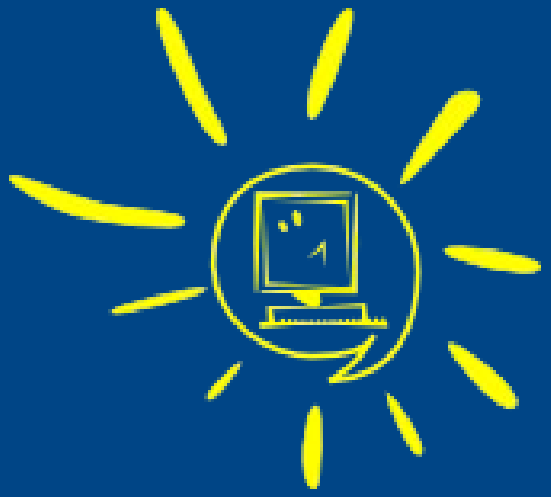


Логическое программирование на языке Prolog

Корябкин Иван

Летняя Компьютерная школа
4 августа 2014

Парадигмы программирования

Императивное программирование

Программист указывает *последовательность команд*, которые требуются для решения поставленной задачи.

Этому подходу следуют почти все популярные языки программирования на сегодняшний день: C/C++, Python, Java, Pascal, C#, etc.

Парадигмы программирования

Функциональное программирование

Программист указывает формулу для вычисления решения задачи.

Функциональной парадигме следуют на данный момент такие языки, как Haskell, Erlang, Closure, OCaml, Lisp. Некоторые императивные языки поддерживают функциональные конструкции (Python, например).

Парадигмы программирования

Логическое программирование

Программист описывает саму задачу, не указывая явно метод её решения.

Такой стиль программирования иногда называют **декларативным**.

Язык Prolog

Языков логического программирования не так много. Частично это связано с их низкой популярностью, частично с тем, что новых революционных идей в логическом программировании почти не появляется.

Два наиболее известных языка — это Prolog и Planner. В рамках спецкурса мы рассмотрим первый как более простой.

Язык Prolog: реализации

Существует несколько работающих реализаций Prolog. На занятии все примеры продемонстрированы в среде SWI Prolog, хотя и в других интерпретаторах (GNU Prolog или Visual Prolog) они также должны успешно исполняться.

Запуск программ

Мы будем использовать преимущественно интерактивный shell и задавать точки входа самостоятельно.

Для загрузки программы в память интерпретатора используется комманда:

```
swipl -f program.pro
```

Описание задачи: факты

Prolog оперирует фактами при поиске решения задачи. Факт — это некоторая информация о взаимоотношениях между объектами. Например, выражение

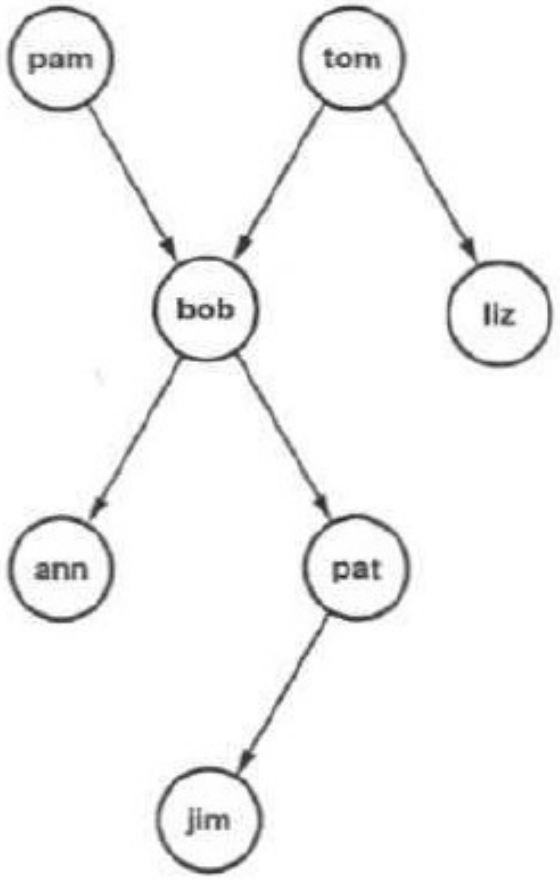
```
parent(bob, alice).
```

скажет Prolog-вычислителю, что *bob* является родителем *alice*.

Описание задачи: факты

В нашей программе мы можем указать и более полный набор фактов. Например, полное генеалогическое древо семьи.

```
parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```



Запросы на генеалогическом древе

Prolog-интерпретатор может отвечать на какие-либо запросы на данном дереве. Например, мы можем задать вопрос, является ли *pam* родителем *bob*. И получить утвердительный ответ:

```
?- parent(pam, bob).
true.
?- parent(bob, jim).
false.
```

Полнота системы

Prolog в своей вычислительной модели придерживается аксиомы о полноте информации. Это значит, что если нам не сообщили об истинности какого-либо высказывания явно и его никак нельзя выразить из имеющихся данных, то оно является ложным.

Таким образом, интерпретатор не имеет права ответить «не знаю». Неизвестность для него эквивалентна ложности.

Запрос с неизвестным результатом

```
?- parent(ivan, maria).
false.
```

Интерпретатор не знает ничего ни про *ivan*, ни про *maria*, и из недостающей информации делает вывод о том, что они не являются родителем и ребенком.

Свободные переменные

Разумеется, сами по себе запросы фактов из базы ничего интересного из себя не представляют. Prolog позволяет указать в качестве одного или нескольких аргументов свободные переменные.

```
?- parent(X, jim).
X = pat.
```

В данном случае интерпретатор не просто нашел, что ответ существует, но ещё и указал значение X.

Свободные переменные

В случае существования нескольких возможных ответов интерпретатор Prolog предложит все возможные.

?- parent(bob, X).

X = ann ;

X = pat.

Или даже так...

```
?- parent(X, Y).
X = pam,
Y = bob ;
X = tom,
Y = bob ;
X = tom,
Y = liz ;
...
```

Основные объекты

Свободные переменные в Prolog начинаются с заглавной буквы или символа нижнего подчеркивания («_»).

Предикаты (parent) и термы (bob, pat, …) обязаны начинаться со строчных букв. Термы также можно указывать в кавычках, если требуется, чтобы они начинались с заглавной буквы.

Более сложные системы фактов

Факты в языке Prolog могут иметь произвольное количество аргументов. Более того, возможно определить предикаты с одинаковым именем, но разным количеством аргументов.

```
everything_fine.
everything_fine(bob).
```

```
?- everything_fine.
true.
?- everything_fine(alice).
false.
```

Более сложные системы фактов

Например, мы можем дополнить нашу систему фактов несколькими дополнительными.

```
female(pam).
male(bob).
male(tom).
female(liz).
female(ann).
female(pat).
male(jim).
```

Более сложные системы фактов

После того, как интерпретатор Prolog узнал о том, что существа описанные в генеалогическом древе имеют пол, мы вполне можем ввести какие-либо более сложные ассоциации.

Например, отношение grandfather(X, Y).

Отношения на основе правил

Однако, вся информация, которая нужна программисту для получения сведений о том, является ли X дедушкой Y может быть легко выражена в трех запросах:

- male(X),
- parent(X, Z),
- parent(Z, Y).

Очевидно, что вносить избыточную информацию в наши данные не имеет смысла.

Отношения на основе правил

Prolog позволяет описать факт, выполнимость которого зависит от нескольких правил.

```
grandfather(X, Y) :-
    male(X), parent(X, Z), parent(Z, Y).
```

При доказательстве факта grandfather(X, Y) будет произведен перебор с отсечениями всех возможных вариантов решения задачи.

Анонимная переменная

В некоторых случаях нам требуется указать свободную переменную, значение которой в ответе не требуется найти. Например, можно определить предикат has_children(X) как

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

Анонимная переменная

В таком случае при ответах на запрос интерпретатор будет выдавать, какой именно ребенок есть у X.

```
?- has_children(bob).
Y = ann ;
Y = pat.
```

Анонимная переменная

В данной задаче вместо свободной переменной Y можно использовать анонимную переменную «_».

```
has_children(X) :- parent(X, _).
```

```
?- has_children(bob).
true.
```

Числа и арифметические операции

Свободные переменные также могут быть отождествлены с произвольными числами (как целыми, так и вещественными).

```
is_negative(X) :- X < 0.
```

```
?- is_negative(-6).
true.
```

Числа и арифметические операции

Для операций сравнения (<, >, >=, =<) в Prolog необходимо, чтобы значения аргументов-выражений были вычислимы на очередном шаге перебора. В частности, попытка доказать is_negative(X), к сожалению (или счастью), приведет к ошибке.

Однако, для операции сравнения (=) данное ограничение не выполняется. В частности, потому, что в Prolog операция «=» имеет неарифметическую природу.

Операция равенства

Демонстрация данного факта.

?- 3 + 2 > 2 + 3.

false.

?- 3 + 2 >= 2 + 3.

true.

?- 3 + 2 = 2 + 3.

false.

Операция равенства

Возможно, данный запрос пояснит природу операции равенства.

```
?- X = 2 + 3.
X = 2+3.
```

В качестве альтернативы операции сравнения, которая не вычислят значения выражений-аргументов, существует операция **is**.

Операция is

Операция **is** вычисляет только значение правого выражения, однако с помощью нее можно писать почти произвольные арифметические функции.

```
?- X is 3 + 2.
X = 5.
```

Предикаты для вычисления арифметических функций

Постольку поскольку, Prolog является языком отношений между объектами, в нем не существует возможностей для возврата целочисленных значений из функций. Вместо этого используется вспомогательный параметр предиката.

Например, вычисление квадратного корня может быть осуществлено предикатом sqrt(X, Y), который выполняется, если Y является квадратным корнем из X.

```
?- sqrt(5, X).
X = 1.7320508075688772.
```

Пример: вычисление НОД

Небольшой пример, в котором описан предикат, вычисляющий наибольший общий делитель первых двух аргументов.

```
gcd(X, 0, X).
gcd(X, Y, R) :- Y > 0,
                Xn is mod(X, Y),
                gcd(Y, Xn, R).
```

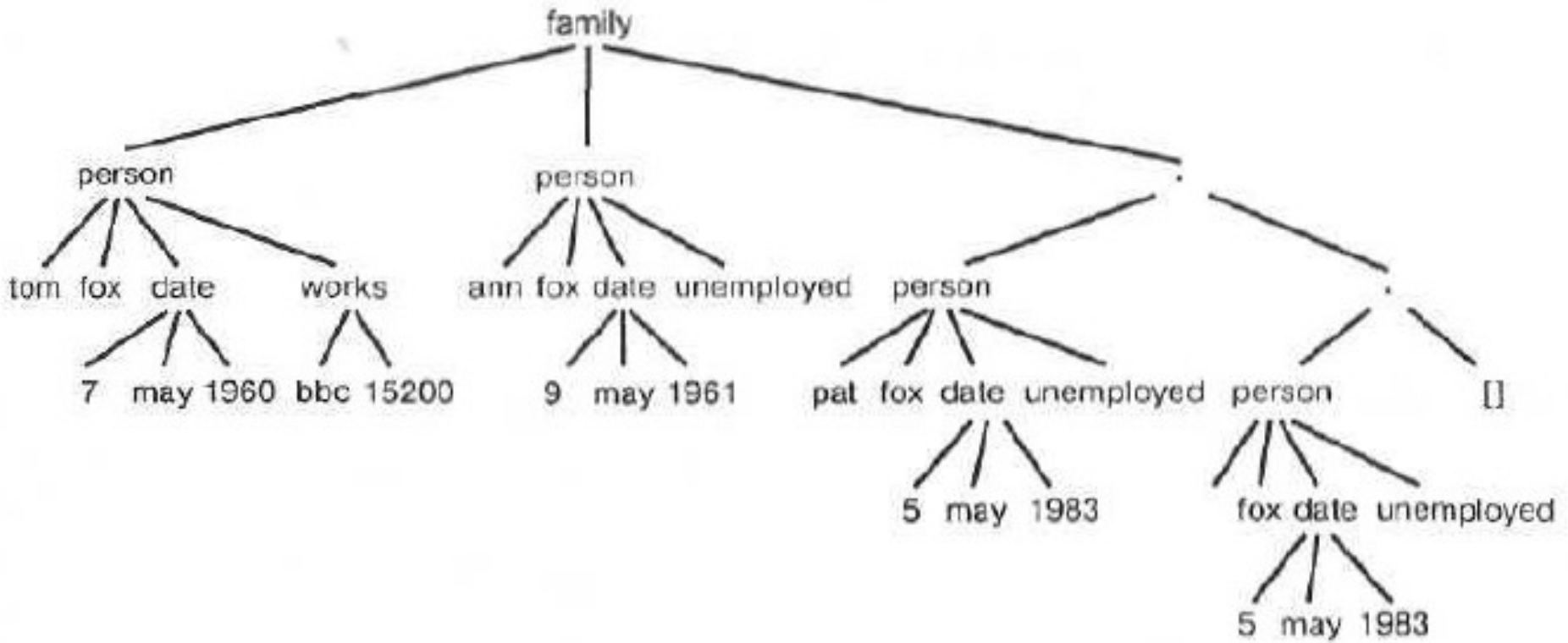
Структуры

Термы языка Prolog можно объединять в произвольные структуры. Структура — это:

- либо одиночный терм;
- либо терм, в скобках после которого перечисляются структуры.

Таким образом структуры в Prolog являются некоторым аналогом деревьев из термов.

Структуры


family
person
person
.
tom fox date
works
ann fox date unemployed
person
.
7 may 1960
bbc 15200
9 may 1961
pat fox date unemployed
person
[]
5 may 1983
fox date unemployed
5 may 1983

Структуры

Приведем пример использования структур, введя структуру вида point(x, y) и структуру segment(p1, p2). И напишем, предикат, проверяющий отрезок на вертикальность и горизонтальность, а также на совпадения концов отрезка.

```
vertical(segment(point(X, _), point(X, _))).
horizontal(segment(point(_, Y), point(_, Y))).
is_point(segment(X, X)).
```

Структуры

В приведенном примере нам удалось в одном аргументе передать всю информацию об отрезке. Более того, за счёт сопоставления с образцом, нам не пришлось прибегать к использованию условий.

Данная техника используется во многих языках программирования и называется pattern matching (сопоставление с образцом).

Списки

Список — это простая структура данных для хранения последовательности произвольных элементов в Prolog.

Список записывается в квадратных скобках, а его элементы перечисляются через запятую.

Например,

```
[ann, joe, 3, X, bill, 1.1]
```

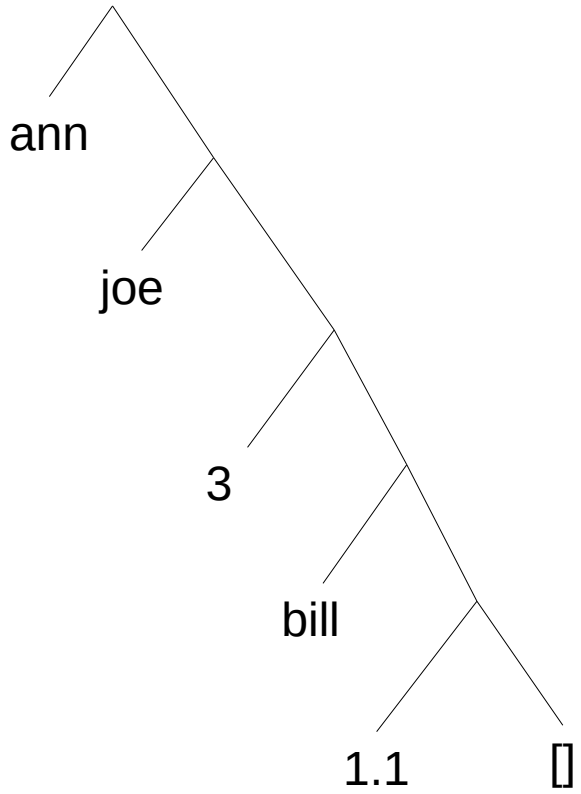
Списки

Первый элемент списка называется его «головой», а список, состоящий из элементов, кроме первого, — «хвостом».

Например, в предыдущим примере, головой был элемент *ann*, а хвостом — [joe, 3, X, bill, 1.1].

Списки

Данная терминология связана с тем, что список представляется в памяти набором вложенных пар.



Шаблоны и списки

Сопоставление с образцом в Prolog позволяет задавать в качестве шаблонов не только список целиком, но и его части.

Некоторые примеры шаблонов:

- [ ] — пустой список;
- [X] — список из одного элемента X;
- [X, Y] — список из двух элементов Y;
- [H | T] — список, в котором H — голова, а T — хвост.

Шаблоны и списки

Более сложные примеры:

- [X, Y | _] — список, в котором первые два элемента — X и Y;

- [_ | _] — непустой список.

Простые примеры работы со списками

Опишем несколько простых предикатов для работы со списками.

```
length([], 0).
length([_ | T], L) :-
    length(T, Lx), L is Lx + 1.
```

Простые примеры работы со списками

Деление списка на два по принципу «четный-нечетный».

```
zip([], [], []).
zip([X], [X], []).
zip([X, Y | H], [X | H1], [Y | H2]) :-
    zip(H, H1, H2).
```

Накапливающий параметр

В некоторых случаях достаточно сложно написать чисто рекурсивную процедуру. В таких случаях часто используется техника «накапливающего параметра». Например, в предикате reverse.

```
reverse([], R, R).
reverse([H | T], X, R) :- reverse(T, [H | X], R).
reverse(L, R) :- reverse(L, [], R).
```

Накапливающий параметр

На предыдущем слайде мы использовали вспомогательный второй параметр, в котором «накапливали» ответ для возврата на последнем шаге через третий.

Упражнение. Реализуйте предикат flatten, который преобразует вложенные друг в друга списки в один линейный список. Например:

[A, [B, C, [D]], E, [F, G]] → [A, B, C, D, E, F, G].

Отсечения

При написании многих программ на Prolog возникает избыточное количество случаев, которые бывает необходимо отсечь, дабы избежать полного перебора.

Напишем простой пример с предикатом max:

```
max(A, B, A) :- A >= B.
max(A, B, B) :- A < B.
```

Отсечения

В данном случае мы были вынуждены явно прописать взаимно исключающие условия. Однако, этого можно избежать с помощью оператора отсечения.

При переборе возможных вариантов Prolog сохраняет так называемые «точки возврата». Это позиции в дереве перебора после каждого корректного отождествления.

Отсечения

Программист на Prolog имеет право указать в качестве одного из условий оператор отсечения — !

Данный оператор имеет следующую семантику: он уничтожает все точки возврата с момента начала вычисления текущего предиката.

Отсечения

Это значит, что если существует правило вида:

```
X :- A, B, C, D, !, E, F, G.
```

то после успешного нахождения решения для A, B, C и D, интерпретатор забудет, что для них, возможно, существовали ещё какие-то решения. Таким образом, будет рассмотрено только первое решение для A, B, C, D. Если с ним не будет найдено решение для E, F, G, то ответом будет являться false.

Отсечения

Приведем пример использования отсечения на примере max:

```
max(A, B, C) :- A > B, !, C = A.
max(_, B, B).
```

Отсечения

Ещё один пример использования: опишем предикат friend(X, Y), который будет выполняться, если X является другом Y.

Допустим, *lena* считает своим другом кого угодно, кроме *vasya*. Тогда определение предиката friend для *lena* можно записать следующим образом:

```
friend(lena, vasya) :- !, fail.
friend(lena, _).
```

Отсечения

Подобным образом реализуется предикат not:

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

Встроенные предикаты

В примере был использован встроенный предикат fail, который заведомо никогда не выполняется.

Несколько полезных встроенных предикатов, для контроля выполнения вычислений: var(X), atom(X), integer(X), float(X), nonvar(X), compound(X).

Упражнение. Написать предикат member, который проверяет элемент X на наличие в списке L.

Встроенные предикаты

Существует также предикат высшего порядка findall(X, G, L), который находит все такие X, удовлетворяющие условию G и отождествляет их со списком L.

Например, таким образом можно найти всех друзей Ильи:

```
?- findall(X, friend(ilya, X), L).
L = [dima, anya, andrey, denis].
```

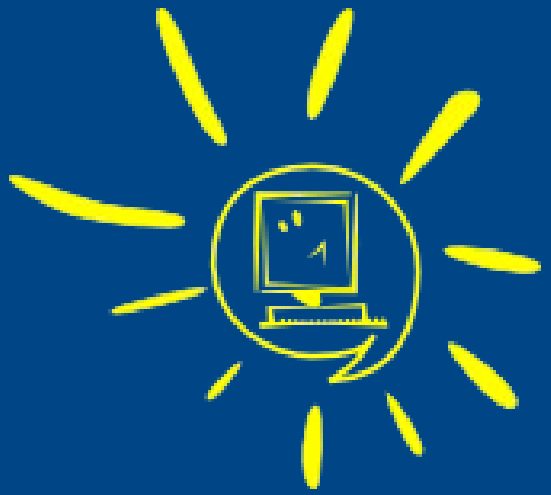
Ввод и вывод

Prolog обладает несколькими встроенными предикатами для осуществления ввода/вывода данных.

- read(X) — считывает один терм со стандартного ввода;
- write(X) — выводит терм на стандартный вывод;

Что почитать?

- И. Братко «Алгоритмы искусственного интеллекта на языке Prolog», Вильямс, 2004.
- http://swi-prolog.org/ → Tutorials



Спасибо за внимание!