GeekBrains

Логическое программирование (лекция 5)

Парадигмы программирования и языки парадигм

Оглавление

Вступление

Добрый день, коллеги! Меня зовут Александр Левин. Поздравляю вас, вы дошли до последней лекции курса “Парадигмы программирования и языки парадигм”! Сегодня мы изучим очень интересную - логическую парадигму (логическое программирование, ЛП). План на сегодня:

- Обзор логического программирования
    - Контекст и определения
    - Примеры задач

    - Сферы применения
    - Языки программирования
- История вкратце
- Примеры кода
- Преимущества и недостатки
- Подведение итогов

Если после окончания лекции у вас останутся вопросы, вы как всегда сможете написать их мне в Telegram по контакту, который я оставлю в конце лекции.

Термины, используемые в лекции

Логическое программирование — это парадигма программирования, которая основана на формальное логике

Логическое высказывание — это

Логический вывод — это

Предикат — это

Формальная логика — это

Логика предикатов — это

Обзор логического программирования

Начинаем развеивать загадочность над логическим программированием.

Контекст и определения

В предыдущей лекции мы разбирали функциональное программирование. **Логическое программирование**, также как и функциональное является **декларативным**. То есть, программы **логического программирования** представляют собой верхнеуровневое описание некого результата, который мы хотим получить, а последовательность шагов при этом реализовывается машиной где-то под капотом. Но если в функциональном программировании “черными ящиками” мы называли математические функции (или “отображения”), то в случае логического программирования желаемый результат описывается с помощью **формальной логики**. Соответственно, ваша программа состоит из **логических высказываний** и **правил вывода**, а исполнение представляет собой процесс **логического вывода**.

Рассмотрим несколько определений:

- 💡 **Логическое программирование** — это декларативная парадигма программирования, которая основана на представлении программ в виде правил вывода и фактов (базы знаний), на языке **формальной логики**.
- 💡 **Логическое программирование** — совокупность идей и понятий программирования, основанная на автоматическом доказательстве теорем; а также раздел дискретной математики, изучающий принципы **логического вывода** информации на основе заданных фактов и правил вывода.
- 💡 **Формальная логика** - это наука, занимающаяся анализом структуры высказываний и доказательств, обращающая основное внимание на форму в отвлечении от содержания.

Основное удобство логического программирования заключается в упрощении синтаксиса. Поскольку **формальная логика** (в частности, **математическая логика**) - это известный и отработанный инструмент, можно даже сказать - “язык”, с которым хорошо знакомо большое количество людей, то удобно на нем описывать логические выражения. Помимо этого, на **языке формальной логики** описывать желаемый результат гораздо проще, чем на языке программирования, поскольку формальная логика ближе по форме к человеческому мышлению, не говоря о

кейсах, когда задача с трудом описывается в императивном стиле. Например, задача для программмы может формулироваться в форме “докажи, что утвереждение X верно, если известно что верно Y, Z” или “найди такие значения, для которых верны утверждения A, B, C”. Согласитесь, что довольно удобно формулировать задачу в таком виде! Хотя это конечно зависит от задачи ;). И совсем скоро мы разберем несколько примеров для которых это высказывание верно.

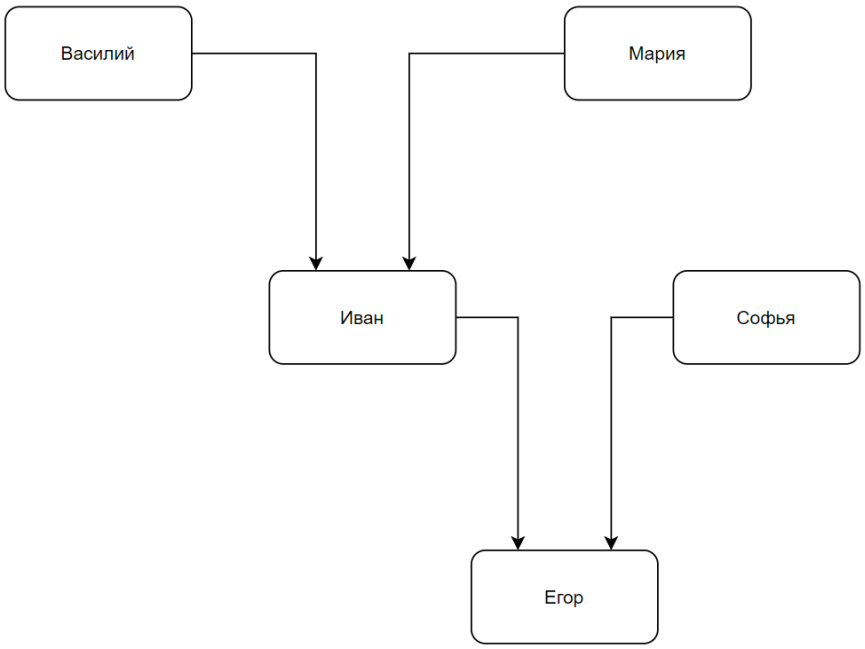
Таким образом, **логическое программирование** является удобным интерфейсом, через который можно общаться с компьютером через какой-то синтаксис (в завимости от языка программирования), основанный на **формальной логике**. И не нужно думать о том, как будет происходить поиск решения - логический язык сделает это за вас, на основе имеющихся фактов и правил вывода. Программисту остается только описать логическую модель в терминах объектов, их свойств и отношений между ними (да, в этом есть сходство с ООП).

Теперь, разберем ещё одну важную концепцию, которая лежит в основе логического программирования - логику предикатов. **Логика предикатов** - это **формальная система**, которая используется для рассуждений о свойствах и отношениях между объектами. В данной системе объекты описываются с помощью переменных, а отношения между объектов - с помощью предикатов и правил вывода. **Предикат** — это высказывание, которое может быть истинным или ложным в зависимости от значений переменных. **Логика предикатов** отличается от формальной логики введением именно *этого* понятия (**предиката**). Также, помимо уже известных логических операторов вводятся кванторы: всеобщности ∀ и существования ∃ (обычно читаются как “для всех” и “существует такой” соответственно). Кванторы и предикаты значительно расширяют базовый функционал формальной логики, и таким образом, **логика предикатов** является необходимым как бы *промежуточным* шагом в переходе от **формальной логики** к **логическому программированию**.

По уровню описания задачи, **логическая парадигма** находится ещё выше функциональной: ещё дальше от конкретной реализации и ещё ближе к описанию результата. Так, **логическая парадигма** помогает бороться с возврастающе сложным синткасисом программ, а значит и снижает количество допускаемых ошибок. Конечно, в общем случае, это упрощение происходит за счет производительности. В общем случае, самый популярный логический язык **Prolog** по скорости сопоставим с языком **Python**, а иногда даже медленнее него.

Пример: генеалогия

Как же выглядят задачи, решаемые с помощью логической парадигмы? Давайте разберем несколько примеров. Первый простейший пример - определение родственника по генеалогическому древу. Скажем, существует человек с именем Иван, у него есть два родителя: Василий и Мария. Также у него есть жена - Софья и сын - Егор.



И мы, зная эту информацию, хотим получить ответ на следующий вопрос: как зовут бабушку и дедушку Егора по Ивану? Попробуйте дать ответ на этот вопрос самостоятельно.

Ответ, конечно же: Мария и Василий. Скорее всего, глядя на картинку или просто прочитав условие задачи вы быстро дали правильный ответ. И скорее всего, для вас он являлся очевидным, но вопрос в том почему ваш ответ был именно таким? Вы даже не задумались о том, как сделали несколько шагов в формальной логике. Во-первых, вы знали определение слов “бабушка” и “дедушка” - это родители родителя женского и мужского пола соответственно. Во-вторых, вы сделали несколько логических переходов. Если вы смотрели на картинку, то нашли родителей Егора, увидели там Ивана, нашли родителей Ивана и просто перечислили их. Если на текст, то точно также вы уже знали, что у Ивана было два родителя и знали их имена. А также вы знали, что Егор является сыном Ивана, следовательно бабушка и дедушка Егора по Ивану - это просто родители Ивана.

Эту же задачу (и другие подобные ей) можно легко решить с помощью логического программирования! Ведь очень удобно было бы ограничиться описанием фактов, правил перехода и четко-сформулированной задачей, вместо разработки сложных

алгоритмов, описанных на длинных полотнах кода. Вместо этого, логический язык программирования получит четкие инструкции и найдет ответ самостоятельно. В рамках данного примера логическо решение выглядело бы следующим образом:

1. Перечисляем начальные факты:
    a. Иван является родителем Егора мужского пола
    b. Софья является родителем Егора женского пола
    c. Василий является родителем Ивана мужского пола
    d. Мария является родителем Ивана женского пола
2. Даем определения (правила переходов):
    a. Бабушка человека по родителю X - это родитель женского пола родителя X
    b. Дедушка человека по родителю X - это родитель мужского пола родителя X
3. Формулируем задачу (задаем вопрос):
    a. Кто является бабушкой Егора по Ивану?
    b. Кто является дедушкой Егора по Ивану?

Ключевой момент: основная изюминка данного подхода заключается в том, что я не передавал программе информацию о том кто является бабушкой и дедушкой Егора напрямую. Я передал исключительно информацию о родителях + определил понятие “бабушек” и “дедушек”. Зная это, машина может сама проделать те же операции что делали вы, когда минуту назад смотрели на эту задачу. Вот в этом и заключается сила данного подхода.

Конечно, в этом простом примере, для нас ответ был очевиден. Но иногда, при решении задач, мы не знаем ответ сходу, но зато знаем по какой логике работает контекст, в котором этот ответ можно получить. При этом мы не хотим разрабатывать сложный алгоритм, а вместо этого готовы описать эту самую логику и передать в компьютер. В этом случае, наш выбор - это **логическое программирование**.

Кстати, теперь вы видите, почему данная парадигма является декларативной: программа описана скорее в виде результата, который необходимо получить, чем в виде последовательности шагов, плюс у нас нету доступа к памяти. Также сюда же можно добавить отсутствие порядка в перечислении фактов и правил. Неважно про что вы напишете раньше: про то, что Софья является матерью Егора или про то, что Мария является матерью Ивана.

Пример: Судоку

Менее тривиальный пример (решение которого сходу неочевидно) и хорошо подходящий для демонстрации логического программирования - решение Судоку. Судоку - это японская головоломка, состоящая из клетчатого поля 9 на 9. В некоторых клетках этого поля уже стоят какие-то цифры. Стороны судоку разделены на три равные части, образуя 9 “мини-квадратов” со стороной 3 x 3. Такие “мини-квадраты” будем называть “блоками”.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Решение головоломки - это полностью заполненный Судоку, то есть в каждой клетке должна стоять ровно одна цифра, с выполнением следующих условий:

- Каждая строка содержит все цифры от 1 до 9
- Каждый столбец содержит все цифры от 1 до 9
- Каждый блок 3 x 3 содержит все цифры от 1 до 9

Существует великое множество статей на тему Судоку и многих вопросов с ним связанных. Например, не любой предзаполненный Судоку возможно решить, то есть, есть такие способы составить головоломку, чтобы невозможно было заполнить его верно. А ещё решений может быть больше одного. Но это тема для отдельного курса или вашей следующей исследовательской работы.

В интернете вы можете найти множество алгоритмов для решения данной задачи. Но поскольку мы рассматриваем **логическое программирование**, нам интересен не столько алгоритм решения, сколько способ описать задачу. Сейчас мы попробуем это сделать псевдокодом похожим на логический стиль.

```
 1 Sudoku:
 2     - имеет сторону len: 9
 3     - содержит строки rows: [1,2,3,4,5,6,7,8,9]
 4     - содержит столбцы cols: [a,b,c,d,e,f,g,h,i]
 5     - клетка cell лежит по координатам из [rows, cols]
 6     - каждый cell содержит значение value [1..9] или NULL
 7     - содержит мини-квадраты blocks:
 8     [
 9         block_1: [a, b, c] x [1, 2 ,3]
10         block_2: [d, e, f] x [4, 5, 6]
11         block_3: [g, h, i] x [7, 8, 9]
12     ]
13 Solved_Sudoku:
14     - for each rows, cols, blocks:
15         список value = списку цифр [1..9]
16
17 Task:
18     - получить Sudoku, для которого верно:
19         each cell ≠ NULL
```

Конечно, псевдокод - это не язык программирования, поэтому эту же задачу мы решим позже в разделе “Примеры кода”. А пока, давайте разберем этот псевдокод более детально.

- Строка 1: даем предикату название Sudoku
- Строка 2: говорим, что у судоку сторона должна быть равна 9
- Строки 3 и 4: даем имена столбцам и строкам
- Строка 5: определяем cell
- Строка 6: говорим, что cell содержит значение в интервале от 1 до 9
- Строка 7: создаем и определяем предикат blocks
    - Строки 8 - 12: говорим что такое blocks, а именно, из каких строк и столбцов он состоит
- Строки 13 - 15: говорим, что список значений в rows, cols и blocks должен совпадать со списком цифр от 1 до 9 (без строгого сохранения порядка)
- Строки 17 - 19: пишем запрос о том, что хотим получить такой предикат Sudoku, для которого все клетки не пусты.

Естественно, реализация в логической парадигме может существенно отличаться от данного псевдокода. Но важно то, что при его написании сохраняются основные принципы логического стиля - программа состоит из высказываний (правил и фактов), которые могут быть истинны или ложны, а решение задачи - это доказательство верности тех или иных высказываний с использованием исходных фактов и множества правил для их выведения.

Сферы применения

Несмотря на то, что на сегодняшний день **логическая парадигма** не входит в топы рейтингов по популярности, она применяется во многих областях, в том числе в следующих:

- Анализ данных и искусственный интеллект
    - Обработка естественного языка - например, анализ тональности или генерация текстов
- Теория вычислений
- Метапрограммирования
- Разработка языков программирования
- Разработка экспертных систем

Также, оно является крайне удобным инструментом при

- прототипировании сложных систем,

особенно, на ранних этапах. То есть, когда необходимо быстро разработать и показать сложноустроенный функционал, который использует множество взаимосвязанных факторов и переменных. Например, также как в примере выше, вас попросили показать работоспособность какого-нибудь кастомного алгоритма поиска решенного Судоку, а реализовывать его с нуля у вас нет времени.

Языки программирования

Давайте перечислим какие языки существуют в логической парадигме.

Чистые логические языки:

- **Prolog** - самым известным из логических языков является язык **Prolog** (Пролог), который был разработан в 1970-х годах и используется как в академических кругах, так и в индустрии. Но конечно он далеко не единственный в своем роде логический язык.
- **Datalog** - декларативный логический язык, очень похож на Prolog, в основном используется в предобработке данных, извлечении информации, работе с сетями. Особенно активно - для работы с “дедуктивными базами данных”.
- **CHR** (Constraint Handling Rules) - выпущен в 1991 году, декларативный язык, используется для решения задач с ограничениями (англ.- constraints). Саму программу называют “обработчик ограничений” (англ. - “constraint handler”), а внутри приводится список ограничений. Используется также как и **Oz** - для ИИ, робототехники, распределенных систем и других сложных задач где требуется высокая скорость.

Языки поддерживающие логическую парадигму:

- **Mercury** - выпущен в 1995 году, поддерживает функциональную и логическую парадигмы. С точки зрения синтаксиса он является гибридом Haskell и **Prolog**. Основное отличие Mercury от Prolog - скорость, т.к. он был создан, чтобы стать быстрее **Prolog-а**. Используется для реализации сложных программ в таких областях как авиация и транспорт.
- **Oz** - выпущен в 1991 году, мультипарадигменный язык. Его основное отличие в том, что он поддерживает **логическую**, **функциональную**, **структурную** и **процедурную** парадигмы, и даже **ООП**! Применяется в основном в сфере ИИ, робототехники и распределенных вычислений.

История вкратце

Теперь вкратце обсудим как и почему появилась логическая парадигма. Парадигма логического программирование появилась сравнительно недавно.

Теоретическая основа

До создания логических языков, как всегда - было слово. Для того, чтобы появились языки программирования, в которых алгоритмы решения выстраиваются “автоматически” для заданной задачи, сначала необходимо доказать что это вообще возможно сделать хотя бы в теории. Таким образом, В 1960 году ученые Мартин Дэвис (англ. - Martin Davis) и Хилари Путнэм (англ. - Hilary Putnam) представили алгоритм для “разрешения” серии логических утверждений с помощью компьютера - **алгоритм Дэвиса-Путнэма** (Davis–Putnam algorithm). Затем, в 1965 году Джон Алан Робинсон (англ. - John Alan Robinson) опубликовал статью, в которой продемонстрировал усовершенствование данного алгоритма, которое значительно снижало сложность поиска решения. Подробнее вы можете прочитать в соответствующих статьях, ссылки вы найдёте в конце этого конспекта.

Planner

Теперь, когда появилась теоретическая и алгоритмическая база, стало возможным создавать их реализацию - язык программирования. Первым языком **логического программирования** считается язык **Planner**, разработанный в Лаборатории искусственного интеллекта Массачусетского технологического института (англ. MIT's Artificial Intelligence Laboratory) Карлом Хьюттом (англ. Carl Hewitt). Плэнэр впервые был представлен в 1969 году в публикации PLANNER: A LANGUAGE FOR PROVING THEOREMS IN ROBOTS. **План** в языке Planner - это совокупность **фактов** и **правил вывода**. Главное нововведение этого языка - возможность по имеющемуся **плану** автоматически вывести результат. В Плэнере это было реализовано с помощью метода **backtracking**, и он же в дальнейшем будет использован в **Prolog**-е.

Prolog

В 1972 году два профессора Роберт Ковальски (Robert Kowalski) и Ален Кольмероэ (фр. - Alain Colmerauer) выпустили **Prolog** (англ. - Programming in Logic, франц. - programmation en logique). Prolog - это первый чистый язык **логической парадигмы**, который до сих пор является самым популярным в ней. Кстати, является полным по Тьюрингу. Программа получает задачу и пытается доказать, является ли полученное **утверждение** следствием из правил и фактов. Соответственно, для того, чтобы доказать **утверждение** состоящее из предпосылок (**предикатов**), необходимо доказать, что все предпосылки верны. В этом случае, **утверждение** считается доказанным. То есть, как и раньше, программа получает на вход какие-то **факты**, а также **правила**, с помощью которых можно выводить новые **факты**. Помимо этого, программа получает задачу, которую она пытается решить самостоятельно. И магия заключается в том, что алгоритм решения выстраивается автоматически, основываясь на теоретической основе, которую мы упомянули выше.

После Prolog-а

Логическое программирование не так широко распространено как императивные стили или функциональное программирование, но тем не менее оно живет и продолжает развиваться. В 1986 была создана Ассоциация логического программирования (The Association for Logic Programming) с официальной миссией “внести свой вклад в развитие логического программирования, связать его с другими формальными, а также с гуманитарными науками и продвигать его использование в академии и индустрии по всему миру”. В 1990е было разработаны новые языки логического программирования, например, Oz (1991), Mercury (1995) и Alice (1998). В индексе TIOBE по состоянию на апрель 2023 года язык Prolog занял почётное 48 место, обойдя в топ-50 языков язык ABAP и Bash. А по состоянию на май 2023 года, встал на 43 место.

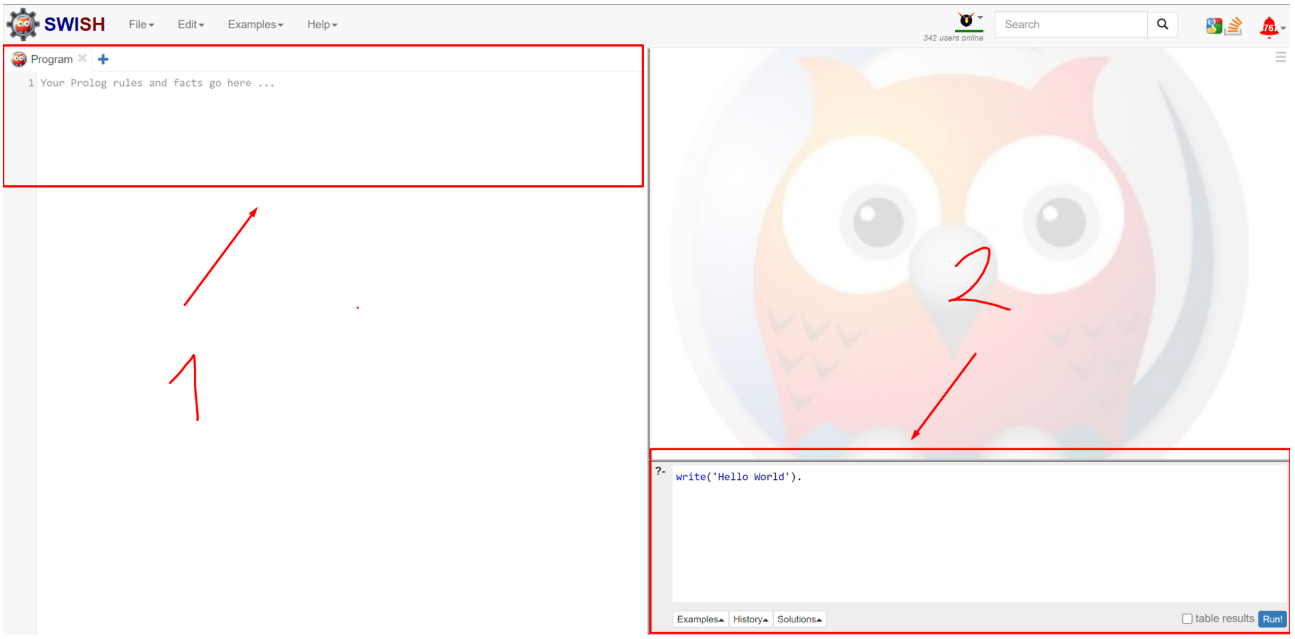
39	CFML	0.29%
40	TypeScript	0.27%
41	Groovy	0.26%
42	ABAP	0.24%
43	Prolog	0.23%
44	PL/SQL	0.22%
45	ML	0.22%
46	Bourne shell	0.22%
47	Forth	0.22%
48	Crystal	0.21%
49	Bash	0.21%
50	Apex	0.19%

Примеры кода

Разберем практические примеры, чтобы ещё глубже погрузиться в логическую парадигму.

Prolog и Hello World

Поскольку Prolog не пользуется большой популярностью даже среди опытных разработчиков, начнем с разбора простейшей “Hello World” программы на языке Prolog. Для упрощения процесса воспользуемся онлайн-компилятором Пролога - SWISH, который можно найти по ссылке https://swish.swi-prolog.org/. Интерфейс компилятора Пролога состоит из двух частей:

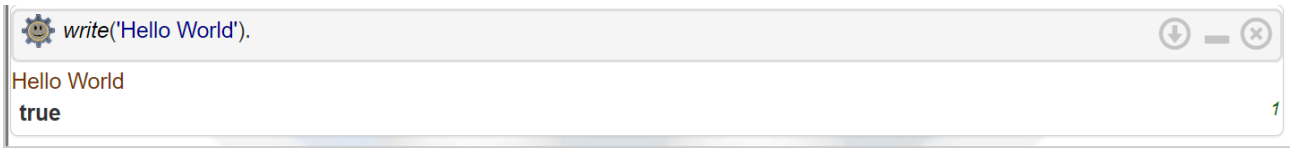


1. Код программы - сюда мы будем записывать факты и логические правила для вывода новых
2. Консоль - сюда мы будем писать **запрос** (или интуитивнее можно сказать - “вопрос”), который мы хотим исполнить.

Отлично. Попробуем исполнить простейший запрос, вывод текста “hello world” на экран. Для это в консоль необходимо написать следующую команду:

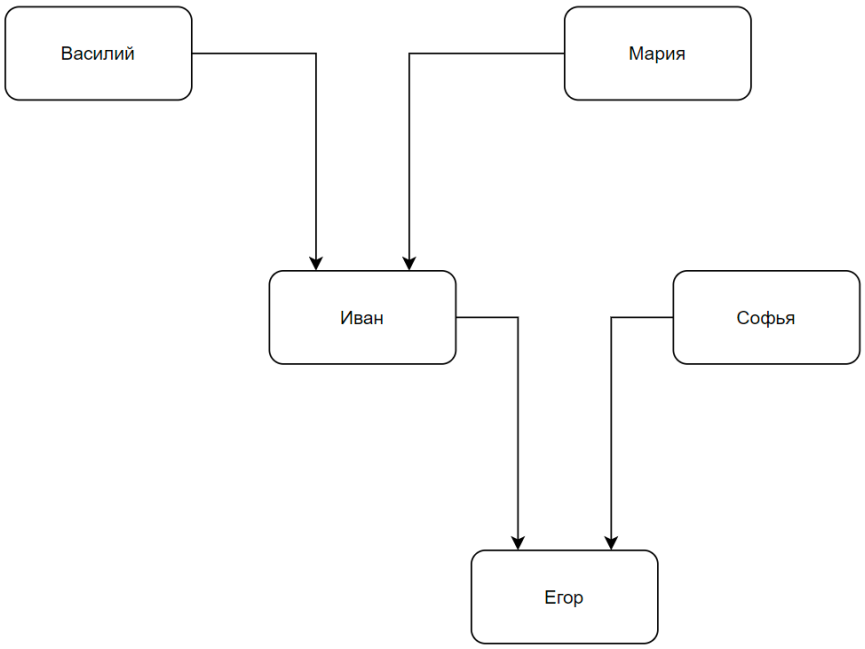
```
1 write('Hello World').
```

И нажать кнопку Run в консоли интерфейса SWISH (справа внизу). И да здравствует магия! Мы запустили свою первую программу в Прологе:



Генеалогия - продолжение

Теперь попробуем что-нибудь посложнее. Например, реализовать задачу из первого примера про генеалогию. Напомню, что в этом примере мы рассматривали следующую семью:



Про которую мы хотели установить как зовут бабушку и дедушку Егора по Ивану. Мы также получили псевдокод в логической парадигме, который верхнеуровнево выглядит следующим образом:

1. Перечисляем начальные факты
2. Даем определения (правила переходов)
3. Формулируем задачу (задаем вопрос)

Давайте посмотрим как это выглядит на языке Prolog. Сначала перечислим в нашей программе факты:

```
male(egor).
male(ivan).
male(vasiliy).
female(sofya).
female(maria).
parent(ivan, vasiliy).
parent(ivan, maria).
parent(egor, sofya).
parent(egor, ivan).
```

В Прологе в конце высказывания ставится точка, прямо как в естественном языке. *male*, *female* и *parent* - это названия предикатов, то есть ключевые слова, которые отражают истинность какого-либо факта. В нашем случае сперва перечисляется пол наших участников, а именно то, что Егор, Иван и Василий - мужчины, Софья и Мария

- женщины. Далее мы уточняем, что родитель Ивана - Василий, а также родитель Ивана - Мария, затем также рассказываем про родителей Егора. По сути к этому моменты мы восстановили генеалогическое древо. Пока что мы не знаем ничего про то, что такое мама, папа, бабушка и дедушка, но уже знаем кто чей родитель. Давайте теперь выведем необходимые правила. Для введения нового правила необходимо написать новые предикат, который мы хотим определить, а затем двоеточие и тире, после которых следует определение. Начнем с определения отца.

```
father(X,Y):- parent(X,Y), male(Y).
```

Что здесь написано. Человек с именем *Y* является отцом человека с именем *X*, в том случае, если два предиката верны. Поскольку запятая между предикатами означает логическое “И”, то оба условия должны выполняются одновременно:

1. Предикат *parent*: Человек с именем *Y* - родитель человека с именем *X*
2. Предикат *male*: Родитель (человек с именем *Y*) является мужчиной

Или, простыми словами, для ребенка *X*, его отцом является его родитель мужского пола. Давайте теперь то же самое сделаем для остальных определений: мама, бабушки и дедушки.

```
mother(X,Y):- parent(X,Y), female(Y).
grandfather(X, Z):- parent(X, Y), father(Y, Z).
grandmother(X, Z):- mother(X, Y), parent(Y, Z).
```

С матерью все понятно - это то же самое, что отец, но родитель женского пола, а не мужского. С дедушкой уже становится интереснее. Человек с именем *Z* для человека *X* считается *дедушкой* в том случае, если верны два следующих высказывания:

1. У человека *X* есть родитель *Y*
2. Отцом *Y* является *Z*

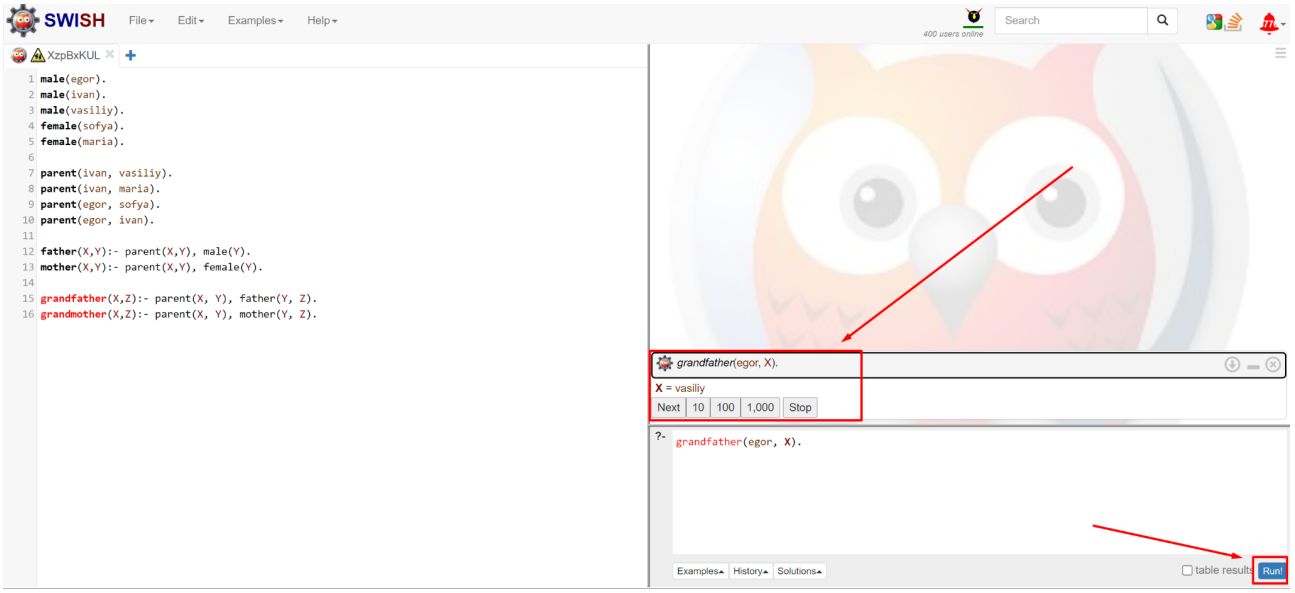
Логика точно такая же, есть два высказывания через запятую (то есть между ними стоит логическое “И”). Но есть одна тонкость. Мы заранее не знаем, существует ли вообще такой человек *Y*, т.к. *Y* - это просто имя переменной. Что же произойдёт, если мы передадим такой *Y*, про который программа не знает ничего? Ответ прост - в этом случае, *parent(X, Y)* просто вернет *false*, поэтому и запрос *grandfather(X, Z)* вернёт *false*, что можно интерепретировать буквально: человек с именем *Z* не является дедушкой *X*, поскольку во время исполнения программы - не было доказано обратное.

Теперь, когда у нас есть и факты и правила, осталось написать запрос: кто же является дедушкой Егора?

```
1 ?- grandfather(egor, X).
```

Запросы в Прологе всегда начинаются со знака вопроса и тире “**?-**”. По сути в данном запросе мы просим у Пролога “найти все такие *X*, для который предикат *grandfather(egor, X)* верен” - то есть вернет *true*.

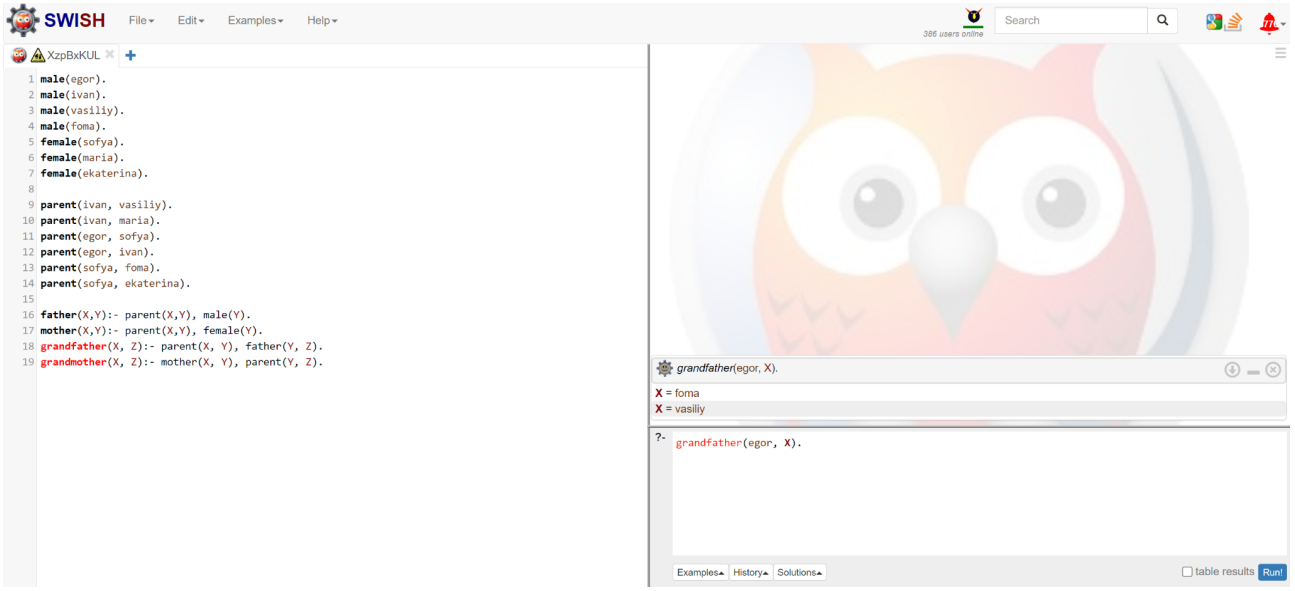
Обращаю внимание на то, что в компиляторе SWISH, который я показывал до этого знак вопроса и тире уже поставили за вас, поскольку для запросов выделено отдельное окно. Поэтому в SWISH ставить ?- не нужно, оно там уже стоит. Давайте запустим все целиком и посмотрим что получится.



Как мы видим, после нажатия “Run”, появилось окно вывода, из которого следует, что X имеет имя Василий, что и требовалось доказать 🙂. Также, вы видите что Prolog спрашивает, проходится ли дальше по родителям Егора, чтобы найти ещё одного дедушку, поскольку видит, что родителей у Егора *два*. Конечно, мы не рассказали программе про родителей Софьи, поэтому если нажать любую из этих серых кнопок, кроме *Stop*, Prolog честно вернет false, поскольку для Софьи предикат grandfather возвращает false ровно по той логике, которую мы обсуждали выше, когда разбирали кейс в котором родителя Y не существует. Здесь родитель Y конечно существует, но про отца у родителя Y - программа не знает. Отсюда и второй false на выводе. Давайте теперь добавим родителей Софьи и посмотрим что получится. Добавим в нашу программу (в часть с фактами) вот такие строчки:

```
1 parent(sofya, foma).
2 male(foma).
3 parent(sofya, ekaterina).
4 female(ekaterina).
```

Здесь, когда мы рассказали про папу и маму Софьи, давайте запустим тот же самый запрос и сразу нажмём *Next* на выводе.



Вуа-ля! Теперь все по-честному: у Егора два дедушки, как мы и хотели! Можем также спросить и про бабушек. Для обозначения логического ИЛИ в Прологе используется точка с запятой:

```
1 grandfather(egor, D); grandmother(egor, B).
```

Попробуйте исполнить данный запрос самостоятельно.

Судоку - продолжение

Ранее мы рассмотрели Судоку как пример задачи, которая может быть поставлена и решена в рамках логической парадигмы, а также псевдокод одного из решений. Напомню, что решение головоломки - это полностью заполненный квадрат Судоку 9x9, то есть в каждой клетке должна стоять ровно одна цифра, и при этом:

- Каждая строка содержит все цифры от 1 до 9
- Каждый столбец содержит все цифры от 1 до 9
- Каждый блок 3 x 3 содержит все цифры от 1 до 9

Давайте теперь разберем решение этой задачи на языке **Prolog**. Для разработки решения нам понадобится библиотека “CLP(FD): Constraint Logic Programming over Finite Domains”, поэтому наша новая программа начнется со строки импорта данной библиотеки:

```
1 :- use_module(library(clpfd)).
```

Данный модуль предоставляет некоторые встроенные функции, которые мы рассмотрим ниже. Теперь напишем определение Судоку и разберем его построчно:

```
 1 sudoku(Rows) :-
 2     length(Rows, 9), maplist(same_length(Rows), Rows),
 3     append(Rows, Vs), Vs ins 1..9,
 4     maplist(all_distinct, Rows),
 5     transpose(Rows, Columns),
 6     maplist(all_distinct, Columns),
 7     Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
 8     blocks(As, Bs, Cs),
 9     blocks(Ds, Es, Fs),
10     blocks(Gs, Hs, Is).
```

В первой строке мы говорим, что предикат sudoku верен для параметра Rows, если верно следующее.

- Строка 2: длина Rows равна 9 (*length* - это процедура вычисляющая длину списка), то есть, всего 9 строк. И каждая строка в свою очередь имеет ту же самую длину, что и Rows. То есть первая строка - это в чистом виде определение матрицы 9 на 9.
- Строка 3: все значения в квадрате 9 на 9 принадлежат интервалу от 1 до 9. Поскольку Rows - это список строк (а по факту - список списков), процедура append в данном случае превращает список списков в один список, то есть удаляет уровень вложенности, рассматривая все числа квадрата как единый массив. Так мы проверили, что все числа являются цифрами от 1 до 9.
- Строка 4: значения во всех строках различны. Процедура *maplist* - означает применение к каждому элементу списка, как и почти одноименный - наш любимый **функциональный** map. А процедура *all_distinct* - попарно сравнивает все элементы внутри массива.
- Строки 5 и 6: то же самое применительно к столбцам. Все значения внутри столбцов обязаны быть разными. Процедура *transpose* - это транспонирование, и теперь, то что называлось *Rows*, стало называться Columns, после чего мы можем проверить значения в *Columns* также как и в предыдущем шаге - через *maplist(all_distinct, Columns)*.
- Строки с 7 по 10: определяем блоки. Сначала даем названия элементам массива *Rows* (то есть собственно строкам), а затем говорим, что blocks верен для As, Bs, Cs, для Ds, Es, Fs и для Gs, Hs, Is. Визуально можно сказать, что мы провели в квадрате Судоку две горизонтальные линии: одна линяя отделяет 3 строку от 4, а вторая 6 строку от 7. Так мы получили три подмножества строк, которых теперь можем попросить что-то ещё, но это мы сделаем дальше.

Фух! Продолжаем. Теперь осталось определить *блок* и добавить условие про то, что все числа внутри одного *блока* должны быть разными.

```
1 blocks([], [], []).
2 blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
3         all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
4         blocks(Ns1, Ns2, Ns3).
```

Давайте также построчно прочитаем что здесь написано.

- Строка 1: Предикат blocks определяется для трех списков (то есть трёх строк).
- Строка 2: Теперь становится интереснее. В blocks каждый из трех списков, начинается с трех элементов, например, первый начинается с N1, N2, N3. А оставшаяся часть просто называется Ns1 и в данном шаге не будет рассматриваться. То есть мы говорим, возьмем три списка, и из каждого из трех список возьмем только три ПЕРВЫХ элемента. И вот только теперь, когда мы понимаем, что у нас в руках именно БЛОКИ, мы можем дать им определение.
- Строка 3: Все значения внутри блока должны быть разными. Можно сказать, что это основное тело данного определение - то ради чего все и затевалось.
- Строка 4: Обрабатываем Ns1, Ns2 и Ns3 (то, что осталось от списков) - для них должно работать ровно то же самое, что и для N1, N2, .. , N9. Отмечу, что таким образом в данной Prolog-программе реализована **рекурсия** (вызов процедуры из самой себя), которая подробно была разобрана в предудыщей лекции по функциональному программированию. В данном случае мы даем определение блоку как набору правил который работает для частей входного массива, а для оставшейся части - работает этот же набор правил.

Осталось ввести саму головоломку и решить её. У меня уже есть подготовленный вариант. Давайте введем предзаполненный квадрат в виде факта.

```
1 problem([[_,_,_,_,_,_,_,_,_],
2          [_,_,_,_,_,3,_,8,5],
3          [_,_,1,_,2,_,_,_,_],
4          [_,_,_,5,_,7,_,_,_],
5          [_,_,4,_,_,_,1,_,_],
6          [_,9,_,_,_,_,_,_,_],
7          [5,_,_,_,_,_,_,7,3],
8          [_,_,2,_,1,_,_,_,_],
9          [_,_,_,_,4,_,_,_,9]]).
```

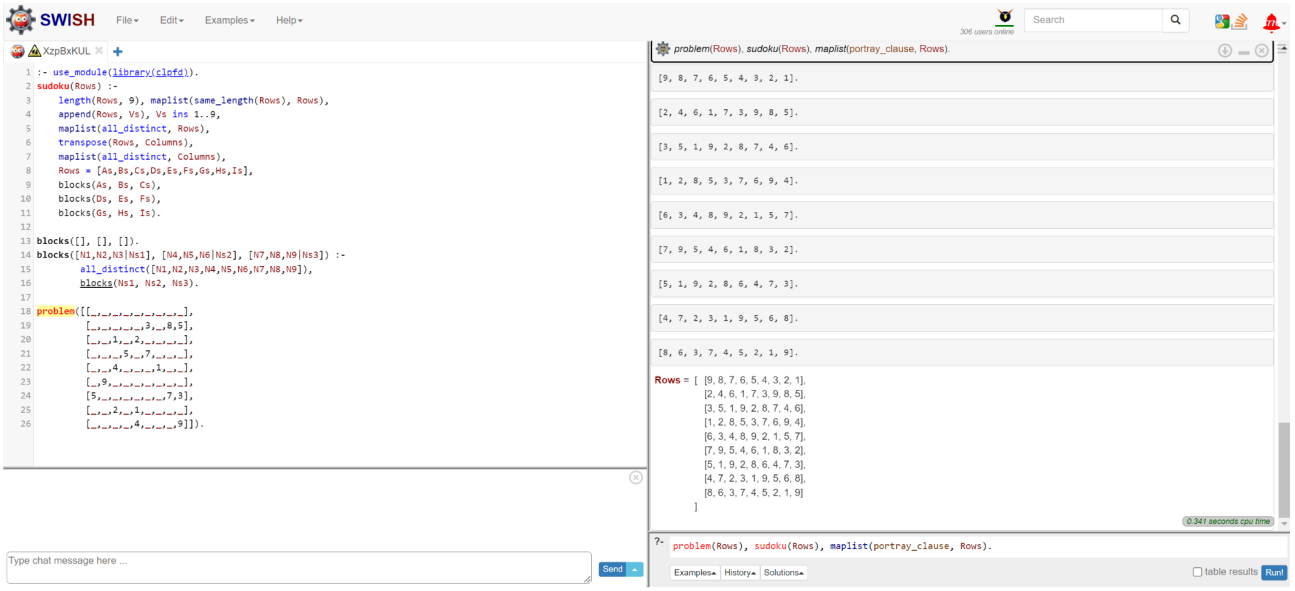
Теперь осталось написать запрос. Будет достаточно вот такого запроса, но только без знака вопроса и тире:

```
1 ?- problem(Rows), sudoku(Rows)
```

А если хотите, чтобы ответи выводился построчно, то вот можно добавить ещё один *maplist* со встроенной процедурой *portray_clause*:

```
1 ?- problem(Rows), sudoku(Rows), maplist(portray_clause, Rows).
```

Что написано в запросе: мы хотим найти такой *Rows*, для которого верно *problem(Rows)*, а также верно *sudoku(Rows)*. Поскольку для предиката *problem* уже закреплены некоторые презаполенные ячейки, то эти значения сто процентов и будут использованы, и для них Пролог будет решать второй предикат - *sudoku(Rows)*. На следующем скриншоте виден построчный вывод, за которым следует решённый Судоку целиком:



Данный пример можно найти по ссылке, которую я оставлю в конспекте. Попробуйте прогнать его целиком.

Загадка Эйнштейна

Логическая задача, Скорее не имеющая к самому Эйнштейну никакого отношения, но при этом не менее известная, иногда её ещё называют “головоломка о зебре”. Я приведу тот самый оригинальный вариант, который впервые был опубликован в журнале Life в 1962 году. И так, загадка Эйнштейна имеет следующие вводные:

1. На улице стоят пять домов.
2. Англичанин живёт в красном доме.
3. У испанца есть собака.
4. В зелёном доме пьют кофе.
5. Украинец пьёт чай.
6. Зелёный дом стоит сразу справа от белого дома.
7. Тот, кто курит Old Gold, разводит улиток.
8. В жёлтом доме курят Kool.

9. В центральном доме пьют молоко.
10. Норвежец живёт в первом доме.
11. Сосед того, кто курит Chesterfield, держит лису.
12. В доме по соседству с тем, в котором держат лошадь, курят Kool.
13. Тот, кто курит Lucky Strike, пьёт апельсиновый сок.
14. Японец курит Parliament.
15. Норвежец живёт рядом с синим домом.

Собственно задача заключается в том, чтобы ответить на следующие
Вопросы: Кто пьёт воду? Кто держит зебру?

Плюс, даны комментарии с уточнениями от авторов:

- каждый из пяти домов окрашен в свой цвет, а их жители — разных национальностей, владеют разными животными, пьют разные напитки и курят разные марки американских сигарет
- в утверждении 6 справа означает справа относительно вас

Получается, что мы получили на вход множество фактов и ограничений (правил вывода новых фактов), после чего вас спрашивают ещё один факт, который, естественно, заранее в условии не приведен. Получается, что для решения задачи необходимо как-то перебирать факты, проверяя ограничения, выводя новые факты и в итоге за несколько итераций прийти к тому факту, который мы искали изначально. Как удачно это ложится на парадигму логического программирования! Давайте напишем решение для данной задачи. Проговорим несколько технических деталей:

- Как мы уже видели в предыдущем примере с Судоку, можно использовать нижнее подчеркивание “_” для обозначения места для переменной в предикате, про которую мы ничего не знаем, кроме того, что она там есть
- Для решения этой задачи удобно использовать встроенную функцию *member*

Начнем с перечисления фактов. Будем использовать следующую конструкцию. Каждый дом представляет собой список атрибутов. С другой стороны, существует список домов, который содержит списки атрибутов каждого дома. То есть, есть некоторый список *houses*, отдельным элементом которого является *h* - список атрибутов одного дома. Мы также знаем, что длина *houses* равна 5, поскольку домов в этой задаче пять, и что на домах есть есть отношение соседства. Ещё мы знаем, что каждый дом описан 5 атрибутами: национальность, домашнее животное, марка сигарет, напиток и цвет дома.

В данном месте я предлагаю вам сделать паузу и попробовать перечислить факты в компиляторе Prolog-а самостоятельно. А через несколько секунд мы обсудим как это можно было сделать. И так, переходим к обсуждению решения в Прологе.

```
 1 houses(Hs) :-
 2     length(Hs, 5),                                          %  1
 3     member(h(english,_,_,_,red), Hs),                       %  2
 4     member(h(spanish,dog,_,_,_), Hs),                       %  3
 5     member(h(_,_,_,coffee,green), Hs),                      %  4
 6     member(h(ukrainian,_,_,tea,_), Hs),                     %  5
 7     next(h(_,_,_,_,green), h(_,_,_,_,white), Hs),           %  6
 8     member(h(_,snake,winston,_,_), Hs),                     %  7
 9     member(h(_,_,kool,_,yellow), Hs),                       %  8
10     Hs = [_,_,h(_,_,_,milk,_),_,_],                         %  9
11     Hs = [h(norwegian,_,_,_,_)|_],                          % 10
12     next(h(_,fox,_,_,_), h(_,_,chesterfield,_,_), Hs),      % 11
13     next(h(_,_,kool,_,_), h(_,horse,_,_,_), Hs),            % 12
14     member(h(_,_,lucky,juice,_), Hs),                       % 13
15     member(h(japanese,_,kent,_,_), Hs),                     % 14
16     next(h(norwegian,_,_,_,_), h(_,_,_,_,blue), Hs),        % 15
17     member(h(_,_,_,water,_), Hs),       % one of them drinks water
18     member(h(_,zebra,_,_,_), Hs).       % one of them owns a zebra
```

И так, читаем построчно:

- Строка 1: определение предиката *houses* для списка домов *Hs*. То есть предикат houses верен тогда, когда верно все что написано ниже для отдельного списка *Hs*. При запуске, Hs - это именно то, что будет подбирать Пролог.
- Строка 2: Здесь и далее, по порядку перечисляем известные нам факты. Первый факт, “на улице стоят 5 домов”, записано через встроенную процедуру поиска длины списка: *length(Hs, 5)*.
- Строка 3: Сообщаем Прологу, что “Англичанин живет в красном доме”. То есть, списку *Hs* принадлежит такой элемент *h*, который содержит национальность “англичанин”, 3 нефиксированных атрибута, и “красный” цвет дома. Далее делаем то же самое в строках 4-6.
- Строка 7: Теперь мы хотим сообщить, что “зелёный дом стоит сразу справа от белого дома”. Для этого будем использовать предикат *next*, который мы определим позже и который будет указывать на соседство двух домов *h* в списке *Hs*. В данном случае предикат вернет True, если дома с последним атрибутом green и последним атрибутом white - соседи.
- Строка 10: Говорим, что “в центральном доме пьют молоко”. Про молоку я думаю уже все понятно, а как указать на центральность? Можно прямо в лоб уточнить формат большого списка *Hs* следующим образом: в списке *Hs* первые два элемента - нефиксированы, потом идет дом, в котором пьют

молоко, а затем ещё два нефиксированных дома. Так дом, в котором пьют молоко - гарантировано стоит по центру.

- Строка 11: Делаем почти то же самое, что в предыдущей строке, но с использованием вертикальной черты. Так можно не перечислять все 4 объекта, а просто поставить вертикальную черту, и тогда это будет означать “все остальное, что идет после первого”. А если целиком, то первый дом - дом с норвежцем, а затем идут нефиксированные элементы.
- Все факты ниже приводятся по уже известным нам принципам.

И всё, почти готово! Осталось только определить *next* и задать Прологу вопрос. Давайте это сделаем.

```
1 next(A, B, Ls) :- append(_, [A,B|_], Ls).
2 next(A, B, Ls) :- append(_, [B,A|_], Ls).
```

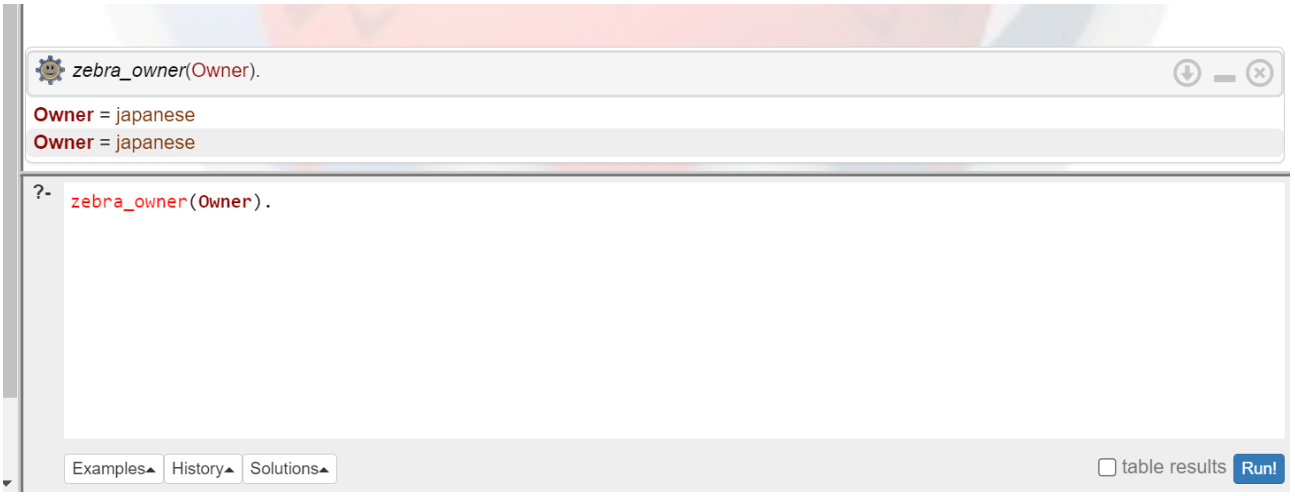
Давайте подумаем, что означает для двух элементов “быть соседями” в списке домов? “Быть соседями” - это одна из двух ситуаций: есть список, в нем либо сначала идет элемент A, а затем элемент B, либо наоборот - сначала B, а затем A. При этом расположение оставшихся элементов не имеет значения. Теперь прочитаем то, что написано у нас. Слово *append* означает **конкатенцию** списков. На всякий случай напомню, что конкатенация списков - это склеивание двух списков в один. Теперь определим *next*. Предикат *next* зависит от A, B и списка Ls и верен, тогда когда верно, что Ls представляет собой **конкатенацию** любых элементов (одного или нескольких) и списка который состоит из A и B (последовательность важна), а затем идут любые элементы (или не идут, если их нет). Следующее утверждение - это то же самое, но наоборот: сначала идёт A, а затем B. То есть логика очень простая: Ls должен быть таким списком, в котором присутствует конкатенция “любых элементов”, “A, B” и “любых элементов”, либо конкатенция “любых элементов”, “B, A” и “любых элементов”. Отлично! Осталось оформить запрос.

```
1 zebra_owner(Owner) :-
2     houses(Hs),
3     member(h(Owner,zebra,_,_,_), Hs).
```

Кто же держит зебру? Определим предикат для хозяина, который живет в доме с зеброй. Поскольку нам необходимо вычислить национальность хозяина, то пишем что *zebra_owner* зависит от *Owner*-а. Это такой предикат, для которого верно следующее:

- Дом с хозяином этой национальности и зеброй присутствует в списке *Hs*.
- Предикат *houses* верен для списка *Hs* (то есть должны быть верны все факты, которые мы с вами перечислили до этого).

Все готово! Приводим в качестве запроса наш предикат *zebra_owner* и Жмем Run, чтобы получить результат.



Ответ: в доме, где держат зебру живёт японец. Попробуйте сделать то же самое для ответа на второй вопрос - “Кто пьёт воду?”.

Преимущества и недостатки ЛП

Рассмотрим преимущества и недостатки логической парадигмы.

Преимущества

- Естественность и удобство синтаксиса
- Встроенный функционал для поиска решений
- Инкапсуляция знаний

Основное преимущество логического программирования - это его декларативность вкупе с привязкой к **формальной логике**. Вам не нужно думать о том, как именно реализовать тот или иной алгоритм. Достаточно только понять синтаксис какого-нибудь логического языка и описать на нем вашу задачу. И если вы уже знакомы с **формальной логикой**, то эти усилия для вас сводятся к минимуму. Таким образом, вы тратите немного, по сравнению с другими парадигмами, времени, которое уходит только на то, чтобы описать задачу, вместо полноценной реализации алгоритма.

Также, с помощью логического программирования вы можете сохранять некоторые факты и результаты в виде логических высказываний, что делает их крайне пригодными для хранения накопленных “знаний”.

Недостатки

- Низкая производительность
- Ограниченность синтаксиса
- Узкое сообщество

Самый главный недостаток этой прекрасной парадигмы - это низкая скорость работы логических языков из-за не всегда эффективной реализации алгоритмов. Особенно это заметно при использовании ЛП на больших объемах данных. Ирония в том, что как правило логическое программирование используется как раз там, где объёмы данных большие (ИИ, машинное обучение, экспертные системы).

Помимо этого, логические языки как правило ограничены в синтаксисе, хотя и являются полными по Тьюрингу. В некоторых случаях это может усложнить разработку.

И конечно же, логические языки не так широко распространены как функциональные и, тем более, не так широко как императивные. Отсюда вытекает малочисленность сообщества и, соответственно, ограниченность библиотек, готовых решений и поддержки, которую можно от этого сообщества получить.

Итоги лекции

В сегодняшней лекции мы разобрали логическое программирование, а именно:

- Обзор логического программирования
    - Контекст и определения
    - Примеры задач для
    - Сферы применения
    - Языки программирования
- История вкратце
- Примеры кода
- Преимущества и недостатки
- Подведение итогов

Если у вас остались вопросы, вы можете написать их мне в Telegram: @alexlevinML

Итоги курса

Поздравляю! Вы прослушали последнюю лекцию курса “**Парадигмы программирования и языки парадигм**”. Мы прошли большой путь от разбора слова “**парадигма**” через привычные императивные парадигмы к менее привычным декларативным. Из императивных стилей мы изучили: структурное, процедурное и объектно ориентированное программирование. Из декларативных стилей: функциональное (где немного затронули язык Haskell) и логическое (где разбирали примеры на Prolog-е).

В самой первой лекции мы разбирали историю про Давида и Голиафа. И главной целью этого курса была демонстрация различных способов “думать о поставленной задаче”, а также преимуществ и недостатков каждого из этих способов в различных ситуациях. Мы убедились в том, что способов решить одну и ту же задачу и получить один и тот же результат, может быть не просто больше одного, а бесконечно много. Но человеческий мозг очень хорошо приспособлен для того, чтобы примерно понимать какой способ подобрать для конкретной ситуации, при наличии у него опыта о подобных выборах и результатах, к которому они привели. Поэтому, будьте “гибким Давидом”, выбирайте те пути, которые считаете наиболее эффективными и со временем результат будет улучшаться и все получится!

Надеюсь, что данный курс оказался для вас интересным и полезным. Желаю удачи и успехов и увидимся в других курсах!

Что можно почитать еще?

1. Братко Иван - Программирование на языке Пролог для искусственного интеллекта
2. Статья на Хабре - “Что такое логическое программирование и зачем оно нам нужно”
3. PySwip: Python - SWI-Prolog bridge enabling to query SWI-Prolog in your Python programs
4. Алгоритм поиска кратчайшего пути A* на языке Prolog

Используемая литература

1. SWISH - Online Prolog Compiler
2. Пример решения Судоку на языке Prolog
3. Пример решения Загадки Эйнштейна
4. Prolog модуль CLP(FD): Constraint Logic Programming over Finite Domains
5. Davis M., Putnam H. - A Computing Procedure for Quantification Theory (1960)
6. John Alan Robinson - A Machine-Oriented Logic Based on the Resolution Principle (1965)
7. Carl Hewitt - PLANNER: A LANGUAGE FOR PROVING THEOREMS IN ROBOTS (1969)
8. Статья - Логика: предикатная, формальная и сентенциальная. Кванторы и возникновение информатики