58. Язык логического программирования ПРОЛОГ. (Щербаков)

Обзор

Alain ColmerauerProlog - это язык программирования с декларативной логикой. Он был создан Аленом Колмерауэром и Робертом Ковальски примерно в 1972 году как альтернатива доминировавшим в Америке языкам программирования Lisp. Это попытка создать язык программирования, который позволяет выражать логику вместо тщательно прописанных инструкций для компьютера. Пролог имеет превосходный механизм сопоставления шаблонов, а также хорошее управление памятью. Он очень часто используется в приложениях искусственного интеллекта.

Выполнение программы на языке Prolog эквивалентно перебору дерева возможностей и определению объектов, удовлетворяющих заданным правилам. Это эквивалентно доказательству с помощью предложения. Prolog имеет отличный механизм обратного хода, поэтому при исследовании дерева он всегда может вернуться на другую ветвь, если текущая ветвь не содержит ответа. Этот подход эквивалентен поиску по глубине, который известен как эффективный алгоритм для обхода деревьев с большим количеством узлов.

Иногда может существовать несколько ответов, которые являются истинными в данных обстоятельствах. Это происходит потому, что программа не завершается, как только найден первый ответ, а продолжает работать до тех пор, пока не будет проверено все дерево возможностей.

Пролог - это разговорный язык, то есть пользователь и компьютер ведут "беседу". Сначала пользователь задает объекты и отношения, существующие между ними. Затем Prolog может ответить на вопросы об этом наборе данных.

Пролог имеет очень компактный синтаксис. С точки зрения разработчиков, сложность программирования на Prolog заключается в нечеткости человеческого мышления. Пролог облегчает кодирование, так как синтаксис очень короткий и точный.

Основы языка Пролог

Пролог состоит из фактов и правил. Факты определяют природу объекта. Правила - это свойства, которые могут быть получены из отношений.

Факты:

Объект, определяемый фактом, может принадлежать к определенной категории. Например, "Джон - мужчина" может быть закодирован как male(john). Факты также представляют отношения, которые являются истинными в данной области. Эти факты могут включать более одного объекта. Например, утверждение parent(john, tom). Это означает, что существует отношение между Томом и Джоном, называемое родительским. Порядок пунктов (объектов) в этом отношении имеет значение. Это утверждение означает, что Джон является родителем Тома, но обратное утверждение не верно.

После установления нескольких фактов можно задавать вопросы об этих отношениях. Например, на вопрос ?- parent(john,tom) будет получено "да", а на ?- parent(jane,tom) - "нет", поскольку ни один факт не говорит ничего о Джейн по отношению к Тому. Вопросы также могут быть комбинированными, когда ответ зависит от истинности нескольких фактов.

Правила:

Правила - это свойства, которые могут быть получены из отношений. Новое отношение может быть выведено из уже существующих отношений. Например, из нового отношения "Джон - отец Тома" следует, что Джон - родитель Тома, а Джон - мужчина, при условии, что отношения "родитель" и "мужчина" уже определены.

Это может быть закодировано следующим образом; father(john,tom) :- male(john), parent(john,tom). Символ ":-" означает "если". В данном контексте, если RHS истинно, то LHS также истинно.

Запятая между ними представляет собой конъюнкцию вычислений. Это означает, что оба отношения male и parent должны быть истинными, чтобы заключить отношение father.

Переменные:

Переменная представляет объект в условиях, когда имя не имеет значения. Важно, существует ли объект, отвечающий критериям, или нет. Это означает, что можно задавать такие вопросы, как "Кто дети Джона?". Например, следующий код: ?- parent(john,X) будет означать, существует ли X такой, что Джон является родителем X. Если X существует, Prolog вернет одно из возможных значений X, иначе ответ будет отрицательным. Если задать вопрос, Prolog также может вернуть все другие возможные ответы, если они существуют. Prolog может различать переменные и имена конкретных объектов, поскольку любое имя, начинающееся с заглавной буквы, считается переменной.

Рекурсия в Prolog:

Рекурсия - это техника, которая позволяет пользователю реализовать структуры данных, такие как деревья, где для ответа на вопрос необходимо различное количество шагов. Рекурсия использует несколько случаев, включая базовый случай, хотя количество случаев обычно всего два.

Например, отношение "ancestor", которое находит предков переменной X. Предком может быть непосредственный родитель X, в противном случае необходимо проверить родителя parent X и так далее.

ancestor(X, Y) :- parent(X, Y).                  
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).

Как Prolog отвечает на вопросы?

Вопрос обычно представляет собой последовательность одной или нескольких целей. Удовлетворить цель означает продемонстрировать, что она логически следует из набора уже определенных правил и фактов. Если вопрос содержит переменные, Prolog также найдет возможные значения этих переменных, чтобы сделать вопрос истинным (если таковые существуют)

Prolog интерпретирует все заданные факты и правила как набор аксиом, а вопрос пользователя - как предположительную теорему. Затем он пытается решить теорему, чтобы показать, что она логически следует из аксиом. Пролог следует цепочке правил в обратном порядке. Так, если дана теорема, то проверяется, можно ли заменить голову вычисления какими-либо другими правилами, которые, соответственно, становятся новыми целями. Если данная теорема верна, то цепочка должна завершиться, как только цели могут быть заменены на уже установленные факты. Любая неудачная попытка произвести замену вызывает обратный ход и попытку альтернативного пункта.

Вот пример того, как Prolog может быть использован для решения очень известной рекурсивной головоломки: башни Ханоя:

### hanoi(N) :- move(N, left, centre, right). move(0, \_, \_, \_) :- !. move(N, A, B, C) :- M is N-1, move(M, A, C, B), inform(A, B), move(M, C, B, A). inform(X, Y) :- write([move, a, disc, from, the, X, pole, to, the, Y, pole]),nl.

Метапрограммирование

Метапрограмма - это программа, которая принимает другие программы в качестве данных. Большинство декларативных языков программирования, особенно Prolog, достаточно мощны, чтобы обладать другими языками и парадигмами программирования. Это происходит потому, что в Prolog реализована функция манипулирования символами. Эта возможность позволяет заменить термин или аргумент предикатом без потери общего логического смысла. Например:

### good\_at\_prolog(X) :- good\_at(X,prolog).

Для реализации метапрограммирования не требуется модификация синтаксиса, так как предикаты могут быть заменены терминами, клаузы - списками предикатов и так далее. Интерпретатор также не нуждается в модификации, поскольку вычисления не меняют своего логического смысла.

Помимо возможности реализации других языков, Prolog также является метаинтерпретатором (интерпретатором для Prolog, написанным на Prolog). Эта особенность декларативного метапрограммирования делает его очень полезным для создания прототипов, реализации новых идей и разработки новых языков.

Метаинтерпретатор Prolog

Метаинтерпретаторы отличаются от обычных интерпретаторов тем, что предоставляют дополнительную функциональность. Они вводят программу и цель и выводят цель относительно программы. Это эквивалентно доказательству того, что программа логически подразумевает цель; поэтому метаинтерпретатор отслеживает выполнение и создает дерево доказательства. Для того чтобы составить дерево доказательств, необходимо разбить "размер зерна" интерпретатора. Это делается с помощью встроенного предиката из библиотеки Prolog. Это позволяет разделить вычисления на голову и тело и проанализировать их по отдельности. Тело, в зависимости от его сложности, может быть единичной клаузой или правилом, которое будет снова разбито на последовательность целей. Голова, однако, никогда не должна быть переменной. Эта особенность трассировки в Prolog ограничивает количество вызовов подцелей и гарантирует, что интерпретатор никогда не выполнит бесконечный цикл и не исчерпает память. Таким образом, последовательность разбиения других целей должна в конце концов закончиться.

For example:  
Computation: **(Head, Body)**  
Body: **(Goal, Other Goals)**  
Other goals: **(unit computation)**