Prolog é um paradigma de programação desenvolvido em 1970 que é inteiramente baseado em cláusulas de primeira ordem. Embora, tais cláusulas comumente são compostas pelos operadores AND(^) e OR(v), usando a forma normal delas somos capazes de obter as cláusulas de Horn, cláusulas que usam apenas operadores AND. Para fazer a conversão é muito simples pois observe nos exemplos a seguir:

- C_1 ou C_2 => A é o mesmo que: C_1 => A e C_2 => A.
- C_1 => A ou B é o mesmo que C_1 ^ ¬(B) => A e C_1 ^ ¬(A) => B.
- C_1 => A ^ B é o mesmo que C_1 => A e C_1 => B.

Assim nós podemos fazer conversões em quaisquer conjuntos de cláusulas de primeira ordem.

Como toda linguagem, há uma sintaxe pré determinada que deve ser obedecida para que ocorra a comunicação, possibilitando a compreensão do destinatário acerca da mensagem enviada pelo remetente. Em Prolog, a sintaxe básica vista em aula consiste em cláusulas e perguntas, de tal modo que cada cláusula possui um objetivo. Os objetivos são compostos por predicados e expressões, tal que este último é aquele que possibilita a inserção de variáveis, números e funções. Existem também alguns predicados primitivos que não são cálculos e realizam as suas operações imediatamente, como o *print, plus, minus* e *less.*

Após a determinação de uma sintaxe básica, partimos agora para a interpretação lógica do funcionamento de um interpretador Prolog: Em suma, a função do interpretador é provar objetivos e para isso faz uso de um mecanismo de escolha de valores usando as regras de inferência definidas pelas cláusulas, isto é, em (infer? g) é necessário achar valores para todas as variáveis de g tais que é possível provar g utilizando as cláusulas existentes.

Esse “mecanismo” utilizado é um algoritmo de resolução de cláusulas e funciona da seguinte forma: Em uma entrada do tipo (infer? g) primeiramente é buscado uma cláusula do tipo (infer g from H_1 H_2 … H_n) e caso não encontre retorna ‘*not satisfied’.* Agora, se a cláusula for encontrada, então são feitas tentativas de resolver cada um dos H_i pela ordem que foram inseridos, no nosso caso, em ordem crescente partindo de 1 à n, e fazemos o mesmo procedimento. Veja o seguinte exemplo:

1 -> (infer G from H_1 H_2)
2 -> (infer H_1 from H_2)
3 -> (infer H_2)
4 -> (infer? G)
‘Satisfied’

Note que em 4 ocorre a pergunta e então busca-se por 1 e então tenta-se provar H_1 encontrando 2, que acarreta na tentativa de prova H_2, que está em 3. Logo após, tenta-se provar H_2, então busca-se por 3 e, assim, a saída retorna *‘Satisfied’,* mas observe que se a linha 1 fosse escrita da seguinte maneira (infer G from H_2 H_1), observe que os passos da explicação acima estariam invertidos. Isso denota um comportamento linear do Prolog que pode causar loops infinitos por causa da definição circular de cláusulas, ou seja, a visão procedural faz parte da definição da linguagem.

Veja a seguir um exemplo que contém um loop infinito:

1 -> (infer A from B C D)
2 -> (infer B from C)
3 -> (infer C from B)
4 -> (infer D)
5 ->(infer? A)

Seguindo os passos do interpretador, usamos a cláusula 1 e obtemos o objetivo B, em B encontramos a cláusula 2 e obtemos o objetivo C, em C encontramos a cláusula 3 e obtemos o objetivo B, em B encontramos a cláusula 2 e obtemos o objetivo C, em C encontramos a cláusula 3 e obtemos o objetivo B, … . Por isso a ordem das cláusulas é importante.

É evidente que na visão procedural, ao utilizarmos as definições de substituição e definição dadas em aulas, temos que para satisfazer um objetivo g, verificamos as cláusulas C_i em ordem, tal que se acharmos uma cláusula C_k = (infer g' from h_1 .. h_n onde g unifica com g' através da substituição S0, tentamos satisfazer S0(h_1) e se falharmos, continuamos com C_(k+1), mas caso fosse satisfeito, prosseguiríamos com a substituição S1 em S0(S1(h_2)) e assim por diante até (S0(S1(… Sn-1(h_n) …). Caso isso ocorra, dizemos que g é bem sucedido com a substituição S = S0<>S1<>...<>Sn-2<>Sn-1.

Fazendo uso das informações desenvolvidas acima, vamos mostrar a implementação de operações de listas e conjuntos:

Listas

Member

->(infer (member X (cons X M)))

->(infer (member X (cons Y M)) from (member X M))

Verifica se X é igual (cons X M), tal que se ocorrer X e (cons Y M), então ignoramos o valor de Y e verificamos com o resto da lista.

Addtoend

->(infer (addtoend X nil (cons X nil)))

->(infer (addtoend X (cons Y M) (cons Y N)) from (addtoend X M N)

O caso trivial trata da adição de um valor à uma lista nula que é (cons X nil); o segundo caso trata da adição em uma lista não nula, então o resultado é dado por (cons Y N), tal que N é o (addtoend X M N), ou seja, é o resto da lista com o X já inserido no final.

Reverse

->(infer (reverse nil nil))

->(infer (reverse (cons X L) M) from (reverse L N) (addtoend X N M)

O caso base define que a reversão de uma lista nula é a própria lista; e o segundo define que a reversão da lista dada por (cons X L) é a reversão da lista L, que é dada por M, e então basta que X seja adicionado ao final da lista M.

Append

->(infer (append nil L L))

->(infer (append (cons X L) M (cons X N)) from (append L M N))

Primeiramente, juntar algo nulo com uma lista não nula é a própria lista; e segundamente, dado que a lista L começa com um valor, temos (cons X L) e um valor M a ser inserido, então recursivamente acessamos todos os elementos de L até que se chegue ao final para inserir M em N, enquanto que os demais valores anteriores foram armazenados em N.

Conjuntos

União

-> (infer (uniao X nil X))

->(infer (uniao X (cons Y M) (cons Y N)) from (uniao X M N))

->(infer (uniao nil X X))

->(infer (união (cons X M) Y (cons X N)) from (união M Y N))

Definimos primeiramente o caso base da recursão; Em seguida fazemos que a união de X com (cons Y M) é o (cons Y N), tal que N é a união de X com o que sobrou de M; As duas últimas servem para ajudar a reconhecer quando a ordem do conjunto não condiz com a cláusula.

Intersecção

->(infer (inter X nil nil))

->(infer (inter X (cons Y M) (cons Y N)) from (member Y X) (inter X M N))

->(infer (inter X (cons Y M) N) from (inter X M N))

Definimos o caso base da recursão dizendo que X faz intersecção com ele mesmo; A segunda diz que Y está na intersecção de X com Y se Y é membro de X e verifica para o resto da lista; a terceira serve para o caso em que Y não é membro de X, então ele é ignorado e continua-se a verificação com os demais elementos da lista

Remoção

->(infer (remove X nil nil))

->(infer (remove X (cons X L) M) from (remove X L M))

-> (infer (remove X (cons Y L) (cons Y M)) from (remove X L M))

O caso trivial define que a remoção de X em uma lista nula é uma lista nula; o segundo define que a remoção de X de (cons X L) é M, que a remoção de X em L; o terceiro caso trata para o caso em que X não é o primeiro elemento da lista, então ocorre uma unificação, dado que X é diferente de Y e, então, remove-se X de L retornando em M.

O mundo dos blocos consiste na determinação de um domínio dado por uma mesa e três blocos a serem empilhados. Assim, dado um posicionamento dos blocos, queremos planejar seus movimentos para atingir outro estado pré-definido. Para isso, a modelagem exige mais sofisticação quanto às cláusulas: *block* e *different* servem para distinguir elementos; *(clear X State)* é verdadeiro se não há nada em cima de C; *(on X Y State)* retorna verdadeiro de X está em cima de Y no estado; *update* define como o movimento de um bloco muda o estado; *(move X Y Z)* indica que o bloco X deve ser movido de Y para Z; *legalMove* define quando um movimento pode ser executado num estado.
Então, vamos mostrar algumas das implementações usadas no estudo de caso do mundo dos blocos:

(infer (block a))

Define um bloco.

(infer (different a b))

Verifica se são blocos diferentes.

(infer (different X table) from (block X))

Verifica se X é diferente da mesa.

(infer (update (move X Y Z)(cons (on X Y) S) (cons (on X Z) S)))
(infer (update (move X Y Z)(cons (on U V) S1)(cons (on U V) S2)) from (different X U) (update (move X Y Z) S1 S2))

Faz a mudança.

(infer (clear X nil))
(infer (clear X (cons (on B Y) State) from (different X Y) (clear X State))

Verifica se não há bloco em cima de X ou X não está empilhado em algum lugar.

(infer (on X Y State) from (member (on X Y) State))

Verifica se X está em cima de Y se (on X Y) for membro da lista State.

(infer (legal-move (move B P1 table) State) from (on B P1 State) (different P1 table) (clear B State))

Verifica se é possível fazer um movimento através do asseguramento de certas condições, por exemplo: Para mover B de P1 para a mesa é necessário que: B esteja em cima de P1; P1 seja diferente da mesa; Não tenha nada em cima de B.

(infer (legal-move (move B1 P B2) State) from (block B2) (on B1 P State) (different P B2) (different B1 B2) (clear B1 State) (clear B2 State))

Verifica se é possível fazer um movimento através do asseguramento de certas condições, por exemplo: Para mover B1 de P para B2 é necessário que: B2 seja um bloco; B1 esteja em cima de P; P seja diferente de B2; B1 seja diferente de B2; Não tenha ninguém em cima de B1; Não tenha ninguém em cima de B2.

(infer (transform State1 State2 Plan) from (transform State1 State2 (cons State1 nil) Plan) (infer (transform State State Visited nil)) (infer (transform State1 State2 Visited (cons

Move Moves)) from (legal-move Move State1) (update Move State1 StateInt) (non-member StateInt Visited) (transform StateInt State2 (cons StateInt Visited) Moves))

Transform é o predicado principal porque retorna verdadeiro se a sequência de movimentos no plano nos leva do Estado1 para o Estado2. Para fazer essa verificação é necessário definir que: Um estado é capaz de chegar nele mesmo com uma lista nula de movimentos; Fazemos a tentativa de um movimento inicial qualquer entre todos aqueles que são possíveis; Realizamos o movimento se ele estiver correto; Verifica se o estado já foi visitado, tal que se for repetido o algoritmo faz um backtracking para escolher outro movimento disponível; Se o passo anterior deu certo, então ele é incluído na lista de estados e refazemos os passos até que StateInt e State2 sejam iguais.

*Note que o algoritmo descrito faz uso da força bruta para encontrar uma resposta, então ele exige grande poder computacional. Em aula vimos a versão 2, que se utiliza de uma heurística que sugere movimentos legais. No entanto, não vou descrevê-la para poupar que este texto fique grande demais.

Existe também a possibilidade de fazer multiplicações se supormos a soma e subtração definindo algumas propriedades: Todo número multiplicado por 0 resulta em 0; A multiplicação 'X*Y' é obtida através da soma de 'Y' X vezes. Veja abaixo o código da implementação:

(infer (mult 0 Y 0))

(infer (mult X Y R) from (minus X 1 X-1) (mult X-1 Y R-INT) (plus R-INT Y R)).

Usando a mesma estratégia podemos fazer o fatorial em prolog ao definirmos algumas propriedades: O fatorial de 0 é 1; O fatorial de N é dado pela multiplicação de todos os sucessores menores ou iguais à N que pertencem aos inteiros positivos. Assim:

(infer (fat 0 1))

(infer (fat N R) from minus (N 1 N-1) (fat N-1 R-int)(mult N R-int R)

*Note que essas duas implementações não funcionam com variáveis em todas as posições porque fazem uso de predicados primitivos estritamente definidos.

Além disso, podemos fazer algoritmos de ordenação. Abaixo serão descritos dois códigos que implementam uma versão usando a força bruta e outra que usa o método do quicksort:

(infer (naive-sort L M) from (permutation (L M)) (ordered M))

(infer (permutation nil nil))

(infer (permutation L (cons H T)) from (append V (cons H U) L) (append V U W) (permutation W T))

(infer (ordered nil))

(infer (ordered (cons A nil)))

(infer (ordered (cons A (cons B Rest))) from (<= A B) (ordered (cons B rest))) (infer (<= X X)) (infer (<= X Y) from (less X Y))

Este algoritmo constrói todas as permutações de uma dada lista ao escolher arbitrariamente valores para preencherem V e U em L, removendo-os e retornando W, que passa por uma verificação para definir se o resto é uma permutação recursivamente e em seguida verifica se está ordenada recursivamente.

(infer (quicksort nil nil))

(infer (quicksort (const H T) S) from (partition H T T1 T2) (quicksort T1 T1Ord) (quicksort T2 T2Ord) (append T1Ord (cons H T2Ord) S)

O quicksort pode ser escrito de forma relativamente fácil porque é muito semelhante à própria definição do método. Note que o primeiro passo foi

definir a base da recursão; o segundo determina que o pivô é o primeiro elemento da lista e particiona a lista em duas partes menores ou iguais e maiores que o pivô; o terceiro passo consiste na ordenação da partição T1; o quarto passo consiste na ordenação da partição T2; e o quinto passo consiste na união das duas partições.

(infer (partition H (cons A X) (Cons A Y) Z) from (<= A H) (partition H X Y Z))
(infer (partition (cons A X) Y (cons A Z) from (less H A) (partition H X Y Z))
(infer (partition H nil nil nil))
As partições foram definidas de tal modo que são formadas duas listas de <= e > em relação ao pivô e, para isso, é feita uma recursão em todos os elementos da lista comparando-os com o pivô e, em seguida, após a determinação em qual das partições o elemento entra, é feita uma nova chamada recursiva nas partições formadas até que cheguemos ao caso base.

Na última aula foi dito que o Prolog utilizado em aula havia algumas alterações para facilitar a compreensão e aplicabilidade, então foi reservado alguns minutos para explicar como ele é de verdade: Em suma, o formato das cláusulas é diferente, o que impacta em uma sintaxe distinta:

(infer G) => G
(infer G from A B … Z) => G :- A B … Z
(p x1 … xn) ➔ p(x1, …..xn)
(plus x y z) ➔ z is x + y.
(cons x1 (cons x2 ….(cons xn nil)…) ➔ [x1, x2,...,xn]
(cons x l) => [x | l]

A semântica também é diferente porque as operações primitivas englobam aritmética, comparações, negações, condicionais e cut/fail sem o uso dos parênteses, como visto nos exemplos acima. O cut and fail são utilizados como recursos adicionais para impedir a continuidade do backtracking caso uma determinada condição seja encontrada. Veja:

not-equal(X,Y) :- equal(X,Y), ! , fail
not-equal(X,Y).
equal(X,X)

Note que a implementação do not-equal consiste na verificação de que os valores de X e Y são iguais e, caso seja verdadeiro, ocorre um cut seguido de fail, enquanto que se for falso o backtracking nos redireciona para a próxima cláusula, retornando verdadeiro.