Московский авиационный институт

(национальный исследовательский университет)

**Курсовая работа**

**«Логические языки как путь к автоматическому решению задач**

**компьютером»**

Выполнил: студент группы М8О-106Б-20

Почечура Артемий Андреевич

Москва

2021

**1 Введение в предмет логического программирования**

На протяжении всей истории человечество сталкивалось с различными проблемами, и всегда стоял вопрос, как их решить? В решении некоторых из них на помощь приходила математика. Ещё в Древнем Египте, Греции и других странах уже решали задачи с использованием алгебры и геометрии. Но с течением времени, задачи становились всё более тяжёлыми, требовалось всё больше непростых вычислений. Люди пытались облегчить их решение создавая различные приспособления: началось всё с примитивных счётов, а закончилось сложнейшими компьютерами. Но если со счётами человеку приходилось много вычислений держать в памяти, то компьютер с лёгкостью берёт эту задачу на себя. Но для этого компьютеру необходимо поставить задачу и найти какой-то способ чтобы объяснить машине, как задачу нужно правильно решать. Таким образом и появились языки программирования. Но и к этому вопросу люди подошли по-разному. В настоящее время существует несколько основных парадигм программирования, а именно: логическое, императивное. Императивное предусматривает выполнение сформулированных инструкций. Но проблема состоит в том, как правильно создать такие инструкции. Математик не сможет без необходимых знаний написать эти инструкции для решения имеющейся задачи. Но тут может помочь логическое программирование. Оно позволяет подойти к программированию и решению задач с точки зрения математики.

Логическое программирование исходит из того, что компьютер способен работать по логическим построениям, присущим человеку. Логическое программирование – это подход к программированию, при котором программа создаётся исходя из правил, но без явного указания последовательности их применения.

Под логическим программированием для создания программного обеспечения подразумевают использование логики предикатов (рассуждений), которые помогают в решении возникшей проблемы.

Обычное предложение из русского языка может быть трансформировано в предложение логики предикатов. Для этого необходимо устранить все ненужные слова из предложения (оставив только ключевые), после чего преобразовать предложение так, чтобы свойство стояло первым, а обладающие данным свойством объекты были сгруппированы после него. При этом объекты должны быть параметрами, на которых это свойство действует. Например, «машина красная», на языке логического программирования будет записана как «красная (машина)».

**1.1. Предпосылки появления логического программирования**

Необходимостью развития логического программирования стало существование неформализованных задач. К неформализованным, или плохо структурированными называются задачи, которые обладают одной или несколькими из нижеследующих характеристик:

1) Задачи не могут быть заданы в числовой форме (данные представлены в символьном виде).

2) Не существует точного метода решения.

3) Есть возможность выбора, но нет алгоритмического решения.

4) Противоречивость исходных данных и знаний о предметной области.

К предпосылкам появления и развития логического программирования относят следующие недостатки традиционного подхода:

1) Некоторые понятия плохо формализуются с помощью алгоритмов.

2) Не все виды понятий можно представить программой.

3) Некоторые понятия и отдельные запросы определены частично или не полностью.

4) Программы теряют способность к модификации с увеличением их размеров.

5) Программы не могут определять или доопределять понятия.

6) Программы выводят только запланированные результаты и/или обрабатывают ошибочные ситуации, но альтернативные решения не предусматриваются.

Для решения каждой из этих проблем можно постараться применить логическое программирование, и создать на его основе программу. Поэтому становиться интересно найти другой подход (в отличие от традиционного) для решения определённых проблем.

Логическое программирование позволяет использовать компьютер пользователем, который не обладает знаниями в области программирования, но хорошо ориентируется в изученных им областях знаний, и владеющего профессиональными специфическими навыками.

**1.2. Историческая справка о языке Пролог**

Первый логический язык программирования - Planner - появился в 1969 году. Но более популярным стал Prolog, разработанный Аленом Кольмеро в 1971 году.  Название произошло от словосочетания ПРОграммирование при помощи ЛОГики (PROgramming in LOGic). Инструментом для автоматического доказательства теорем использовался метод исчисления предикатов [1]. Позже их стали применять в качестве языков программирования общего назначения.

Важной особенностью логических языков программирования является наличие встроенной базы данных (БД).

В настоящее время Пролог – язык, который предназначен для создания приложений, использующих методы и средства искусственного интеллекта, и для создания экспертных систем.

Несмотря на то, что языки логического программирования имеют небольшое распространение, тем не менее их применяют при разработке трансляторов и искусственного интеллекта [2], также они могут использоваться для разработки любых десктопных и мобильных приложений, а также сайтов. Например, датская компания PDC, разрабатывающая Visual Prolog, применяет его при программировании авиационных и медицинских систем [3]. На Prolog разрабатываются инструменты оптимизации программ – т.к. выяснилось, что это гораздо удобнее чем на С++, при этом в основном используют SWI Prolog.

С помощью Visual Prolog можно создавать приложения с графическим интерфейсом, используемым в операционной системе Microsoft Windows.

**2 Практическое применение логического программирования**

Обычно программирование начинают изучать именно с языков Python, C/C++/C#, Java, поэтому все привыкли, что программа выглядит как-то так:

Начать

Команда 1

Команда 2

Если УСЛОВИЕ

Команда 3

Иначе

Команда 4

Закончить

Этот яркий, но малоинформативный пример призван показать, что *обычно* команды выполняются одна за другой, при условии, что следующие инструкции (команды) будут использовать результат работы предыдущих. Также возможно описывать собственные команды, код которых будет написан подобным образом. Компьютер выполняет то, что запрограммирует программист, по четкому алгоритму, последовательно выполняя инструкции, и он не сможет, подобно человеку, делать выводы, строить ассоциативные ряды.

Рассмотрим простой пример – знаменитое высказывание Аристотеля:

*Всякий человек смертен.*

*Сократ - человек.*

*Следовательно, Сократ смертен.*

Выглядит логично. Но есть ли способ научить компьютер делать выводы как Аристотель? И вот тут приходит на помощь Prolog, который так часто встречается в Интернете при поиске информации о логическом программировании. Несложно догадаться, что Prolog является самым популярным чисто логическим языком программирования.

Логическое программирование основано на логике предикатов. Вспомним, что основными понятиями в логическом программировании являются: ***Предикат*** (условно говоря) – это функция от нуля или более переменных, возвращающая значение логического типа (истина) или ложь. ***Факт***– это предикат с определенным набором параметров и заведомо известным значением.

Здесь стоит повториться: очевидным применением логического программирования является ***работа с базами данных***. При помощи логического языка возможно естественным образом описывать запросы, комбинируя предикаты, причем научить писать такие запросы можно даже человека, совершенно не знакомого с логическим программированием!

**3 От слов к делу! Примеры программ**

Начнём с примера, призванного наглядно продемонстрировать, почему логическое программирование удобно, и за что же его любят математики. А именно, опишем правила вычисления производной:

d(X,X,1) :- !. % производная X по X = 1

d(T,X,0) :- atomic(T). % производная константы = 0

d(U+V,X,DU+DV) :- d(U,X,DU), d(V,X,DV). % производная суммы = сумме производных

d(U-V,X,DU-DV) :- d(U,X,DU), d(V,X,DV).

d(-T,X,-R) :- d(T,X,R).

d(C\*U,X,C\*W) :- atomic(C), C\=X, !, d(U,X,W). % производная константы, умноженной на выражение = константе на производную от выражения

d(U\*V,X,Vd\*U+Ud\*V) :- d(U,X,Ud), d(V,X,Vd). % производная произведения

d(U/V,X,(Ud\*V-Vd\*U)/(V\*V)) :- d(U,X,Ud), d(V,X,Vd).

Запустим:

?- d((x-1)/(x+1),x,R).

R = ((1-0)\*(x+1)-(1+0)\*(x-1))/((x+1)\*(x+1)).

Пусть производная получилась довольно громоздкой, но мы и не ставили цель её упростить. Главное, из примера видно, что правила вывода производной на Prolog-е описываются очень близким образом к их математическому представлению. Чтобы сделать подобное на привычных языках программирования, пришлось бы вводить понятие дерева выражений, описывать каждое правило в виде функции и т. д. Тут же мы обошлись 8-ю строками. Но здесь важно остановиться и задуматься: компьютер не начал работать как-то иначе, он все ещё обрабатывает последовательности команд. Стало быть, те самые деревья, которые где-то все-таки должны быть зашиты, чтобы программа работала, действительно присутствуют, но в неявном виде. Деревья эти именуют "деревьями вывода", именно они позволяют подбирать нужные значения переменных, перебирая все возможные варианты их значений.

Давайте на простом примере рассмотрим, что из себя представляет перебор.

Зададим условие:

speciality(X,tech\_translator) :- studied\_languages(X), studied\_technical(X). % X - технический переводчик, если изучал языки и технические предметы

speciality(X,programmer) :- studied(X,mathematics), studied(X, compscience). % X - программист, если изучал математику и компьютерные науки

speciality(X,lit\_translator) :- studied\_languages(X), studied(X,literature). % X - литературный переводчик, если изучал языки

studied\_technical(X) :- studied(X,mathematics). % X изучал технические предметы, если изучал математику

studied\_technical(X) :- studied(X,compscience). % ...или компьютерные науки

studied\_languages(X) :- studied(X,english). % X изучал языки, если изучал английский

studied\_languages(X) :- studied(X,german). % ...или немецкий

studied(petya,mathematics). % Петя изучал математику

studied(petya,compscience). % ...компьютерные науки

studied(petya,english). % ...и английский

studied(vasya,german). % Вася изучал немецкий

studied(vasya,literature). %...и литературу

Зададим вопрос компьютеру – Петя или Вася технический переводчик?:

?- speciality(X,tech\_translator).

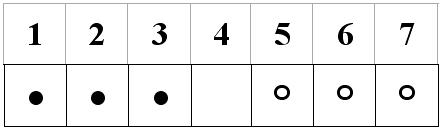
X = petya ;

X = petya ;

false.

Ответ: Петя, и в то же время Петя – ложь… Что-то не так!

Попробуем разобраться. На самом деле, перебирая все варианты значений *X*, Пролог пройдёт по такому дереву:



Дерево будет обходиться в глубину, то есть сначала рассматривается всё левое поддерево для каждой вершины, затем правое. Таким образом, Пролог дважды докажет, что Петя - технический переводчик, но больше решений не найдёт и вернёт false. Стало быть, половина дерева нам, в общем-то, была не нужна. В данном случае, перебор не выглядит ошибочным, всего-то обработали лишнюю запись в базе. Чтобы показать ошибочность перебора, рассмотрим другой пример:

Представим, что перед нами в ячейках расположены три чёрных и три белых шара (см. рисунок выше), которые требуется поменять местами. За один ход шар может или передвинуться в соседнюю пустую клетку, или в пустую клетку за соседним шаром ("перепрыгнуть" его). Решать будем ***поиском в ширину*** в пространстве состояний (состоянием будем считать расположение шаров в ячейках). Суть этого метода заключается в том, что мы ищем все пути длины 1, затем все их продления, затем продления продлений и т. д., пока не найдем целевую вершину (состояние). Почему поиск в ширину? Он первым делом выведет самый оптимальный путь, то есть самый короткий. Как может выглядеть код решения:

% Обозначения: w - белый шар, b - чёрный, e - пустая ячейка

is\_ball(w). % w - шар

is\_ball(b). % b - шар

near([X,e|T],[e,X|T]) :- is\_ball(X). % если фишка рядом с пустой ячейкой, то можно переместиться

near([e,X|T],[X,e|T]) :- is\_ball(X).

jump([X,Y,e|T],[e,Y,X|T]) :- is\_ball(X), is\_ball(Y). % если за соседним шаром есть пустая ячейка, то можно переместиться

jump([e,Y,X|T],[X,Y,e|T]) :- is\_ball(X), is\_ball(Y).

% предикат перемещения. Мы или рассматриваем первые элементы списка, или убираем первый элемент и повторяем операцию

move(L1,L2) :- near(L1,L2).

move(L1,L2) :- jump(L1,L2).

move([X|T1],[X|T2]) :- move(T1,T2).

% предикат продления текущего пути. Если из состояния X можно перейти в состояние Y и

% Y не содержится в текущем пути, то Y - удачное продление

prolong([X|T],[Y,X|T]) :- move(X,Y), not(member(Y,[X|T])).

% Первый аргумент - очередь путей, второй - целевое состояние, третий - результат, то есть найденный путь

bdth([[X|T]|\_],X,R) :- reverse([X|T], R). % Поиск в ширину нашел решение, если первый элемент пути совпадает с целью (путь наращивается с начала, так что перевернем результат)

bdth([P|QI],Y,R) :- bagof(Z,prolong(P,Z),T), append(QI,T,QO), !, bdth(QO,Y,R). % Ищем все возможные продления первого пути и кладём в очередь, рекурсивно запускаем поиск

bdth([\_|T],Y,R) :- bdth(T,Y,R). % Если продлений на предыдущем шаге не нашлось, то есть bagof вернул false, убираем первый путь из очереди

bsearch(X,Y,R) :- bdth([[X]],Y,R). % Здесь использован предикат bdth

% Предикат, который решает нашу задачу и выводит результат и длину найденного пути на экран

solve :- bsearch([w,w,w,e,b,b,b],[b,b,b,e,w,w,w],P), write(P), nl, length(P, Len), write(Len), nl.

Если попытаться вызвать предикат *solve*, то, в лучшем случае получим ошибку, в худшем - среда зависнет. Дерево здесь (с учётом лежащих в памяти путей) настолько велико, что переполнит стек, так и не давая ответ. Это же может происходить и в императивных (процедурных (обычных)) языках программирования. Очевидно, что на решение той же задачи на Питоне или Си (без использования библиотек) ушло бы на порядки больше времени. Для полноты картины рассмотрим решение данной проблемы, а после перейдем к тому, какие же задачи решаются подобным образом.

Со стороны улучшения алгоритма можно предложить использовать поиск в глубину. Но как же, он ведь не даст оптимального результата? Сделаем просто: ограничим глубину поиска. Так мы точно не забьём стек и, возможно, получим ответ. Поступим так: проверим, есть ли пути длины 1, затем длины 2, затем длины 4 и т. д. Получим так называемый ***поиск с итерационным заглублением***:

% Первый аргумент - текущий путь, второй - целевое состояние, третий - результат, то есть найденный путь

dpth\_id([X|T],X,R,0) :- reverse([X|T], R). % Успешное окончание поиска

dpth\_id(P,Y,R,N) :- N > 0, prolong(P,P1), N1 is N - 1, dpth\_id(P1,Y,R,N1). % Если счётчик >0, то уменьшаем его и продолжаем поиск рекурсивно

generator(1). % Изначально предикат вернет 1

generator(N) :- generator(M), N is M + 1. % Рекурсивно получаем 2, 3, 4 и т. д.

isearch(X,Y,R) :- generator(D), dpth\_id([X],Y,R,D). % функция вызывает поиск от каждого натурального значения глубины.

Во-первых, здесь стоит обратить внимание, что мы не используем очереди, а также внешних предикатов (кроме *reverse*). Это потому, что поиск в глубину естественен для Пролога. Во-вторых, пусть мы и делаем вроде как "лишние" действия, то есть для каждого нового значения глубины проходим по всем путям заново, мы практически не теряем в скорости относительно поиска в ширину (может в несколько раз, но не на порядок), при этом значительно экономим память. В-третьих, мы наконец-то получаем ответ, и это самое главное.

С другой стороны, задачу можно было бы ***решить, не меняя код поиска***, а лишь изменив правила перемещения шаров. Обратим внимание, что нам заранее известны входные и выходные данные. Приглядевшись становится понятно, что нет никакого смысла в движении фишек "назад". Действительно, если чёрным нужно встать в правые позиции, то какой смысл делать ходы влево? Перепишем предикаты движения:

near([w,e|T],[e,w|T]).

near([e,b|T],[b,e|T]).

jump([w,X,e|T],[e,X,w|T]) :- is\_ball(X).

jump([e,X,b|T],[b,X,e|T]) :- is\_ball(X).

Хм, код стал даже проще. Запустив мы убедимся, что поиск (оба варианта), во-первых, работает, во-вторых, работает быстро, в-третьих, работает быстро и выводит результат. Это приводит к успеху, задача решена! Работа программы похожа на работу ***искусственного интеллекта***. Программа получает входные данные и желаемый результат, а затем сама ищет, как его достигнуть.

**4 Зачем и где можно применять логическое программирование**

Рассмотрим применяемость логического программирования на примерах решения различных задач.

**Анализ естественного языка**

Человек при составлении и восприятии текста ориентируется не только на набор слов и их значений, но и на свою картину мира. К примеру, если в базе лежит факт "Коля починил Оле компьютер", то на вопрос "Разбирается ли Коля в компьютерах?" программа не сможет ответить, не смотря даже на то, что решение вроде как "на поверхности". Именно из-за низкой скорости и особенностей использования логических языков они не применяются там, где с лёгкостью справляются "нейросети" (например, поиск совпадений на картинках). Но вот задачи синтаксического разбора, текстовой аналитики и т.п. на логических языках решать очень даже комфортно.

**Поиск решений**

Задача с Петей и Васей, а также задача с шарами (см. главу 3) - примеры поиска решений. Представим, что у нас есть сведения о некоей компьютерной сети и нам нужно понять, можно ли тем или иным способом её взломать (обойти защиту). Данную задачу достаточно лаконичным образом возможно описать на логических языках и запустить процесс поиска решений. Помимо переборных задач, хорошо будут решаться те задачи, где потребуются логические рассуждения (например, анализ кода или естественного текста).

**Мета-программирование**

С помощью языков логического программирования (например, Пролога) можно описать свой собственный язык, и придумать в нём определённые правила. Например, стоит задача анализа определённых научных данных. В одной команде работают биолог и программист. Получается, программист должен постоянно советоваться с биологом о том, что и как ему делать, ведь биологическим формулам и законам программистов не учат. Поэтому программисту было бы очень важно написать простой язык, на котором биолог сможет сам записать свои формулы, не привлекая программиста. Вот тут и пригодится логическое программирование.

Здесь крайне важно отметить, что решения задач с помощью логических языков оказываются насколько специфичны и сложны, настолько и удобны (если речь не идёт об узкоспециализированных решениях). На практике, программа на императивном языке вероятнее всего обгонит аналогичную программу на логическом языке, но затраты на написание кода в ряде случаев (в том числе описанных выше) сократятся в разы. На практике не часто можно столкнуться именно с Prolog-ом. Он, конечно, выразителен (можно описывать сложные вещи просто), хорошо расширяется, на нем легко можно написать надежный код и решать определенные задачи (в т. ч. просто логические задачи), но в нём присутствует и ряд недостатков: Пролог сильно уступает по скорости императивным языкам, а также не особенно поддерживается и не развивается в настоящее время, что делает его применение на практике практически нецелесообразным.

**5 Заключение**

Подтверждением возможности создания коммерческих продуктов с применением языков логического программирования может служить реализация известного пакета Autocad с использованием языка Лисп. Другим примером является Канадская национальная система бронирования авиабилетов которая реализована на языке Python, поддерживающем логическую парадигму.

Многие идеи логического программирования нашли свое отражение в развиваемой компанией Microsoft технологии .NET. Разработчики .NET решили добавить в C# и в другие языки платформы .NET проблемно-ориентированное расширение LINQ, поддерживающее интегрированный язык запросов. В LINQ можно использовать l-функции, которые представляются в виде деревьев выражений, а в C# также в определенной степени поддерживаются идеи логического программирования.

Последнее время интерес к языкам логического программирования заметно растет в связи со свойственной им возможностью естественного распараллеливания вычислений, что делает их гораздо более перспективными в условиях появления многоядерных вычислительных машин.

Наконец, данные языки обладают еще одним преимуществом, имеющем место только в теоретическом программировании (Computer Science). Благодаря их декларативной семантике программы на этих языках представляют собой уже готовые математические объекты, которые могут исследоваться и обрабатываться с использованием математических методов. Это особенно важно т.к. программы, написанные на императивных или объектно-ориентированных (ОО) языках, практически не поддаются исследованию такими методами. И хотя сейчас эта область исследований находится еще в начальной стадии, возможно, именно с её помощью произойдёт научный прорыв в будущем.

**6 Список литературы**

1. Решение логических задач на Prolog. URL: https://pro-prof.com/archives/1299.
2. Сергиевский Г.М., Волченков Н.Г.: Функциональное и логическое программирование. М.: Академия, 2010.
3. Официальный сайт компании PDC. URL: https://www.pdc.com
4. Визуальный пролог. URL: https://ru.abcdef.wiki/wiki/Visual\_Prolog
5. Prolog – удивительный язык программирования. URL: https://habr.com/ru/post/124636.
6. Что такое логическое программирование и зачем оно нам нужно. URL: https://habr.com/ru/post/322900.
7. Логическое программирование. URL: https://ru.wikipedia.org/wiki/ Логическое\_программирование.
8. Лекции Сошникова Д.В. и Левинской М.А. Московский авиационных институт.