1. Publicação nº	2. Versão	3. Data	5. Distribuição
INPE-4291-PRE/1162		*Agosto 1987*	☐ Interna ☒ Externa ☐ Restrita

4. Origem	Programa
DPI	*ANIMA*

6. Palavras chaves - selecionadas pelo(s) autor(es)

INTELIGÊNCIA ARTIFICIAL
SISTEMAS ESPECIALISTAS
LINGUAGEM PARA SISTEMAS ESPECIALISTAS
SISTEMAS DE REGRAS
ALGORITMO RETE

7. C.D.U.: *681.3.019*

8. Título *INPE-4291-PRE/1162*

UMA IMPLEMENTAÇÃO DA LINGUAGEM OPS5 PARA COMPUTADORES COMPATÍVEIS COM O IBM-PC

10. Páginas: *10*

11. Última página: *09*

12. Revisada por

Nelson Mascarenhas

Nelson D. A. Mascarenhas

9. Autoria

Flávio Roberto Dias Velasco

Flávio Roberto Dias Velasco

Assinatura responsável

13. Autorizada por

Dr. Marco Antonio Raupp
Diretor Geral

14. Resumo/Notas

A linguagem OPS5 é uma linguagem de programação baseada em regras destinada a implementação de sistemas especialistas. As características mais importantes da linguagem OPS5 são: a eficiência (pelo uso do algoritmo Rete), a generalidade (se presta a várias aplicações) e mecanismos poderosos de casamento de padrões. Este trabalho descreve uma implementação da linguagem OPS5 para micro-computador (IBM-PC e compatíveis). É mostrada a estrutura interna da implementação e dada ênfase às decisões de projeto.

15. Observações *Trabalho submetido para apresentação no 4º Simpósio Brasileiro de Inteligência Artificial, de 13 a 16 de outubro de 1987, Uberlândia, MG*

UMA IMPLEMENTAÇÃO DA LINGUAGEM OPS5 PARA COMPUTADORES COMPATÍVEIS COM O IBM-PC

Flávio Roberto Dias Velasco
Ministério da Ciência e Tecnologia - MCT
Instituto de Pesquisas Espaciais - INPE
Departamento de Processamento de Imagens - DPI
Caixa Postal 515, S. J. Campos, S. Paulo

1. Introdução

A linguagem OPS5 é uma linhguagem de programação baseada em regras criada por Forgy (1981) destinada ao desenvolvimento de sistemas especialistas. OPS5 teve como precursoras a linguagem PSG, desenvolvida por Allen Newell, e as linguagens OPS e OPS4 desenvolvidas pelo próprio Forgy e por John McDermott (apud Hayes-Roth et al. 1983).

A característica mais importante da linguagem OPS5 é a eficiência de execução, conseguida através do uso do algoritmo Rete (Forgy 1983). O ciclo de execução de um sistema de regras de produções consiste, normalmente, de três fases:

1. casamento ("matching"): para todas as produções, seus lados esquerdos são comparados aos dados armazenados (fatos, conjecturas e objetivos) para determinação se o lado esquerdo é satisfeito por um conjunto de dados (Um lado esquerdo pode ser satisfeito por vários conjuntos de dados e um mesmo conjunto de dados pode satisfazer vários lados esquerdos.);

2. resolução de conflito: escolhe-se uma "instância", ou seja, uma produção e um conjunto de dados que satisfaça a produção; se não houver uma produção nessas condições, o ciclo pára;

3. disparo ("firing"): as ações do lado direito da produção escolhida são executadas para os dados da instância.

A cada ciclo os dados armazenados que compõem a memória de trabalho são alterados muito pouco. O algoritmo Rete deriva sua eficiência por não casar todos os dados a cada ciclo (só os elementos alterados são recomputados) e por compartilhar testes iguais mesmo entre regras diferentes.

Além da eficiência, a linguagem OPS5 tem a propriedade da generalidade, podendo ser utilizada em várias aplicações diferentes com diferentes estilos de programação. É relativamente fácil, por exemplo, implementar diferentes estratégias de solução de

problemas, mesmo dentro de um mesmo programa. Outra vantagem do OPS5 é o poderoso mecanismo de casamento de padrões ("pattern matching") que dispõe. OPS5 permite que padrões extremamente complexos sejam casados eficientemente (Hayes-Roth et al. 1983).

OPS5 tem sido usada com sucesso em inúmeras aplicações. A mais famosa dessas é o sistema especialista R1 para configuração de computadores VAX (McDermott 1980). OPS5 foi usada também para a análise de imagens (McKeown et al. 1985) que é a motivação principal para a implementação descrita neste trabalho.

A estrutura deste trabalho é a seguinte. A Secção 2 dá uma visão geral da linguagem OPS5. A rêde que é compilada a partir das produções e que é o cerne do algoritmo Rete é descrita na Secção 3. A estrutura do intérprete é apresentada na Secção 4, dando ênfase às decisões de projeto principalmente quando estas diferem da implementação sugerida por Forgy (1983). A metodologia utilizada na implementação, uma breve comparação com outras implementações e os planos futuros de continuidade do trabalho são discutidos na última secção.

2. Uma visão geral da linguagem OPS5

Um programa OPS5 é dividido em duas secções: a secção de declarações e a secção de produções. Na primeira são definidos os elementos de dados (ou simplesmente "elementos") que comporão a memória de trabalho e declaradas
as funções externas chamadas. A secção de produções contém as regras que são compostas de nome, lado esquerdo (lista de condições) e lado direito (lista de ações).

A Figura 2.1 mostra um programa simples em OPS5 extraído de Brownston et al. (1985). O programa acha num banco de nomes de pessoas e suas filiações quais são os ancestrais de para uma pessoa cujo nome é fornecido pelo usuário do programa.

No programa são definidos três tipos de dados diferentes: "Person" com três atributos, "Request" com um atributo e "Start", sem atributo algum. Os dados usados no programa devem ser necessariamente de um desses três tipos. É possível definir um elemento de memória sem fornecer todos os seus atributos: os atributos não fornecidos têm valor por falta ("default") igual à "nil".

```
    declaracoes

(literalize Person name father mother)

(literalize Request target)

(literalize Start)

;    producoes

(p FindAncestors::Initialize
     ( ( Start ) (initialize) )
-->
     (remove (initialize))
     (write (crlf) |Please type the first name of a person|
          (crlf) |whose ancestors you would like to find:| (crlf))
     (make Request ^target (accept)))

(p FindAncestors
     (Request ^target ((name) () nil) )
     (Person ^name (name) ^mother (mother-name) ^father (father-name))
-->
     (make Request ^target (mother-name))
     (make Request ^target (father-name) ))

(p FindAncestors::Print
     ( (Request ^target ( (name) () nil ) ) (request1) )
-->
     (write (crlf) (name) is an ancestor )
     (remove (request1) ))
```

Fig. 2.1. Um programa simples em OPS5

O programa da Figura 2.1 tem três produções. O átomo "p" serve para identificar que se trata de uma produção. O lado esquerdo é separado do direito pelo átomo "-->". Todo o conjunto é ladeado por um par de parênteses.

A primeira produção, FindAncestors::Initialize, dá início à execução do programa. Seu lado esquerdo é satisfeito caso o dado "(Start)" esteja presente na memória. As ações executadas são a remoção deste elemento, o envio de mensagem pedindo ao usuário que entre com o nome de uma pessoa e a criação de um dado do tipo "Request" cujo valor do atributo "target" é o nome fornecido. O operador " é aposto ao nome do atributo para caracterizá-lo como tal.

A segunda produção, FindAncestors, procura por um dado do tipo "Request" com objetivo ("target") definido (1a. condição), tal que o objetivo conste do banco de dados (2a. condição). O disparo desta regra faz que duas novas requisições sejam adicionadas: uma correspondente à mãe (1a. ação) e outra ao pai (2a. ação).

A terceira e última produção imprime os ancestrais (por definição uma pessoa é ancestral de si mesma) e remove as requisições.

As regras 1 e 3 removem objetivos (requisições) da memória de trabalho. Não é preciso remover dados da memória para impedir que uma produção dispare uma segunda vez com os mesmos dados. Um mecanismo embutido no OPS5 ("refraction") impede naturalmente que isto aconteça.

A escolha de qual produção executar pode ser feita de duas maneiras: ou através da regra "LEX" ou através da regra "MEA". A regra LEX ordena as instâncias

(produção + dados) de acordo com os seguintes critérios: 1) antiguidade dos dados - é dada preferência aos dados mais recentes; 2) especificidade das produções - produções com mais testes do lado esquerdo são executados antes. Por exemplo, no programa da Fig. 2.1, sempre que a segunda produção é satisfeita a terceira também o é (o conjunto de condições da regra 3 é subconjunto do conjunto de condições da regra 2). Contudo, a segunda regra é sempre escolhida antes da terceira por ser mais específica.

A regra MEA ("means-ends analysis") verifica primeiro a antiguidade do dado que satisfaz à primeira condição. Após este passo a regra é semelhante à regra LEX. O uso da regra MEA faz sentido quando a primeira condição testa a existência de um dado objetivo ("goal"). Neste caso os objetivos mais recentes são perseguidos antes, independentemente da antiguidade dos dados e especificidade das produções.

Além das condições mostradas no programa da Fig. 2.1, é possível ter em OPS5 condições negativas, que são precedidas por um sinal de menos ("-"). A função da condição negativa é inibir o disparo da regra caso haja pelo menos um dado que a satisfaça na memória de trabalho.

Há dois tipos de variáveis em OPS5: variáveis associadas a condições e variáveis a associadas a valores. Os dois tipos são representados por átomos da mesma forma: um sinal de menor, uma cadeia de caracteres e um sinal de maior. No exemplo da Fig. 2.1, <initialize> e <request1> são variáveis do primeiro tipo e permitem que ações (como "remove" e "modify") façam referência aos dados que satisfazem as condições. As variáveis <name>, <mother-name> e <father-name> são ligadas ("bound") aos valores dos atributos dos elementos que casam as condições onde elas aparecem. Variáveis deste tipo podem ser usadas no lado esquerdo para restringir valores numa mesma condição ou em condições diferentes.

A linguagem OPS5 dispõe de uma série de facilidades para desenvolvimento que podem ser usadas interativamente. É possível, por exemplo, examinar a qualquer instante o conteúdo do conjunto de conflito ("conflict set") que contém as produções que são satisfeitas e os dados correspondentes. É possível, também, executar o programa por um número de ciclos dado. A memória de trabalho pode ser examinada em sua totalidade ou só os elementos que satisfazem uma condição fornecida. Pontos de quebra podem ser inseridos que param a execução quando uma regra é escolhida; pode-se adicionar e remover elementos da memória e verificar que elementos satisfazem as condições de uma dada produção (individualmente ou em conjunto).

3. A rêde de produções

As produções de um programa OPS5 ao serem carregadas são compiladas e gerada uma rêde a partir dos lados esquerdos dessas. A rêde é um grafo orientado, acíclico, com um único nó maximal (a raiz) e vários nós minimais, um para cada produção. A rêde gerada correspondente ao programa da Fig. 2.1 é mostrado na Fig. 3.1.

```
NETWORK:

    1: ROOT
    5: FORK 8
    2: TCLS Start
    3: TMNL FindAncestors::Initialize

    8: FORK 7
    4: TCLS Request
    6: TCT8 1 nil
   14: FORK 13
    9: JOIN 1
   11: TER7 1 1 1
   12: TMNL FindAncestors

   13: NOOP
   15: TMNL FindAncestors::Print

    7: TCLS Person
   10: MERG 9
```

Figura 3.1. Rêde de produções

Na Fig. 3.1 cada linha corresponde a um nó e cada nó está associado a um número. Em geral, cada nó tem um único sucessor que é o nó da linha seguinte (à excessão do nó "MERG", cujo sucessor é dado explicitamente). Nós terminais (TMNL) não têm sucessores e nós forquilha (FORK) têm dois sucessores (o da linha seguinte e o dado explicitamente).

Os outros nós que aparecem na rêde da Fig. 3.1 são: o nó raiz (ROOT) que não tem antecessor, o nó que testa a classe (tipo) do elemento (TCLS), o nó que compara o valor de um atributo com uma constante (TCTi, onde i indica a comparação que é feita: 7 significa igualdade, 8 significa desigualdade, etc.), o nó com duas entradas (JOIN) que serve para fazer a conjunção de condições positivas, o nó que compara valores de atributos de duas condições diferentes (TERi) e, finalmente, o nó "NOOP" ("no operation") criado para facilitar a geração da rêde. A numeração dos nós, a propósito, dá a ordem em que os mesmos foram gerados.

Os nós do tipo TCTi têm dois parâmetros. O primeiro fornece o índice do atributo e o segundo a constante a ser comparada. Os argumentos dos nós do tipo TERi dão a condição (positiva) na qual a variável

ocore pela primeira vez, o índice do atributo desta ocorrência e, por fim, o índice do atributo da condição que está sendo juntada. Nós do tipo JOIN têm um único argumento que dá o número de nós TERi que vêm adiante na rêde.

Há três outros tipos de nós que não aparecem na Fig. 3.1: o nó para conjunção de condições negativas (NJON), o nó usado para implementação de disjunções (OFRK) e o nó para teste de variáveis que aparecem mais de uma vez numa mesma condição (TRAi).

A rêde da Fig. 3.1 mostra como diferentes condições compartilham testes. É o caso das produções FindAncestors e FindAncestors::Print, que compartilham os testes "TCLS Request" e "TCT8 1 nil".

Sempre que um dado é criado ou removido, a rêde é percorrida a partir da raiz. Modificações de dados são tratadas como uma remoção seguida de uma adição. A rêde é percorrida e os dados passados adiante quando os testes são satisfeitos. Os nós FORK indicam bifurcações cujos dois caminhos devem ser seguidos. Os nós JOIN possuem memórias locais que armazenam os elementos ou conjuntos de elementos que chegaram até o nó. A memória esquerda armazena os conjunto de elementos ("tokens") adicionados à memória que satisfizeram todas as condições positivas que chegam diretamente ("pela esquerda") ao nó JOIN; a memória direita armazena os elementos que chegam através um nó do tipo MERG ("pela direita"). São passados adiante (como novos tokens) os conjuntos de tokens e elementos da memória direita que passam todos os testes TERi associados ao nó JOIN. Quando um token atinge um nó terminal, uma instância é criada e adicionada ou retirada do conjunto de conflito conforme o caso.

4. Estrutura do intérprete

Os módulos principais do intérprete para o OPS5 são o analisador léxico, o analisador sintático, o executor de comandos de alto nível, o construtor da rêde de produções, o intérprete da rêde e o executor de ações dos lados direitos.

O analisador léxico é responsável pelas tarefas que normalmente são atribuição de analisadores léxicos: ignorar espaços, linhas em branco, comentários, reconhecer átomos que têm funções especiais, etc. Os átomos simbólicos encontrados são armazenados numa tabela (acesso "hash") e depois referenciados pelo seus índices na tabela. Isto permite que testes de igualdade entre átomos sejam feitos de modo eficiente, só pela comparação dos índices.

O análise sintática dos comandos é feita por um analisador sintático recursivo descendente. Este gera

uma forma interna (árvore binária) equivalente ao comando. A Fig. 4.1 mostra a árvore gerada para a segunda produção do exemplo da Fig. 2.1. De um modo geral, a subárvore esquerda (left) de um nó define-o com mais detalhes e a subárvore direita (rght) representa um item sintático de mesmo nível de abstração. No lado esquerdo, o sinal "+" indica que se trata de uma condição positiva; no lado direito, o número 30 é o código da ação "make".

```
lado esquerdo:

+
left:Request
     left:target
          left:NULL
               left:=
                    left:(name)
                    rght:()
                         left:nil
rght:+
     left:Person
          left:name
               left:NULL
                    left:=
                         left:(name)
               rght:mother
                    left:NULL
                         left:=
                              left:(mother-name)
                    rght:father
                         left:NULL
                              left:=
                                   left:(father-name)

lado direito:

30
left:Request
     left:target
          left:(mother-name)
rght:30
     left:Request
          left:target
               left:(father-name)
```

Figura 4.1. Forma interna gerada

O analisador sintático é responsável pela coleta de informações sobre variáveis, classes e atributos. A atribuição de índices aos atributos é feita ao ser encontrada a primeira produção, após a definição de todas as classes, já que atributos de mesmo nome de classes diferentes compartilham os mesmos índices.

A árvore gerada pelo analisador sintático é passada ao construtor da rêde se o comando for uma produção e ao executor de comandos caso contrário. O executor de comandos interpreta diretamente os comandos enquanto o construtor da rêde transforma os lados esquerdos das produções na rêde. Ao fim de cada produção tem-se uma rêde que implementa corretamente as produções encontradas até então. A adição de uma nova produção é feita pelo percorrimento da rêde para cada condição. Uma nova derivação é criada (via FORK)

se nenhum dos caminhos implementa a condição ou um nó tipo JOIN, NJON, TMNL ou OFRK é encontrado.

O intérprete da rêde, que implementa o algoritmo Rete, adiciona ou retira elementos da memória de trabalho. Os elementos são "bombeados" na rêde e os nós são visitados enquanto os testes do tipo TCTi e TRAi derem resultados verdadeiros. Ao fim destes testes, necessariamente teremos um JOIN (ou NJON) diretamente ou através um nó MERG ou, então, um nó TMNL. Neste ponto o elemento é transformado num "token" (de um único elemento), armazenado na memória local do JOIN e, se, junto com outro token da memória oposta, passar em todos os testes do JOIN, segue adiante na rêde. Quando um nó TMNL é encontrado, o token é transformado numa instância e adicionado ou subtraído do conjunto de conflito. Este, o conjunto de conflito, é mantido constantemente ordenado segundo a regra em vigor (LEX ou MEA).

O executor de ações é semelhante ao executor de commandos na medida que interpreta diretamente a árvore gerada pelo analisador sintático.

5. Conclusões

Intérpretes para a linguagem OPS5 são comummente escritos em LISP (MACLISP, FRANZLISP etc.) que são, por sua vez interpretados. Embora com isto se ganhe todo o ambiente de programação LISP, a eficiência é consideravelmente reduzida. Um outro enfoque de implementação é o utilizado pela DEC na implementação da linguagem OPS5 em computadores VAX (Digital Equipment Corporation, 1983), onde foi usada a linguagem BLISS. Esta última implementação tem um ganho em eficiência considerável em relaçào às implementações LISP. Enquanto, tipicamente, num VAX 11/780, o OPS5/LISP dispara 1 a 5 regras por segundo, o OPS5/BLISS dispara 5 a 12 regras por segundo (Hillyer e Shaw, 1986).

A linguagem escolhida para a implementação foi a linguagem C por permitir uma implementação eficiente e por prever recursos, como tipos estruturados, ponteiros e alocação dinâmica de memória, que permitem a implementação de tipos abstratos de dados. Para a implementação do intérprete foram definidos 14 tipos abstratos: nomes (átomos simbólicos), constantes (átomos ou números), listas, árvores, classes, atributos, variáveis, produções, índices, elementos, tokens, conjunto de tokens, nós e instâncias. Foi definido também um encapsulamento de dados que implementa a abstração de conjunto de conflito.

A aplicação pretendida para o intérprete é o seu uso na construção de sistemas especialistas para análise de imagens de satélite (recursos naturais e

meteorologia). Para isto facilidades adicionais serão implementadas para integração do OPS5 com o SITIM (sistema para processamento de imagens, Souza et al., 1986) e com o SGI (sistema geográfico de informações, Erthal et al., 1986). Além dessas interfaces, os planos futuros para o OPS5 incluem a criação de um ambiente de programação com facilidades para a criação de interfaces com o usuário e de sistemas de explanação.

REFERÊNCIAS BIBLIOGRÁFICAS

BROWNSTON, L.; FARRELL, R.; KANT, E.; MARTIN, N., "Programming expert systems in OPS5", Addison-Wesley, Reading, Massachusetts, 1985.

DIGITAL EQUIPMENT CORPORATION, "OPS5 for VAX: User's Guide", Digital Equipment Corporation, Maynard, Massachusetts, março 1984.

ERTHAL, G.; CÂMARA, G.; OLIVEIRA, M. O. B.; FELGUEIRAS, C.; PAIVA, J. A. C., "O banco de dados geográficos do INPE", 1o. Simpósio Brasileiro de Banco de Dados, Rio de Janeiro, 1986.

FORGY, C. L., "OPS5 User's Manual", relatório técnico CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, julho 1981.

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

HAYES-ROTH, F.; WATERMAN, D.A.; LENAT, D. B., "Building expert systems", Addison-Wesley, Reading, Massachusetts, 1983.

HILLYER, B. K.; SHAW, D. E., "Execution of OPS5 production systems on a massively parallel machine", Journal of Parallel and Distributed Computing", vol. 3, pp. 236-268, 1986.

MCDERMOTT, J., "R1: an expert in the computer systems domain", Proceedings of the 1st. National Conference of the American Association for Artificial Intelligence, pp. 269-271, 1980.

MCKEOWN Jr., D. M.; HARVEY, Jr., W. A.; MCDERMOTT, J., "Rule-based interpretation of aerial imagery", IEEE Transactions on Pattern Analysis and Machine Intelligence", vol. PAMI-7(5), pp. 570-585, setembro 1985.

SOUZA, R. CC. M.; MENDES, C. L.; GARRIDO, J. P.; CÂMARA, G., "Evolução da família de sistemas de tratamento de imagens do INPE", IV Simpósio Brasileiro de Sensoriamento Remoto, Gramado, agosto de 1986.