Contents

1 Логическое программирование **1**

2 I. Введение в PROLOG **2**
2.1 Примеры 2
2.1.1 Расширение программы 4

3 II. Рекурсивные определения правил **4**
3.1 Переменные и их значение 6
3.2 GNUProlog 6
3.3 Некоторое определения 7
3.4 Операторы 7
3.5 Примеры задач 8
3.5.1 Вычисление факториала 8
3.5.2 Числа Фиббоначи 8
3.5.3 Задача 1 9
3.5.4 Задача 2 9
3.5.5 Задача 3 9
3.5.6 Задача 4 9

4 III. Списки **9**
4.1 Примеры работы со списком 10
4.1.1 Соединение двух списков 10
4.1.2 Получение n-го элемента списка. 10
4.1.3 Последний элемент списка 11
4.1.4 Разделение списка на два подсписка 11
4.1.5 Добавление элемента в список 12
4.1.6 Удаление элемента из списка 12
4.1.7 Сортировка списка 12

5 IV. Отладка **13**
5.0.1 Задача с бананами 13

6 Отсечение **15**
6.1 `if then else` 15
6.2 Оператор `not`. 15
6.3 Оператор повтора 16

1 Логическое программирование

Преподаватель - *Сергей Васильевич Родионов*

Адрес: sv-rodion@mail.ru

Облако с материалами

Среда - GNUProlog v 1.4.5

Литература:

- Братко. Программирование для искуственного интелекта
- Стирлинг, Шапиро. Программирование на языке PRLOOG

2 I. Введение в PROLOG

PROLOG - Programming in Logic. Логическое программирование - один из видов программирования.

Большинство задач ИИ - переборные; PROLOG не ориентирован на решение переборных задач, поэтому он не получил широкого распространения.

PROLOG позволяет задать корректное математическое описание задачи; если это получается - задача решается автоматически. Это отличается от **императивных** парадигм программирования, где нужно задать последовательность решения задачи.

Программист задает законы предметной области и **вопрос**. Ответ на вопрос может быть только **да** или **нет**, значения переменных и т.п. - побочные результаты.

2.1 Примеры

Запишем факт, что Том является родителем Боба:

```
parent(tom,bob).
```

Зададим вопрос:

```
?-parent(tom,bob).
```

Получим `yes`.

Расширим родственные связи:

```
parent(tom,bob).
parent(ann,bob).
parent(tom,liza).
parent(bob,mary).
parent(bob,luk).
parent(luk,kate).
```

Вопрос:

```
?-parent(tom,liza).
```

Сначала производится сравнение вопроса с первым фактом. Ответ `No`, переход к следующему факту. На третьем факте ответ - `yes`.

Вопрос:

```
?-parent(X,liza).
```

`X` - переменная. Производится сравнение: 1. `X=tom, liza=bob` > `No` 2. `X=ann, liza=bob` > `No` 3. `X=tom, liza=liza` > `yes`

PROLOG даст ответ:

```
X=tom
yes
```

Вопрос:`? - parent(tom,X)` Будет ответ:

```
X=bob
```

Если здесь нажать `;`, будет:

```
X=liza
yes
```

Если нажать `Enter`, будет выведен только первый ответ.

Вопрос *кто является прародителем* `luk`*?*:

```
?-parent(X,Y),parent(Y,luk).
```

Здесь запятая играет роль **И**.

```
X=tom
Y=bob;
X=ann
Y=bob
yes
```

PROLOG выводит все возможные наборы переменных на каждой строке.

Вопрос: *кто является правнуками* `tom`*?*

```
?-parent(tom,X),parent(X,Y),parent(Y,Z).
```

Ответ:

```
X=bob
Y=luk
Z=kate
yes
```

Вопрос: *Верно ли, что* `bob` *и* `liza` *имеют общего родителя*?

```
?-parent(X,bob),parent(X,liza).
```

Ответ:

```
X=tom
yes
```

Поскольку результат `X=tom`, можно использовать *анонимную переменную*:

```
?-parent(_,bob),parent(_,liza).
```

2.1.1 Расширение программы

Определим пол всех людей. Это можно сделать так:

```
female(kate).
make(tom).
```

или так:

```
gender(kate, feminine).
gender(tom, masculine).
```

Расширим программу понятием “потомок”:

```
offspring(bob,tom).
```

Это можно определить в виде **правила**:

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

Это все конструкции в PROLOG: * Факты * Правила * Вопросы

Факт описывает условие которое всегда верно. Правило читается справа налево.

Правило вывода:

$$\frac{A, A \to B}{B}$$

Программа “Мама”:

```
parent(tom,bob).
parent(ann,bob).
parent(tom,liza).
parent(bob,mary).
parent(bob,luk).
parent(luk,kate).
male(tom).
male(bob).
female(ann).
female(liza).
female(mary).
male(luk).
female(kate).
mother(X,Y):-parent(X,Y),female(X).
```

3 II. Рекурсивные определения правил

Факты:

```
parent(tom,bob).
parent(ann,bob).
parent(tom,liza).
```

```
parent(bob,mary).
parent(bob,luk).
parent(luk,kate).
```

Напишем программу “Предшественник”:

```
predecessor(X,Y):-parent(X,Y).
predecessor(X,Y):-parent(X,Z),predecessor(Z,Y).
```

Если X - родитель Y или X - родитель Z, который является предшественником Y, то X - предшественник Y.

```
?-predecessor(tom,mary).
yes
```

Рассмотрим, как PROLOG будет доказывать это утверждение. Пронумеруем все правила и факты:

```
1) parent(tom,bob).
2) parent(ann,bob).
3) parent(tom,liza).
4) parent(bob,mary).
5) parent(bob,luk).
6) parent(luk,kate).
7) predecessor(X,Y):-parent(X,Y).
8) predecessor(X,Y):-parent(X,Z),predecessor(Z,Y).
```

PROLOG анализирует первые 6 фактов, в которых имя предиката не совпадает. Ответ будет No.

Выполним условную трассировку: `(#шага). #правила.`, * означает возврат к правилу.

```
% Совпало имя предиката
(1). 7. X=tom,Y=mary -> parent(tom,mary) -> No.
% 7-е правило не выполняется.
(2). 8. X=tom,Y=mary -> parent(tom, Z)
% Снова переход к первому факту.
(3). 1. Z=bob -> yes.
% Первая часть правила 8 доказана. Переход ко второй части
(2*). 8. X=tom,Y=mary -> parent(tom, Z),Z=bob -> predecessor(bob,mary).
(4). 7. X=bob,Y=mary -> parent(bob,  mary)
(5). 1. parent(bob,mary) ?= parent(tom,bob) -> No
% Проводится проверка прочих правил. На 8-м шаге будет yes
(8). 4. parent(bob,mary) ?= parent(bob,mary) -> yes.
(4*). 7. X=bob,Y=mary -> parent(bob,  mary) -> yes.
(2**). 8. X=tom,Y=mary -> parent(tom, Z),Z=bob -> predecessor(bob,mary) -> yes.
```

3.1 Переменные и их значение

Если переменной присвоено значение, это значение не может быть изменено в той же ветви доказательства. Имя переменной имеет смысл только в рамках одного правила.

Правило 8 в рассмотренном примере - рекурсивное. При программировании на PROLOG чаще всего используются такие правила.

При возврате происходит переход на новую ветвь доказательства, поэтому значения переменных могут изменится.

Правила 7 и 8 можно объединить так:

```
predecessor(X,Y):-parent(X,Y);parent(X,Z),predecessor(Z,Y).
```

3.2 GNUProlog

GNUProlog ищет тексты исходных программ в рабочей директории. Расширение файлов - `.pl`.

После запуска GNUProlog выведет строку вида `?-`. Открыть файл можно так:

- `consult(file)` - открыть файл
- `write` - вывод на экран
- `listing` - для проверки загруженного текста программы.
- `reconsult(file)` - не работает

```
?-X=2,write(X).
2
X=2
yes
```

При таком вводе:

```
?-X=2.
?-write(X).
16
yes
```

Значение переменной не определено, выводится то, что лежит в памяти.

Рассмотрим программу:

```
fallible(X):-man(X).
man(socrates).
?-fallible(socrates).
yes
```

Все люди ошибаются. Сократ - человек. Ошибается ли Сократ? Да.

```
?-fallible(plato).
no
```

Платон не человек:

```
?-man(plato).
no
```

3.3 Некоторое определения

Атом - последовательность латинских символов в нижнем регистре, цифр и нижнее подчеркиваний. Можно заключить в одинарные кавычки

Переменная: * Последовательность латинских букв, цифр и подчеркиваний, начинающаяся с большой буквы. * Последовательность латинских букв, цифр и подчеркиваний, начинающаяся с подчеркивания. * Подчеркивание (анонимная переменная).

Структуры - составные термы PROLOG. Например, дата может быть представлена в виде дня, месяца и года:

```
date(3, march, 2020).
```

Функтор

```
d(t(334,m),list)
```

Количество параметров функтора - **арность**

3.4 Операторы

Пример - описание геометрических фигур на плоскости.

```
triangle(point(0,0),point(0,4),point(3,0)).
```

Для трехмерного представления можно использовать `point(X,Y,Z).`

```
?-X=2+2
X=2+2
yes
```

`2+2` - выражение. сложения не происходит, `X` равен этой строчке.

`+` - функтор, который может быть представлен и так: `+(a,b)`. Следующие записи эквивалентны.

```
(a+b)+(c+d) <=> +(+(a,b),+(c,d))
```

Для вычисления есть специальный оператор `is`.

```
?- X is 2+2.
X=4
yes
```

Такое выражение выведет ошибку:

```
?-X is X + 1.
```

Доступные операторы: * = - присваивание * is * <, >, ,>=, <= * \= - не равно * == - сравнение * =:= - сравнение с вычислением левых и правых частей.

```
?-2 == 1+1.
no
?-2 =:= 1+1.
yes
```

3.5 Примеры задач

3.5.1 Вычисление факториала

```
fact(0, 1).
fact(N,V) :- N > 0, N1 is N-1, fact(N1, V1), V is V1 * N.
```

Программирование на PROLOG напоминает доказательство по индукции. Первое правило - база индукции, второе - переход от x_n к x_{n-1}.

3.5.2 Числа Фиббоначи

$$x_0 = 1; x_1 = 1$$

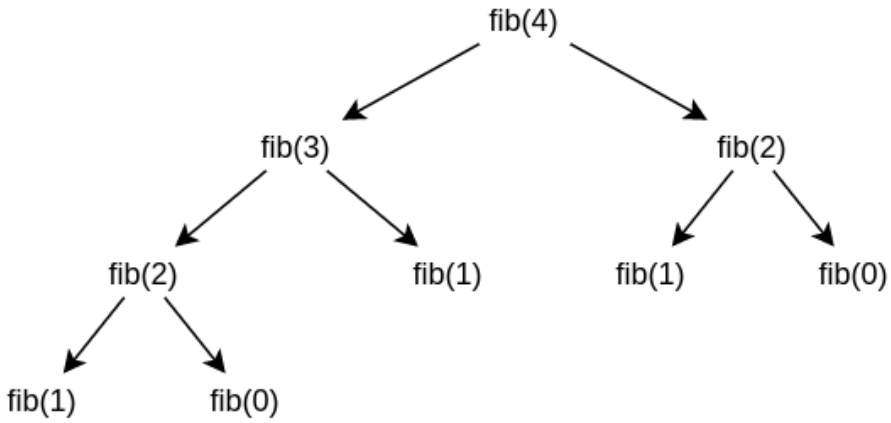
$$x_{N+2} = x_{N+1} + x_N$$

Программа должны состоять из трех строчек.

```
fib(N, V)
```

```
fib(0, 1).
fib(1, 1).
fib(N, V) :- N > 1, N1 is N - 1, fib(N1, V1), N2 is N - 2, fib(N2, V2), V is V1 + V2.
```

Построим граф вычислений:



Такие вычисления неэффективны - `fib(2)` вычисляется 2 раза.

3.5.3 Задача 1

Создать предикат, вычисляющий неотрицательную степень целого числа.

```
pow(_, 0, 1).
pow(N, P, R) :- N > 0, P1 is P - 1, pow(N, P1, V), R is V * N.
```

3.5.4 Задача 2

Создать предикат, вычисляющий по натуральному числу N сумму чисел от 1 до N.

```
sum(_, 0).
sum(N, R) :- N1 is N - 1, sum(N1, R1), R is N + R1.
```

3.5.5 Задача 3

Имеется доска размером 5×5. Обезъяна может двигаться вправо и вверх.

Определить, может ли обезъяна добраться до банана.

```
start(1,1).
stop(4,4).
go :- start(X, Y), move(X, Y).
move(X,Y) :- stop(X, Y).
move(X,Y) :- X < 5, X1 is X + 1, move(X1, X).
move(X, Y):- Y < 5, Y1 is Y + 1, move(X, Y1).
```

3.5.6 Задача 4

Предикат, вычисляющий по натуральному числу N сумму нечетных чисел, не превосходящих N.

```
sum(0, 0).
sum(1, 0).
sum(N, R) :- 0 is mod(N, 2), N > 0, N1 is N - 2, sum(N1, R1), R is R1 + N.
sum(N, R) :- 1 is mod(N, 2), N > 0, N1 is N - 1, sum(N1, R).
```

4 III. Списки

Список - последовательность термов, перечисленных через запятую в квадратных скобках. Это основная структура данных Prolog.

Список делится на две части - **голову** и **хвост**. Для отделения головы и хвоста используется вертикальная черта. Хвост - всегда список.

Рассмотрим предикат `member`, который позволяет определить, относится ли элемент к списку. Обычно он уже встроен в Prolog.

```
member(Elem, [Elem|_]).
member(Elem, [_|Tail]) :- member(Elem, Tail).
```

Если элемент относится к списку `Tail`, то он является элементом списка, к которому добавлена голова. В пустом списке голову и хвост выделить нельзя.

```
| ?- member(a, [b, a, c]).
true ?
yes

| ?- member(a, [b, a, a]).
true ?
yes

| ?- member(a, [b, c, X]).
X = a
yes

| ?- member(X, [a, b, c]).
X = a ? a
X = b
X = c
yes
```

4.1 Примеры работы со списком

4.1.1 Соединение двух списков

```
conc([], Q, Q).
conc([HP|TP], Q, [HP|TR]) :- conc(TP, Q, TR).
```

Пусть есть список:

```
[jack,jim,jim,tim,jim,bob]
```

Надо найти списки слева и справа от `jim`.

```
| ?- conc(L, [jim|R],[jack,jim,jim,tim,jim,bob]).

L = [jack]
R = [jim,tim,jim,bob] ? ;

L = [jack,jim]
R = [tim,jim,bob] ?

yes
```

4.1.2 Получение n-го элемента списка.

```
get(0, [H|_], H).
get(N, [_|T], L) :- N1 is N - 1, get(N1, T, L).
```

```
| ?- get(0, [a,b,c],X).

X = a ?

yes
| ?- get(1, [a,b,c],X).

X = b ?

yes
| ?- get(2, [a,b,c],X).

X = c ?

yes
```

4.1.3 Последний элемент списка

```
fin([T], T).
fin([_|T], R) :- fin(T, R).
```

```
| ?- fin([a, b, c, d, e], X).

X = e ?

yes
```

4.1.4 Разделение списка на два подсписка

В первый список должны войти числа, меньшие N, во второй - оставшиеся элементы.

```
split(N, [T], [T], []) :- T < N.
split(N, [T], [], [T]) :- T >= N.
split(N, [H|T], [H|L1], L2) :- H < N, split(N, T, L1, L2).
split(N, [H|T], L1, [H|L2]) :- H >= N, split(N, T, L1, L2).
```

Более простое решение:

```
split(_, [], [], []) :- !.
split(N, [H|T], [H|L1], L2) :- H < N, split(N, T, L1, L2).
split(N, [H|T], L1, [H|L2]) :- split(N, T, L1, L2).
```

```
| ?- split(3, [1,2,3,4,5],X,Y).

X = [1,2]
Y = [3,4,5] ?

yes
```

4.1.5 Добавление элемента в список

```
add(I,L,[I|L]).
```

```
| ?- add(1, [], X).

X = [1]

yes
| ?- add(1, [1,2,3,4], X).

X = [1,1,2,3,4]

yes
```

4.1.6 Удаление элемента из списка

```
del(_, [], []) :- !.
del(I, [I|T], T) :- !.
del(I, [H|T], [H|R]) :- del(I, T, R).
```

```
| ?- del(1, [1,2,3],X).

X = [2,3] ?

yes
| ?- del(2, [1,2,3],X).

X = [1,3] ?

yes
| ?- del(2, [1,2,3,2,1,2,3],X).

X = [1,3,2,1,2,3] ? a

X = [1,2,3,1,2,3]

X = [1,2,3,2,1,3]
```

4.1.7 Сортировка списка

```
del(_, [], []) :- !.
del(I, [I|T], T) :- !.
del(I, [H|T], [H|R]) :- del(I, T, R).

ordered([]) :- !.
ordered([_]) :- !.
ordered([H|[H1|T]]) :- H =< H1, ordered([H1|T]), !.
```

```
min([X], X) :- !.
min([H|T], X1) :- min(T, X1), X1 < H.
min([H|T], H) :- min(T, X1), X1 >= H.

sort_m(X, X) :- ordered(X), !.
sort_m(L, [M|T1]) :- min(L, M), del(M, L, L1), sort_m(L1, T1).
```

5 IV. Отладка

Режимы отладки:

- `debug`/`nodebug`
- `trace`/`notrace`
- `spy`/`nospy`

`leash` - настройка `spy`: * `full` - полный список * `half` - `call`, `redo`; * `loose` - `call`; * `none`; * `tight` - `call`, `fail`, `redo`, `exception`.

Отладка включается автоматически при выполнении с включенной трассировки или при выполнении `debug` на предикат, для которого включен `spy`.

В процессе трассировки можно выполнять следующие команды:

- `Enter` - переход к следующей строке доказательства
- `a` - abort
- `h`, `?` - полная справка по командам

5.0.1 Задача с бананами

Имеется доска размером 5×5. Обезьяна может двигаться вправо и вверх.

Вывести кратчайший путь до банана.

```
start(1,1).
stop(4,4).
go(Path) :- start(X, Y), move(X, Y, [], Path).
move(X, Y, P, [m(X, Y) | P]) :- stop(X, Y).
move(X, Y, From, To) :- X < 5, X1 is X + 1, move(X1, Y, [m(X, Y) | From], To).
move(X, Y, From, To) :- Y < 5, Y1 is Y + 1, move(X, Y1, [m(X, Y) | From], To).
```

Первый параметр - `From` - инициализируется пустым списком и накапливает каждый вновь выполненный шаг `m(X, Y)`.

Второй аргумент - `To` - рекурсивно возвращает результат, когда добрались до банана.

```
| ?- go(P).

P = [m(4,4),m(4,3),m(4,2),m(4,1),m(3,1),m(2,1),m(1,1)] ? a

P = [m(4,4),m(4,3),m(4,2),m(3,2),m(3,1),m(2,1),m(1,1)]

P = [m(4,4),m(4,3),m(3,3),m(3,2),m(3,1),m(2,1),m(1,1)]

P = [m(4,4),m(3,4),m(3,3),m(3,2),m(3,1),m(2,1),m(1,1)]

P = [m(4,4),m(4,3),m(4,2),m(3,2),m(2,2),m(2,1),m(1,1)]

P = [m(4,4),m(4,3),m(3,3),m(3,2),m(2,2),m(2,1),m(1,1)]

P = [m(4,4),m(3,4),m(3,3),m(3,2),m(2,2),m(2,1),m(1,1)]

P = [m(4,4),m(4,3),m(3,3),m(2,3),m(2,2),m(2,1),m(1,1)]

P = [m(4,4),m(3,4),m(3,3),m(2,3),m(2,2),m(2,1),m(1,1)]

P = [m(4,4),m(3,4),m(2,4),m(2,3),m(2,2),m(2,1),m(1,1)]

P = [m(4,4),m(4,3),m(4,2),m(3,2),m(2,2),m(1,2),m(1,1)]

P = [m(4,4),m(4,3),m(3,3),m(3,2),m(2,2),m(1,2),m(1,1)]

P = [m(4,4),m(3,4),m(3,3),m(3,2),m(2,2),m(1,2),m(1,1)]

P = [m(4,4),m(4,3),m(3,3),m(2,3),m(2,2),m(1,2),m(1,1)]

P = [m(4,4),m(3,4),m(3,3),m(2,3),m(2,2),m(1,2),m(1,1)]

P = [m(4,4),m(3,4),m(2,4),m(2,3),m(2,2),m(1,2),m(1,1)]

P = [m(4,4),m(4,3),m(3,3),m(2,3),m(1,3),m(1,2),m(1,1)]

P = [m(4,4),m(3,4),m(3,3),m(2,3),m(1,3),m(1,2),m(1,1)]

P = [m(4,4),m(3,4),m(2,4),m(2,3),m(1,3),m(1,2),m(1,1)]

P = [m(4,4),m(3,4),m(2,4),m(1,4),m(1,3),m(1,2),m(1,1)]
```

```
(4 ms) no
```

Можно обернуть список с помощью `reverse` или сохранять координаты при возврате из рекурсии:

```
start(1,1).
stop(4,4).
go(Path) :- start(X, Y), move(X, Y, [], Path).
move(X, Y, P, [m(X, Y) | P]) :- stop(X, Y).
move(X, Y, From, [m(X, Y) | To]) :- X < 5, X1 is X + 1, move(X1, Y, From, To).
move(X, Y, From, [m(X, Y) | To]) :- Y < 5, Y1 is Y + 1, move(X, Y1, From, To).
```

В таком случае путь будет выведен в обратном порядке:

```
| ?- go(P).

P = [m(1,1),m(2,1),m(3,1),m(4,1),m(4,2),m(4,3),m(4,4)] ? ;

P = [m(1,1),m(2,1),m(3,1),m(3,2),m(4,2),m(4,3),m(4,4)] ? ;

P = [m(1,1),m(2,1),m(3,1),m(3,2),m(3,3),m(4,3),m(4,4)] ?
```

6 Отсечение

Отсечение отключает механизм возврата, т.е. запрещает передоказательство (поиск альтернатив) в том правиле, где оно применяется.

6.1 `if then else`

Пример - оператор `if then else` - без отсечения его реализовать невозможно.

```
if(B,C,_) :- B,!,C.
if(_,_,D) :- D.
```

```
| ?- if(1 > 0, write('1'), write('2')).
1

yes
| ?- if(1 < 0, write('1'), write('2')).
2

(1 ms) yes
| ?-
```

6.2 Оператор `not`.

```
not(X) :- X, !, fail.
not(_).
```

6.3 Оператор повтора

```
repeat.
repeat :- repeat.
```