Школа современного программирования

Соловьев Игорь Павлович

ИЗУЧЕНИЕ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ

НА ПРИМЕРЕ ЗАДАЧ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Задачи *искусственного интеллекта* (ИИ) - естественная и многообещающая область применения компьютеров и науки о них. Уже в начале 50-х годов, почти сразу после появления первых компьютеров, многие группы исследователей разных стран пытались расширить традиционную область применения компьютеров (сложные математические и экономические расчеты) такими задачами, которые обычно оставляли живому интеллекту. Сюда можно отнести распознавание образов (текстовых, графических, объемных), причем как восприятие образов, так и их интерпретацию, решение математических задач и автоматическое доказательство теорем, всевозможные игры (в том числе и азартные), понимание естественного языка, медицинский диагноз, составление расписаний и многое другое.

У перечисленных областей исследования имеется ряд характерных черт. Прежде всего, это *символьная* форма представления знаний о предметной области. Символы и цепочки символов используются для представления объектов самой различной природы - текстовой, графической, числовой. Еще одна важная особенность задач ИИ - отсутствие (незнание) простого и эффективного алгоритма решения. Обычно в таких задачах возникает проблема *выбора* направления решения из многих вариантов в условиях неопределенности. Говорят, что задачам ИИ присущ *недетерминизм.*

Обратимся к тому, какие средства формализации задач ИИ и поиска их решения предлагает информатика.

Императивный стиль программирования...

НЕМНОГО ИСТОРИИ

Несколько первых поколений компьютеров развивались в русле так называемой архитектуры фон Неймана. Конструкция таких машин предполагает наличие центрального процессора с локальной памятью процессора, обычно называемой регистровой, и достаточно большой однородной памяти (массива ячеек), называемой оперативной или основной, в которой хранится последовательность исполняемых инструкций и данных (программа). В целом (с небольшими вариациями) работа машины подчиняется следующей циклической схеме. Процессор извлекает из основной памяти очередную инструкцию, выбирает из основной памяти необходимые данные и загружает их в регистровую память, вы-

Декларативный стиль программирования...

полняет логическую или арифметическую инструкцию над содержимым регистров и отсылает полученные значения из регистров в основную память.

Архитектура фон Неймана фактически предопределила направление развития средств создания программного обеспечения вычислительной техники. Программы, написанные для таких машин, представляют собой упорядоченные последовательности инструкций, дополненные небольшим набором команд передачи управления. Языки программирования высокого уровня, реализующие основные принципы архитектуры фон Неймана, получили название *императивных* (императив - приказание, инструкция). Примерами подобных языков можно считать Фортран, Алгол 60, Алгол 68, PL/1, Паскаль, Си и многие другие.

Императивные языки оказали большое влияние на развитие науки и практики программирования. Они позволяли создавать довольно большие программные комплексы за приемлемое время. В учебной аудитории эти языки оказались достаточно хороши для изучения относительно несложных или небольших по объему реализации алгоритмов. Однако нужно ясно осознавать, что эти языки в значительной мере аккумулируют в себе технические особенности первых поколений вычислительной техники.

Технологическая революция в области вычислительной техники и потребности практики в новых приложениях привели к нарастанию сложности программных комплексов и одновременно к появлению серьезных претензий к надежности программного обеспечения, создаваемого в рамках архитектуры фон Неймана. Ограниченность традиционной архитектуры с самого начала понимали и исследователи в области ИИ. Совместными усилиями теоретиков и практиков в конечном счете были разработаны новые подходы в области создания программных средств решения алгоритмических задач. Обратимся к одному из таких направлений - логическому программированию (ЛП).

Математическая логика использует отточенный формальный язык для представления знаний об объектах той или иной предметной области, включая явные средства выражения гипотез и суждений. Подобные качества роднят логику и искусство программирования. Понятно, что идея непосредственного применения логики в качестве средства программирования возникла практически одновременно с первыми императивными языками. Главная особенность такого подхода состоит в том, что программа (логическая) состоит из набора утверждений (аксиом), а вычисление, выполняемое под управлением такой программы, представляет собой логический вывод некоторого целевого утверждения - искомого результата. Вывод производится из аксиом программы по правилам математической логики, причем эти правила применяются автоматически, программист не должен их специально указывать.

...довольно часто и авторы пасуют перед своими программами...

Часто стиль программирования, проповедуемый в рамках направления ЛП, называют *декларативным* (в противоположность императивному), поскольку целевое («вычисляемое») утверждение программы заранее декларирует (объявляет) искомый результат. При этом программист в своей программе не должен описывать шаг за шагом весь процесс вычислений, доверяя поиск решения *логической машине вывода.*

Привлекательность применения логики в программировании состоит прежде всего в том, что в результате постепенного уточнения формулировки задачи она приобретает все более ясную форму, понятную как создателю программы, так и ее возможным читателям (потребителям). Особенно хорошо язык логики подходит для формулирования задач ИИ. Все это объясняется тем, что язык логики опирается на общие законы человеческого мышления, а не на технические особенности кодирования для вычислительной машины того или иного типа.

Даже самые активные энтузиасты императивного программирования не станут утверждать ничего подобного о своих программах, поскольку детальная проработка сложных программ, написанных на традиционных языках, зачастую настолько их усложняет (и даже запутывает), что никто, кроме авторов, понять их не может (довольно часто и авторы пасуют перед своими программами).

Временем рождения современных реализаций идеи ЛП принято считать начало 70-х годов. К этому времени после целой череды экспериментальных языков группой Алана Колмероэ в Марселе была создана (еще неэффективная) реализация языка, заменившего последовательные вычисления машины фон Неймана на логический вывод. Интересно, что новый язык, названный Прологом (ПРОграммирование ЛОГическое), предназначался для анализа текстов, написанных на естественном языке, то есть для решения задач, обычно относимых к области ИИ. Приблизительно в те же годы были разработаны и теоретические основы нового направления в программировании. Основными результатами в этой области мы обязаны Алану Робинсону и Роберту Ковальскому.

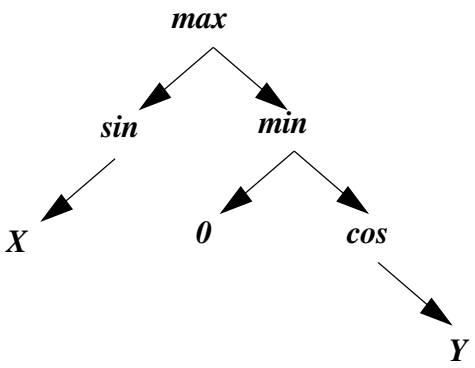

Рисунок 1. Древовидная структура терма *max(sin(X), min(0,cos(Y)))*

Первая реализация Пролога, выполненная, кстати, на Фортране, заинтересовала специалистов, но не получила широкого распространения по причине низкой эффективности (поскольку работала лишь как интерпретатор и не имела компилятора). Мешал распространению Пролога и накопившийся к этому времени у специалистов (в основном американских) общий скепсис по отношению к идее ЛП, поскольку все реализации предшественников Пролога были также неэффективны.

Настоящая революция в этой области произошла в конце 70-х, когда Дэвид Уоррен из Эдинбургского университета создал первый компилятор для языка Пролог. Этот компилятор работал настолько эффективно, что скепсис специалистов немедленно сменился всеобщим энтузиазмом. С тех пор и до настоящего времени направление ЛП успешно развивается и поддерживается как профессионалами, так и просто любителями программирования. Реализация Уоррена получила название *эдинбургской* и стала фактическим стандартом Пролога - наиболее распростаненного языка ЛП. Отметим, что эта реализация была выполнена на машине DEC-10. По этой причине ее иногда называют Пролог-10.

ПРОЛОГ

Познакомимся теперь поближе с особенностями программирования на языке Пролог, причем за основу возьмем именно эдинбургский вариант реализации.

Синтаксис прологовских программ чрезвычайно прост и прозрачен, даже немного аскетичен. Программа состоит из упорядоченного набора предложений:

предложение$_1$
предложение$_2$
...
предложение$_n$

причем каждое из предложений может принадлежать одному из двух типов - *фактов* и *правил*.

Правила записываются в одной из двух эквивалентных форм:

заключение ← *посылка.*

или

заключение :- *посылка.*

причем последняя форма используется чаще. Смысл правила: если верна посылка, то верно и заключение. Пример правила:

X < *Y* :- *X* = *1 и Y* = *2.*

Прочитать такое правило можно следующим образом: "*X* меньше *Y*, если *X* = 1 и *Y* = 2". Здесь же заметим, что имена переменных в Прологе начинаются с прописной буквы.

Факты записываются в виде правил без посылок:

заключение.

Смысл предложения без посылки: факт - это безусловная истина. Пример факта:

1 = 1.

Основной синтаксической единицей, кирпичиком программных конструкций прологовских программ, является терм. Терм - это константа (число или символ), переменная, либо математическое выражение вида $f(t_1,...,t_n)$, где число аргументов выражения $n \geq 0$, аргументы $t_1,...,t_n$ - термы, f - некоторая символьная константа, называемая (главным) *функтором* или *функциональным* именем данного терма. Терм последнего вида называется также *функциональным*. Примеры термов: число 5, символ *banana*, функциональные термы *sort(1), father(x, y), max(sin(X), cos(Y))* и т.п. В частном случае при $n = 0$ функциональный терм приобретает вид *f()*, причем многие реализации языка разрешают писать просто *f*.

Имена переменных в языке Пролог начинаются с прописной латинской буквы, а символьные константы (иногда их называют *идентификаторами*) начинаются со строчной латинской буквы. В записи имен переменных и символьных констант могут использоваться также десятичные цифры и знак подчеркивания, например *Box_Pos_1* или *monkey_has_banana.*

Некоторые функциональные термы, образованные с помощью общеупотребительных в математике функциональных имен, наряду со стандартной префиксной записью допускают и эквивалентную инфиксную. Например, можно писать +(1, 2) и 1 + 2, =(*a, b*) и *a* = *b* и т.п. Термы именно такого типа встретились в приведенном выше примере прологовского правила.

При наличии некоторой доли фантазии в синтаксической записи терма можно усмотреть древовидную структуру. На рисунке 1 изображена структура терма *max(sin(X), min(*0, *cos(Y)))* в виде стилизованного дерева. Корень этого дерева расположен вверху, ветви ориентированы сверху вниз, листьями являются простейшие аргументы функциональных термов - константы и переменные. В узлах дерева, служащих развилками, находятся функциональные имена.

Важно подчеркнуть, что терм (структура) - это лишь «неодушевленная» последовательность символов, а не математическое выражение, которое следует вычислить. Смысл, который вкладывается в каждый терм, определяется его местом в конкретной программе.

Вернемся к структуре прологовского предложения. Заключения и посылки фактов и правил строятся из термов. С синтаксической точки зрения правила имеют следующий вид:

t_0 :- t_1, t_2,..., t_n.

где терм t_0 представляет заключение правила, называемое также заголовком правила, а термы t_1, t_2,..., t_n представляют посылку правила. Как мы видим, посылка может состоять из нескольких термов, разделенных запятыми. Факты состоят только из одного заголовка, на них можно смотреть как на правила с пустой посылкой. Все термы t_i, из которых строятся факты и правила, трактуются как *утверждения*, запятые, разделяющие утверждения в посылках - как логические связки *и*, знак :- трактуется как логическое следование (читается справа налево).

Приведем пример очень простой программы на языке Пролог (с подобных примеров начинаются многие учебники по этому языку).

```
отец(николай, евгений).
отец(николай, василий).
отец(евгений, александр).
отец(евгений, иван).
отец(василий, алексей).
сын(X, Y) :- отец(Y, X).
дед(X, Y) :- отец(X, Z), отец(Z, Y).
брат(X, Y) :- отец(Z, X), отец(Z, Y).
```

Наша программа содержит знания о родственных связях некоторой семьи (небольшой фрагмент таких знаний). Поскольку символьные константы в языке

Пролог принято начинать со строчной буквы, нам пришлось именно таким образом записывать используемые в программе собственные имена. (Кроме того, оговоримся, что реальные системы программирования часто ограничивают использование букв русского алфавита: их нельзя использовать в функциональных именах, а символьные константы в таких случаях заключаются в апострофы.)

Программа содержит пять фактов с главным функтором *отец*: *николай* - отец *евгения*, *николай* - отец *василия* и т.д., а также три правила. Первое утверждает: если некто *Y* - отец *X*, то *X* - сын *Y*. Второе: если некто *X* - отец некоторого *Z* и этот же *Z* - отец *Y*, то *X* - дед *Y*. Третье правило интерпретируется аналогичным образом.

Считается, что группы фактов и правил с одинаковым главным функтором задают определение одного свойства или отношения (называемого также *предикатом*). В нашем примере пять фактов определяют отношение *отец*: два человека находятся в этом отношении, если в программе явно представлен соответствующий факт, либо его можно доказать на основании содержащейся в программе информации. На рисунке 2 представлена графическая иллюстрация определенных в программе семейных отношений.

Предположим, что в нашем распоряжении имеется некоторая система программирования на языке Пролог и что эта система запущена и работает в диалоге с нами. Введем в систему текст нашей программы. Что можно вычислить или *вывести*, как принято говорить в Прологе, основываясь на такой программе? Для того, чтобы это узнать, программе нужно задать вопрос или сформулировать *цель*. Записывается это в форме

?- t_1, t_2,..., t_n,

где последовательность термов t_1, t_2,..., t_n имеет тот же смысл, что и в посылках правил - это набор утверждений, одновременная истинность которых должна быть выведена из программы.

Например, на вопрос

?- *отец(николай, василий).*

программа ответит *да (yes)*, а на вопрос

?- *отец(василий, александр).*

программа ответит *нет (no)*. Ответы на оба вопроса программа дает после просмотра (сверху вниз) своей базы знаний и непосредственного сравнения содержания вопроса (в данном случае одного терма) с заголовками содержащихся в ней предложений (в данном конкретном случае достаточно просмотреть только факты).

Еще одна форма вопроса:

?- *отец(николай, X).*

Здесь вместо второго аргумента проверяемого (выводимого, доказываемого) утверждения записана переменная. Это указание программе найти значение переменной *X*, при котором утверждение *отец(николай, X)* станет истинным. Поиск решения снова осуществляется последовательным просмотром предложений и

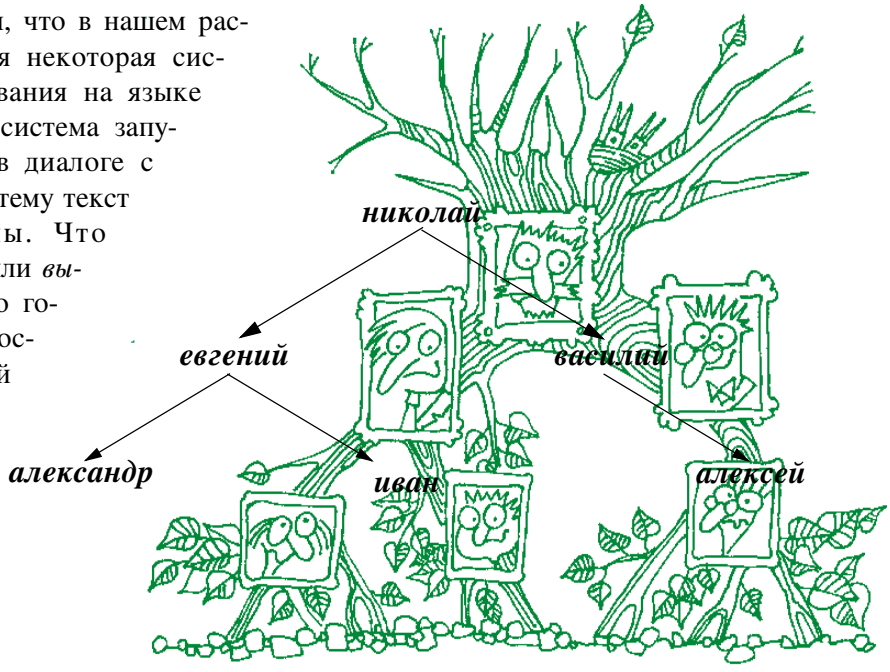

Рисунок 2. Семейная база данных.

сопоставлением терма *отец(николай, X)* с заголовками правил. Поскольку переменная *X* не имеет значения, она может быть сопоставлена с любым значением второго аргумента того или иного заголовка. Первый подходящий заголовок - *отец(николай, евгений)*, поэтому на наш вопрос программа ответит *X = евгений*.

Мы знаем, что наша задача имеет и второе решение. Если после получения первого решения ввести в систему знак ‘;’ (стандартное соглашение эдинбургского Пролога), то поиск решения будет продолжен с той позиции в базе, в которой он был остановлен. Очередное решение - *X = василий*. Если после этого мы еще раз попросим систему продолжить поиск, то получим ответ *по*.

До сих пор для ответов на наши вопросы использовались только факты. Для того, чтобы показать, как работают правила, выясним, кто является дедом *алексея*:

?- *дед(A, алексей).*

Последовательный просмотр предложений программы приводит к тому, что терм *дед(X, иван)* сопоставляется с заголовком соответствующего правила, переменная *A связывается* с его первым аргументом - переменной *X*, значение *иван* связывается со вторым аргументом - переменной *Y*, а наш запрос (исходная цель) сводится к новой цели:

?- *отец(A, Z), отец(Z, алексей).*

состоящей из двух подцелей.

Процессом вычислений по заданной нами программе руководит *логическая машина вывода*, то есть машина, каждый шаг которой опирается на правила математической логики. Благодаря этим правилам доказательство заключения некоторого правила автоматически сводится к доказательству его посылки. Так что, если доказательство новых подцелей нашей программы закончится успехом, успешным будет и доказательство исходной цели.

Подцель *отец(A, Z)* дает решение *A = николай, Z = евгений*. Но доказательство подцели *отец(евгений, алексей)* заканчивается неуспехом!

Мы знаем, что наша задача имеет решение, но этого не знает машина вывода. Однако эта машина имеет особый механизм поиска - бэктрекинг (механизм возврата), с помощью которого мы, в конце концов, получим решение (это не всегда возможно в реальных системах логического программирования).

В случае, когда вычисления заходят в тупик, машина вывода прослеживает цепочку вывода в обратном направлении, пытаясь найти *точку выбора* - такую подцель, при доказательстве которой можно было бы найти иное решение. В нашей задаче такой подцелью является *отец(A, Z)*. Машина *пересматривает* свой выбор: ранее найденное решение отменяется, поиск решения данной подцели продолжается с той точки, где он был остановлен. В результате находится новое решение *A = николай, Z = василий.* Тогда вторая подцель переформулируется:

?- *отец(василий, алексей).*

Доказательство данной подцели, а с ней и исходной цели, заканчивается успехом. Помимо успеха машина вывода возвращает и значения переменной *A*, содержащейся в исходном запросе: *A = николай*, другими словами, машина вывела, что дед *алексея - николай.*

Рассмотренный нами тип запроса с неопределенными значениями переменных (в Прологе говорят - *несвязанных* переменных) относится к числу так называемых *экзистенциальных*. Логическая машина вывода обладает возможностью не только проверять истинность того или иного утверждения, но также и *подбирать* значения параметров, при которых утверждения становятся истинными (это называют также *поиском решения* проблемы).

ИНТЕЛЛЕКТ

Наш первый пример логической программы достаточно прост и служит единственной цели - показать основные особенности языка Пролог. Хорошим примером, демонстрирующим применение идеи логического программирования для решения задач искусственного интеллекта, может служить известная задача “обе-

зьяна и банан".

Представим себе комнату, в которой томится голодная обезьяна. В комнате к потолку подвешен банан, причем довольно высоко. Обезьяна с пола не может дотянуться до банана. Бедное животное мечется по комнате, пытаясь найти способ добраться до плода. У окна этой комнаты находится ящик, которым обезьяна, если она достаточно сообразительна, могла бы воспользоваться как подставкой. Доберется ли обезьяна до банана?

Мы не предлагаем ставить опыты на животных. Вместо этого представим себе поиск решения данной проблемы как игру для одного игрока, в которой определен набор допустимых состояний, исходное состояние, заключительное состояние и правила поведения игрока (обезьяны) или правила перехода из одного состояния в другое.

Каждое состояние определяется взаимным расположением в комнате обезьяны, ящика и банана. Для определенности выделим в комнате три возможных положения, в которых могут располагаться обезьяна и ящик - у двери, у окна и под бананом. Назовем эти положения, соответственно, *at_door*, *at_window* и *under_banana*. Кроме того, обезьяна может находиться на полу (*on_floor*) или взобраться на ящик (*on_box*). Предположим, что в начале игры обезьяна находится у двери. Заключительное состояние - счастливая обезьяна держит в лапах банан, стоя на ящике.

Перечислим действия, которые может предпринять обезьяна, другими словами - правила перехода нашей игры. Обезьяна может переходить (перебегать?) по полу из одного положения в другое, передвигать ящик из одного положения в другое, если ящик находится там же, где обезьяна, забираться на ящик и спускаться с него, а также схватить банан, если она стоит на ящике прямо под бананом.

Процесс (игра) развивается в дискретном времени. Это означает, что обезьяна, находясь в некотором состоянии, выполняет одно допустимое действие (ход) и немедленно оказывается в новом состоянии, и т.д.

Опишем состояние игры синтаксическими средствами языка Пролог. Это можно сделать несколькими различными способами. Выберем следующий. Пусть текущее состояние определяется парой объектов - состоянием обезьяны и состоянием ящика. Состояние обезьяны можно представить синтаксической структурой (термом) вида *monkey(HPos, VPos, Want_banana)*, где *monkey* - имя структуры, а в скобках перечислены три переменные, обозначающие, соответственно, горизонтальную позицию обезьяны (*HPos*), вертикальную позицию обезьяны (*VPos*) и наличие у обезьяны банана (*Want_banana*). Состояние ящика описывается его положением на полу в комнате, для чего достаточно одной переменной, например *Box_Pos*. Конкретные имена переменных в программе могут немного изменяться в зависимости от контекста.

Определим значения, принимаемые указанными переменными в программе, которую мы собираемся написать на языке Пролог. Для этого воспользуемся записью вида

переменная ∈ *множество значений*

Итак,

$HPos \in \{at\_door, at\_window, under\_banana\}$,

$BoxPos \in \{at\_door, at\_window, under\_banana\}$,

$VPos \in \{on\_floor, on\_box\}$,

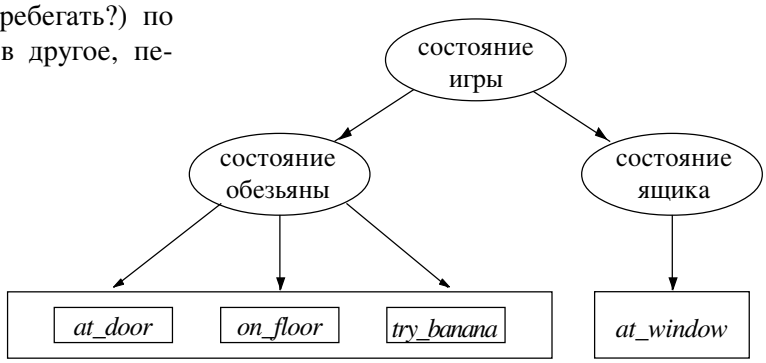

Рисунок 3. Исходное состояние игры "Обезьяна и банан".

Want_banana ∈ {*try_banana, has_banana*}.

На рисунках 3 и 4 в виде древовидных структур изображены исходное и заключительное состояния нашей игры, соответственно.

Приведем полный текст нашей программы:

```
% обезьяна и банан
step(monkey(under_banana, on_box,
    try_banana), under_banana,
    grasp,              % схватить банан
    monkey(under_banana, on_box,
    has_banana), under_banana).
step(monkey(HPos, on_floor, Want_banana),
    HPos,
    climb,              % залезть на ящик
    monkey(HPos, on_box, Want_banana),
    HPos).
step(monkey(HPos1, on_floor, Want_banana),
    HPos1,              % передвинуть ящик
    move(HPos1,HPos2),  % из HPos1 в HPos2
    monkey(HPos2, on_floor,Want_banana),
    HPos2).
step(monkey(HPos1, on_floor, Want_banana),
    Box_Pos,            % перейти
    go(HPos1,HPos2),    % из HPos1 в HPos2
    monkey(HPos2, on_floor, Want_banana),
    Box_Pos).
obtain(monkey(_, _, has_banana), _).
                % есть банан - завершить поиск
obtain(Monkey1, Box1) :- step(Monkey1,Box1,
                % выполнить очередной шаг
    Step,
    Monkey2,Box2),
    obtain(Monkey2,Box2).
                % продолжить поиск
goal(Monkey_Pos, Box_Pos) :-obtain(monkey
(Monkey_Pos, on_floor, try_banana),Box_Pos).
```

Текст программы начинается с комментария (цепочка символов, начинающаяся со знака %). Далее следует определение отношения *step*, описывающее правила перехода из одного состояния в другое. Первые два, а также четвертый и пятый аргументы отношения *step* указывают состояния обезьяны и ящика до очередного хода и после, третий аргумент - описание действия, совершаемого обезьяной. Так, первое предложение констатирует, что обезьяна может схватить (*grasp*) банан, если она стоит на ящике прямо под бананом. Второе предложение разрешает обезьяне залезть (*climb*) на ящик, если она стоит на полу прямо под бананом и рядом с ней находится ящик. Третье и четвертое предложения разрешают обезьяне передвинуть ящик или перейти самой из одной позиции в другую.

Далее описывается предикат *obtain* - основная движущая сила нашей игры. Первое предложение в определении - факт, описывающий заключительное состояние: обезьяна держит в лапах банан (*has_banana*). Второе предложение имеет два условия и, соответственно, порождает две подцели: сначала выполняется очередной шаг игры, затем вызывается предикат *obtain* (рекурсивно) с параметрами нового состояния игры, и т.д., пока не будет достигнуто заключительное состояние.

Последний предикат *goal* включен в программу для удобства. С его помощью можно сформировать цель

?- goal(at_door, at_window).

и запустить игру в начальном состоянии: обезьяна - у двери, ящик - у окна. Предикат *goal* формирует параметры для подцели *obtain* и передает ей управление. Получив такой запрос, наша программа ответит *yes*, что будет означать успешное окончание игры - обезьяна получит свою награду.

Наша программа описывает процесс игры исключительно логическими сред-

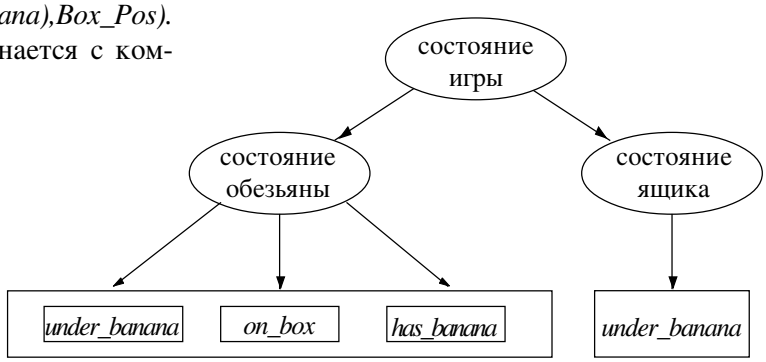

Рисунок 4. Заключительное состояние игры "Обезьяна и банан".

ствами в виде отношений участников и атрибутов игры. Это описание оказалось весьма кратким и выразительным, что еще раз оправдывает закрепившееся за подобным стилем программирования название *декларативный.*

Декларативный стиль позволяет нам указать, что именно мы хотим узнать или вывести из программы, не уточняя, как именно будет достигнута наша цель. Конкретная последовательность шагов, решающая поставленную задачу, определяется логической машиной вывода по правила математической логики и может быть весьма нетривиальной.

Жизнь, как всегда, богаче любых схем. Получив ответ *yes* в задаче про обезьяну и банан, мы, тем не менее, хотим узнать немного больше: как долго искала решение логическая машина и какие конкретные шаги все же ей пришлось предпринять (например, для того, чтобы не делать ошибок). Помогает получить ответы на такие вопросы встроенный отладчик, который принято включать в системы логического программирования. Запустив такой отладчик, мы может проследить всю трассу логического вывода и непосредственно увидеть точки возврата и выбора. Так, в нашей задаче мы можем увидеть, как обезьяна добегает до окна, забирается на ящик, слезает с него, затем двигает ящик под банан, снова забирается и, наконец, достигает заветной цели.

Доберется ли обезьяна до банана?

Как мы видим, путь к достижению цели может быть непрямым. Рекомендуем поэкспериментировать с предложенной программой, изменяя последовательность ее предложений.

В заключение отметим, что в настоящее время существует много реализаций языков логического программирования для различных операционных систем. Мы не беремся выделить лучшую из них, поскольку в каждой есть свои особенности. Скажем лишь, что программы, включенные в текст данной статьи, написаны и отлажены в системе Amzi! Prolog, работающей под управлением операционной системы Windows 95/98. Бесплатную 90-дневную копию этой программы можно найти в Интернете по адресу http:// www.amzi.com.

Литература.

1. И. Братко. Программирование на языке Пролог для искусственного интеллекта. М., 1990.
2. У. Клоксин, К. Меллиш. Программирование на языке Пролог. М., 1987.

НАШИ АВТОРЫ

Соловьев Игорь Павлович,
доцент кафедры информатики
мат.-мех. факультета СПбГУ.