Логическое программирование

язык ПРОЛОГ

Механизм сопоставления и поиска с возвратом

Пролог воспринимает в качестве программы некоторое описание задачи или базы знаний и сам производит логический вывод, а также поиск решения задач, используя при этом поиск с возвратом и унификацию.

Унификация представляет собой процесс **сопоставления** цели с фактами и правилами базы знаний.

Решение задачи называют процессом согласования цели.

Цель может быть согласована, если она может быть сопоставлена с заголовком какого-либо предложения базы. Для этого предикат цели и предикат заголовка предложения должны иметь одинаковое имя и арность.

Пример

1. **отец (коля, миша).**
2. **отец(коля, саша).**
3. **брат(миша, саша).**
4. **брат(вася, коля).**

Запрос (Цель)

?- **отец(коля, саша).** Согласуется с предложением (фразой) №2.

?-**отец (коля, X).**

Согласуется с предложениями (фразами) № 1 и №2, при этом конкретизируется переменная X.

Правила унификации и согласования термов

- переменная согласуется с константой или структурой. В результате эта переменная становится конкретизированной, т.е. принимает значение этой константы или структуры;
- переменная согласуется с переменной, при этом они обе становятся одной и той же переменной (т.е. интерпретатор дает им одно и то же внутреннее имя);
- анонимная переменная "_" согласуется с любым объектом;
- константа согласуется с константой, если они идентичны;
- структура согласуется с другой структурой, если они имеют одинаковые функторы, а компоненты поддаются унификации.

Пример

отец (коля, миша).
отец(коля, саша).
брат(миша, саша).
брат(вася, коля).
дядя(A, B):-
брат(A, C),
отец(C, B).

Рассмотрим запрос «дядя» к программе:
?-дядя(вася, X).

Механизм сопоставления и поиска с возвратом

- После того, как фраза, согласующаяся с запросом, будет обнаружена, она активируется. При этом каждая подцель, входящая в тело фразы, обрабатывается также, как и исходный запрос. Если тело согласующейся фразы является пустым, т.е. она представляет собой факт, то запрос сразу же оказывается успешным.
- Если интерпретатор не может найти фразу согласующуюся с целью, то он возвратится назад. Он возвращается к последней успешной подцели, ликвидирует конкретизацию любых переменных, явившуюся результатом успешной обработки этой подцели, и приступает к поиску в множестве фраз текущей программы заголовка другой фразы, которая согласуется с данной подцелью.
- При успешном выполнении запроса пользователя интерпретатор выводит на терминал значения всех тех переменных, входящих в состав запроса, которые были конкретизированы в результате процесса обработки, или слово «да» (true), если переменные в запросе

Поиск фразы и конкретизация переменных

Как только в стеке появился запрос, интерпретатор приступает к поиску множества фраз с тем же самым именем предиката и с тем же количеством аргументов, что и у запроса

Состояние интерпретатора:

активные запросы **фразы программы**

?-дядя(вася, X).

дядя(вася, X):-

брат(вася, C),

отец(C, X).

После унификации запроса с заголовком фразы интерпретатор переходит к телу фразы. Если тело фразы пустое, т.е. фраза является фактом, то запрос сразу же является успешным. Если тело фразы не пустое, то интерпретатор помещает в стек запросов каждую подцель, входящую в тело, и, в свою очередь, обрабатывает ее. Если все подцели из тела фразы будут успешно обработаны, то и весь исходный запрос будет успешным.

Поиск первой подцели

Тело правила «дядя» состоит из двух подцелей «брат (вася, С)» и «отец (С, Х)». Первая подцель «брат (вася, С)» помещается в стек запросов:

активные запросы **фразы программы**

?-дядя(вася, Х). **брат(вася, С),**
дядя(вася, Х):-
отец(С, Х).

?- брат (вася, С).

активные запросы *фразы программы*

?-дядя(вася, Х). *- указывет на -* *дядя(вася, Х):-*
брат(вася, С),
отец(коля, Х).

?- брат (вася, С). *- указывет на -* *брат(вася, коля).*

Поиск второй подцели

активные запросы **фразы программы**

?-дядя(вася, X). **- указывет на -** **дядя(вася, X):-**
брат(вася, коля),
отец(коля, X).

?- брат(вася, коля). **- указывет на -** **брат(вася, коля).**

?- отец (коля, X). **- указывет на -** **отец (коля, миша).**
отец (коля, саша).

Интерпретатор пытается согласовать подцель «отец (коля, X).» с фразой «отец (коля, миша)». Эта попытка приводит к успеху, в результате чего конкретизируется переменная X, которая получает значение «миша».

Состояние интерпретатора:

активные запросы **фразы программы**

?-дядя(вася, миша). **- указывет на -** **дядя(вася, миша):-**
брат(вася, коля),
отец(коля, миша).

?- брат(вася, коля). **- указывет на -** **брат(вася, коля).**

?- отец(коля, миша). **- указывет на -** **отец(коля, миша).**
отец(коля, саша).

Интерпретатор выводит значение переменной **X=миша.** При этом весь стек активных запросов сохраняется.

Неудача запроса и возврат назад

- Если активный запрос достигнет конца соответствующего множества фраз (базы знаний) и не найдет сопоставимой фразы, то он завершится неудачей. Если активный запрос является подцелью составного запроса и не является первой подцелью, то интерпретатор возвратится назад, чтобы повторно проанализировать предыдущую подцель составного запроса. Когда интерпретатор возвращается назад, то ликвидируются все конкретизации переменных, выполненные последним активным запросом. Если же активный запрос является первой подцелью составного запроса, то неудача активного запроса приводит к неудаче всего составного запроса.
- Указание интерпретатору вернуться назад - ввод символа «;». Вод символа «;» означает отказ от только что полученного ответа. Это заставляет интерпретатор возвратиться назад и приступить к поиску другого ответа.

Пример

отец (коля, миша).
отец(коля, саша).
брат(миша, саша).
брат(вася, коля).
дядя(A, B):- **/* правило*/**
брат(A, C),
отец(C, B).

Рассмотрим запрос «дядя» к программе:
?-дядя(коля, X). , который сопоставляется правилом
Подцелями правила являются
брат(коля, C).
отец(C, B).
Цель **брат(коля, C).** не сопоставляется с фразами базы знаний, запрос завершается неудачей.

Пример возврата

отец (коля, миша).
отец(коля, саша).
брат(вася, петя).
брат(миша, саша).
брат(вася, коля).
дядя(A, B):- **/* правило*/**
брат(A, C),
отец(C, B).

Рассмотрим запрос «дядя» к программе:
?-дядя(вася, X). , который сопоставляется правилом
Подцелями правила являются
брат(вася, C).
отец(C, B).
Цель **брат(вася, C).** сопоставляется с фразой **брат(вася, петя).**
Переменная С конкретизируется «петя». Цель **отец (петя, B).** терпит неудачу – происходит возврат, конкретизация переменной **C** отменяется, ищется следующая фраза, сопоставимая с фразой **брат(вася, C).**

Процедура в Прологе

Процедура — это несколько правил, заголовки которых содержат одинаковые предикаты. Так, например, два правила

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

max(X, Y, Y) :- X < Y.

реализуют процедуру нахождения наибольшего из двух чисел и использует одинаковый предикат max/3, вида max(число1, число 2, макс_число).

Считается, что между всеми правилами одной процедуры неявно присутствует соединительный союз «или», то есть все правила процедуры дизъюнктивно связаны между собой.

Рекурсивные процедуры

Рекурсию можно применять для достижения того же эффекта, какой реализуется при употреблении итеративных управляющих конструкций, например, цикл «пока» в процедурных языках. В рекурсивном правиле более сложные входные аргументы выражаются через менее сложные. Классическим примером рекурсивного определения в Прологе может служить программа предок, состоящая из двух правил:

предок (A, B):-
родитель (A, B).
(фраза 1)

предок (A, B):-
родитель (C, B),
(фраза 2)
предок (A, C).

Или

предок (A, B):-
родитель (A, B).

предок (A, B):-
родитель (A,C),
предок (C, B).

Рекурсивные процедуры

Фраза (1) процедуры «предок» определяет исходный вид этой процедуры. Как только данная фраза станет истинной, дальнейшая рекурсия прекратится.

Фраза (2) - это рекурсивное правило. При каждом вызове данное правило поднимается на одно поколение вверх. Подцель «родитель(С, В)», входящая в тело правила, вырабатывает значение переменной С, которое затем используется в рекурсивной подцели «предок (А, С)».

Если поменять порядок следования подцелей в теле фразы (2), то такая процедура называется процедурой с левой рекурсией, так как во фразе (2) рекурсивная подцель стоит первой (т.е. располагается слева от всех подцелей).

Интерпретатор Пролога не может надежно обрабатывать леворекурсивные процедуры, что обусловлено природой заложенной в него стратегии решения задач.

Примеры рекурсии: вычисление факториала числа

```
fact(1,1).                  /* факториал единицы равен единице */
fact(N,F):- N>1,            /* убедимся, что число больше единицы */
N1 is N-1,
fact(N1,F1),                /* F1 равен факториалу числа, на единицу меньшего исходного
    числа */
F is F1*N.                  /* факториал исходного числа равен произведению F1 на само число */
```

Запрос:

```
?- fact(5,X).
X = 120 ;
false.
```

Структура – это составной терм

имя (арг1, арг2, ..., аргn) ,

где арг1, ..., аргn сами являются термами (т.е. константами, переменными или структурами).

Структуры - это определяемые программистом объекты произвольной сложности. Имя структуры называется **функтором**, а аргументы в скобках - **компонентами**.

Функтор объединяет компоненты в структуру.

Примеры:

дата(1, май, 1998).

Дата(День, Месяц, Год).

Все структурные объекты можно представлять в виде деревьев, где функтор - корень дерева, а ветви - компоненты.

Разные структуры могут иметь одинаковые функторы, если они различаются количеством компонентов, например:

точка(X , Y) - точка в двумерном пространстве;

точка (X , Y , Z) - точка в трехмерном пространстве.

В одной программе - это разные структуры.

Списки

Списки рассматриваются в Прологе как специальный частный случай двоичного дерева. В Прологе существует два способа изображения списков:

пустой список – это атом, изображается **[] ;**

непустой список – рассматривается как структура, состоящая из двух частей:

первый элемент – **голова** (ею может быть любой объект, например, переменная или структура);

остальная часть списка называется **хвостом**, им может быть только список.

Голова соединяется с хвостом при помощи специального функтора « . »

. (Голова, Хвост)

Хвост может быть пустым, либо иметь свои собственные голову и хвост:

.(мяч, .(кукла, .(машинка, [])))

Представление списков

В Прологе используются специальные средства для списковой нотации, позволяющие представлять списки в более лаконичном виде:

[Элемент1, Элемент2,…].

Примеры: **а) [мяч, кукла, машинка]**

б) Увлечения1=[теннис, музыка],
Увлечения2=[еда, пиво],
Список= [миша,Увлечения1,коля, Увлечения2].
Список= [миша, [теннис, музыка] ,коля, [еда, пиво]].

На практике бывает удобно трактовать хвост списка как самостоятельный объект. Тогда используется следующая списковая нотация:

[Голова | Хвост].

Например. **L=[a,b,c],** тогда можно написать **Хвост=[b,c] L=[a | Хвост]** или **[Элемент1, Элемент2,…| Остальные].**

Можно перечислить любое количество элементов списка, затем поставить | и перечислить остальные элементы.

Пример**: [a,b,c]= [a | [b,c]] =[a,b | [c]] =[a,b,c | []].**

Пример программы со списками

/*Исходный вид: когда 1-й список пуст, то результат конкатенации – это второй список*/

konkat([],S,S).

/*Рекурсивное правило*/

konkat([X|S1],S2,[X|S3]):-konkat(S1,S2,S3).

Эта программа обладает большой гибкостью и ее можно использовать в различных целях.

Запрос **?- konkat ([a, в] , [c, d] ,S).**

позволяет получить список, являющийся результатом конкатенации: S=[a, в, c, d].

Разбиение заданного списка на две части всеми возможными способами:

?- konkat (S1 ,S2, [a, в, c]).

Получим:

S1=[], S2=[a,b,c];

S1=[a], S2=[b,c]; S1=[a, b], S2=[c]; S1=[a, b, c], S2=[];

найти все элементы, предшествующие заданному элементу списка, и следующие за ним:

?- konkat (До , [c | После], [a, в, c, d, e]).

До=[a, b], После=[d, e]

Продолжение примера

б) элемент списка, предшествующий данному, и элемент, следующий за данным элементом:

?- konkat (_ , [До, c, После | _], [a, в, c, d, e]).

До= b, После= d

в) удаление хвоста списка, начиная с заданной цепочки элементов:

?- konkat (Список , [c, c, c | _], [a, в, c, c, c, d, e]).

Список=[a, в]

г) используя конкатенацию, можно задать отношение принадлежности элемента списку:

принадлежит (X, S):- konkat (_ , [X | _], S).

То есть X принадлежит S, если список S можно разбить на два списка таким образом, чтобы элемент X являлся головой второго из них.

Отрицание в Прологе

Информация о фактах, которые не являются истинными, или об отношениях, которые не соблюдаются, называется негативной. Негативная информация обычно не хранится в Пролог-программах в явной форме. Вместо этого считается, что вся информация, отсутствующая в текущем множестве фраз, является ложной. Система не различает отсутствующую в программе фразу и доказуемо неистинную фразу. По умолчанию Пролог-система всегда руководствуется правилом, которое называется предположением о замкнутости мира:

если фраза P отсутствует в текущей программе, то это значит, что представлено отрицание P.

Отрицание в явной форме представляет собой встроенный предикат:

not (аргумент) ,

где аргумент – это запрос, значение которого после обработки заменяется на противоположное.

fail - специальная цель, встроенный предикат, который всегда терпит неудачу.

Этот встроенный предикат всегда приводит к неудаче. Когда он является частью составного запроса, то заставляет интерпретатор вернуться назад и попробовать найти новые конкретизации для переменных. Этот процесс будет продолжаться до тех пор, пока не исчерпаются соответствующие факты программы, после чего весь составной запрос потерпит неудачу:

```
not (P):- P, ! , fail.
not (P):- true.
```

true – цель, которая всегда истинна.

Примеры

Определим сельского жителя как человека, не являющегося ни горожанином, ни жителем пригорода:

горожанин (иван).

житель_пригорода (софья).

сельский_житель (X):- not (горожанин (X)), not (житель_пригорода (X)).

Мэри любит всех животных, кроме змей:

likes(mary,X):-annimal(X),not(snake(X)).

different(X,Y):-not(X=Y).

Средства управления ходом выполнения программы

Пространство поиска запроса – это множество всех возможных ответов, рассматриваемых интерпретатором при выполнении запроса. Автоматический перебор – полезный программный механизм, но иногда требуется его ограничить или полностью исключить для повышения эффективности программы.

Встроенный предикат “сократить” – "**!**" (отсечение) применяется для уменьшения размера пространства поиска запроса, т.е. ограничивает перебор всех возможных вариантов ответов или полностью подавляет его.

Он дает указание интерпретатору не возвращаться назад далее той точки, где стоит этот предикат. Предикат "! " - это псевдоцель, которая всегда истинна. Этот предикат по разному действует на составной запрос и на множество фраз, образующих процедуру.

Предикат «отсечение» - !

Предикат “отсечение” является одной из подцелей составного запроса, например:

?- a(X), b(Y), ! , c(X, Y, Z).

При выполнении этого запроса, если подцели a(X) и b(Y) окажутся успешными, интерпретатор пройдет через предикат !. После этого интерпретатор не сможет возвратиться назад к подцелям, стоящим перед !, и обязан использовать при выполнении подцели c(X,Y,Z) те значения X и Y, которые они получили при конкретизации. Если подцель c(X,Y,Z) окажется неуспешной при данных конкретизациях X и Y, то и весь запрос потерпит неудачу. Если предикат ! будет последней подцелью составного запроса, то это гарантирует получение только одного ответа на запрос.

Пример 1: отказ от поиска несуществующих ответов

Путь задана база данных «возраст»:

возраст(петров,18).

возраст(сидоров, 16).

возраст(иванов, 18).

Создадим запрос: **?- возраст (петров, X).** Если не использовать отсечение, то после того, как будет найдено значение возраста, система будет пытаться найти другие ответы, что нерационально при большой базе данных «возраст». Если использовать отсечение в запросе, можно сократить пространство поиска.

Если Запрос ?- возраст (петров, X), ! .

Можно записать правила с отсечением других вариантов ответов. Тогда, если интерпретатор дойдет до этого предиката, то он не сможет вернуться назад для рассмотрения остальных фраз программы:

возраст(петров,18):- ! .

возраст(сидоров, 16) :- !.

возраст(иванов, 18) :- !.

Запрос ?- возраст (петров, X) .

Пример 2. Влияние отсечения на процедуру

```
a(1):-write('один').
a(X):-b(X),write('два').
a(3):-write('три').
b(21).
b(22).

?- a(N).
один
N = 1 ;
два
N = 21 ;
два
N = 22 ;
три
N = 3 ;
false.
```

Поместим отсечение в середину второго правила:

```
a(1):-write('один').
a(X):-b(X),!,write('два').
a(3):-write('три').
b(21).
b(22).
```

Результат запроса ?- a (N).

```
один
N = 1 ;
два
N = 21 ;
false.
```

Пример 3. Поиск наибольшего из двух чисел.

Если подцель, стоящая перед отсечением, окажется ложной, то система перейдет к рассмотрению следующего предложения процедуры, в противном случае произойдет переход через отсечение, и тогда следующие предложения этой процедуры рассматриваться не будут.

```
max(X,Y,X):- X>Y,!.
max(X,Y,Y):- Y>X,!.
max(X,_,X):- write("Числа равны"),!.
```

Еще пример

Пусть некоторая Мэри любит всех животных.
Это записывается:
likes(mary,X):-annimal(X).

Мэри не любит змей:
likes(mary,X):-snake(X),!,fail.

fail - специальная цель, встроенный предикат,
который всегда терпит неудачу.

Комментарии

- Программы на Прологе (Prolog) могут содержать комментарии. Комментарии можно помещать в любом месте программы. На длину комментариев нет ограничений.
- Комментарии обрамляются символами **/*** и ***/**.
- Пример:
- */* Это программа на Прологе */*

Встроенные предикаты: Ввод и вывод

read(*-Term*) - осуществляет ввод терма с входного потока. Термами являются числа, строки (в одинарных кавычках) и списки (в том числе списки символов и кодов символов).

readln/1 - считывает символы до конца строки, формируя список слов (в одинарных кавычках);

get_char/1 - считывает с потока один символ, используя синхронный ввод (если ввод осуществляется с клавиатуры, то функция не начнет работу, пока пользователь не нажмет Enter);

read(*+Stream, -Term*) - считывает данные из потока и размещает их в *Term*;

readterm(Тип_Аргумента, Аргумент) - считывает терм;

write(*+Term*) – выводит терм в текущий поток;

write(*+Stream, +Term*) – выводит терм в поток *Stream*

writeln(*+Term*) - эквивалентно write(Term), nl. /* nl - переход на следующую строку*/

Передавать программы Пролог-системе можно при помощи двух встроенных предикатов.

consult («имя файла») - добавляет новые предложения в программу из файла.

reconsult («имя файла») – действует так же, как и предыдущий предикат, но при этом переопределяет ранее введенные определения отношений, не затрагивая тех отношений, о которых в файле ничего не сказано.

Встроенные предикаты: Ввод и вывод

Удобно обрабатывать строки на SWI Prolog в виде списка кодов символов (они хранятся при этом в юникоде). Для ввода строк удобно использовать предикат **rd_line**.

Правило считывает список кодов с текущего устройства ввода:

rd_line(Str):- get_code(H),

(code_type(H, end_of_line), !, Str = []; code_type(H, end_of_file), !, Str = []; Str = [H|T], rd_line(T)).

Предикат get_code возвращает код символа (аналогично get_char). Если текущий символ представляет собой конец строки или файла – результатом является пустая строка (пустой список кодов). В противном случае выполняется рекурсивный вызов для считывания остальных символов.

Для ввода из файла нужно установить файл в качестве основного устройства ввода.

see/1, принимает ‘имя файла’, открывает файл для чтения, устанавливает его в качестве текущего потока ввода. После вызова предиката весь ввод осуществляется с открытого файла;

s**een/0**, закрывает файл, ассоциированный с текущим потоком ввода. После вызова этого предиката ввод осуществляется с клавиатуры;

tell/1, тоже самое, что see, но открывает файл для записи и устанавливает в качестве потока вывода;

told/0, закрывает файл открытый tell. После вызова, весь вывод попадает на экран.

Такой вариант не сработает если вам надо работать одновременно с несколькими файлами. В этом случае надо использовать версии функции read, get_char и т.п., принимающие имя открытого файлового потока в качестве дополнительного аргумента. Открыть поток можно функцией **open**, а закрыть – **close**, работа с потоками в SWI Prolog мало отличается от других языков.