia. Esta construção permite ao projetista da base de conhecimento descrever cada classe como uma especialização (sub-classe) de outras classes mais genéricas. Assim, automóveis podem ser descritos como veículos mais um conjunto de propriedades que distinguem automóveis de outras classes de veículos. A Figura 3 ilustra a taxonomia de um *frame*; as linhas retas representam o relacionamento classe-subclasse e as linhas pontilhadas representam relacionamentos classe-membro. *Veículo* é uma subclasse de *objetos físicos* e de *produtos*, enquanto *Paulo e Carlos* são membros de *homem*.

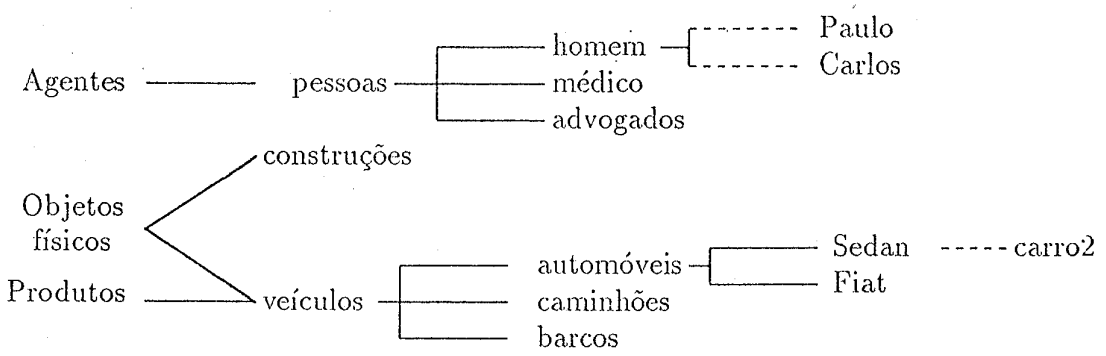

Figura 3: Taxonomia de um *frame*

Redes Semânticas e *Frames* se diferem nas seguintes propriedades:

1. Propriedades de estrutura de dados

   (a) Em uma Rede Semântica, os nodos não possuem informações associadas, com exceção de seus nomes.

   (b) Em um sistema *Frames*, um nodo possui uma grande quantidade de informações na forma de *slots*.

2. Propriedades de relações

   (a) Em uma Rede Semântica toda espécie de relação é automaticamente manuseada de acordo com um conjunto de métodos de herança que pode ser definido pelo programador da rede.

   (b) Em um sistema de *Frames* há normalmente uma única classe de relações que sempre permite a herança de propriedades.

As diferenças que existem nestes dois tipos de representação são tão insignificantes que um sistema programado para Redes Semânticas pode ser usado para construir um sistema baseado em *Frames* e vice-versa.

Uma das mais importantes propriedades de um sistema *Frame* é a possibilidade de definição de *ações* que podem ser executadas sob certas condições. Por exemplo, computar o valor atual de um parâmetro ou produzir um efeito secundário na estrutura da rede, sempre que o valor de um *slot* é requisitado e \ ou substituído. Estas *ações* normalmente são denominadas de *demons*, e são programadas como programas procedurais em alguma linguagem de alto nível.

Algumas das vantagens do uso de *frames* são:

- Provê uma representação estrutural concisa de relacionamentos úteis.
- Suporta uma definição concisa através da técnica de especialização que é fácil de usar pela maioria dos especialistas em um domínio.

A desvantagem deste modelo está no fato de que ele não provê diretamente facilidades para descrever declarativamente como o conhecimento armazenado nos *frames* é usado. Tradicionalmente, a única maneira de associar o comportamento dependente do domínio aos *frames* tem sido juntando-se a eles, sob várias formas, procedimentos escritos em alguma linguagem de programação como, por exemplo, em *LISP*, como ocorre nas linguagens de representação *KL-ONE* e *KRL*. Além disto, facilidades adicionais são necessárias em tais sistemas para descrever declarativamente as regras de inferência dependentes do domínio, regras de análise de decisão, ações que podem ser tomadas no domínio por vários agentes, simulações do comportamento de objetos, etc.

A forma de representação mais popular e eficaz utilizada para descrever declarativamente o conhecimento comportamental dependente do domínio em sistemas baseados em conhecimento são as regras de decisão *pattern/action*, denominadas Regras de Produção. Os Sistemas de Produção são, de fato, um subconjunto do Cálculo de Predicados com um componente adicional para indicar como as informações contidas nas regras devem ser utilizadas durante o raciocínio. As regras de produção podem ser facilmente entendidas por especialistas e possuem um grande poder de expressão. Este modelo será objeto do próxima seção.

Exemplos de sistemas baseados nos tipos de representações apresentados nesta seção são *KRL, UNITS, KL-ONE, KRIPTON* [FIKES 85].

2.1.2 Representação Baseada em Regras

Nos Sistemas de Produção, o conhecimento se expressa por meio de regras. As regras são constituídas de premissas e ações que devem se realizar se as premissas são satisfeitas.

O formato básico de uma regra é:

```
if P1 &  ... Pn, then Q1 & ... Qn.
```

onde P1, ..., Pn são as premissas, as vezes chamadas de condições, antecedentes ou lado-esquerdo de uma regra e Q1, ..., Qn são as ações, conclusões, consequentes ou muitas vezes chamadas lado-direito de uma regra. As premissas e as conclusões podem ser pares atributo-valor, por exemplo, *IDADE = 20*, onde *IDADE* é um atributo e 20 é um valor. As premissas e as conclusões podem ainda ser triplas objeto-atributo-valor, por exemplo, *(Maria IDADE 15)*, onde *Maria* é o objeto.

As regras podem usar os raciocínios para-frente ou para-trás, muitas vezes denominados encadeamento regressivo e progressivo, respectivamente. Sucintamente, o raciocínio regressivo começa na ação de uma regra e tenta satisfazer a "condição". O raciocínio progressivo se inicia com a "condição" de uma regra e infere a "ação" se a "condição" é satisfeita. O encadeamento regressivo é útil na prova de um objetivo em particular e o encadeamento progressivo é útil na busca do que pode ser inferido de uma dada proposição.

Os principais componentes da arquitetura de um Sistema de Produção são: uma memória de trabalho, uma base de conhecimento e uma máquina de inferência, conforme ilustra a Figura 4.

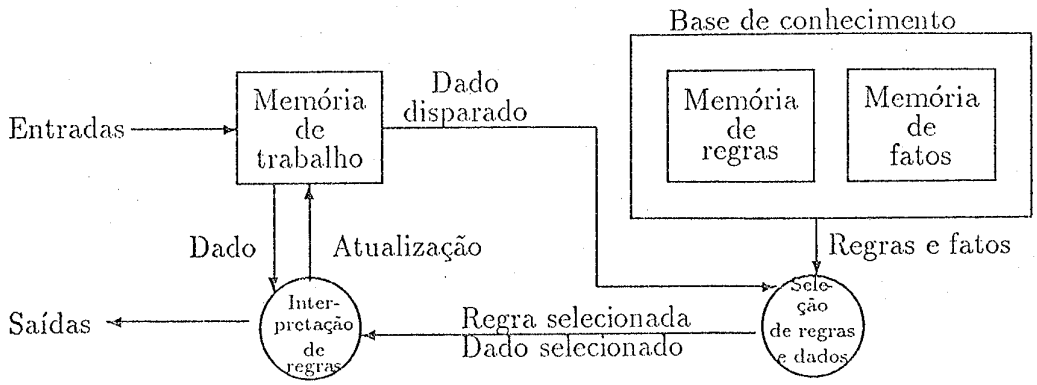

Figura 4: Componentes básicos de um Sistema de Produção

A memória de trabalho é uma base de dados globais de símbolos representando fatos e asserções referentes ao problema. Os dados são instâncias de objetos que representam objetos físicos, fatos relacionados com o domínio de aplicação ou objetos conceituais relacionados com a estratégia de solução do problema. As asserções armazenam inferências prévias baseadas em regras. O conteúdo da memória é o principal determinante do fluxo de controle do programa.

A base de conhecimento contém regras e fatos. Regras sempre expressam uma condição com dois componentes, um antecedente e um consequente. A interpretação de uma regra se resume em: se o antecedente pode ser satisfeito, o consequente também pode sê-lo. Quando o consequente define uma ação, o efeito de satisfazer o antecedente é selecionar uma ação para ser executada. Quando o consequente define uma conclusão, o efeito é inferir a conclusão. Como o comportamento de todos os Sistemas de Produção é derivado deste regime simples, suas regras sempre especificarão o verdadeiro comportamento do sistema quando dados são fornecidos para solucionar um problema em particular.

Fatos, o outro tipo de dado contido na base de conhecimento, expressam asserções sobre propriedades, relacionamentos, proposições, etc. Ao contrário das regras as quais os sistemas de produção interpretam como imperativas, fatos são normalmente estáticos e inativos. Assim, muito embora em muitos contextos, fatos e regras sejam logicamente permutáveis, no contexto dos Sistemas de Produção eles são bem distintos.

A máquina de inferência determina que regras são relevantes para uma dada configuração de memória de trabalho, e entre todas elas seleciona uma para ser aplicada. Esta seleção faz parte da estratégia de controle e se chama resolução de conflito.

Os sistemas de Inteligência Artificial baseados em produções são distintos entre si sob vários aspectos, mas há algumas características do formalismo destes sistemas que podem ser generalizadas como a modularidade e a uniformidade.

- Modularidade: Uma qualidade óbvia destes sistemas reside no fato de que produções individuais em uma base de dados podem ser adicionadas, removidas ou substituídas independentemente. Elas se comportam como peças de conhecimento independentes. Muito embora a substituição de uma regra possa modificar o desempenho do sistema, ela pode ser efetuada sem ter que se preocupar com efeito direto sobre as demais, porque regras não se invocam diretamente.
- Uniformidade: Como a informação deve ser codificada dentro de um padrão rígido, ela pode ser mais facilmente entendida por outra pessoa ou por outra parte do sistema.

Como mencionado anteriormente, a representação baseada em regras é simples, possui características modulares, é fácil de modificar, permite a introdução de novas regras, facilita a construção de um protótipo rápido e a programação experimental. Entretanto, algumas das características deste método de representação, embora apareçam como vantagens, acarretam em importantes limitações. Por exemplo, a adição incontrolada de regras pode levar a interações indesejadas e resultados imprevisíveis. Além disto, a simplicidade das formas da linguagem é por outro lado uma restrição ao poder de expressão da mesma, como pode ser visto nas limitações dos conectivos lógicos nas condições, etc.

Outros aspectos deste modelo, apontados como desvantagens, são a ineficiência e a opacacidade. A ineficiência se refere a execução do programa. A forte modularidade e uniformidade das produções resultam em uma grande sobrecarga no seu uso na resolução de um problema. O segundo aspecto, a opacacidade diz respeito à dificuldade de seguir o controle na solução de um problema. Faltam mecanismos de abstração. Os algoritmos são menos aparentes que eles poderiam ser se fossem expressos em uma linguagem de programação convencional. Em outras palavras, muito embora o conhecimento *situação-ação* possa ser expresso naturalmente nos sistemas de produção o conhecimento algorítmico não o é naturalmente. Dois fatores contribuem para este problema: o primeiro é o isolamento das regras (elas não ativam uma a outra), e segundo, o tamanho uniforme de cada regra (não existe nada como uma hierarquia de subrotinas, na qual uma regra possa ser composta de várias sub-regras). Chamadas de subrotinas e funções, características comuns em linguagens de programação, poderiam tornar o fluxo de controle mais fácil de ser seguido.

As regras de produção não provêem uma adequada facilidade de representação para a maior parte das aplicações dos Sistemas de Conhecimento. Em particular, seu poder de expressão é inadequado para definir termos e para descrever objetos de um domínio, bem como o relacionamento estático entre os mesmos.

A maior insuficiência das regras de produção reside em áreas que são efetivamente manejadas por *frames*. A integração entre *frames* e as linguagens de regras de produção tem obtido sucesso na formação de facilidades de representação híbrida, que combinam as vantagens de ambos componentes destas técnicas de representação. Exemplo desta união são os sistemas *LOOPS e KEE* que serão objeto de discussão

na próxima seção. Estes sistemas têm mostrado como a linguagem de representação *frame* pode servir como uma base poderosa para a linguagem de regras.

Exemplos de sistemas baseados neste tipo de representação são *PLANNER, XCON, MYCIN, PROSPECTOR, AGE, M1, S1*, etc. [FIKES 85].

2.1.3 Representação Baseada em Lógica

A idéia fundamental da programação em lógica é a programação descritiva. Na engenharia de software tradicional constrói-se um programa especificando-se as operações que devem ser efetuadas para a solução de um problema, ou seja, dizendo *como* o problema deve ser solucionado. As suposições através da qual o programa é baseado normalmente são deixadas implícitas. Na programação em lógica constrói-se o programa descrevendo sua área de aplicação, ou seja, dizendo *o que* é verdade. As suposições são explícitas, mas a escolha das operações é implícita [GENESERETH 85].

Uma descrição desta natureza se transforma em um programa quando é combinada com um procedimento de inferência independente da aplicação. Aplicando-se tal procedimento à descrição de uma área de aplicação, permite-se à máquina obter conclusões sobre a mesma e responder questões, mesmo que estas respostas não estejam armazenadas explicitamente na descrição. Esta capacidade é a base para a tecnologia de programação em lógica. A Figura 5 ilustra a configuração de um sistema típico de programação em lógica. No coração do programa se encontra o procedimento de inferência, independente da aplicação, o qual aceita perguntas de usuários, acessa fatos na base de conhecimento (descrição) e produz as conclusões apropriadas.

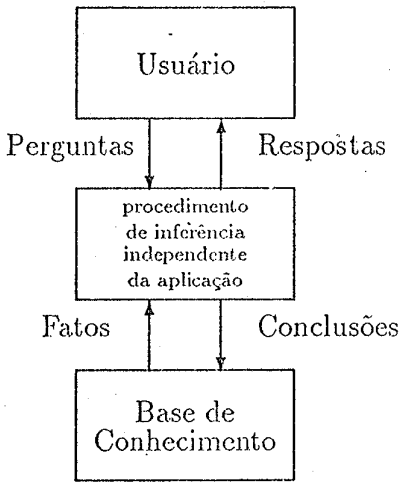

Figura 5: Sistema de Programação em Lógica

O fato do procedimento de inferência ser independente da base de conhecimento que ele acessa acarreta em uma série de vantagens. A principal delas diz respeito ao desenvolvimento incremental. A medida que novas informações sobre uma aplicação

são descobertas, elas podem ser adicionadas à base de conhecimento do programa e portanto incorporada ao programa como um todo, sem a necessidade de revisão ou desenvolvimento de novo algoritmo.

A segunda vantagem é a explicação. Através do raciocínio automático é fácil armazenar as decisões tomadas durante a resolução de um problema. Assim é possível ao programa explicar ao usuário como ele resolveu cada problema e, portanto, porque ele acredita que o resultado esteja correto.

O uso do formalismo lógico torna fácil a emulação de sistemas baseados em regras, com encadeamento progressivo ou regressivo e objetos estruturados. A base de conhecimento e as informações para solucionar problemas podem ser representados com este formalismo como fatos e regras, a partir dos quais é possível extrair inferências solidamente fundadas automaticamente.

Além das vantagens apontadas têm-se:

- Lógica parece ser a maneira natural de representar declarativamente o conhecimento; ela sempre corresponde ao nosso entendimento intuitivo de um domínio.
- Lógica é precisa. Existem métodos padrões para determinar o significado de uma expressão no formalismo lógico.
- Lógica é flexível. Como a lógica não compromete a natureza do processo que realmente faz deduções, um fato particular pode ser expresso de uma única maneira, sem ter que considerar seus possíveis usos.
- Lógica é modular. Asserções lógicas podem ser inseridas a uma base de dados independentemente de cada uma; conhecimento pode crescer incrementalmente a medida que novos fatos são descobertos e adicionados. Em outros sistemas de representação, a adição de um novo fato pode afetar o tipo de dedução a ser efetuada.

A maior desvantagem do sistema baseado em lógica advém também da separação da representação e processamento. A dificuldade com a maior parte dos sistemas de Inteligência Artifical atualmente reside na parte heurística do sistema, ou seja, na determinação de como usar os fatos armazenados no banco de dados do sistema, e não na decisão de como armazená-los. Assim, separando estes dois aspectos simplesmente adia o tratamento do problema.

A representação baseada em lógica utiliza-se da Lógica de Predicados de Primeira Ordem como linguagem de representação.

Um elemento básico da Lógica de Predicados é uma expressão do seguinte tipo:

P(X1 .....Xn) onde P representa o predicado e X1.....Xn os seus argumentos.

A Lógica de Predicados se chama de primeira ordem quando os argumentos não podem por sua vez serem predicados.

Exemplos de sistemas baseados neste tipo de representação são *PROLOG, MRS*, etc. [FIKES 85].

3 Ferramentas Disponíveis

A construção de sistemas baseados em conhecimento pode ser encarada de diversas maneiras. Em alguns casos utiliza-se como forma de representação linguagens de alto nível, como *PASCAL* e *FORTRAN*; em outros casos usa-se linguagens como *LISP* e PROLOG, ou ainda usa-se "software" especializado.

Dadas as dificuldades encontradas no entendimento, depuração e manutenção, durante a fase de análise conceitual dos sistemas, os projetistas de Sistemas Especialistas concluíram que era conveniente separar totalmente o código que implementava a máquina de inferência e as outras funções de controle conexas, daquela que implementava a base de conhecimento. Como consequência desta separação, a máquina de inferência pode ser re-utilizada com outras bases de conhecimento, para dar lugar ao desenvolvimento de novos sistemas baseados em conhecimento, sem necessidade de re-codificá-los. Nasceram assim esqueletos dos primeiros sistemas: *EMYCIN, EXPERT, PROSPECTOR* etc. Estes foram os primeiros passos na construção de um conjunto de sistemas que facilitam o desenvolvimento de Sistemas Especialistas, permitindo que o engenheiro do conhecimento se concentre no aspecto realmente mais importante de seu trabalho, a aquisição e representação do conhecimento. Estes sistemas foram denominados de ferramentas.

Devido a grande variedade de sistemas existentes é muito difícil classificá-los [HARMON 85]. Assim é mais apropriado considerá-los como um conjunto amplo de elementos formando um contínuo, que vai desde as linguagens convencionais de alto nível, passa pelos ambientes e se estende até as ferramentas. A Figura 6 mostra alguns destes sistemas e a sua posição relativa dentro do contínuo mencionado.

Linguagens de alto nível		Ambientes	Ferramentas	
LISP	PROLOG	OPS5	KEE	EMYCIN
	INTERLISP		LOOPS	S1
FORTRAN			ART	M1
	PASCAL		MRS	EXPERT

Figura 6: Contínuo Linguagem-Ferramenta

Neste contínuo, as linguagens de alto nível são as mais flexíveis, entretanto é mais difícil usá-las na construção rápida de protótipos de novos sistemas especialistas.

Os ambientes de programação estão em uma situação intermediária entre as linguagens e ferramentas, tanto no que diz respeito a flexibilidade, quanto a facilidade de uso.

As ferramentas são muito menos flexíveis, pois grande parte das decisões que o engenheiro do conhecimento deveria tomar, agora é encorporada na ferramenta. Embora as ferramentas sejam muito mais rígidas, elas oferecem, não só a máquina

de inferência mas também a administração do raciocínio inexato, os mecanismos de explicação, o manejo do diálogo com o usuário, os mecanismos de depuração e outras facilidades para a construção rápida de protótipos.

Em uma situação intermediária entre o ambiente e as ferramentas encontram-se as ferramentas híbridas, como *KEE, LOOPS, ART, MRS* etc. Elas são mais flexíveis que as ferramentas puras, entretanto mais difíceis de usar. Para cada uma das funções do sistema especialista elas contêm diferentes opções que o engenheiro do conhecimento deve selecionar e combinar. Elas são também mais flexíveis em suas possibilidades de integração com o software convencional.

LISP, como indicado na Figura 6, é uma linguagem, *INTERLISP* é uma versão de *LISP* contendo um grande número de pacotes de rotinas, portanto está bem mais próxima do conceito de ambiente do que de linguagem. *PROLOG* e *OPS5* são linguagens muito menos flexíveis que *LISP*, e que proveem uma máquina de inferência e uma série de outras facilidades úteis na construção de um sistema especialista.

A descrição dos componentes da Figura 6 será apresentada em quatro sub-seções, onde, uma tratará das linguagens, outra do ambiente e as duas últimas das ferramentas híbridas e puras.

3.1 Linguagens

As linguagens podem ser classificadas em linguagens convencionais e linguagens de inteligência artificial [WATERMAN 83]. As linguagens convencionais são aquelas essencialmente projetadas para manusear operações numéricas, tais como *FORTRAN* e *PASCAL*. Elas normalmente consistem em sequências de comandos e subrotinas. As linguagens de inteligência artificial são aquelas contendo características tais que facilitam a construção de sistemas baseados em conhecimento. Elas são projetadas, por exemplo, para manipular processamento simbólico. Dentro desta classe se encontram *LISP* e *PROLOG*. Estas duas linguagens são bem diferentes; *LISP* é uma linguagem funcional, enquanto *PROLOG* é uma linguagem baseada em lógica. Entretanto, uma característica comum entre estas linguagens é que as mesmas resultam ineficientes no "hardware" convencional, uma vez que o mesmo foi projetado para as linguagens convencionais orientadas para o cálculo. Para solucionar este problema, a partir de 1974 tem-se trabalhado intensamente no projeto e construção das chamadas máquinas *LISP* e máquinas *PROLOG*, existindo hoje, algumas versões comerciais [SHEIL 86].

3.1.1 LISP

LISP foi proposta por John Mc.Carthy [McCARTHY 65] e até recentemente era a única linguagem usada em inteligência artificial nos Estados Unidos. *LISP* consiste em um grupo de módulos, onde cada um é especializado na execução de uma tarefa em particular. Em *LISP* não há uma diferença significativa entre dados e programa. É uma linguagem recursiva, dados e programas são representados como listas, onde

as mesmas podem ser aninhadas. Uma lista é um conjunto de itens, onde itens de uma lista são os "membros" da lista. As funções de listas realizam "operações" nas listas. Algumas funções importantes são *CAR, CDR, CONS* e *APPEND. CAR* retorna o primeiro membro de uma lista. *CDR* devolve o resto de uma lista após o primeiro membro ter sido removido. *CONS* anexa um elemento ao começo de uma lista e *APPEND* junta duas listas, fazendo delas uma só.

LISP é uma linguagem bem flexível, permite a definição de novas funções em termos das funções básicas existentes, tornando possível, por exemplo, escrever em *LISP* um compilador *LISP*.

As estruturas de dados básica em *LISP* são os átomos e as "S-expressions". Cada átomo possui uma estrutura própria que contém informações sobre o átomo, incluindo seu nome, seu valor e qualquer outra propriedade que o programador deseje. Uma "S-expression" é uma estrutura de dados composta de dois campos, onde cada um deles contém um apontador para outro objeto *LISP*.

As funções em *LISP* que efetuam testes e devolvem os resultados T (true) ou NIL (falso) são chamadas predicados. A função *LISP* que atribui valor a uma variável é chamada *(SETQ X Y)* onde, *X* é instanciado no valor de *Y*. A função *COND* permite testar um conjunto de condições e depois executar alguma ação se a condição associada for verdadeira. A administração de memória em *LISP* é dinâmica e completamente automática.

Dada esta flexibilidade, foram criados uma grande variedade de ambientes *LISP* tais como:

```
MacLISP
   FranzLISP (UNIX)

InterLISP
     InterLISP-D (XEROX)
     Vax LISP    (DEC)
```

3.1.2 PROLOG

PROLOG foi inicialmente desenvolvida em 1972 por A. Colmerauer e P. Roussel [COLMERAUER 82] , na Universidade de Marseille. *PROLOG* é uma linguagem de programação que implementa uma versão simplificada do Cálculo de Predicados. O primeiro compilador eficiente foi desenvolvido na Universidade de Edinburgh. Como *LISP*, *PROLOG* foi projetada para manipulação simbólica. *PROLOG* é eficiente em processamento de listas.

PROLOG foi criada especialmente para responder perguntas sobre base de conhecimentos consistindo em regras e fatos. Em *PROLOG* o encadeamento das regras é efetuado regressivamente. Além disto, ele se utiliza de outra técnica, conhecida como "backtraking", rastreamento para trás. O encadeamento regressivo, definido na Seção 2.1.2, é uma técnica na qual uma conclusão ou consequência é assumida

como verdadeira, e depois uma base de conhecimento de regras e fatos é examinada para verificar se ela se apóia essa suposição. Se a suposição não estiver correta, o rastreamento para trás é usado para desfazer a suposição original e substituí-la por uma outra.

Em essência, uma computação em *PROLOG* é simplesmente o controle de deduções lógicas. Ele usa um sub-conjunto da Lógica de Predicados de Primeira Ordem, denominado Cláusulas de Horn.

3.1.3 INTERLISP

E' um conjunto integrado de ferramentas para o desenvolvimento de programas em *LISP* [LUCENA 87]. *INTERLISP* inclui procedimentos uniformes para a manipulação de erros, correção automática de erros, um editor de tela e um sistema de arquivos. O editor de *INTERLISP* opera diretamente sobre a estrutura de lista do programa e dos dados. O editor é uma função *LISP*, permitindo assim, ao usuário definir comandos do editor como macros ou mesmo definir macros de edição que ativam programas em *LISP*. Mesmo os programas em *INTERLISP* podem chamar o editor e lhes fornecer comandos. O pacote de depuração de *INTERLISP* oferece facilidades para interromper e acompanhar a execução de funções. Após a ocorrência de um erro, um novo *"read-evaluate-print loop"* é invocado, no qual a pedido do usuário um acompanhamento da parte do programa anterior ao erro é efetuado. Além disto, o usuário pode pedir para avaliar qualquer comando *LISP*. O sistema de arquivos presente em *INTERLISP* auxilia, em particular, o acompanhamento das funções que realmente estão em uso [TAPPEL 82].

As ferramentas são integradas de duas maneiras:

(a) Todas as ferramentas estão a disposição do programador em quase todos os estados da computação.

(b) Uma única ferramenta é usada para um dado propósito em todos os estados.

3.2 Ambientes

3.2.1 OPS5

As principais vantagens de *OPS5* [BROWNSTON 85] são a sua estabilidade e eficiência. Suas principais desvantagens são a falta de uma boa interface com o usuário, facilidades de edição e explicação. *OPS5* tem somente uma memória de dados. Esta memória pode ser de qualquer tamanho e consiste em um conjunto de unidades chamadas elementos de memória. Estes elementos são ordenados pelo seu tempo de criação e pelas modificações mais recentes. Os atributos de dados são do tipo elemento-atributo-valor e vetores. Cada elemento de memória pode conter somente um atributo vetor. Variáveis são disponíveis para permitir o casamento entre condições e para permitir que valores sejam usados do lado direito da regra. Em algumas versões do sistema, regras podem ser criadas em tempo de execução.

Fatos em *OPS5* são representados como objetos com atributos e valores. Regras ou produções são representadas como *IF-THEN* comandos. A máquina de inferência para OPS5 é simples. Regras são comparadas aos elementos da memória até que uma regra seja disparada e nova informação colocada na memória. Então o ciclo recomeça.

O esquema básico de inferência é o encadeamento progressivo. *OPS5* provê ainda facilidades para depuração e resolução de conflito. *OPS5* originalmente foi implementada em *LISP*.

3.3 Ferramentas Híbridas

3.3.1 KEE "The Knowledge Engineering Environment"

O sistema *KEE* "Knowledge Engineering Environment" é uma ferramenta híbrida. Ela possui um conjunto integrado de ferramentas de "software" que lhe tem permitido o uso em diferentes áreas.

A memória de dados de *KEE* é um sistema baseado em *frames* organizados como objetos [COX 86]. O usuário pode declarar objetos genéricos, e instâncias destes objetos genéricos automaticamente herdarão seus atributos. As regras são muito simples na forma. O antecedente é uma simples conjunção de predicados sobre os valores dos atributos. O consequente preenche o valor de um ou mais atributos. O usuário pode examinar e trocar os valores dos atributos enquanto o sistema está sendo executado, uma vez que a interface com o usuário é executada como um processo separado. O esquema básico de inferência é o encadeamento regressivo, mas o usuário pode modificar consideravelmente a estrutura de controle usando os mecanismos de "displays" gráficos.

O sistema *KEE* é implementado em Máquinas *LISP* com facilidades de janelas múltiplas, "display" gráfico e um "mouse". *KEE* é um exemplo de programação orientada por objetos [COX 86]. Fatos e regras são representados por objetos ou *frames* com rótulos para os *slots* contendo valores ou meios para obter valores.

3.3.2 LOOPS

LOOPS é um ambiente de "software" desenvolvido pelo Centro de Pesquisa da Xerox de Palo Alto, que incorpora uma variedade de construções diferentes de engenharia do conhecimento em um pacote unificado. As construções incluem programação orientada por objeto, um esquema de gerenciamento da base de conhecimento e um pacote de regras.

LOOPS possui um bom editor de textos com facilidades para depuração. As regras são organizadas em seções modulares chamadas conjunto de regras. Os conjuntos de regras consistem em um conjunto de regras ordenadas e uma estrutura de controle.

3.3.3 ART "The Automated Reasoning Tool"

ART é um pacote para o desenvolvimento de sistemas baseados no conhecimento. O pacote contém quatro componentes principais: uma linguagem baseada em conhecimento para expressar os fatos e seus relacionamentos, um compilador para converter a linguagem baseada em conhecimento para *LISP*, um aplicador o qual constitui a máquina de inferência e um ambiente de desenvolvimento que inclui facilidades de depuração.

A máquina de inferência, ou aplicador de conhecimento, é descrito como um raciocinador oportunista. Isto significa que *ART* pode raciocinar com o encadeamento progressivo, regressivo, ou através de explícitos comandos procedurais. Regras afetam a direção da inferência. Neste caso, a máquina de inferência se movimenta oportunisticamente, dependendo do tipo do resultado intermediário. *ART* é escrito em *LISP* e é executado nas máquinas *LISP* produzidas pela *XEROX*.

3.3.4 MRS "Meta-level Representation System"

O "Meta-level Representation System" (MRS) [RUSSELL 85] é um sistema de programação em lógica. Sua principal característica inclui um esquema de controle flexível utilizando raciocínio para frente, raciocínio para trás e uma técnica denominada resolução [RICH 83] para provar teoremas, além da habilidade de representar "metalevel knowledge", isto é, conhecimento sobre ele mesmo. E com tal característica, *MRS* pode ser utilizada para implementar virtualmente todo tipo de aplicação de Inteligência Artificial em uma variedade de arquiteturas. Os fatos em *MRS* possuem forma similar a chamada de função em *LISP*. O relacionamento entre objetos é seguido pelos objetos. Por exemplo, *(father Jerry Jim)* significa que *Jerry* é pai de *Jim*. Vários relacionamentos, tais como as operações de comparações e aritméticas já estão embutidas na definição de *MRS*. Uma regra em *MRS* consiste em uma palavra chave *if*, uma parte condição e uma parte ação. Um comando *if* significa que a ação é verdadeira se a condição for verdadeira, ou seja, "ação" é executada somente se a "condição" for verdadeira. Variáveis podem ser usadas nas proposições e regras. As proposições são fatos descrevendo os objetos que representam a área do conhecimento. As regras usam as proposições para inferir novas proposições. Os fatos na parte de condição de uma regra podem ser conectados com um comando *and*, *(and* $<$ *fact1* $>$ $\cdots$ $<$ *factn* $>$*)*. Um comando *and* é verdadeiro quando todos os fatos, indo de 1 até n o são. Os comandos *or* e *not* possuem formas similares ao *and*. O exemplo a seguir ilustra uma regra em *MRS* e significa que o fato 3 é verdadeiro se o fato 1 for verdadeiro e o fato 2 for falso.

(if (and (fact1) (not (fact2))) (fact3)).

Conjuntos diferentes de proposições e regras podem ser colocadas em teorias diferentes. Em *MRS* uma teoria é um conjunto de fatos em sua própria biblioteca. Isto

melhora sua eficiência porque teorias inativas não são pesquisadas, ou seja, dividindo o conhecimento total do problema em áreas diferentes, a atenção pode ser focalizada apenas em uma porção relevante do problema em um dado instante.

O processo de computação em *MRS* para solucionar um problema se resume em produzir novos fatos a partir dos fatos originais. As regras que determinam os fatos que podem ser adicionados a partir de fatos existentes são denominadas Regras de Inferência. Por exemplo, a regra básica de inferência usada frequentemente por *MRS* é denominada *Modus Ponens*:

```
(if A B), A
-----------
     B
```

cuja interpretação é: se A implica em B, e A é verdadeiro, então B pode ser deduzido. Exemplo, se é verdade:

```
(if (CurrentYear 1985) (Age John 22)) and (CurrentYear 1985)
```

pode-se concluir (Age John 22) e solucionar o problema.

A razão porque *MRS* utiliza esta regra de inferência é devido a dois fatores, primeiro, porque a grande parte do conhecimento é naturalmente expresso via *if*, segundo, porque mesmo que não seja, ele pode normalmente ser reescrito de tal forma.

Existem outras formas de raciocínio em *MRS* como os encadeamentos para-frente e para-trás. Além da habilidade de mudar o tipo de encadeamento, progressivo e regressivo, *MRS* controla sua maneira de trabalhar através de *meta-level rules*, ou seja, regras sobre regras. Assim *MRS* pode ser adaptada a situações diferentes substituindo-se as *meta-level rules*. Redefinindo estas regras sobre regras, *MRS* pode inclusive raciocinar diferentemente. A maneira em que as proposições são armazenadas e acessadas no banco de dados também pode ser mudado.

MRS foi implementado em *INTERLISP*, sua sintaxe lembra a sintaxe de *LISP*. Funções em *LISP* podem ser chamadas por *MRS* e, *LISP* pode executar os comandos de *MRS* assim como manipular os resultados retornados por estes comandos.

MRS está disponível em três máquinas diferentes. Elas são: *DEC-20* rodando *MacLISP*, *VAX* rodando *FranzLISP* sobre o *UNIX* de Berkeley e "*Symbolics LISP machine LM-2/3600*".

3.4 Ferramentas Puras

3.4.1 EMYCIN

É uma ferramenta para a construção de um sistema de consultas baseado em *MYCIN*. *MYCIN* é um sistema especialista que incorpora o conhecimento de especialistas em meningite. Grande parte do conhecimento em *EMYCIN* é representado

por regras de produção e a sua estratégia básica de controle é o encadeamento regressivo. Dentre as facilidades de *EMYCIN*, destaca-se o programa explicador, um depurador e uma biblioteca de testes.

3.4.2 M.1

M.1 [HARMON 85] foi desenvolvida por uma firma americana denominada Teknowledge Inc., fundada por um grupo de pesquisadores da Universidade de Stanford. *M.1* foi liberada ao público em Junho de 1984 e pertence à família de ferramentas derivadas de *EMYCIN*. Muito embora *EMYCIN* tenha sido codificada em *LISP*, a primeira versão de *M.1* foi elaborada em *PROLOG*, e a versão atual está codificada em *C*. *M.1* utiliza a representação baseada em regras e o encadeamento se realiza regressivamente. As expressões são normalmente uni-valoradas mas podem ser declaradas multi-valoradas. A interface com o usuário é simples. O diálogo se realiza em termos compreensíveis por si mesmos, e portanto não requerem o entendimento de um vocabulário próprio do sistema.

M.1 foi projetada para auxiliar o desenvolvimento de sistemas baseados no conhecimento que manuseiam consultas baseadas em diagnóstico/prescrição. Os fatos são representados como pares atributo-valor. As regras de produções representam o conhecimento heurístico. Uma regra testa o valor de uma expressão e a partir dele conclui valores para outras expressões. A característica fundamental de M.1 é o seu suporte para regras "variáveis". Esta técnica permite a inserção de várias regras similares escrevendo apenas uma regra genérica, e então adicionando uma série de informações de uma maneira similar a uma tabela "look-up".

A interação do usuário com *M.1* é feita via respostas a perguntas feitas pelo sistema. A base de conhecimento de *M.1* é montada fora da ferramenta com o auxílio de qualquer editor de texto.

3.4.3 S1

S.1 é um pacote integrado de software. Fatos em *S.1* são armazenados como triplas objeto-atributo-valor. Objetos, chamados classes, podem ser agrupados dentro dos tipos de classes. Membros de um tipo de classe herdam um conjunto de atributos em comum. Atributos podem ser arranjados em hierarquias quando uma propriedade de uma classe é uma precondição para outra. Similarmente, hierarquias de valores declaram relacionamentos entre valores. Outros relacionamentos são representados como regras de produções. A máquina de inferência em *S.1* é o encadeamento regressivo. O controle do processo de raciocínio é facilitado pelos blocos de controle. Um bloco de controle é um procedimento explícito que indica como a consulta deve prosseguir. O usuário se comunica com *S.1* via um "mouse" ou entrando informações via console. *S.1* foi implementado em *LISP*.

3.4.4 EXPERT

É uma ferramenta para projetar e construir sistemas baseados em consultas [HARMON 85]. *EXPERT* armazena fatos como pares atributo-valor. Fatos são classificados de duas maneiras: "findings" (descobertas) ou hipóteses. "Findings" são dados de observação que são fornecidos ao sistema. Como exemplo de "findings" temos as observações ou medidas tais como altura e pressão sanguínia de pacientes. Hipóteses são conclusões que podem ser deduzidas pelo sistema, como por exemplo, diagnóstico.

Os relacionamentos e heurísticas são armazenados como regras de produções agrupados em três categorias.

- "F-F" regras que ligam um fato encontrado com outro fato encontrado. São usados para estabelecer um controle local sobre uma sequência de perguntas. Exemplo de uma regra deste tipo, "se A é verdade então B é falso".
- "F-H" regras que relacionam fatos encontrados com hipóteses
- "H-H" regras que ligam uma hipótese com outras hipóteses.

O controle em *EXPERT* é estabelecido de acordo com a ordem e a categoria das regras. A interface com o usuário é muito simples. *EXPERT* é implementado em *FORTRAN.*

4 Esboço de desenvolvimento do Sistema Especialista para o Otimizador Local

4.1 Introdução

Esta seção focaliza o problema do entendimento humano e a condução do problema, apresentando as técnicas usadas no processo de desenvolvimento de um Sistema Baseado em Conhecimento para um Otimizador "peephole", e o tipo de recurso que deve ser considerado quando se propõe a desenvolver um compilador otimizante. Entretanto, o objetivo principal desta seção é mostrar como o conhecimento capturado de um especialista na área de otimização de código pode ser representado utilizando-se uma das ferramentas apresentadas neste texto, no caso, *MRS.* A escolha desta ferramenta, apresentada na Seção 3.3.4, se deve ao fato de ser ela a que melhor se adapta as características do problema em questão. O exemplo utilizado nesta seção foi extraído de Warfield [WARFIELD 88].

Otimização "peephole" ou Local é uma técnica utilizada para melhorar a qualidade do código objeto gerado por compiladores. O otimizador procura substituir uma ou mais instruções por uma instrução mais rápida ou menor. O termo "peephole" foi derivado do fato de que o otimizador, na tentativa de melhorar o desempenho do programa objeto somente vê uma pequena janela "peephole" de instruções de máquina adjacentes sem usar nenhum conhecimento global do programa.

A missão deste Sistema Especialista é reconhecer quais instruções podem ser otimizadas a partir da descrição das instruções da máquina alvo. Este Sistema deve poder ser redirecionado para outro computador substituindo-se a descrição das instruções e a descrição dos modos de endereçamento.

As etapas consideradas no desenvolvimento do sistema baseado em conhecimento para um otimizador local são descritas abaixo. Elas seguem mais ou menos a ordem sugerida na Seção 2.

Etapas 1 e 2 incluem a identificação do problema, seleção de uma ferramenta adequada e análise do conhecimento que deve ser incluído no sistema; Etapa 3 inclui o projeto do sistema; Etapa 4 envolve a criação da base de conhecimento e o teste da mesma. Mais duas etapas (5 e 6) seriam necessárias para completar o sistema. Etapa 5 inclui a expansão do sistema, os testes e revisões até que ele faça exatamente o que foi requerido e, finalmente, Etapa 6 consiste na manutenção necessária do sistema. Entretanto no exemplo elas não serão mostradas.

Usualmente nas etapas 1 e 2, o especialista conversa com o cliente a respeito do problema, pedindo-lhe para descrever o tipo de programa desejado, as restrições de implementação, etc. A partir desta conversa, recomenda-se algumas maneiras através das quais o objetivo pode ser obtido e o tipo de ferramenta adequada para o problema em questaão.

Em *MRS*, o conhecimento capturado sobre um determinado domínio pode ser distribuído em teorias diferentes com o objetivo de obter maior eficiência. Este desmembramento permite focalizar a atenção somente sobre partes relevantes do problema em um dado instante.

Baseado nas considerações acima, três teorias são usadas para o otimizador local redirecionável:

(a) Teoria de descrição de instruções:

Esta teoria descreve o que cada instrução faz em termos das proposições entendidas pelo MRS. Constantes denotando espaço e tempo são definidas para cada instrução para auxiliar as regras na determinação de instruções eficientes. Também pode ser incluído na teoria de descrição de instruções um fato que especifica quando o espaço ou o tempo é preferível na determinação de uma instrução mais eficiente.

(b) Teoria de modos de endereçamento:

Esta teoria contém proposições que descrevem os modos de endereçamento e suas classes. Os modos de endereçamento podem ser agrupados em classes particulares.

(c) Teoria de otimização:

Todas as regras estão nesta teoria. Usando o raciocínio para trás e um "peephole" de duas instruções, as regras encontram todas as otimizações possíveis usando definições de instruções contíguas. As regras pegam todos os pares de instruções da teoria de descrição de instruções, as combinam e tenta encontrar uma única instrução mais eficiente para substituir a combinação das instruções

originais. O otimizador de regras retorna uma lista de todas as otimizações possíveis e as condições que devem ser verdadeiras para que o otimizador funcione. Novas regras são criadas a partir daquelas que o otimizador retorna. Raciocínio para frente é usado com estas novas regras quando se está otimizando o código objeto de um programa compilado. Além de descobrir todas as combinações de instruçò es que podem ser otimizadas, o otimizador também considera desvio sobre desvio e modos de endereçamento especiais.

As seções seguintes ilustram como se representa o conhecimento adquirido nestas teorias.

4.2 Definição de instruções

A teoria de definição de instruções contém a definição de cada instrução. O programa e os exemplos ilustrados neste texto se utilizam do repertório de instruções do *VAX*.

Uma instrução é denotada por uma proposição "instruction". Por exemplo, a instrução *beql* é representada da seguinte maneira:

```
(instruction beql)
```

As instruções se classificam em instruções de desvio e outras instruções. As instruções de desvio necessitam somente da "condição" para permitir a verificação do registrador de condição da máquina. A "condição" significa que um desvio para um rótulo é efetuado se o registrador de condição estiver ligado, senão o programa continua com o próximo comando.

A instrução *beql* testa se o registrador de código de condição está ligado; e a instrução *bneq* testa se o mesmo está desligado.

Exemplos da implementação destas instruções são:

```
(instruction beql)
(set beql z)

(instruction bneq)
(reset bneq z)
```

onde z representa o código de condição "zero".

Outras instruções usam as proposições "set" e "reset" para descreverem quais códigos de condição são ligados para cada instrução. Estas instruções são usadas com as instruções de desvio condicional para descreverem suas ações. As informações não definidas em instruções de desvio são inferidas do fato de que ela é um desvio. Uma instrução de desvio incondicional não necessita as proposições "set" ou "reset". Por exemplo,

(instruction jmp)

O registrador de condição não necessita uma descrição completa, pois ele é usado somente para determinar se uma otimização pode acontecer. As proposições "set" e "reset" são usadas para descrever o que uma instrução faz com o código de condição. Por exemplo, a instrução "addb3", a qual liga os registradores N e Z dependendo do resultado da adição, é implementada com a proposição *(set addb3 NZ)*.

Regras que não são instruções de desvio requerem uma descrição mais detalhada. Elas necessitam informações sobre registradores de condição, especificação de tamanho, espaço e tempo; além dos operandos à esquerda "left" e o primeiro operando à direita, "right1". Se a instrução é lógico-aritmética, ainda é necessário um segundo operando à direita, "right2".

A especificação do tamanho representa o tamanho do operando, 1 para "byte", 2 para palavra ou 4 para palavra longa. Por exemplo, a instrução "addb3" é uma instrução de 1 *byte* e é implementada por:

(*size addb3 1*)

Na arquitetura do VAX todas as instruções em uma otimização devem ter operandos do mesmo tamanho para que a otimização seja válida.

As constantes especificando tempo e espaço na especificação de uma instrução são arbitrárias, mas devem ser consistentes com todas as instruções. O espaço e o tempo da instrução "addb3" pode ser implementado por:

```
(space addb3 3)
(time  addb3 3)
```

O fato "prefer" considera o *tempo* ou o *espaço* na descoberta de otimizações. A ausência deste fato indica que não há preferência. Se não há preferência, a combinação das constantes de tempo e espaço de duas instruções deve ser maior do que as constantes de tempo e espaço da terceira instrução. Uma preferência é indicada pelo fato "prefer" tal como:

```
(prefer time)
```

As especificações *left*, *right1* e *right2* se referem ao comando de atribuição que descreve cada instrução. Por exemplo, um comando de atribuição para uma instrução "addb3" é implementado da seguinte forma:

```
(left   addb3 operand3 all-modes)
(right1 addb3 operand2 all-modes)
(right2 addb3 operand1 all-modes)
```

onde a primeira proposição estabelece que o "register-deferred" é indireto através de um registrador. A segunda proposição estabelece que o modo de endereçamento autodecremento é indireto e que efetua algum tipo de operação aritmética.

Modos de endereçamento especiais necessitam ser definidos completamente para que as otimizações que usam estes modos de endereçamentos possam ser encontradas. Sua definição é idêntica a definição de instruções aritméticas. O modo de endereçamento autodecremento é definido por:

```
(mode autodecrement)
(operation autodecrement subtraction)
(left   autodecrement operand2 register)
(right1 autodecrement operand2 register)
(right2 autodecrement operand1 size)
```

e significa que autodecremento subtrai o tamanho do operando de um registrador.

A proposição *(constant-mode immediate)* significa que o modo de endereçamento imediato pode ser usado com constantes. Esta proposição é usada para auxiliar as regras na construção de otimizadores que usam constantes, tais como o modo de endereçamento autodecremento.

4.4 Definições de Regras

As regras tentam encontrar todas as otimizações possíveis usando certos critérios. Uma otimização é considerada válida somente se as constantes referentes ao tempo e tamanho das instruções originais são ambas maiores do que as constantes de espaço e tempo da nova instrução. Além disto, o efeito que uma instrução na otimização tem sobre o código de condição deve ser equivalente.

Um desvio condicional seguido por um desvio incondicional pode ser combinado em um desvio condicional cuja condição seja o oposto da condição que a instrução original usa e cujo rótulo seja o mesmo que o rótulo especificado no desvio incondicional na instrução original. Por exemplo, dada a sequência de instruções:

```
    beql L1
    jmp  L2
L1:
```

O efeito da instrução *beql* é definido por *(set beql Z)*. As regras sabem que o complemento de um *set* é *reset*. Uma instrução de desvio é então procurada para efetuar um *reset* sobre *Z*. A instrução *bneq* é encontrada e substitui-se as duas instruções originais como ilustrado abaixo,

```
    bneq L2
L1:
```

A especificação "left" se refere ao operando LHS (*left hand side*) do comando de assinalamento. As especificações "right1" e "right2" se referem aos operandos do RHS (*right hand side*) de comando de atribuição. Sendo neste exemplo, *operand3* o lado esquerdo, *operand2* o operando *right1* e o *operand1* o operando *right2*. "All-modes" indica que os três operandos podem ser usados com qualquer modo de endereçamento. Um modo de endereçamento para cada um dos operandos deve ser especificado na definição do operando e pode ser o próprio nome de um modo de endereçamento, uma constante que representa o valor do operando ou ainda a palavra chave "size" se o valor do operando depende do seu tamanho. A Seção 4.3 apresenta um exemplo para este caso mostrando a definição do modo de endereçamento autodecremento.

A especificação "operation" informa que operação uma instrução aritmética executa, por exemplo, para "addb3" tem-se:

```
(operation addb3 addition)
```

Instruções não aritméticas não necessitam as especificações "operation" nem "right2"

4.3 Definições de Modos de Endereçamento

Os modos de endereçamento são distintos das instruções pelo uso de uma proposição de "mode" ao invés de uma proposição "instruction". Por exemplo, um modo de endereçamento de autodecremento é implementado por:

```
(mode autodecrement)
```

Modos de endereçamento também podem ser agrupados usando a especificação "subset". Por exemplo, o modo de endereçamento autodecremento é um subconjunto de todos os modos de endereçamento e este fato é implementado como ilustra o exemplo

(mode-subset autodecrement all-modes).

Durante a verificação por uma otimização válida estas proposições são usadas para determinar se os operandos das instruções usam ou não o mesmo modo de endereçamento.

Se um modo de endereçamento é indireto, este fato pode ser especificado usando a proposição "indirect". Por exemplo, o modo de endereçamento autodecremento e o "register-deferred" são implementados por:

```
(indirect register-deferred register)
(indirect autodecrement operation)
```

As instruções que não são de desvios são combinadas primeiramente classificando-se duas instruções em um dos três grupos:

(a) Nenhuma das duas instruções é uma instrução lógico-aritmética.

(b) A primeira instrução é lógico-aritmética mas a segunda não.

(c) A primeira instrução não é lógico-aritmética e a segunda o é. Este grupo é subdividido em três subgrupos:

   i. Os dois operandos da segunda instrução "right1" e "right2" podem ser combinados com o "left" operando da primeira instrução que não é lógico-aritmética.

   ii. Somente o operando "right1" da segunda instrução pode ser combinado com o operando "left" da primeira instrução.

   iii. Somente o "right2" operando da segunda instrução pode ser combinado com o "left" operando da primeira instrução.

Os três grupos apresentados são as únicas combinações possíveis de duas instruções.

Nos exemplos mostrados a seguir, assuma que o operando "left" da primeira instrução é igual ao operando "right1" da segunda instrução e que este operando está morto, isto é, um operando que não é mais necessário.

Por exemplo, no grupo 1, duas instruções *movb*

movb a, b — b é o "left" operando
movb b, c — b é o "right1" operando

poderiam ser substituídas por uma terceira instrução: *movb a,c*

No grupo 2, por exemplo, uma instrução *addb3* e uma instrução *movb*

addb3 a, b, c — c é o "left" operando
movb c, d — c é o "right1 operando

poderiam ser substituídas por: *addb3 a, b, d*

No grupo 3(a), por exemplo, uma instrução *movb* e uma instrução *addb3*

movb a, b — b é o "left" operando
addb3 b, b, c — ambos "right" operandos

poderiam ser substituídas por: *addb3 a, a, c*

No grupo 3(b), por exemplo, uma instrução *movb* e uma instrução *addb3*

movb a,b — b 'e o "left" operando
addb3 b, c, d — b 'e o "right1" operando

poderiam ser substituídas por: addb3 a, c, d

No grupo 3(c), por exemplo, uma instrução *movb* e uma instrução *addb3*

movb a, b — b é o "left" operando
addb3 c, b, d — b é o "right2" operando

poderiam ser substituídas por: *addb3 c, a, d*

A descoberta de regras para instruções que usam modos de endereçamento especial é feita de maneira diferente. Uma vez que já são conhecidas as operações que os modos de endereçamento especial executam, uma instrução que efetua a mesma operação é encontrada. Esta instrução passa a ser a primeira instrução na otimização e tem os operandos restritos aos operandos que são especificados no modo de endereçamento.

Os exemplos apresentados nesta Seção e os que serão mostrados na Seção 4.5 ilustram como regras de otimizações são deduzidas a partir dos critérios aquí estabelecidos.

4.5 Resultados do uso de um Sistema Baseado em Conhecimento

Os resultados de um Sistema Baseado em Conhecimento são colocados na forma de regras e postos em uma teoria chamada regras. O encadeamento progressivo é usado para otimizar o código objeto usando estas regras. Cada linha do código objeto é inserida como um fato na biblioteca após ter sido substituída para a forma de *MRS*. Um exemplo para esta nova forma é:

clrb @R3
se transforma em:
(instruction1 (clrb (register-deferred 3)))

A "instruction1" ou "instruction2" é inserida para cada instrução de tal forma que *MRS* saiba qual instrução na sequência ela é. Os operandos da instrução são substituídos por nomes simbólicos do modo de endereçamento usado nos operandos. Os mesmos são seguidos por alguma constante pertencente aos operandos. No exemplo da instrução "clear" a constante é 3.

Após duas instruções terem sido declaradas, *MRS* pesquisa automaticamente as regras para encontrar uma regra que casa as duas instruções. Se uma for encontrada, *MRS* declara a nova instrução otimizada na biblioteca. Esta nova declaração pode ser recuperada da biblioteca para ser usada na substituição de duas instruções originais no código objeto. Após cada otimização de duas instruções ser tentada, as duas instruções originais junto com a terceira instrução, se houver, devem ser removidas da biblioteca.

As regras de otimização são da forma:

```
(if (and(instruction1 $x)
        (instruction2  $y)
        (dead $1))
        (instruction3 $z))
```

onde $ inicia um nome de variável em *MRS*.

Este exemplo significa que a terceira instrução é verdadeira se a primeira instrução "instruction $x" e a segunda instrução "instruction $y" são ambas verdadeiras. Além disto, o registrador $1 deve estar também morto se a condição "dead" aparece na regra.

Para os dois desvios condicionais *bleq* e *beql* definidos anteriormente, o sistema baseado em conhecimento deduz duas otimizações:

```
(if (and (instruction1 (bleq $1)) (instruction2 (jmp $2)))
   (instruction3 (beql $2)))

(if (and (instruction1 (beql $1)) (instruction2 (jmp $2)))
   (instruction3 (bleq $2)))
```

Para a instrução *movb* definida, a regra do Grupo 1 deduz a seguinte otimização:

```
(if (and (instruction1 (movb $1 $2))
 (instruction2 (movb $2 $3))
 (dead $2))
 (instruction3 (movb $1 $3)))
```

Para a instrução *addb3* definida, a regra do Grupo 2 deduz a seguinte otimização:

```
(if (and (instruction1 (addb3 $1 $2 $3))
 (instruction2 (movb $3 $4))
 (dead $3))
 (instruction3 (addb3 $1 $2 $4)))
```

Para a instrução *addb3* definida, a regra do Grupo 3(a) deduz a seguinte otimização:

```
(if (and (instruction1 (movb $1 $2))
 (instruction2 (addb3 $2 $2 $3))
 (dead $2))
 (instruction3 (addb3 $1 $1 $3)))
```

Para a instrução *addb3* definida, a regra do Grupo 3(b) deduz a seguinte otimização:

```
(if (and (instruction1 (movb $1 $2))
 (instruction2 (addb3 $2 $3 $4))
 (dead $2))
 (instruction3 (addb3 $1 $3 $4)))
```

Para a instrução *addb3* definida, a regra do Grupo 3(c) deduz a seguinte otimização:

```
(if (and (instruction1 (movb $1 $2))
 (instruction2 (addb3 $3 $2 $4))
 (dead $2))
 (instruction3 (addb3 $3 $1 $4)))
```

Uma vez definidos os modos de endereçamentos autodecremento, autoincremento e as instruções *addb2*, *clrb*, *subb2*, as regras para modos de endereçamento especial descobrem as seguintes otimizações:

```
(if (and (instruction1
           (subb2 (immediate 1) (register $1)))
         (instruction2
           (clrb (register-deferred $1))))
         (instruction3 (clrb (autodecrement $1))))

(if (and (instruction1
           (addb2 (immediate 1) (register $1)))
         (instruction2
           (clrb (register-deferred $1))))
         (instruction3 (clrb (autoincrement $1))))
```

Instruções adicionais na Teoria de Instruções permitirão ao otimizador encontrar outras regras similares a estes exemplos.

5 Conclusão

O objetivo deste texto foi dar uma visão geral dos tópicos mais relevantes na construção de um sistema especialista, tais como as técnicas existentes para a representação do conhecimento, a metodologia usada para a aquisição de dados, as linguagens, ferramentas, além de mostrar a aplicabilidade dos Sistemas Baseados em Conhecimento no contexto de otimização de código.

Com respeito às linguagens, percebe-se que têm ocorrido inúmeros debates relacionados com a existência ou não de uma linguagem de representação universal que seja boa o suficiente para todos os problemas em qualquer domínio. Nestes debates, reivindicações tem sido feita para Lógica do Predicado a este respeito [KOWALSKI,

79], [GENESERETH 85]; Fikes e Kehles [FIKES 85] defendem uma visão da representação centrada em objetos, enquanto regras de produção tem sido fortemente defendida por Newell e Simon [NEWELL, 72], Hayes-Roth [HAYES-ROTH 85] entre outros. Entretanto tudo o que se pode dizer no momento é que não existe evidência de que tal linguagem exista ou mesmo um argumento forte que justifique a sua existência.

Isto não significa que a comunidade de Inteligência Artificial esteja sendo dividida em campos competitivos por estas três metodologias, muito embora cada uma delas apresente vantagens em certos domínios específicos, por exemplo; *lógica*, onde o domínio pode ser prontamente axiomatizado e onde modelos casuais completos estão disponíveis; *regras*, onde a maior parte do conhecimento pode ser convenientemente expresso com heurística experimental e *frames*, onde descrição de estruturas complexas é necessária para definir adequadamente o domínio, a visão atual é de síntese e não de exclusividade. Ambos sistemas, baseados em regras e lógica normalmente incorporam estruturas *frame-like* para facilitar a representação de grandes quantidades de informações baseadas em fatos, e sistemas baseados em *frames* como o *KEE* permitem ambos, regras de produção e cálculo de predicado, serem armazenados e ativados dos *frames* para efetuar a inferência. A próxima geração de ferramentas de representação do conhecimento pode inclusive auxiliar usuários na seleção de metodologias apropriadas para cada classe em particular de conhecimento e então automaticamente integrar várias metodologias assim selecionada em um contexto consistente para conhecimento.

Percebe-se também que o problema central na arquitetura de um sistema especialista está intimamente relacionado ao esquema de representação e ao tipo de controle escolhido. É imprescindível que o projetista decida que esquema de representação lhe permitirá codificar o conhecimento de tal forma que seja fácil sua subsequente aplicação e também saiba escolher o tipo de controle que melhor se adapte a natureza do espaço da busca escolhido.

Estas decisões deveriam ser tomadas baseadas em princípios, mas infelizmente na maioria das vezes isto não ocorre pois considerações tais como, configuração do "hardware" existente, perfil do programador disponível e outros fatores colocam uma série de restrições sobre as decisões que deveriam ser postas em prática.

O objetivo do exemplo apresentado na Seção 4 é mostrar como o conhecimento humano capturado de um especialista em otimizaçõ de código pode ser representado usando uma das ferramentas apresentadas neste texto. O exemplo demonstra que um Sistema Baseado em Conhecimento pode ser usado por um Otimizador Local. Para redirecionar o Sistema Especialista para outra máquina basta trocar a descrição das instruções na teoria de instruções e a descrição dos modos de endereçamento na teoria de modos de endereçamento.

O Sistema Especialista pode também manusear modos de endereçamento especial "jump" sobre "jump". A técnica usada para estes dois casos é única na maneira em que as otimizações são descobertas. Otimizações que usam modos de endereçamento especial são descobertas começando com a última instrução presente na regra de otimização e encontrando as duas instruções que são necessárias para realizar a

tarefa da última instrução na regra. Este método é um melhoramento em relação a outras técnicas, por exemplo, Davidson e Fraser [DAVIDSON 84], uma vez que em seu modelo é necessário combinar as duas instruções antes de encontrar uma terceira instrução.

Na ferramenta *MRS*, para encontrar outras otimizações a pesquisa e o reconhecimento de padrão são efetuados automaticamente, o que torna o uso de um Sistema Especialista mais desejável. Assim, a quantidade de trabalho requerido para um programa para construir e manter um otimizador local redirecionado é reduzido usando um sistema especialista. Um Sistema Especialista pode fazer inferências a partir de fatos conhecidos. Isto significa que menos código é necessário ser escrito para um otimizador local redirecionado que usa um Sistema Especialista. Regras adicionais podem ser incluídas ao Sistema Especialista sem mudar o resto do programa.

Bibliografia

[ALFONSECA89] ALFONSECA, Manuel, "*Frames, Semantic Networks and object-oriented programming in APL2*" IBM J. Res. Develop. Vol. 33 no. 5, September 1989.

[BARR 82] BARR, A. and Feigenbaum, E. A., THE HANDBOOK OF ARTIFICIAL INTELLIGENCE. Vol 1. Los Altos, California:Morgan Kaufman. Eds. 1982.

[BUCHANAN 83] BUCHANAN B. G, et alii., em "Constructing an Expert System", in BUILDING EXPERT SYSTEMS, editores Hayes-Roth, E. et alii, 1983.

[BRACHMAN 83] BRACHMAN R. J, et alii., *"What Are Expert Systems ?"*, in BUILDING EXPERT SYSTEMS, editores Hayes-Roth, E. et alii, 1983.

[BROWNSTON 85] BROWNSTON, Lee, Farrell, R., Kant E., PROGRAMMING EXPERT SYSTEMS IN OPS5. Addison-Wesley Pub. Co., Inc. - 1985.

[CARNOTA 88] CARNOTA R. J. and Teszkiewicz A. D., SISTEMAS EXPERTOS Y REPRESENTACION DEL CONOCIMIENTO. EBAI. Edição EBAI - 1988.

[COHEN 85] COHEN, J. & HICKEY, T. J., *"Parsing and compiling using PROLOG"*, Computer Science Department, Ford Hall, Brandeis University Waltham, Massachusetts, 1985.

[COLMERAUER 82] COLMERAUER, A., *"PROLOG-II, Manuel de reference et modele theorique"*, Groupe d'Intelligence Artificielle Universite d'Aix-Marseille II, 1982.

[COX 86] COX, Brad J., OBJECT ORIENTED PROGRAMMING An Evolutionary Approach, ADDISON-WESLEY PUBLISSHING COMPANY, 1986.

[FIKES 85] FIKES, Richard and KEHLER Tom em The Role of Frame-based Representation in Reasoning, Communications of the ACM, September 1985 Volume 28 Number 9.

[DAVIDSON 84] DAVIDSON, J. W, & FRASER C. W., *"Automatic Generation of Peephole Optimizations."*, ACM SIGPLAN Notices, June 1984, Vol. 19, No. 6.

[GENESERETH 85] GENESERETH Michael R. and GINSBERG Matthew L. em Logic Programming, Communications of the ACM, September 1985 Volume 28 Number 9.

[HARMON 85] HARMON, P. & KING D., EXPERT SYSTEMS. John Wiley & S. 1985.

[HAYES-ROTH 85] HAYES-ROTH Frederick *Rule-based Systems*, Communications of the ACM, September 1985 Volume 28 Number 9.

[HAYES-ROTH 83] HAYES-ROTH Frederick & WATERMAN, D. A., *"An Overview of Expert Systems"*, in BUILDING EXPERT SYSTEMS, editores Hayes-Roth, E. et alii, 1983.

[HARADVALA 84] HARADVALA, S., KNOBE, B. and RUBIN, N., *"Expert Systems for High Quality Code Generation"*. Proceedings of the First IEEE Conference on AI Aplications, 1984, pp. 310-313.

[JACKSON 88] JACKSON Peter. INTRODUCTION TO EXPERT SYSTEMS. Addison-Wesley. 1988.

[LUCENA 87] LUCENA, C. J. P, INTELIGENCIA ARTIFICIAL E ENGENHARIA DE SOFTWARE, PUC - RJ. 1987.

[McCARTHY 65] McCARTHY, et al., LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA, 1965.

[RICH 83] RICH, Elaine, Artificial Intelligence, McGraw-Hill Book Company, N. Y., 1983.

[SHEIL 86] SHEIL, Beau, *"Power tools for programmers"*, in Readings in Artificial Intelligence and Software Engineering, Edited by RICH, Charles & WATERS, R. C., 1986.

[RUSSELL 85] RUSSELL, Stuart, *"The Compleat Guide to MRS"*, Stanford Knowledge Systems Laboratory, Stanford University Report no. KSL-85-12, June 1985.

[TAPPEL 82] TAPPEL, Steve, WESTFOLD Stephen & BAR Avron *"Programming Languages for AI Research"*, Departament of Computer Science, Stanford University Report no. STAN-CS-82-935, October 1982.

[WARFIELD 88] WARFIELD, Jay W. & BAUER, Henry R., *"An Expert System for a Retargetable Peephole Optimizer"*, ACM Sigplan Notices, Vol. 23, No. 10, 1988.

[WATERMAN 83] WATERMAN, D. A., & HAYES-ROTH, *"An Investigation of Tools for Building Expert System"*, in BUILDING EXPERT SYSTEMS, editores Hayes-Roth, E. et alii, 1983.