Пролог

[PROLOG](#h.bsrl2549na13)

[Списки](#h.33k1d5knb7l0)

[Примеры](#h.gd3ulh2w53te)

[Математические вычисления](#h.49b1i9c1234q)

[Управление перебором](#h.z7asi8yaaaj5)

[Отрицание при достижении цели](#h.7ugapc37jyqh)

[Задача поиска в пространстве состояний](#h.2bnzn9meemjq)

[Представление задачи на Prolog](#h.jm99nsgu2hbd)

[Алгоритмы решения](#h.y1nb5hwjw87)

[Поиск в глубину](#h.opwtm75etxbm)

[Поиск в ширину](#h.2bc67smr6cdt)

[Другие варианты](#h.uoosv1yn41bv)

[Реализация экспертных систем на Prolog](#h.rgipmgacsyj9)

[Системы, основанные на правилах](#h.feesk5xstgco)

[Системы, основанные на семантических сетях](#h.m1w9gv5lr2op)

[Системы, основанные на фреймах](#h.3pnbz4ih7msd)

# PROLOG

Это был 1970 год. Мы писали ИИ как могли.

* Символьные вычисления (легко создаются типы данных) Данных мало, типов много.
* Ориентирован на задачи доказательства
* Пролог - это интерпретатор. Даже типы входных данных могут быть разные
* Взаимодействие по принципу вопрос-ответ

Что мы можем делать с интерпретатором:

1. Сообщать факты
2. Сообщать правила вывода
3. Формулировать вопросы (является ли факт истинным, и если да, то при каких условиях)

Основные понятия пролога:

1. Отношение - базовое понятие пролога

parent(bob, tom) , parent - имя отношения, добавили информацию в базу знаний

Могут быть отношения без аргументов is\_spring

1. ?-parent(bob,tom) - задать вопрос “Является ли отношением?”

?-parent(bob, X) - задать вопрос с переменной Х, ответ например, Х=tom

цель, которую нужно доказать

?-parent(X,Y) - ответ набор пар значений X, Y

Встроена возможность полного перебора.

1. Формулируем правило

Голова:-цель1, цель2... цельN..

granparent(X,Y) :- parent(X,Z), parent(Z,Y)

Рекурсивное правило:

offspring(X,Y):-parent(Y,X)

offspring(X,Y):-parent(Y,Z),offspring(X,Z)

Все переменные с большой буквы или подчеркивания, данные - с маленькой, анонимная переменная - знак подчеркивания (важно, чтобы значение существовало, но использовать его не будем). Вместо подчеркивания интерпретатор подставит произвольное, не используемое имя. Например:

has\_child(X):-parent(X,\_)

4. Структура - записывается так же как отношение

birthday(john, date(1,apr,1970) )

К структурам напрямую обращаться нельзя. Структура называется функтор.

?-birthday(john, date(D,M,Y)) - при каких D, M, Y значение истинно?

?-birthday(john, X)- X = date(1, apr, 1970).

Операция согласования.

?-birthday(A,B) - ищем такие факты А и В из базы знаний, которые согласуются. Результатом согласования является ответ на вопрос “согласуются или нет два аргумента”, и если согласуются, то при каких значениях переменных

Правила согласования:

1. Два аргумента согласуются, если они являются константами и равны
2. Два аргумента согласуются, если они являются переменными. Переменные связываются между собой
3. Два аргумента согласуются, если S - переменная, Т - константа, при этом S конкретизируется значением Т. То же самое, если Т - уже конкретизированная переменная

?-parent(X,Y), parent(Y,Z)

1. Два аргумента согласуются, если они являются структурами с одним и тем же функтором, с одним и тем же количеством компонент и попарно согласуются соответствующие компоненты

?-birthday(P, date(\_,\_,1970)) - выведет имена всех 1970 года рождения

?-birthday(X, D), D=date(1,jan,Y) - кто родился 1 января

Оператор сопоставления:

(выражение),Y=A

Используется при доказательстве чего-нибудь и подборе нужных пар.

vertical(seq(point(X1,Y1),point(X2,Y2))):-X1=X2

<=>

vertical(seq(point(X,Y1),point(X,Y2)))

Декларативное значение программы.

?-цель(A,B,C,D)

Цель истинна, если существует необходимое предложение программы (определение факта или правила с необходимой конкретизацией переменных). При этом, если предложение является правилом, то истинна каждая цель, указанная в хвосте правила.

**Общий алгоритм поиска ответов**

На вход сама программа, как набор предложений и список целей для доказательства G1 … GN

Если набор целей пуст, то ответ истинен.

Если нет, то начинаем искать по базе знаний согласования с G1. Ищутся факты или правила, у которых голова согласуются с целью G1. Сработало какое-то правило, оно имеет вид H:-B1..Bm. Если это факт, а не правило, то m=0. При согласовании H c G1 происходит конкретизация значений переменных. Эта конкретизация объединяется со входной конкретизацией. Список целей заменяется на B1..Bm. Запустили задачу рекурсивно.

При выходе из рекурсии переходим к следующему предложению программы.

Процедурное значение:

1. Важен порядок фактов и правил.
2. Важен порядок целей в предложении.

Примеры программ, проверяющих, является ли Х предком Y

Способы 1 и 2 работают

Способ 3 не отвечает на отрицательные ответы (циклится)

Способ 4 циклится всегда

Сначала факты, потом правила. В правилах должны сначала проверяться простейшие цели, потом более сложные.

Списки

Варианты записи:

* .(a, .(b, .(c, [])))
* [a, b, c]
* [a | [b, c] ]
* [a, b | [c] ]
* [a, b, c | [] ]
* [a,b,c|Tail]

## Примеры

**conc(A,B,L)** % Является ли список L конкатенацией списков A+B

conc([], L, L).

conc([X|L1],L2,[X|L3]) :- conc(L1, L2, L3).

**member(X, L)** (v1.0)

member(X, [X|\_]).

member(X, [\_|L]) :- member(X, L).

**member(X, L)** (v2.0)

member(X,L) :- conc(\_, [x|\_], L).

member(X,Y,L) :- conc(\_, [X,Y|\_], L).

**before(X, L1, L)** % L1 - список, идущий до элемента Х в списке L?

before(X, L1, L) :- conc(L1, [x|\_], L).

**add\_head(X, L, L1)***(является ли L1 списком, получающимся в результате добавления X в начало списка L)*

add\_head(X, L, [X|L]).

**add\_tail(X, L, L1)**

add\_tail(X, [], [X]).

add\_tail(X, [H|L1], [H|L2]) :- add\_tail(X,L1,L2).

add\_tail(c,[a,b],L) L=[a,L1]

add\_tail(c,[b],L1)

add\_tail(c,[],L2) L1=[b,L2]

**del(X, L, L2)** % при удалении X из L получается L2 ?

del(X, [X|L], L).

del(X, [Y|L], [Y|L2]) :- del(X, L, L2).

**sublist(S, L)** % является ли S подсписком L

sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).

?-sublist(S,[1,2,3]) % 7 вариантов

### Математические вычисления

1 + 2 \* 3 → +(1, \*(2, 3))

X = 1+2 → X = +(1,2)

X is 1+2 → X=3 слева - неконкретизированная переменная, справа - выражение, причем если есть переменные, то они должны быть конкретизированы.

X is Y+1, Y is 2 → ошибка, Y ещё не конкретизирована

X =:= Y – проверка на равенство

X =\= Y – проверка на неравенство

X > Y – проверка на «больше»

**gcd(X,Y,D)** % является ли D наибольшим общим делителем для X и Y

gcd(X,X,X)

gcd(X,Y,D) :- X < Y, Y1 is Y-X, gcd(X,Y1,D)

gcd(X,Y,D) :- X > Y, gcd(Y,X,D)

**len(A,L)** % Вычисление длины списка

len([],0)

len([\_|L],N) :- len(L,N1), N is N1+1

?-len([1,2,3],N) % Если вместо N написать 3, то не работает

**len\_test(L,N)**

len\_test(L,N) :- len(L,N1), N =:= N1

### Управление перебором

{ 0, x < 3

f(x) = { 2, 3 <= x < 6

{ 4, 6 <= x

f(x, 0) :- x < 3

f(x, 2) :- 3 <= x, x < 6

f(x, 4) :- 6 <= x

?-f(1,Y)

Будут перебраны все три правила. А нужно сказать интерпретатору, что правила взаимоисключающие.

**Оператор отсечения** предотвращает проверку других предложений при доказательстве *текущей* цели.

f(x, 0) :- x < 3, ! % дальше не выполняется

f(x, 2) :- x < 6, !

f(x, 4)

**Пример.** Определение максимума.

v1.0 (ошибка при запросе max(3,1,1))

max(X,Y,X) :- X >= Y, !

max(X,Y,Y)

v2.0

max(X,Y,M) :- X >= Y, !, M=X

max(X,Y,M) :- M = Y

**Пример**. Проверка принадлежности

member(X,[X|L]) :- !

member(X,[H|L]) :- member(X,L)

### Отрицание при достижении цели

Цели fail и true.

animal(X)

snake(X)

*Мэри нравятся все животные, кроме змеи*

likes(mary, X) :- snake(X), !, fail

likes(mary, X) :- animal(X)

Оператор not

v1.0

not(P) :- P, !, fail

not(P) :- true

v2.0

not(P) :- P, !, fail; true

diff(X,Y) :- X=Y, !, fail; true

Проблемы, связанные с оператором отсечения:

- становится важен порядок отсечения с точки зрения логики

1)

p:-a,!,b;c

p:-a,!,b

p:-c

p= true , (a and b)|(not a and c)

2)

p:- c;a,!b

p::-c

p:-a,!b

p=true, if c | (a&b)

**Зеленые операторы отсечения** не влияют на логику программы, **красные** — влияют.

Пример с кафе:

good(jean);

expansive(jean);

good(francisco);

два запроса:

good(X), not(expansive(X)).

X=francisco

not(expansive(X)), good(X).

NO

not не умеет перебирать значения, т.е информации в системе нет

**Организация ввода-вывода**

Все программы были устроены так: набор фактов, набор правил и вопросы-ответы.

Необходимо подгружать информацию из файла, БД, уметь ее сохранять, уметь задавать произвольные вопросы, уметь формировать произвольные сообщения.

У интерпретатора несколько входных.выходных потоков. Консоль - один из потоков. Элементы потока - либо отдельные символы или термы.

чтение из потока:

p:-

see(filename),

read\_something,

seen(filename), - прочитали

see(user).- перешли в консоль

запись в поток:

p:-

tell(filename),

write\_something,

told(filename),

tell(user).

read(X), write(X).

read считает только одну строчку, переход на новую nl,

write(‘Hello world!’).

считает куб:

cube:- read(X);

process(X).

process(stop):-!

process(X):=write(X\*X\*X),cube - не будет считать, это структура

process(X):-C is X\*X\*X, write(C), cube

put(c)

get(c) - считает символ, пропуская служебные

- работа с одним символом из потока.

get0(c) - не пропускает служебные символы

От геттерров надо перейти к термам. Используем предикат name(A,L)

A - терм

L - список кодов символов

name(taxi,L) L=’t’,’a’,’x’,’i’

name(X,[122,121])

позволяет проводить операции с терминами на уровне описания.

prefix(X,P):-name(X,XL), name(P,PL),conc(PL,\_,XL).

conc(PL,\_,XL) - проверяет, что начало списка PL совпадает с началом ,XL

В процессе работы программа может подгрузить набор правил из файла.

А либо в результате наименования или в результате чтения. Для проверки ее типа используется проверка: var(X) - является ли неконкретизированной переменной.

nonvar(X) - является ли конкретизированной переменной. (есть значение)

Можно проверить, является ли числом или другим атомом.

atom(X),

integer(X),

float(X),

compound(X) - является ли структурой

В заданном списке считать количество заданных элементов.

X - что ищем

L - сам список

N - результат

conc(\_,[],0)

conc(A,[A|L],N):-,!,conc(A,L,N1), N is N1+1

conc(A,[\_|L],N):-conc(A,L,N)

?-conc(a,[a,b,a,c],N).

N=2, Yes

?- conc(a,[a,b,X,Y],N)

N=3 :-D Yes

?- conc(\_,[1,3,4,4],N)

выведет длину списка

Доработаем - добавим проверку на то, что значение конкретизировано.

conc(\_,[],0)

conc(A,[B|L],N):-nonvar(B),A=B,!,conc(A,L,N1), N is N1+1

conc(A,[\_|L],N):-conc(A,L,N)

Композиция, декомпозиция структур:

T=..L //T- structure, L- List

?- date(7,May)=..L.

=>L=[date,7,May]

?- T=..[f,a,b]

T==f(a,b)

P:-obtain(F,Args), G=..[F | Args], G.

Чето-там по функторы

Операции с базой знаний

Добавляем новые факты и правила динамически:

Предикат asserta - to the start of list

assertb - to the end of list

retract(P) - delete some fact from data base

?- ok

NO

?-assert(ok)

?- ok

YES.

?-parent(bob,X),

T=[parent,bob,X],

retract(T).

удалит всю инфу о детях Боба.

Динамически добавляем не факт, а правило:

assert(P:-a,b,c)

**Работа с конкретизациями**:

bagof(X,P,L) - находит все Х, для которых выполняется Р и пихает в Л

setof(X,P,L) - тоже самое, только с удалением дубликатов

findall(X,P,L) - тоже самое, но истинно, даже, если список пуст.

## Задача поиска в пространстве состояний

Пространство состояний - это граф (может быть бесконечным). Состояние описывается набором параметров.

**Примеры**

* Поиск кратчайшего пути
* Вычисление метрики Левенштейна

Дан некоторый ориентированный взвешенный граф, найти путь с минимальной (максимальной) длиной пути.

### Представление задачи на Prolog

S(X, Y) // наличие перехода из X в Y

S(X, Y, Cost) // путь с ценой

S может быть задана и с помощью правил S:-..., X и Y могут быть сложными структурами.

### Алгоритмы решения

### Поиск в глубину

**Решение #1**

solve(N,P)

solve(N, [N]) :- goal(N)

solve(N, [N|S]) :- S(N, N1), solve(N1, S)

где

goal(N) – функция проверки, является ли состояние N целевым (достигнуто ли целевое состояние)

Недостаток этого решения – возможность зацикливания на циклических графах.

**Решение #2**

solve(N, S) :- df([], N, S)

df(P, N, [N]) :- goal(N)

df(P, N, [N|S]) :- c(N, N1), not member(N1, P), df([N|P], N1, S)

P - список уже пройденных вершин, их не надо рассматривать.

Дополнительное улучшение – ограничение на максимальную глубину пути и на следующей итерации М++. Начало будет выполняться несколько раз. ("Итеративное углубление”)

**Решение #3**

solve(N, S, M) :- df([], N, S, M)

df(P, N, [N], M) :- goal(N)

df(P, N, [N|S], M) :- c(N, N1), M>0, not(member(N1, P)), M1 is M-1, df([N|P], N1, S, M1)

Вызов:

solve2(N, S) :- df2(N, S, 10)

df2(N, S, M) :- solve(N, S, M), !, true; M1 is M+10, df2(N, S, M1)

### Поиск в ширину

Используем очередь. N - вершина, S - каким путем шли в нее. Рассмотрим все переходы из нее и заносим их в очередь. В начале в очереди первый элемент.

**Решение #1**

Solve(N, S) :- bf([[N]], S) // можно хранить путь в обратном порядке и тогда N=S[0], можно не хранить.

bf([[N|P]|\_],[N|P]) :- goal(N), !

bf([P|Q], S) :- extend(P, Paths), conc(Q, Paths, Q2), bf(Q2, S) // Q - конец очереди

extend([N|P], Paths) :- bagof([N1, N|P], (s(N, N1), not(member(N1, P)), Paths)

Paths - дальнейшие варианты перехода.

bagof(X,P,L) L - список таких Х, при которых выполняется преикат Р

### Другие варианты

* Обратный поиск
* Двунаправленный поиск. Две очереди, ищем место встречи.

[картинка]

Начальное состояние (N\*,G\*).

N\* - множество вершин, мы знаем как попасть из N

G\* - множество состояний, из которых мы знаем как попасть в целевое

([N],[G]) -> (N\*,G\*) такой что и там и там есть похожий элемент (стыкуются пути). Когда встретились, решение найдено.

([N,1,2],[G])

т.е. это двунаправленный поиск, но в другом пространстве состояний.

в начале [[N]], [[G]]

[картинка] - граф для решения задачи, по нему обычным поиском.

* Эвристический поиск (А\*) (поиск в ширину + эвристика). Очередь с сортировкой. Для каждого пути f(P) - перспективность пути. Начинаем рассматривать с самого перспективного. Если f=”по порядку”, то это поиск в ширину.

## Реализация экспертных систем на Prolog

**Экспертная система (ЭС)** — это:

* База знаний
* Машина логического вывода
* UI

Всё это взаимодействует.

Предметная область накладывает ограничения на все три пункта.

Правила ЭС могут быть основаны на:

* семантической сети
* фреймах
* правилах

### Системы, основанные на правилах

“Если … то …”. Есть исходные условия, которые истинны.

**Затапливаем соседей**.

kitchen\_dry % на кухне сухо

hall\_wet % вода в гостиной

bathroom\_dry % сухо в ванной

window\_closed

no\_rain

leak\_in\_bathroom :- hall\_wet, kitchen\_dry

problem\_in kitchen

no\_water\_outside :- window\_closed, no\_rain % вода не поступает извне

leak\_in\_kitchen :- problem\_in\_kitchen, no\_water\_outside % утечка на кухне

?-leak\_in\_kitchen

Проблемы:

* Неудобно для обычного пользователя
* Нет возможности управлять ходом рассуждений
* Нет поддержки объяснений

Исправляем:

is\_true(p) :- fact(p)

is\_true(p) :- if(c,p), is\_true(c)

is\_true(and(p1,p2)) :- is\_true(p1), is\_true(p2)

is\_true(or(p1,p2)) :- is\_true(p1); is\_true(p2)

fact(hall\_wet) % запись в БД

......

if(and(hall\_wet, kitchen\_dry), leak\_in\_bathroom)

?-is\_true(...)

Это — **обратный логический вывод**.

Теперь прямой:

forward :- new\_derived\_fact(p), ! , write(p), assert(fact(p), forward; write(‘NO MORE’), nl

new\_derived\_fact(c) :- if(A,C), not(fact(c)), composed(A)

composed(c) :- fact(c)

composed(and(A, B)) : composed(A), composed(B)

composed(or(A, B)) : composed(A); composed(B)

прямой ~ прогнозирующего …

обратный ~ диагностирующего …

Если много разных целей и мало фактов, то надо применять прямой вывод

[картинка: метапродукции генерируют продукции и диагностируют]

Как спрашивать факты у пользователя:

ask(kitchen\_dry, ‘Is kitchen dry?’)

fact(p) :- ask(P, M), ! , retract(ask(P, M)), write(M), …

**Формирование объяснений**

is\_true(p, w)

is\_true(p, p) : fact(p)

is\_true(p, from(c1,p)) :- if(c, p), is\_true(c, c1)

is\_true(and(A,B), from(and(A1, B))) :- is\_true(A, A1), is\_true(B, B1)

?-is\_true(kitchen\_dry, w) % w - “почему?”, может быть несколько.

**Учет неопределённости**

Нечеткие ответы: вместо fact() - given(...,0.5)

is\_true(P, E) :- given(P, E)

is\_true(P, E) :- if(C, P, i), is\_true(C, C , I), E is I\*CI

is\_true(and(A, B), E) :- is\_true(A, A1), is\_true(B, B1), min(A1,A2, E)

### Системы, основанные на семантических сетях

Узлы связаны дугами - это отношения kind\_of

isa(bird, animal) % является ли типом

isa(eagle, bird)

moving\_method(bird, fly)

moving\_method(X, M) :- isa(X, Y), moving\_method(Y, M)

### Системы, основанные на фреймах

bird(kindof, animal)

value(F, S, V) :- Q=..[F, S, V], Q

Q - структура вида F(S,V)

?-value(bird, kindof, V)

value(F, S, V) :-parent(F,P), value(P,S,V)

parent(F,P) :- value(F, K, kindof, P)

value(F, square, V) :- value(F, Height, H)