Подготовка к кр по прологу!!!!

1. **Опишите понятие и структуру фактов в языке Пролог. Раскройте основные возможные типы, опишите понятие атом. Расскажите принцип работы терминальной машины Swi-Prolog, объясните каким образом происходит обработка вопросов.**

Prolog работает с предикатами и высказываниями.

Высказывания- утверждение «истина» или «ложь».

Предикат – высказывание с переменными.

### Понятие и структура фактов в языке Пролог

Факты —определяют отношения или свойства объектов. Структура факта включает имя предиката и аргументы.

База фактов или база высказываний:

- имя\_факта(факт1,…,факт2(аргумент23)).

Пример:

-pr(3)

-pr(5)

-pr(7)

Добавим в память факт о том, что число простое

Assert(pr(5))

И так далее, после этого программа выдаст true, иначе если факта нет, тогда false.

Атом(тип данных)- некая переменная, которая всегда равна его значению.

- Assert(fact(atom)).

True

-fact(x)

X=atom

Факты обычно представляют отношения между объектами, которые доступны программе для обработки запросов и вывода логических заключений.

### Основные типы данных в Прологе

В Prolog существует несколько основных типов данных:

1. Атомы: Это независимые элементы, подобные константам в других языках. Атомы могут быть идентификаторами или строками. Например, 'Иван', 'Мария', true, foo, 'привет мир'.

2. Числа: Включают целые числа и числа с плавающей запятой.

3. Переменные: Переменные начинаются с заглавной буквы или символа подчеркивания \_. Переменные в Prolog не имеют фиксированных значений и могут быть привязаны к значениям других переменных или термов в ходе вычислений.

4. Списки: Списки упорядоченных элементов. Например:

Например: списки Чёрча состоят из:

-head – один элемент

-tail – остальные (весь основной список)

Элементами списка является все что угодно.

-List=[Head/Tail]

Пример: (1,2,3]=[1(Head)/[2,3](Tail)]

5. Термы: Термы — это основная структура данных в Prolog, которая может быть атомом, числом, переменной или сложным термом. Сложные термы имеют вид functor(argument1, argument2, ...).

### Принцип работы терминальной машины Swi-Prolog

1. Загрузка базы данных: Программы в Prolog составляются из фактов и правил, хранящихся в базе данных. При запуске Swi-Prolog загружает эти данные.

2. Обработка вопросов: Пользователь или программа могут задавать вопросы (запросы) базе данных. Запрос — это вопрос о том, что можно логически вывести из базы данных.

3. Привязка и унификация: Основным механизмом обработки запросов является унификация, которая пытается сделать термы одинаковыми, подбирая значения переменных. Если унификация успешна, Prolog может продолжить выполнение правил для поиска решений.

4. Обратный поиск (Backtracking): Если на каком-то этапе выполнения не удается найти решение, Prolog откатывается (отмена сделанных решений до момента неудачи) и пробует альтернативные пути. Этот процесс известен как обратный поиск.

### Как происходит обработка вопросов (запросов)

1. Запрос: Пользователь вводит запрос в интерактивной сессии (например, ?- родитель(иван, Кто).).

2. Унификация: Prolog начинает процесс унификации, пытаясь сопоставить запрос с фактами и правилами в базе данных.

3. Поиск решения: Если есть соответствие, переменные в запросе заменяются значениями, найденными в базе данных.

4. Вывод результата: Если решение найдено, Prolog выдает результат. Если не найдено, Prolog использует механизм обратного поиска, чтобы проверить другие возможные решения.

5. Рекурсивный поиск: Prolog применяет рекурсивный поиск через правила, чтобы найти цепочки отношений, которые могут привести к решению.

Этот процесс продолжается до тех пор, пока

не будут исчерпаны все возможные решения или пока пользователь не прервет выполнение.

1. **Опишите смысл термина унификация, приведите показательные примеры. Объясните, как задаются предикаты, что такое правила и каким образом происходит работа с ними.**

Унификация — это процесс сопоставления двух термов (сущностей) и их сведение к одной форме путем изменения значений переменных. Унификация является краеугольным камнем в работе Prolog и используется для выполнения запросов, вывода и обратного поиска.

Унификация работает путем рекурсивного сопоставления структур термов и их подструктур.

#### Примеры унификации

1. Унификация с атомами:

Prolog

foo = foo. % true, оба атома идентичны

foo = bar. % false, разные атомы

2. Унификация с переменными:

Prolog

X = 42. % true, переменная X привязывается к числу 42

Y = foo. % true, переменная Y привязывается к атом foo

3. Унификация со сложными термами:

Prolog

point(X, Y) = point(1, 2). % true, X=1 и Y=2

foo(1, X) = foo(Y, 2). % true, X=2 и Y=1

foo(bar, Y) = foo(X, baz). % true, X = bar и Y = baz

foo(bar, Y) = foo(X, qux). % false, Y != qux

4. Унификация со списками:

Prolog

[X, Y, Z] = [1, 2, 3]. % true, X=1, Y=2, Z=3

[Head | Tail] = [1, 2, 3]. % true, Head=1, Tail=[2, 3]

[X | Y] = [1, 2, 3]. % true, X=1, Y=[2, 3]

[X, Y | Z] = [1, 2, 3, 4]. % true, X=1, Y=2, Z=[3, 4]

### Предикаты

Предикат — это логическая конструкция, представляющая отношение или свойство объектов. Предикаты задаются с помощью имени и списка аргументов.

Prolog

родитель(иван, мария). % Иван родитель Марии

собака(шарик). % Шарик — собака

Каждый предикат в Prolog определяется своими фактами и может содержать переменные. В том же множестве фактов возможно создание сложных отношений.

### Правила

Правило в Prolog позволяет устанавливать связи между различными фактами и задавать логику на их основе. Правило состоит из головы и тела. Голова — это предикат, который будет истинным, если истинно тело правила. Тело правила — это выражение, состоящее из одного или нескольких предикатов, объединенных логическими связками.

#### Пример правила

Prolog

родитель(X, Y) :- отец(X, Y).

родитель(X, Y) :- мать(X, Y).

Это правило читается как: X является родителем Y, если X отец Y или X мать Y.

### Работа с предикатами и правилами

1. Определение предикатов

Prolog

студент(анна).

студент(иван).

2. Определение правил

Prolog

однокурсники(X, Y) :-

студент(X),

студент(Y),

X \= Y.

### Обработка запросов

Когда задается вопрос (запрос), Prolog пытается унифицировать его с существующими фактами и правилами.

#### Пример вопроса и обработки

Prolog

студент(анна).

студент(иван).

студент(дмитрий).

однокурсники(X, Y) :-

студент(X),

студент(Y),

X \= Y.

?- однокурсники(анна, Кто).

Процесс:

1. Поиск факта: Ищется факт студент(анна) — он существует.

2. Поиск совпадений: Prolog затем проверяет все возможные значения для Кто, которые смогут унифицироваться с предикатом студент(Y) при условии X \= Y.

- Проверка студент(иван) удовлетворяет условиям Кто = иван.

- Проверка студент(дмитрий) — следующий возможный результат Кто = дмитрий.

Таким образом, вывод запроса:

Prolog

Кто = иван; % Первое решение

Кто = дмитрий. % Второе решение

### Автоматизация унификации и обратного поиска

На каждом шагу Prolog автоматически:

- Пытается унифицировать текущие термы.

- При неудаче — инициирует обратный поиск (откат к предыдущему состоянию).

- Генерирует следующее возможное значение переменной.

- Продолжает процесс до тех пор, пока не исчерпаны все возможные пути или пользователь не прервет выполнение.

Таким образом, унификация и последующий обратный поиск позволяют Prolog находить все возможные решения (и варианты) для поставленной задачи.

1. **На примере числовых алгоритмов объясните смысл рекурсии вверх и рекурсии вниз в прологе.**

Рекурсия — это метод программирования, при котором функция вызывает саму себя для решения задачи. В контексте Prolog (и логического программирования в целом) рекурсия используется для определения отношений и вычисления значений через последовательные этапы.

### Рекурсия вверх

Рекурсия вверх, также известная как хвостовая рекурсия, происходит, когда рекурсивный вызов является последней операцией в функции. Это означает, что текущему вызову не нужно ничего делать после возврата рекурсивного вызова, что позволяет производить оптимизацию за счет переиспользования текущего стека.

#### Пример: Факториал

Факториал числа n (обозначается n!) определяется как произведение всех положительных целых чисел от 1 до n. Например, факториал 5 (5!) равен 1 2 3 4 5 = 120.

В Prolog можно реализовать факториал с помощью рекурсии вверх следующим образом:

Prolog

% Базовый случай: 0! = 1

факториал(0, 1).

% Рекурсивный случай:

факториал(N, R) :-

N > 0,

N1 is N - 1,

факториал(N1, R1),

R is N \* R1.

1. Вызов факториал(5, R).

2. Переменная N > 0, поэтому выполняется рекурсивный вызов факториал(4, R1).

3. Этот процесс повторяется до тех пор, пока N не станет равным 0 (базовый случай).

4. Когда достигается базовый случай, возвращает значение 1.

5. Возврат идет «вверх» по стеку вызовов, выполняя умножение на каждом уровне.

### Рекурсия вниз

Рекурсия вниз реализуется путем накопления результата или промежуточного значения и передавания его на следующую итерацию функции. Это позволяет избежать необходимости возврата по стеку вызовов для выполнения операций.

#### Пример: Факториал с аккумулятором

Тот же факториал может быть реализован с использованием аккумулятора, который накапливает результат.

Prolog

% Вспомогательная функция с аккумулятором

факториал(N, A, R) :-

N > 0,

N1 is N - 1,

A1 is A \* N,

факториал(N1, A1, R).

% Базовый случай

факториал(0, A, A).

% Внешний интерфейс: для удобства пользователя

факториал(N, R) :-

факториал(N, 1, R).

1. Вызов факториал(5, R) инициирует факториал(5, 1, R).

2. N > 0, значит, выполняется факториал(4, 5, R).

3. Этот процесс продолжается до тех пор, пока N не станет равным 0.

4. Когда N достигает 0, возвращает аккумуляторное значение A как результат.

Таким образом, в отличие от рекурсии вверх, в рекурсии вниз результирующее значение накапливается и передается через каждый рекурсивный этап.

### Другие примеры рекурсии

#### Сумма чисел от 1 до N

##### Рекурсия вверх (обычная рекурсия)

Prolog

сумма(0, 0).

сумма(N, S) :-

N > 0,

N1 is N - 1,

сумма(N1, S1),

S is S1 + N.

##### Рекурсия вниз (хвостовая рекурсия)

Prolog





% Вспомогательная функция с аккумулятором

сумма(N, A, S) :-

N > 0,

N1 is N - 1,

A1 is A + N,

сумма(N1, A1, S).

% Базовый случай

сумма(0, A, A).

% Внешний интерфейс

сумма(N, S) :-

сумма(N, 0, S).

#### Фибоначчи числа

##### Рекурсия вверх

Prolog

фиб(0, 0).

фиб(1, 1).

фиб(N, F) :-

N > 1,

N1 is N - 1,

N2 is N - 2,

фиб(N1, F1),

фиб(N2, F2),

F is F1 + F2.

##### Рекурсия вниз (с аккумуляторами)

Prolog

% Вспомогательная функция с двумя аккумуляторами

фиб(0, A, \_, A).

фиб(N, A, B, F) :-

N > 0,

N1 is N - 1,

A1 is B,

B1 is A + B,

фиб(N1, A1, B1, F).

% Внешний интерфейс

фиб(N, F) :-

фиб(N, 0, 1, F).

1. **Раскройте на примерах понятие backtracking, оператор отсечения и смысл его применения.**

Backtracking — это метод поиска решений путем систематического перебора всех возможных вариантов. Когда система поиска сталкивается с ошибкой или тупиком, она «откатывается» (англ. backtrack) до предыдущего точки принятия решения и пробует другой путь.

В контексте Prolog, backtracking является основной частью его механизма поиска, позволяющей находить все возможные решения для заданного запроса.

#### Пример: Поиск в списке

Допустим, у нас есть список предикатов о членах семьи.

Prolog

родитель(джон, джеймс).

родитель(мэри, джеймс).

родитель(джон, лиза).

Запрос родитель(джон, Ребенок) выполнит поиск всех детей Джона.

Prolog

?- родитель(джон, Ребенок).

Ребенок = джеймс ;

Ребенок = лиза ;

false.

Для каждого совпадающего предиката, Prolog возвращает значение. Если есть несколько совпадений, Prolog использует backtracking для того, чтобы найти все возможные экземпляры.

### Оператор отсечения (cut)

Оператор отсечения (!), или cut, является мощным инструментом в Prolog, который позволяет управлять поведением backtracking. Он устраняет альтернативные пути после своего появления, таким образом, все альтернативы до этого не рассматриваются.

#### Пример: Предикат максимум

Рассмотрим предикат, который находит максимум из двух чисел.

Prolog

максимум(X, Y, X) :- X >= Y.

максимум(X, Y, Y) :- X < Y.

При запросе максимум(2, 3, M), Prolog выполнит следующее:

1. Сначала проверит максимум(2, 3, 2), но условие 2 >= 3 ложно.

2. Затем проверит максимум(2, 3, 3) и условие 2 < 3 верно, значит, M = 3.

Можно улучшить эту реализацию с помощью cut.

Prolog





максимум(X, Y, X) :- X >= Y, !.

максимум(\_, Y, Y).

1. Сначала проверит максимум(2, 3, 2). Условие 2 >= 3 ложно, поэтому идет дальше.

2. Затем проверяет максимум(2, 3, \_) при X >= 3 после cut, отбрасывая альтернативные решения и экономя ресурсы.

### Смысл и применение оператора cut

Цель оператора cut — оптимизировать поиск в Prolog, избегая ненужных проверок и backtracking, помогая тем самым:

- Сокращать количество проверок

- Обеспечивать более предсказуемое и целенаправленное выполнение программы

- Исключать логически неверные или нежелательные альтернативы

#### Пример: Определение классификации по возрасту

Prolog





классификация(Возраст, ребенок) :- Возраст < 18, !.

классификация(\_, взрослый).

Prolog





?- классификация(15, K).

K = ребенок.

?- классификация(20, K).

K = взрослый.

Благодаря cut, если выполняется Возраст < 18, Prolog не будет анализировать следующее предложение.

1. **Опишите принцип работы со списками Черча в Swi-prolog. Покажите реализации встроенных предикатов работы со списками на основе механизма унификации(append, reverse , nth0).**

### Списки Черча в SWI-Prolog

Списки Черча — это функциональное представление списков. В SWI-Prolog списки Черча можно рассматривать как списки, которые хранятся не в виде структур данных, а как функции, принимающие две операции: `empty` (пустота) и `unit` (единица).

Чтобы работать со списками Черча, достаточно понимать, что они используют понятие свертки (`fold`) для построения списка.

### Принципы работы со списками Черча в Prolog

В Prolog списки Черча можно реализовать как функции, принимающие два аргумента — накопитель и список. Эти функции повторяется для каждого элемента списка и применяют его к накопителю. Разберем это более подробно ниже.

### Реализация встроенных предикатов append, reverse и nth0 на основе механизма унификации

#### 1. Append

\*\*Задача:\*\* Объединение двух списков.

\*\*Реализация:\*\* В Prolog предикат `append/3` объединяет два списка. Мы указываем, что объединение первого списка с пустым возвращает первый список, а объединение двух непустых списков происходит рекурсивно.

```prolog

my\_append([], List, List).

my\_append([Head|Tail], List, [Head|Result]) :-

my\_append(Tail, List, Result).

```

\*\*Примеры использования:\*\*

```prolog

?- my\_append([1, 2], [3, 4], Result).

Result = [1, 2, 3, 4].

?- my\_append([], [1, 2, 3], Result).

Result = [1, 2, 3].

?- my\_append([a, b], [c, d, e], Result).

Result = [a, b, c, d, e].

```

#### 2. Reverse

\*\*Задача:\*\* Реверсирование списка.

\*\*Реализация:\*\* Мы создаем предикат `reverse/2`, который использует аккамулятор для обратного обхода списка.

```prolog

my\_reverse(List, Reversed) :-

my\_reverse(List, [], Reversed).

my\_reverse([], Acc, Acc).

my\_reverse([Head|Tail], Acc, Reversed) :-

my\_reverse(Tail, [Head|Acc], Reversed).

```

\*\*Примеры использования:\*\*

```prolog

?- my\_reverse([1, 2, 3], Reversed).

Reversed = [3, 2, 1].

?- my\_reverse([a, b, c], Reversed).

Reversed = [c, b, a].

```

#### 3. Nth0

\*\*Задача:\*\* Получение n-го элемента списка, (индексация с нуля).

\*\*Реализация:\*\* Предикат `nth0/3` получает n-й элемент списка, индексируя с 0.

```prolog

my\_nth0(0, [Element|\_], Element).

my\_nth0(N, [\_|Rest], Element) :-

N > 0,

N1 is N - 1,

my\_nth0(N1, Rest, Element).

```

\*\*Примеры использования:\*\*

```prolog

?- my\_nth0(2, [a, b, c, d, e], Element).

Element = c.

?- my\_nth0(0, [1, 2, 3, 4, 5], Element).

Element = 1.

1. **Объясните принцип работы со строками. Покажите на примерах основные встроенные предикаты работы со строками.**

В SWI-Prolog строки представляются как списки ASCII-кодов или как атомы. Атомы — это последовательности символов, заключенные в апострофы `'` или двойные кавычки `"`. Строки, заключенные в двойные кавычки, интерпретируются как списки ASCII-кодов символов строки. Это поведение позволяет работать с текстом на уровне как строки, так и отдельных символов.

### Примеры строк в Prolog

1. Атомные строки:

```prolog

A = 'hello'.

B = "hello". % Интерпретируется как список ASCII-кодов, [104, 101, 108, 108, 111]

```

2. Списки ASCII-кодов:

```prolog

L = [104, 101, 108, 108, 111].

```

### Основные встроенные предикаты для работы со строками

#### 1. \*\*atom/1\*\*

Проверяет, является ли переменная атомом.

```prolog

?- atom('hello').

true.

?- atom("hello").

false.

```

#### 2. \*\*atom\_length/2\*\*

Возвращает длину атома.

```prolog

?- atom\_length('hello', Length).

Length = 5.

```

#### 3. \*\*atom\_concat/3\*\*

Конкатенирует два атома.

```prolog

?- atom\_concat('hello', 'world', Result).

Result = helloworld.

```

#### 4. \*\*sub\_atom/5\*\*

Извлекает подстроку из атома.

```prolog

?- sub\_atom('hello\_world', 0, 5, \_, Sub).

Sub = hello.

?- sub\_atom('hello\_world', 6, 5, 0, Sub).

Sub = world.

```

#### 5. \*\*string\_codes/2\*\*

Преобразует строку в список ASCII-кодов и наоборот.

```prolog

?- string\_codes("hello", Codes).

Codes = [104, 101, 108, 108, 111].

?- string\_codes(Atom, [104, 101, 108, 108, 111]).

Atom = "hello".

```

#### 6. \*\*string\_length/2\*\*

Возвращает длину строки.

```prolog

?- string\_length("hello", Length).

Length = 5.

```

#### 7. \*\*split\_string/4\*\*

Разбивает строку на список строк по заданному разделителю.

```prolog

?- split\_string("Prolog,is,fun", ",", "", Parts).

Parts = ["Prolog", "is", "fun"].

```

### Примеры использования предикатов

1. \*\*Конкатенация строк:\*\*

```prolog

concatenate\_strings(S1, S2, Result) :-

atom\_concat(S1, S2, Result).

?- concatenate\_strings('hello', '\_world', R).

R = 'hello\_world'.

```

2. \*\*Извлечение подстроки:\*\*

```prolog

get\_substring(String, Start, Length, Sub) :-

sub\_atom(String, Start, Length, \_, Sub).

?- get\_substring('Prologisfun', 0, 6, Sub).

Sub = 'Prolog'.

```

3. \*\*Расщепление строки:\*\*

```prolog

split\_example :-

split\_string("Prolog;is;awesome", ";", "", Parts),

writeln(Parts).

?- split\_example.

["Prolog", "is", "awesome"]

```

4. \*\*Преобразование строки в список ASCII-кодов:\*\*

```prolog

string\_to\_ascii\_list(String, CodeList) :-

string\_codes(String, CodeList).

?- string\_to\_ascii\_list("hello", Codes).

Codes = [104, 101, 108, 108, 111].

```

5. \*\*Получение длины строки:\*\*

```prolog

length\_of\_string(String, Length) :-

string\_length(String, Length).

?- length\_of\_string("Prolog", L).

L = 6.

1. **Покажите, каким образом происходит построение стандартных комбинаторных объектов средствами Swi-prolog.**

В Prolog можно конструировать различные комбинаторные объекты, такие как списки, множества, перестановки, сочетания и графы. Рассмотрим, как можно использовать предикаты и встроенные функции в SWI-Prolog для построения и обработки этих объектов.

### 1. Списки

Списки являются одним из наиболее широко используемых комбинаторных объектов. В Prolog списки представляются в виде:

Prolog





[Head | Tail]

% Где Head — голова списка, а Tail — хвост списка

#### Примеры:

1. Создание списка:

Prolog





list\_example :-

List = [1, 2, 3, 4, 5],

writeln(List).

?- list\_example.

[1, 2, 3, 4, 5]

2. Конкатенация списков:

Prolog





concat\_lists(L1, L2, Result) :-

append(L1, L2, Result).

?- concat\_lists([1, 2], [3, 4], R).

R = [1, 2, 3, 4].

### 2. Множества

Множества в Prolog обычно представляются в виде списков с уникальными элементами. Предикаты из библиотеки ordsets помогают работать с отсортированными множествами, где элементы уникальны и находятся в порядке возрастания.

#### Примеры:

1. Создание множества:

Prolog





:- use\_module(library(ordsets)).

set\_example :-

ord\_list\_to\_ord\_set([3, 1, 2, 3, 4], Set),

writeln(Set).

?- set\_example.

[1, 2, 3, 4]

2. Объединение множеств:

Prolog





union\_sets(S1, S2, Union) :-

ord\_union(S1, S2, Union).

?- union\_sets([1, 2, 3], [3, 4, 5], U).

U = [1, 2, 3, 4, 5].

### 3. Перестановки

В Prolog можно использовать рекурсивные предикаты для генерации перестановок списка.

#### Пример:

Prolog





permutation([], []).

permutation(L, [H|T]) :-

select(H, L, R),

permutation(R, T).

?- permutation([1, 2, 3], P).

P = [1, 2, 3];

P = [1, 3, 2];

P = [2, 1, 3];

P = [2, 3, 1];

P = [3, 1, 2];

P = [3, 2, 1].

### 4. Сочетания

Сочетания из n по k элементов можно генерировать с помощью рекурсивных предикатов.

#### Пример:

Prolog





combination(0, \_, []).

combination(K, [X|Xs], [X|Ys]) :-

K > 0,

K1 is K - 1,

combination(K1, Xs, Ys).

combination(K, [\_|Xs], Ys) :-

K > 0,

combination(K, Xs, Ys).

?- combination(2, [a, b, c, d], C).

C = [a, b];

C = [a, c];

C = [a, d];

C = [b, c];

C = [b, d];

C = [c, d].

### 5. Графы

В Prolog графы можно представлять в виде списка ребер или в виде списков смежности. Существует библиотека ugraphs, которая помогает выполнять операции над графами.

#### Пример:

Prolog





:- use\_module(library(ugraphs)).

graph\_example :-

Edges = [1-2, 1-3, 2-4, 3-4],

vertices\_edges\_to\_ugraph([], Edges, Graph),

writeln(Graph).

?- graph\_example.

уграф(1-[2,3],2-[4],3-[4],4-[])

1. **Покажите основные принципы реализации переборных алгоритмов на графах**

Переборные алгоритмы на графах включают в себя такие алгоритмы, как поиск в глубину (DFS), поиск в ширину (BFS), поиск кратчайшего пути и другие. В этом обсуждении мы рассмотрим основные принципы реализации таких алгоритмов и как их можно применять с использованием SWI-Prolog.

### Основные принципы

1. Представление графа:

- Множество вершин и ребер.

- Списки смежности для представления связей между вершинами.

- Множество ребер в виде пар (начальная вершина - конечная вершина).

2. Поиск в глубину (DFS):

- Использует стек для хранения состояний посещения вершин.

- Рекурсивный обход графа.

3. Поиск в ширину (BFS):

- Использует очередь для хранения состояний посещения вершин.

- Итерирует по уровням графа.

4. Выбор структуры данных:

- Множество вершин: список или множество.

- Множество ребер: список или пара (начальная вершина - конечная вершина).

- Списки смежности: использовать библиотеку ugraphs для удобства работы.

### Примеры алгоритмов

#### 1. Представление графа с помощью списков смежности:

Prolog





% Пример графа

% Вершины: [1,2,3,4]

% Ребра: [1-2, 1-3, 2-4, 3-4]

% Используем библиотеку ugraphs

:- use\_module(library(ugraphs)).

% задаем граф как список ребер

graph\_edges([1-2, 1-3, 2-4, 3-4]).

#### 2. Поиск в глубину (DFS):

Prolog





% DFS предикат

dfs(Start, Graph, Path) :-

dfs(Start, Graph, [Start], Path).

dfs(Node, \_, Visited, Visited) :-

\+ member(Node, Visited). % Если достигли конца списка

dfs(Node, Graph, Visited, Path) :-

member(Node-Neighbor, Graph),

\+ member(Neighbor, Visited), % Проверяем, посещали ли эту вершину

dfs(Neighbor, Graph, [Neighbor | Visited], Path).

% Пример использования

dfs\_example :-

graph\_edges(Edges),

dfs(1, Edges, Path),

reverse(Path, CorrectPath), % Обратный порядок для корректного вывода

writeln(CorrectPath).

?- dfs\_example.

[1, 3, 4, 2]

#### 3. Поиск в ширину (BFS):

Prolog





% BFS предикат

bfs(Start, Graph, Path) :-

bfs([[Start]], Graph, Path).

bfs([[Node | Path] | \_], \_, [Node | Path]) :-

\+ member(Node, Path). % Если достигли конца списка

bfs([[Node | Path] | Paths], Graph, FinalPath) :-

findall([Neighbor, Node | Path],

(member(Node-Neighbor, Graph),

\+ member(Neighbor, [Node | Path])),

NewPaths),

append(Paths, NewPaths, Paths1),

bfs(Paths1, Graph, FinalPath).

% Пример использования

bfs\_example :-

graph\_edges(Edges),

bfs(1, Edges, Path),

reverse(Path, CorrectPath), % Обратный порядок для корректного вывода

writeln(CorrectPath).

?- bfs\_example.

[1, 2, 4]

1. **Раскройте понятия статические и динамические факты. Поясните на примерах принцип работы с динамическими фактами.**

### Статические Факты

Статические факты в Prolog — это факты, которые не изменяются во время выполнения программы. Они определяются в исходном коде и остаются неизменными на протяжении всего вызова программы.

#### Пример статических фактов:

Prolog





% Предикат "родитель"

родитель(джон, мария).

родитель(джейн, мария).

родитель(джон, том).

родитель(джейн, том).

% Предикат "мужчина"

мужчина(джон).

мужчина(том).

% Предикат "женщина"

женщина(джейн).

женщина(мария).

% Пример запроса

% кто является родителем Марии?

% ?- родитель(X, мария).

% X = джон ;

% X = джейн.

В этом примере факты "родитель", "мужчина" и "женщина" считаются статическими, потому что они определены заранее и не изменяются в процессе выполнения программы.

### Динамические Факты

Динамические факты в Prolog — это факты, которые можно добавлять, изменять или удалять во время выполнения программы. В SWI-Prolog для указания, что предикат динамический, используют директиву dynamic/1.

#### Пример объявления и работы с динамическими фактами:

Prolog





% Объявление предиката "друзья" как динамического

:- dynamic(друзья/2).

% Добавление фактов "друзья" во время выполнения программы

добавить\_друга(Человек1, Человек2) :-

assertz(друзья(Человек1, Человек2)).

% Удаление фактов "друзья" во время выполнения программы

удалить\_друга(Человек1, Человек2) :-

retract(друзья(Человек1, Человек2)).

% Пример использования динамических фактов

:- dynamic(друзья/2).

% Инициализация

инициализация :-

добавить\_друга(том, мэри),

добавить\_друга(джейн, карл),

добавить\_друга(том, сэм).

% Пример запроса

пример :-

инициализация,

writeln('Текущие друзья:'),

forall(друзья(X, Y), (write(X), write(' и '), writeln(Y))),

удалить\_друга(том, мэри),

writeln('Обновленные друзья:'),

forall(друзья(X, Y), (write(X), write(' и '), writeln(Y))),

добавить\_друга(том, мэри),

writeln('После повторного добавления:'),

forall(друзья(X, Y), (write(X), write(' и '), writeln(Y))).

% Запуск примера

?- пример.

#### Объяснение каждого шага:

1. Объявление предиката как динамического:

Prolog





:- dynamic(друзья/2).

Это объявляет предикат друзья/2 как динамический, то есть теперь можно добавлять и удалять факты этого типа.

2. Добавление фактов:

Prolog





добавить\_друга(Человек1, Человек2) :-

assertz(друзья(Человек1, Человек2)).

Здесь используется assertz/1, чтобы добавить новый факт в конец базы данных.

3. Удаление фактов:

Prolog





удалить\_друга(Человек1, Человек2) :-

retract(друзья(Человек1, Человек2)).

Здесь используется retract/1, чтобы удалить факт из базы данных.

4. Инициализация:

Prolog





инициализация :-

добавить\_друга(том, мэри),

добавить\_друга(джейн, карл),

добавить\_друга(том, сэм).

В этой инициализации добавляются несколько фактов для тестирования.

5. Запрос фактов и вывод результатов:

Prolog





пример :-

инициализация,

writeln('Текущие друзья:'),

forall(друзья(X, Y), (write(X), write(' и '), writeln(Y))),

удалить\_друга(том, мэри),

writeln('Обновленные друзья:'),

forall(друзья(X, Y), (write(X), write(' и '), writeln(Y))),

добавить\_друга(том, мэри),

writeln('После повторного добавления:'),

forall(друзья(X, Y), (write(X), write(' и '), writeln(Y))).

Этот код выполняет инициализацию, выводит текущие факты, затем удаляет и снова добавляет факты, выводя результаты на каждом шагу.

1. **Объясните принцип работы предикатов var, nonvar, atom, atomic, name, functor, arg, repeat.**

### 1. var/1

Предикат var/1 используется для проверки, является ли данный аргумент переменной.

#### Пример:

Prolog





?- var(X).

true.

?- var(5).

false.

?- var(a).

false.

### Принцип работы:

- Если аргумент является переменной, предикат возвращает true.

- Если аргумент не является переменной, предикат возвращает false.

### 2. nonvar/1

Предикат nonvar/1 проверяет, что аргумент не является переменной.

#### Пример:

Prolog





?- nonvar(X).

false.

?- nonvar(5).

true.

?- nonvar(a).

true.

### Принцип работы:

- Если аргумент не является переменной, предикат возвращает true.

- Если аргумент является переменной, предикат возвращает false.

### 3. atom/1

Предикат atom/1 проверяет, является ли аргумент атомом.

#### Пример:

Prolog





?- atom(a).

true.

?- atom('hello').

true.

?- atom(Var).

false.

?- atom(123).

false.

### Принцип работы:

- Если аргумент является атомом, предикат возвращает true.

- Если аргумент не является атомом, предикат возвращает false.

### 4. atomic/1

Предикат atomic/1 проверяет, является ли аргумент атомом или числом.

#### Пример:

Prolog





?- atomic(a).

true.

?- atomic(123).

true.

?- atomic(Var).

false.

?- atomic([a, b]).

false.

### Принцип работы:

- Если аргумент является атомом или числом, предикат возвращает true.

- Если аргумент не является атомом или числом, предикат возвращает false.

### 5. name/2 (SWI-Prolog использует эквивалентный предикат atom\_chars/2 или atom\_codes/2)

Предикат name/2 преобразует атом в список символов или наоборот.

#### Пример:

Prolog





% В SWI-Prolog

?- atom\_chars(hello, X).

X = [h, e, l, l, o].

?- atom\_chars(A, [h, e, l, l, o]).

A = hello.

### Принцип работы:

- Если первый аргумент — атом, он преобразуется в список символов и сравнивается со вторым аргументом.

- Если первый аргумент — список символов, он преобразуется в атом и сравнивается с первым аргументом.

### 6. functor/3

Предикат functor/3 извлекает/устанавливает имя и арность (количество аргументов) структуры.

#### Пример:

Prolog





?- functor(point(1,2), F, A).

F = point,

A = 2.

?- functor(T, f, 3).

T = f(\_, \_, \_).

### Принцип работы:

- Если первый аргумент — структура, предикат возвращает её имя и арность.

- Если первый аргумент — переменная, предикат создаёт структуру с указанным именем и арностью.

### 7. arg/3

Предикат arg/3 извлекает аргумент структуры по его позиции.

#### Пример:

Prolog





?- arg(2, point(1,2,3), Arg).

Arg = 2.

?- arg(1, date(2023, 10, 31), Arg).

Arg = 2023.

### Принцип работы:

- Позиция (целое число) передаётся в качестве первого аргумента.

- Структура передаётся в качестве второго аргумента.

- Аргумент, находящийся на указанной позиции, извлекается в третий аргумент.

### 8. repeat/0

Предикат repeat/0 бесконечно генерирует решения, то есть возвращает true неограниченное количество раз.

#### Пример:

Prolog





?- repeat, write('Hello, World!'), nl, fail.

Hello, World!

Hello, World!

Hello, World!

... (бесконечный повтор)

### Принцип работы:

- repeat/0 всегда успешен и вызывает сам себя, генерируя новую точку возврата.

- Полезен в комбинации с другими предикатами для создания циклов или пробных решений.