Представление списков

В силу особенностей задач искусственного интеллекта, в которых предполагается накопление и пере - структурирование знаний, целиком и полностью оправдано использование списков. При этом используются списки, элементы которых сами могут являться списками. Списки, как рекурсивно определенные структуры (определение давалось ранее), удобно обрабатывать рекурсивно. При обработке всех элементов списка, требуется уметь «увидеть» признак конца списка – пустой список (в Prolog []).

Исторически первая синтаксическая форма представления списков как термов (в Prolog) была основана на использовании бинарного функтора “точка” (точечная пара). Чтобы такая пара была списком, достаточно, чтобы 2-ой элемент был списком (пустым), первый элемент может иметь любую структуру. Синтаксическая форма записи списка в виде терма: **“.”(1, [ ])** или **“.”(a, “.”(b, [ ])) .**
Облегченный (лаконичный) синтаксис Prolog позволяет эти списки записать:

[1, []] или **[1]** , [a, b, []] или **[a, b]** . В Prolog – символы **[…]** – это синтаксис (признак) особого терма, ничего, кроме термов Prolog не использует! .
Элементы списка для Prolog это аргументы терма. С такими термами система работает как обычно!. Пустой список (**[ ]**) не образует никакой структуры, по сути, он является особым атомом, обозначающим пустой список. Система Prolog неявно подразумевает его в качестве терминального элемента списка.

В Prolog существует более общий способ доступа к элементам списка. Для этого используется метод разбиения списка на начало и остаток. **Начало** списка – это группа первых элементов, **не менее одного**. Остаток списка – обязательно список (может быть пустой). Для разделения списка на начало, и остаток используется вертикальная черта (|) за последним элементом начала. **Остаток это всегда один** (простой или составной) **терм**. Свойство атома: его нельзя разбить на части. Если начало состоит из одного элемента, то получим: голову и хвост. При таком способе разбиения – список можно разбить на части не единственным способом:
[a, b, c, d] ~ [a, b, c, d | []] ~ [a, b, c | [d]] ~ [a, b | [c, d]] ~ [a | [b, c, d]] других способов разбиения этого списка нет!

То, что второй элемент списка (хвост – остаток) должен быть списком – это свойство списка, а не функтора «точка». Поэтому интерпретатор не обнаруживает ошибки в записи: [a | b]. Ошибка может быть обнаружена только при попытке расчленить хвост. Т.е. не всякая конструкция, внешне использующая списковую нотацию, представляет собой список.

В списковых структурах переменными могут быть представлены и голова и хвост:
[X | Y] или
отдельные элементы: **[**a, X, b | **[Y** | []**]]**.

Каждая из переменных может быть конкретизирована термом, произвольной структуры. В результате, мы можем получить списковую или нет структуру. При этом, каждая переменная может обозначать только один объект (терм). Система Prolog должна подобрать такие термы для последующей конкретизации ими переменных. Как ранее было сказано, она это делает с помощью алгоритма унификации, пытаясь так разделить список на начало и остаток, чтобы унификация была успешна. Списки могут являться аргументами других термов: вопросов, заголовков правил, термов тела правил. Таким образом, очевидно, что система должна уметь сравнивать списки. Естественно, что бессмысленно сравнивать терм – список с термом, например студент. Имеет смысл сравнивать списки со списками.

Примеры унификации списков:

- [X] = [] – унификация невозможна;
- [X] = [a | []] → [X] = [a] – унификация успешна, X = a;

- [X] = [a, b, c] – унификация невозможна;
- [X, b] = [a, b] → [X | [b]] = [a | [b]]] – унификация успешна, {X = a};
- [[X], b] = [a, b] – унификация невозможна, т.к. разная структура 1-х элементов;
- [a | Y] = [a, b, c] → [a | Y] = [a | [b, c]] – унификация успешна {Y = [a | [b, c]]}
- [a | Y] = [a, b | [c]] → [a | Y] = [a | [b, c]] – унификация успешна {Y = [b, c]}

Методы обработки списков

Для обработки списка: его самого или его элементов – системе требуется знать как это сделать, т.е. требуется знание.

Приведем несколько примеров.

1). Предикат list, который позволит проверить: является ли аргумент списком.

Для распознавания, является ли терм L списком, создадим знание о том как это сделать – предикат **list(L):**

По определению **L –**является списком: если он пустой (правило I), или, если он не пустой, но его хвост – является списком (правило II), т.е. это надо проверить. Очевидно, что второе правило определяет рекурсию. После задания вопроса, например list([1, 2, 3]) – с помощью успешной унификации терма вопроса и терма заголовка правила (list([1, 2, 3]) = list(L)), L будет конкретизировано списком [1, 2, 3] и начнется его анализ:

```
I     list(L) :–  L=[].                 «Да», если список пуст, а если нет, то проверить II
II    list(L) :–  L=[H|T], list(T).     если список не пуст,  то проверить хвост
```

Здесь: H – голова, а T – хвост. Но после выделения головы и хвоста (в II), голова далее не используется, а значит ее значение возвращать не надо – можно использовать анонимную переменную. Т.е.:

```
I     list(L) :–  L=[].
II    list(L) :–  L=[_|T], list(T).
```

Далее можно заметить, что первые проверки в обоих правилах являются проверками применимости этих правил, значит, эту проверку можно перенести в заголовки:

```
I     list([]).
II    list( [_ | T] ) :– list(T).
```

Эта версия наиболее эффективна: при выборе правила одновременно выполняется проверка – пустой ли список (если выбрано правило I, являющееся фактом, тогда можно ответить «Да» – на поставленный вопрос), а если нет, то прямо при выборе правила II (с помощью алгоритма унификации) список делится на голову и хвост. И, при проверке истинности тела, происходит анализ хвоста, с помощью этой же процедуры (рекурсия). На каждом очередном шаге рекурсии работа будет происходить с хвостом.

Приведем **схему описания формальной работы системы**, при задании вопроса: list([1, 2, 3]) (процесс согласования цели list([1, 2, 3])).

Введены обозначения:

ТР - текущая резольвента; ТЦ - текущая цель; шаг№№ - шаг начинается с появления очередной резольвенты и заканчивается: либо новой резольвентой, либо выводом – успех или неудача; ПРI или ПРII – попытка применения соответствующего правила, рядом указываем **равенства для унификации** и результат (главные функторы одинаковы, арности одинаковы – **проверить надо унифицируемость аргументов**);

т.к. на каждом шаге рекурсии система создает новые экземпляры нужных переменных, то к имени переменной добавляем номер шага (T1, T2, ...). Используем процедуру:

```
I     list([]).
II    list( [_ | T] ) :– list(T).
```

И вопрос (GOAL): list([1,2,3])

ТР: list([1,2,3])

шаг1: ТЦ: list([1,2,3]) *поиск знания с начала базы знаний*

ПРI: [] = [1, 2, 3] унификация невозможна => возврат к ТЦ, метка переносится ниже
ПРII: [_|T1] = [1, 2, 3] успех (подобрано знание)=> {T1 = [2, 3]}, проверка тела ПРII
ТР: list([2,3]) (резольвента изменилась в 2 этапа)
шаг2: ТЦ: list([2,3]) *поиск знания с начала базы знаний*
ПРI: [] = [2,3] унификация невозможна => возврат к ТЦ, метка переносится ниже
ПРII: [_|T2] = [2,3] успех (подобрано знание) => {T2 = [3]}, проверка тела ПРII
ТР: list([3]) (резольвента изменилась в 2 этапа)
шаг3: ТЦ: list([3]) *поиск знания с начала базы знаний*
ПРI: [] = [3]] унификация невозможна => возврат к ТЦ, метка переносится ниже
ПРII: [_|T3] = [3] успех (подобрано знание) => {T3 = []}, проверка тела ПРII
ТР: list([]) (резольвента изменилась в 2 этапа)
шаг4: ТЦ: list([]) *поиск знания с начала базы знаний*
ПРI: [] = [] успех (подобрано знание) => пустое тело заменяет цель в резольвенте
ТР: пусто; успех – однократный ответ – «Да», *метка – на ПРI*
Отказ от найденного значения (откат), возврат к предыдущему состоянию резольвенты
ТР: list([])
шаг5: ТЦ: list([]) *поиск знания от метки ниже*
ПРII: [_|T5] = [] унификация невозможна =>неудача, *надо включить откат, но метка (метки) в конце процедуры* – других альтернатив нет => **система завершает работу** с единственным результатом – «Да».

2). Предикат member, который позволяет проверить: является ли X элементом L, т.е. необходимо иметь два аргумента. Очевидно, что X надо сравнить с головой, т.е. в списке L надо выделить голову и сравнить с X. Это можно сделать несколькими способами:

а) I member(X, L):– L=[H|T], X=H.
II member(X, L):– L=[H|T], member(X,T).

Но в I : хвост нас не интересует, а X=H можно перенести в предыдущее условие, переход к ПР II произойдет, если в ПР I неудачно. В ПР II: нас не интересует голова, а X надо искать в хвосте T – это следующий шаг рекурсии:

б) т.е. оптимизация:
I member(X, L):– L=[X,_].
II member(X, L):– L=[_|T], member(X,T).

Но разбиение L на части можно выполнить при подборе знания, т.е. в заголовке:

в) поэтому, дальнейшая оптимизация:
I member(X, [X|_]).
II member(X, [_|T]):– member(X, T).

Как видели в предыдущем примере, после получения утвердительного ответа на вопрос, с помощью ПРI, система все-таки анализирует возможность использования и ПРII, хотя очевидно, что уже можно ответить «Да» на поставленный вопрос. Можно на это повлиять:

г) используем отсечение – оптимальный вариант:
I member(X, [X|_]):– !.
II member(X, [_|T]):– member(X, T).

Если подобрано ПРI , то предикат ! станет текущей резольвентой, а он истинный, т.е. система получит «Да». При попытке же отменить ! система завершит использование процедуры, ПРII использоваться не будет.

д) без использования отсечения "оптимального" поведения можно добиться с помощью дополнительной проверки, но в этом случае все-таки происходит анализ ПРII:
I member(X, [X|_]).
II member(X,[H|T]):– X<>H, member(X, T).

3). Объединение списков – предикат **append.** Этот предикат должен используя два списка, например, L1 и L2 получить третий- результирующий список L, т.е. у предиката три аргумента (append(L1, L2, L)), пусть результат – это третий аргумент, тогда вопрос можно задать: append([a, b], [c, d], R). Хотим получить R=[a, b, c, d]. Рассмотрим возможные ситуации:

Если 1-й список – пустой, то результат равен второму списку (ПРI). Если первый список не пуст, то надо переписать **в результирующий список**, на место головы – голову первого списка (ПРII), а хвост результата еще не определен (Т3). После чего, **в голову хвоста результирующего списка** надо переписать голову хвоста 1-го списка (рекурсия), и т.д., пока первый список не станет пустым, после чего остатком результирующего списка должен стать второй список.

I append([], L2, L2).
II append([H|T], L2, [H|T3]):- append(T, L2, T3).

Частичное формирование результирующего списка (переписывание головы первого списка в голову результата – ПРII) обеспечиваем на этапе выбора соответствующего правила (заголовка). Отметим, что сцепление элементов в результирующем списке формально реализуется конкретизацией частей списка (причем не сразу всех). Здесь отсечение в первом правиле можно не использовать, т.к., в случае пустоты первого списка, второе правило выбрано быть не может (хотя попытка и происходит), т.к. первый список (он пустой), в этом случае, разбить на части нельзя.