# Логическое программирование

Пётр Лозов
lozov.peter@gmail.com

12.05.2023

# Историческая справка

| | |
|---|---|
| **1951** | Альфред Хорн выделил и изучил дизъюнкты специального вида, который назвали в честь него (*Америка*) |
| **1970-1975** | Роберт Ковальски показал, что логическое доказательство может быть вычислено[1] (*Великобритания*) |
| **1972** | Ален Колмероэ и Филипп Руссель разработали язык Пролог (*Франция*) |
| **1982-1992** | Один из основных языков разработки базы знаний в национальном проекте “Пятое поколение компьютеров” (*Япония*) |

[1] Predicate Logic as a Programming Language, University of Edinburgh, 1973

# Programmation en logique

- Prolog — аббревиатура от “*programmation en logique*”
- **Язык** и **система** логического программирования
  - Хорновские выражения (предикаты первого порядка)
  - Метод резолюции
  - Унификация
  - Поиск с ограничениями в древовидной структуре
    - Возможно, бесконечной
- Язык декларативного программирования
- Существует множество диалектов
  - Datalog
  - $\alpha$Prolog
  - $\lambda$Prolog
  - . . .

# В индустрии

- IBM Watson
  - Система синтеза ответов для вопросов на естественном языке
  - Сопоставление с образом на деревьях разбора естественных языков
- GeneXus
  - Визуальный кроссплатформенный инструмент разработки
  - Преобразование визуальной программы в код для конкретной платформы
- TerminusDB
  - Графовая база данных
  - Похожа на git
  - Реализована на Prolog

# Где попробовать

- SWI-Prolog
  - Свободная реализация интерпретатора Пролога
  - Есть web-версия
    - https://swish.swi-prolog.org/
- Есть реализация интерпретатора под .NET
  - https://github.com/Slesa/Prolog.NET
- Минималистичная реализация на OCaml
  - https://github.com/dboulytchev/microProlog
- ...

# Начнем с основ

- Программа (база знаний) представляет из себя набор фактов и правил вывода
- Факты и правила можно рассматривать как отношения с математической точки зрения
  - На первый взгляд это функции, возвращающие bool
- Для старта вычисления необходим запрос к базе знаний
- Результатом являются уточнения запроса, выводимые из базы знаний

# Простой пример[2]

```
loves(vincent, mia).
loves(marcellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).

jealous(X, Y) :-
    loves(X, Z),
    loves(Y, Z).
```

[2] https://swish.swi-prolog.org/example/kb.pl

# Простой пример[2]

```
?- loves(vincent, mia) -> true
```

```
loves(vincent, mia).
loves(marcellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).

jealous(X, Y) :-
    loves(X, Z),
    loves(Y, Z).
```

[2] https://swish.swi-prolog.org/example/kb.pl

# Простой пример[2]

```
?- loves(vincent, mia) -> true

?- loves(vincent, pumpkin) -> false
```

```
loves(vincent, mia).
loves(marcellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).

jealous(X, Y) :-
    loves(X, Z),
    loves(Y, Z).
```

[2] https://swish.swi-prolog.org/example/kb.pl

# Простой пример[2]

```
loves(vincent, mia).
loves(marcellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).

jealous(X, Y) :-
    loves(X, Z),
    loves(Y, Z).
```

```
?- loves(vincent, mia) -> true

?- loves(vincent, pumpkin) -> false

?- loves(X, mia) ->
    X = vincent;
    X = marcellus
```

[2] https://swish.swi-prolog.org/example/kb.pl

# Простой пример[2]

```
loves(vincent, mia).
loves(marcellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).

jealous(X, Y) :-
    loves(X, Z),
    loves(Y, Z).
```

```
?- loves(vincent, mia) -> true

?- loves(vincent, pumpkin) -> false

?- loves(X, mia) ->
    X = vincent;
    X = marcellus

?- jealous(X, Y) ->
    X = Y, Y = vincent;
    X = vincent, Y = marcellus;
    X = marcellus, Y = vincent;
    X = Y, Y = marcellus;
    X = Y, Y = pumpkin;
    X = Y, Y = honey_bunny
```

[2] https://swish.swi-prolog.org/example/kb.pl

# Ещё простой пример

```
append([], X, X).
append([X|Xs], Y, [X|Zs]) :-
  append(Xs, Y, Zs).
```

# Ещё простой пример

```
?- append([a], [b,c], [a,b,c]) -> true
```

```
append([], X, X).
append([X|Xs], Y, [X|Zs]) :-
  append(Xs, Y, Zs).
```

# Ещё простой пример

```
?- append([a], [b,c], [a,b,c]) -> true

?- append([a], [b,c], X) ->
    X = [a,b,c]
```

```
append([], X, X).
append([X|Xs], Y, [X|Zs]) :-
  append(Xs, Y, Zs).
```

# Ещё простой пример

```
append([], X, X).
append([X|Xs], Y, [X|Zs]) :-
  append(Xs, Y, Zs).
```

```
?- append([a], [b,c], [a,b,c]) -> true

?- append([a], [b,c], X) ->
    X = [a,b,c]

?- append(X, [c], [a,b,c]) ->
    X = [a,b]
```

# Ещё простой пример

```
append([], X, X).
append([X|Xs], Y, [X|Zs]) :-
  append(Xs, Y, Zs).
```

```
?- append([a], [b,c], [a,b,c]) -> true

?- append([a], [b,c], X) ->
    X = [a,b,c]

?- append(X, [c], [a,b,c]) ->
    X = [a,b]

?- append(X, Y, [a,b,c]) ->
    X = [], Y = [a,b,c];
    X = [a], Y = [b,c];
    X = [a,b], Y = [c];
    X = [a,b,c], Y = [];
```

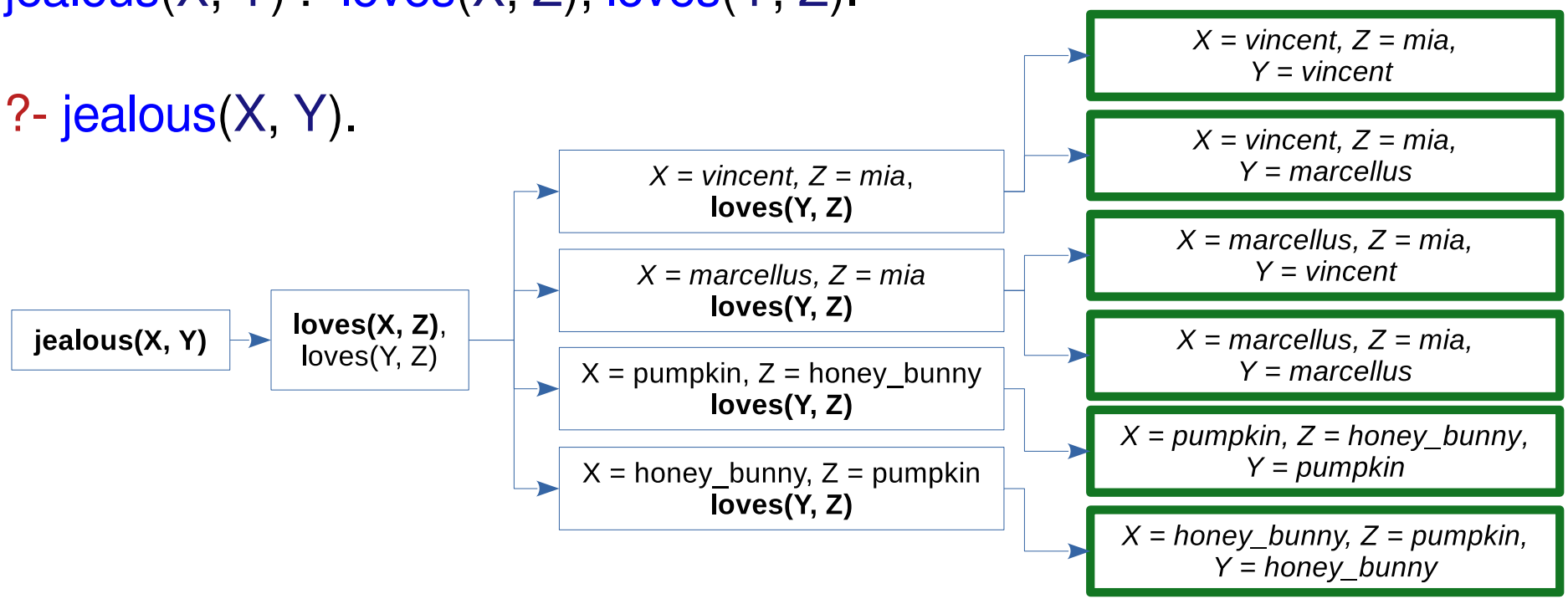
# Как это работает

- Построение вывода от запроса к фактам
  - Метод резолюций
  - Перебор возможных путей поиском в глубину с возвратами
- Унификация
  - Уточнение переменных
  - Проверка, что запрос вложен в факт или в голову правила вывода

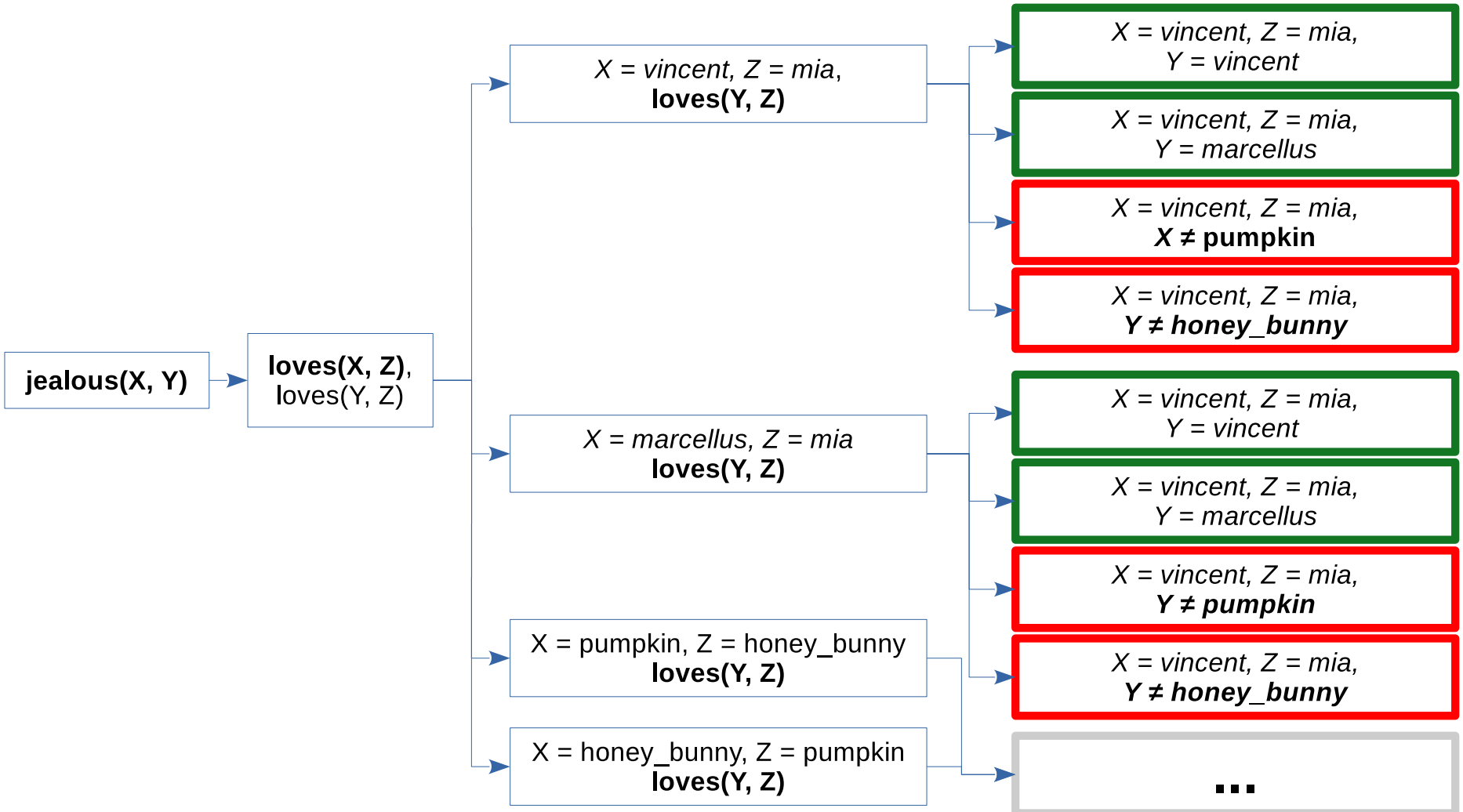
# Дерево поиска

```
loves(vincent, mia).
loves(marcellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
jealous(X, Y) :- loves(X, Z), loves(Y, Z).

?- jealous(X, Y).
```



# Дерево поиска



# Унификация: определения

- Термы $\mathcal{T} = \mathcal{V} \mid \mathcal{C}(\mathcal{T}, \ldots, \mathcal{T})$
  - Функциональные символы (конструкторы) и переменные

# Унификация: определения

- Термы $\mathcal{T} = \mathcal{V} \mid \mathcal{C}(\mathcal{T}, \ldots, \mathcal{T})$
  - Функциональные символы (конструкторы) и переменные
- Подстановка $\Sigma : \mathcal{V} \to \mathcal{T}$
  - Отображение из имен переменных в термы

# Унификация: определения

- Термы $\mathcal{T} = \mathcal{V} \mid \mathcal{C}(\mathcal{T}, \ldots, \mathcal{T})$
  - Функциональные символы (конструкторы) и переменные
- Подстановка $\Sigma : \mathcal{V} \to \mathcal{T}$
  - Отображение из имен переменных в термы
- Применение подстановки $t\Sigma$
  - Замена в терме $t$ переменных на термы в соответствии с подстановкой $\Sigma$
  - node(X, Y) $\{$ X $\leftarrow$ node(leaf, Z) $\}$ = node(node(leaf, Z), Y)

# Унификация: определения

- Термы $\mathcal{T} = \mathcal{V} \mid \mathcal{C}(\mathcal{T}, \ldots, \mathcal{T})$
  - Функциональные символы (конструкторы) и переменные
- Подстановка $\Sigma : \mathcal{V} \to \mathcal{T}$
  - Отображение из имен переменных в термы
- Применение подстановки $t\Sigma$
  - Замена в терме $t$ переменных на термы в соответствии с подстановкой $\Sigma$
  - node(X, Y) $\{$ X $\leftarrow$ node(leaf, Z) $\}$ = node(node(leaf, Z), Y)
- Отношение частичного порядка над подстановками
  - $\Sigma_1 < \Sigma_2$, если $\exists\, \Sigma\ \forall\, t \in \mathcal{T} \quad t\Sigma_1\Sigma = t\Sigma_2$
  - Подстановка меньше, если содержит меньше информации

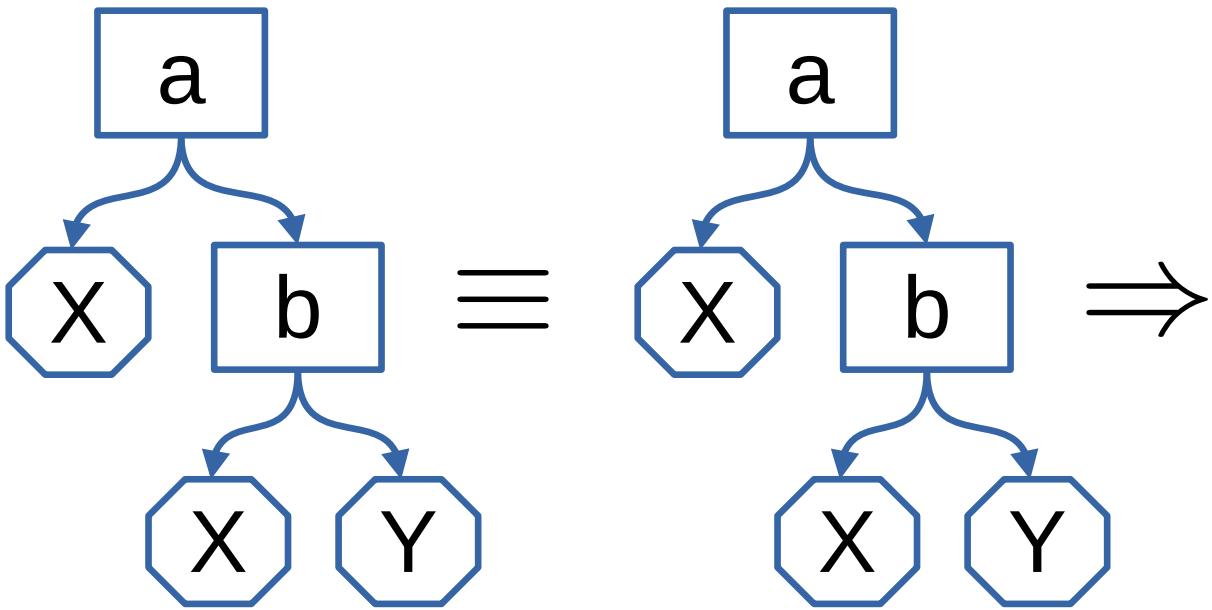
# Унификация

- Унификатор для термов $t_1$ и $t_2$
  - Подстановка $\Sigma$ такая, что $t_1\,\Sigma = t_2\,\Sigma$
  - Унификатора может не быть
  - Унификаторов может быть много

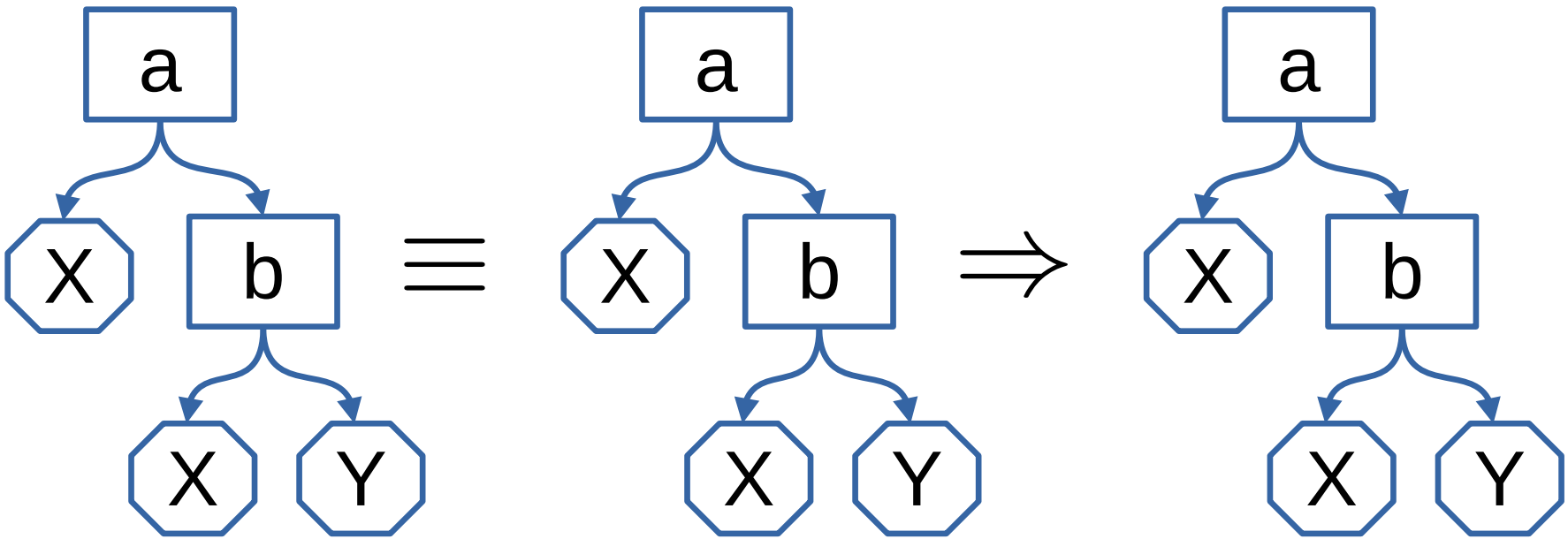
# Унификация

- Унификатор для термов $t_1$ и $t_2$
  - Подстановка $\Sigma$ такая, что $t_1\,\Sigma = t_2\,\Sigma$
  - Унификатора может не быть
  - Унификаторов может быть много
- Унификация *unify* : $\mathcal{T} \to \mathcal{T} \to \Sigma$ *option*
  - Построение наименьшего унификатора для двух термов
  - На самом деле, *unify* : $\Sigma$ *option* $\to \mathcal{T} \to \mathcal{T} \to \Sigma$ *option*
  - В худшем случае экспоненциальна
  - Сопоставление с образцом — частный случай унификации

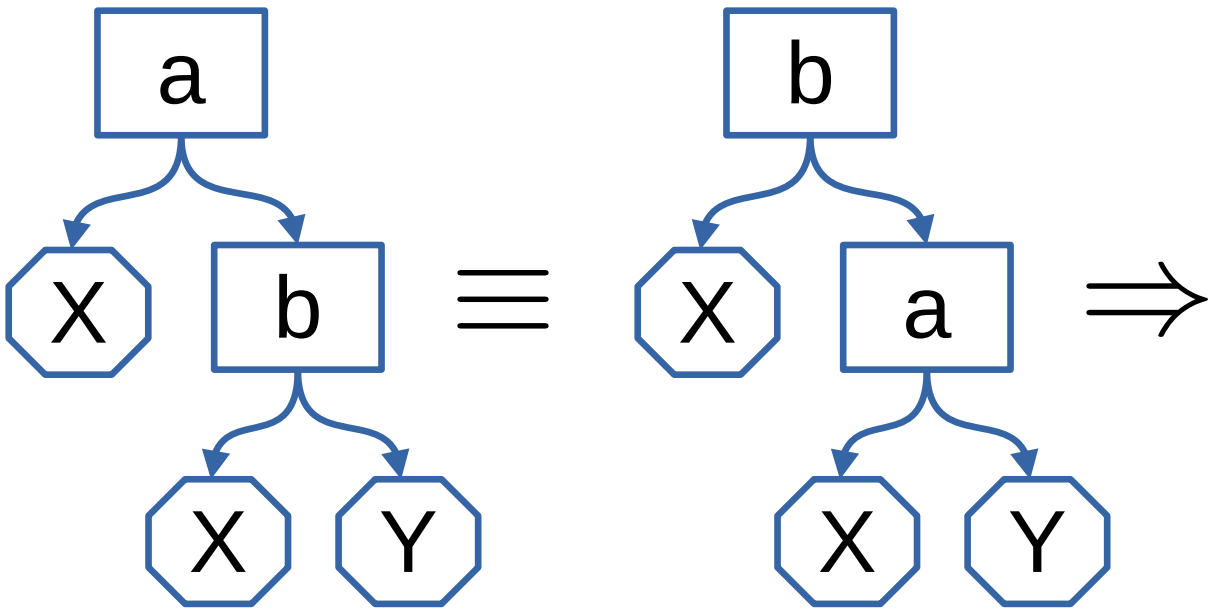
# Унификация: пример 1



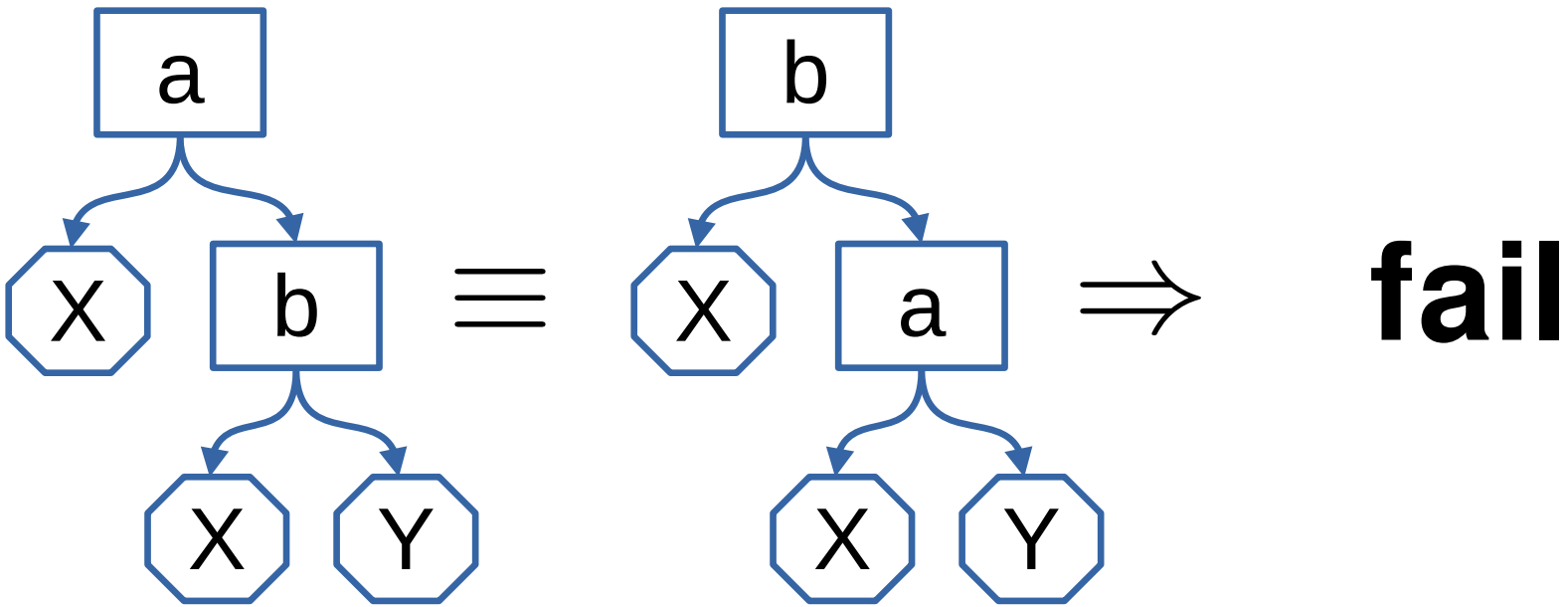
# Унификация: пример 1



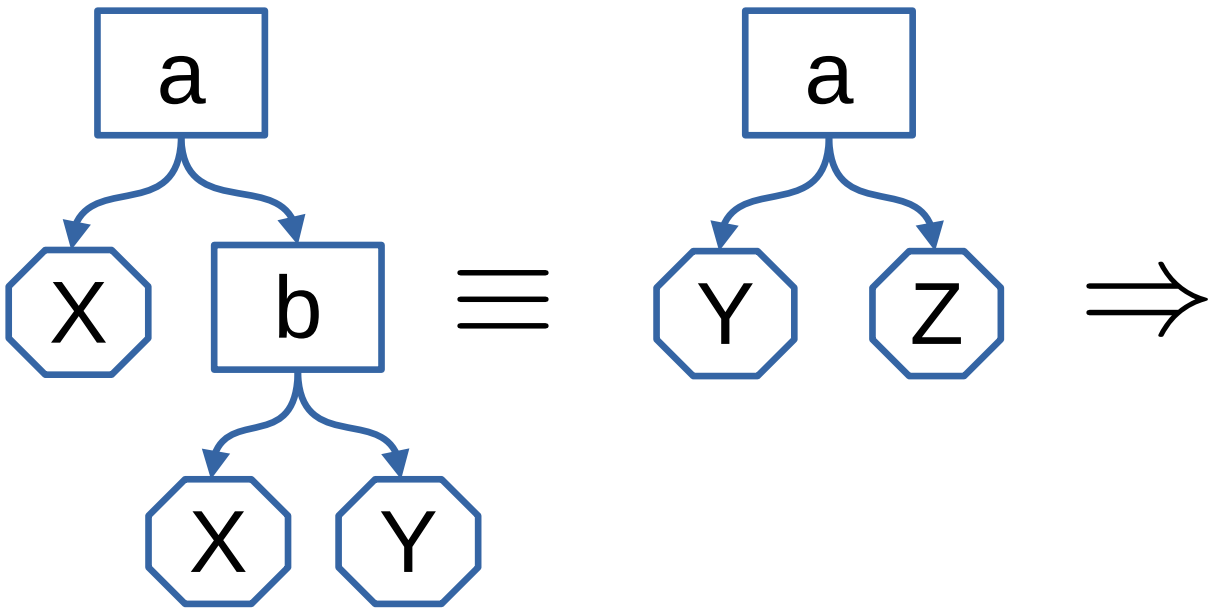
# Унификация: пример 2



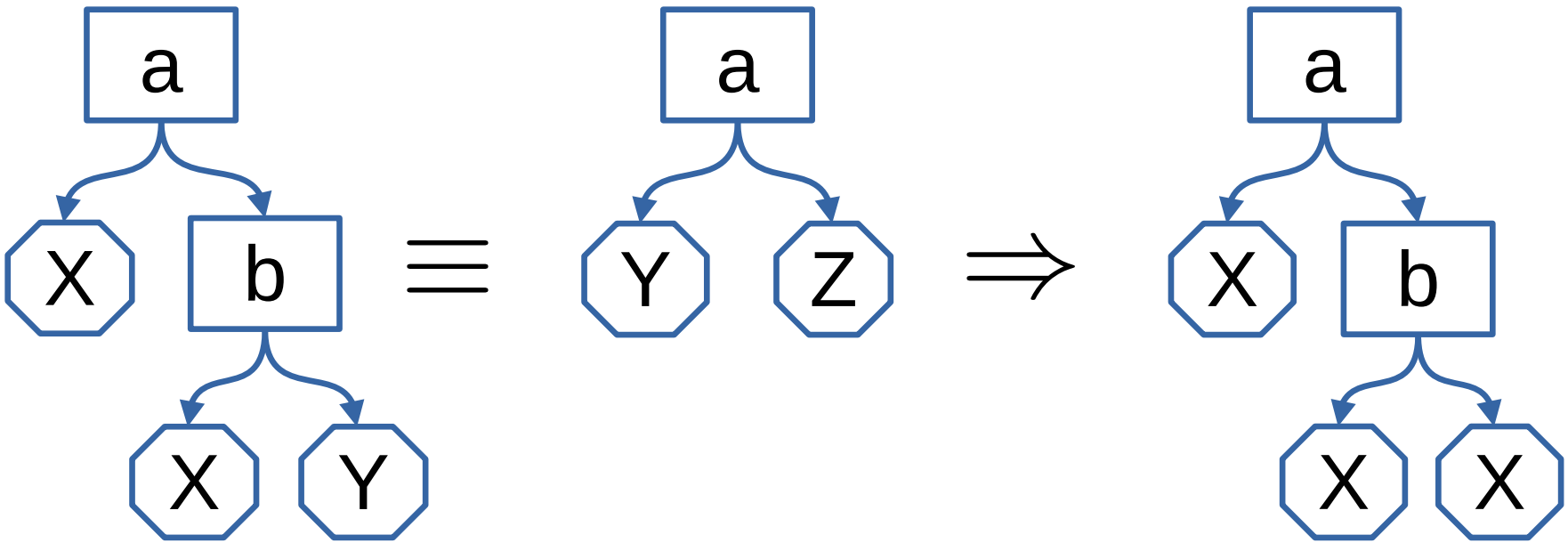
# Унификация: пример 2



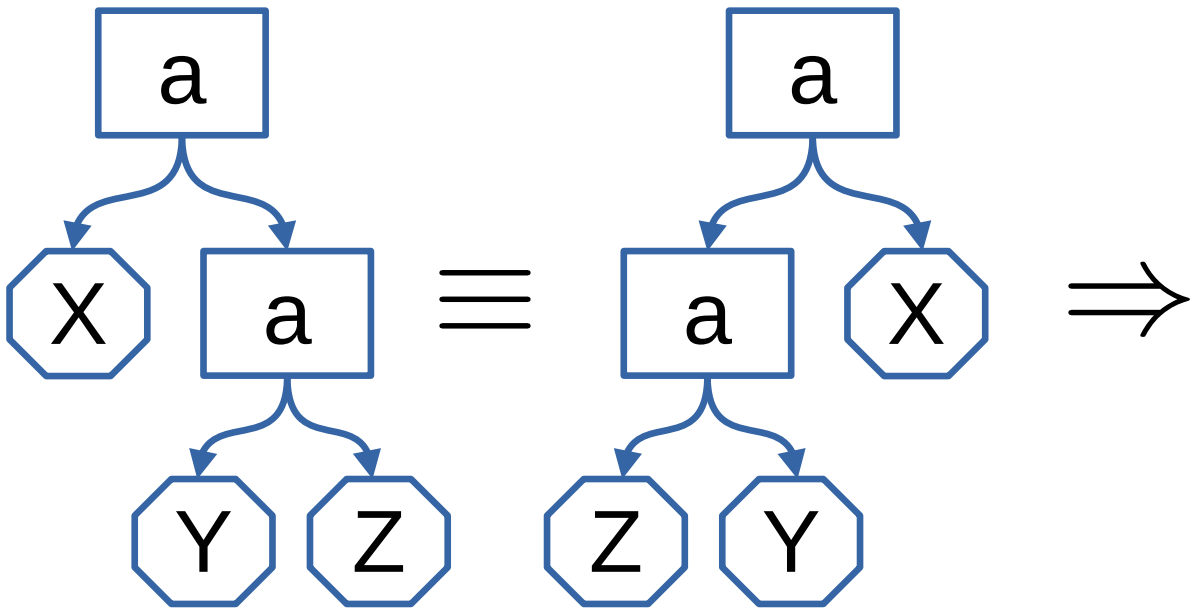
# Унификация: пример 3



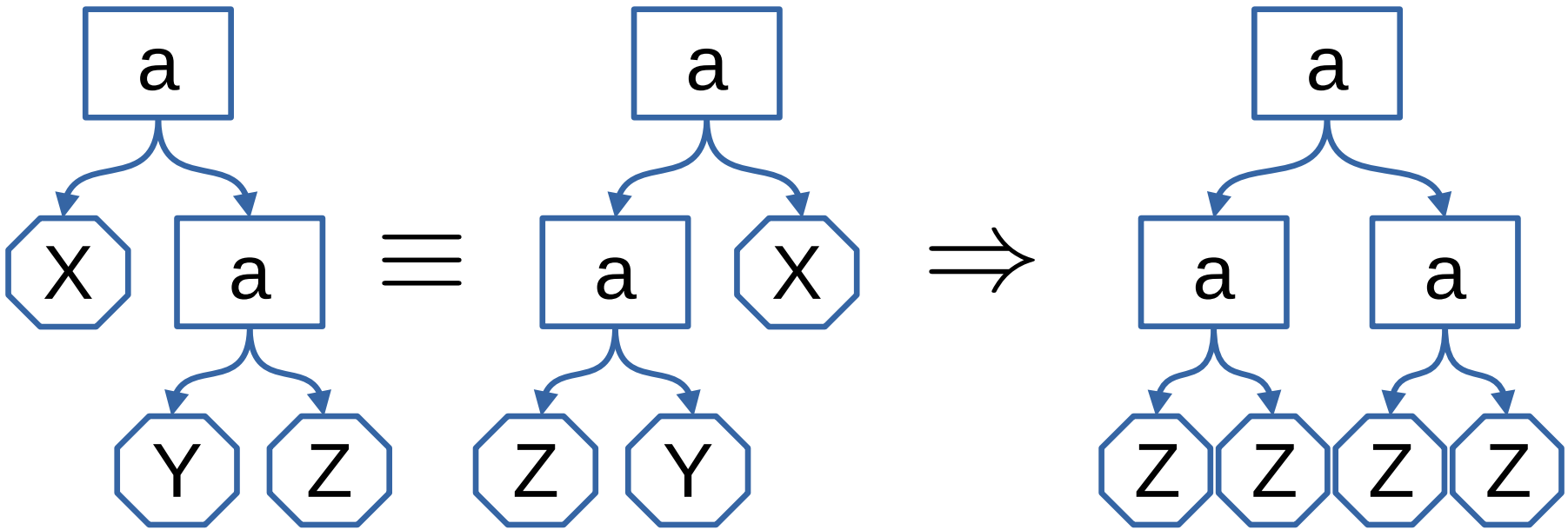
# Унификация: пример 3



# Унификация: пример 4



# Унификация: пример 4



# Унификация: Occurs Check

$$X \equiv f(X)$$

# Унификация: Occurs Check

$$X \equiv f(X)$$

- Некорректно в терминах логики первого порядка
  - Результат такой унификации — *fail*
  - Детекция вложенности термов замедляет алгоритм унификации
  - Не детектируется в большинстве реализаций Пролога

# Унификация: Occurs Check

$$X \equiv f(X)$$

- Некорректно в терминах логики первого порядка
  - Результат такой унификации — *fail*
  - Детекция вложенности термов замедляет алгоритм унификации
  - Не детектируется в большинстве реализаций Пролога
- Конечное представление бесконечного терма
  - Вероятно, существуют задачи, где такое описание может быть полезным

# Неполный поиск

```
:- use_rendering(svgtree).

member(X, leaf(X)).
member(X, node(L, _)) :- member(X, L).
member(X, node(_, R)) :- member(X, R).
```

- Проверка наличия значения в листьях дерева

# Неполный поиск

```
:- use_rendering(svgtree).

member(X, leaf(X)).
member(X, node(L, _)) :- member(X, L).
member(X, node(_, R)) :- member(X, R).
```

- Проверка наличия значения в листьях дерева
- А если попробовать синтезировать все деревья, содержащие заданное значение в листьях?

# Внелогические операторы

- Оператор отсечения (!)
  - Запрещает поиск других решений, если одно решение найдено
  - Позволяет управлять поиском
  - Недекларативен
- assert и retract
  - Операции динамического добавления и удаления фактов и правил из базы знаний
  - Позволяют писать мета-программы, изменяющие сами себя в процессе исполнения
- Negation as Failure
- И многие другие