ДОПОВІДІ
НАЦІОНАЛЬНОЇ
АКАДЕМІЇ НАУК
УКРАЇНИ

ІНФОРМАТИКА

doi: https://doi.org/10.15407/dopovidi2017.10.018
УДК 004.8

А.Ф. Кургаев

Институт кибернетики им. В.М. Глушкова НАН Украины, Киев
E-mail: afkurgaev@ukr.net

Формализация списков в метаязыке нормальных форм знаний

Представлено академиком НАН Украины А.В. Палагиным

Впервые предложена формализация предикатов на списках в метаязыке нормальных форм знаний, базируясь на известных описаниях этих понятий на Прологе, использующих списковый домен. Среди описанных предикатов: добавление элемента в список, удаление элемента, удаление повторов, принадлежность элемента списку, поиск последнего элемента списка, поиск соседних элементов списка, конкатенация списков, реверс и др.

Ключевые слова: *метаязык нормальных форм знаний, список, предикат, определение, рекурсия.*

Постановка задачи. Простота структуры списков, эффективность представления в форме списков разнородной информации в памяти компьютеров и естественность их использования в символьной обработке информации многие годы обеспечивают решающие преимущества и популярность старейшего (после Фортрана) семейства высокоуровневых языков программирования Лисп (LISP, от англ. LISt Processing — “обработка списков”) в решении задач искусственного интеллекта. Единство структуры данных признается достоинством системы программирования: “Лучше иметь 100 функций, которые работают с одной структурой данных, чем 10 функций, работающих с 10 структурами” [1, п. 2.2.1].

Списки позволяют представить практически любые неоднородные и/или иерархические структуры данных, возможные в символьных вычислениях, и поддерживают функциональный стиль программирования. В виде списков удобно представлять формулы, функции, деревья, графы, множества и многие другие сложные объекты. На протяжении истории появился ряд диалектов языка Лисп: Common Lisp, Haskell 98, Scheme, Standard LISP и др., позволяющих использовать, наравне со списками, структуры, определённые пользователем, и такие структуры, как: vector, hash table [1—3].

Список — один из самых простых и полезных типов структур составных объектов логического программирования, позволяющий во многих случаях улучшить “читабельность” программ [4—6].

В метаязыке нормальных форм знаний [7, 8], в отличие от ЛИСПа и Пролога, нет такой встроенной структуры данных, как *список*, эффективность которой обоснована не толь-

ко теоретически, но и обширной практикой использования этих языков при создании систем искусственного интеллекта. Потому, для практического использования метаязыка нормальных форм знаний (НФЗ) важно реализовывать и понятие списка, и предикаты на списках, опираясь на стандартные домены метаязыка НФЗ.

В качестве базовой используем структуру области данных из [7, 8]:

• домен D представлен двумя независимыми массивами — входным INP и выходным OUT, с которыми связаны переменные m и n, принимающие значения текущих координат соответствующих массивов;

• две библиотеки одноименных предикатов — библиотеку анализа над данными из INP и библиотеку порождения над данными из OUT;

• набор системных процедур, управляющих этими элементами домена D:

RB — истинный предикат переключения библиотек;

RIO — истинный предикат переключения массивов INP и OUT;

UIO — истинный предикат объединения/разделения массивов INP и OUT.

Во всех последующих формальных описаниях предикатов на списках использована нотация метаязыка НФЗ [7, 8], включая элементарные операции: распознавание, распознавание со следом и порождение над информационными структурами, которые обозначаются соответственно знаками “?”, “#” и “!”, присоединяемыми к имени понятия, вводящего некоторую информационную структуру.

1. Представление задачи. В терминах метаязыка НФЗ всякая задача формулируется как доказательство категорического суждения $P(x)$, предикат P которого задан именем, а субъект (возможно, многоместный аргумент x) задан последовательностью элементов, ограниченной круглыми скобками:

subject = “( )” / “(“ element (“,”element) ”)”;

Структуру термина element примем подобной в Прологе и в нотации метаязыка НФЗ [7, 8] опишем так:

element = term / list;
list = ′[′ list_content ′]′;
list_content = element (′,′ element) / head comma tail / variable / true;
head = term (′,′ term);
comma = ′,′ / true;
tail = list;
term = natural / variable / atom / structure;
structure = atom ′(′ term (′,′ term) ′)′;
variable = letter1 (letter / letter1 / number);
letter1 = A / B / C / ... / Z;
letter = a / b / c / ... / z;
natural = numeral (numeral);
numeral = 0/1/2/3/4/5/6/7/8/9;
atom = letter (letter / letter1 / numeral);

Список является рекурсивной структурой данных, потому нужны и рекурсивные алгоритмы для его обработки. Главный способ обработки списка — это поэлементный просмотр и обработка списка до его исчерпания. Эти алгоритмы обычно задаются двумя ут-

верждениями: одно определяет, что делать с пустым списком, второе — что делать с обычным списком.

При описании всякой задачи необходимо, прежде всего, определиться с аргументами предиката, принять некоторую структуру области данных, описать процесс анализа конкретного значения аргументов и, далее, — процесс вывода заключения. Среди базовых предикатов на списках: формирование, объединение списков; поиск элемента в списке; вставка элемента в список и удаление из списка и др.

2. Печать списков. Печать элементов списка в Прологе задается двумя утверждениями:

```
write_a_list([ ]).
write_a_list([H|T]):- write(H), nl, write_a_list(T).
```

Первое из них утверждает факт истинности для пустого списка, второе — инициирует печать головы непустого списка и рекурсивный вызов печати хвоста списка. В метаязыке НФЗ этот предикат определяется так:

```
write_a_list='(' '[' L# ']' ')' viv_write_a_list;
viv_write_a_list = ^genL ^X / ^stepL? X write nl! viv_write_a_list;
                                                    /* nl - переход на новую строку */
^genL  = RIO L ']'! RIO;
^stepL =  X comma L# ;
X = term;
L = term (',' term) / true;
```

Здесь, в первом утверждении описан анализ структуры аргумента предиката write_a_list, в процессе которого с переменной L связывается исходное значение списка, и, далее, вызывается предикат viv_write_a_list, первая альтернатива которого утверждает, что пустой список печатать не надо, а вторая альтернатива утверждает поэлементную печать головы списка и рекурсивный вызов предиката viv_write_a_list на сокращенном списке.

3. Добавление элемента в список. У предиката add(X,L,L1]) три аргумента: добавляемый элемент X, начальный список L и результирующий — L1. Проще всего добавить элемент в список — вставить его в самое начало, как голову нового списка L1. В Прологе это записывают в виде факта:

```
add(X,L,[X|L]).
```

Все варианты вывода предиката add(X,L,L1]) (добавление: известного элемента X к пустому или непустому списку L с получением неизвестного списка L1, неизвестного элемента X к известному списку L с получением известного списка L1, неизвестного элемента X к неизвестному списку L с получением известного списка L1, известного элемента X к известному списку L с получением известного списка L1) в метаязыке определим в форме пяти альтернативных определений термина add:

```
add = '(' X# ',' '[' ']' ','? '[' A# ']' ')' writeXS / '(' X# ',' '[' T# ']' ','? '[' A# ']' ')' writeAdd
      / '(' A# ',' '[' L1# ']' ','? '[' L2# ']' ')' ^genL2 controlAdd writeX
      / '(' A# ',' '[' B# ']' ','? '[' L2# ']' ')' ^genL2 bindingX_L1 writeVV
      / '(' X# ',' '[' L1# ']' ',' '['? L2# ']' ')' ^genL2 analysX_L1;
writeXS = A '=' '[' X ']' nl!;
writeAdd = A '=' '[' X ',' T ']' ';' nl!;
^genL2  = RIO L2 ']'! RIO;
```

```
controlAdd = X# analysL1 ^',';
analysL1 = RB L1! RB / ^RB;
bindingX_L1 = X# comma L1# ^',';
writeVV = A '=' X ',' B '=' L1 ';' nl!;
analysX_L1 = RB X comma L1! RB / ^RB;
A = variable;
B = variable;
T = term (',' term);
L1 = term (',' term) / true;
L2 = term (',' term) / true;
```

Первая альтернатива завершается порождением одноэлементного списка, вторая альтернатива — порождением нового списка, составленного из известного элемента с пристыковкой к нему известного списка. Третья альтернатива после успешного вычитания первого списка из второго завершается порождением оставшейся головы второго списка. Четвертая альтернатива после успешного разделения второго списка на его голову и хвост завершается порождением найденных элемента и первого списка. Пятая альтернатива состоит в проверке на эквивалентность второго списка с конкатенацией известного элемента и первого списка.

4. Удаление элемента. Предикат away(X,L,L1) удаления элемента оперирует тремя аргументами: L1 — это список L, из которого изъят элемент X. В Прологе это записывают двумя утверждениями, первое из которых — простой факт, завершающий вычисление, а второй — рекурсивное правило:

```
away(X, [X|T],T).
away(X, [Y|T], [Y|T1]):- away(X,T,T1).
```

Все четыре варианта (удаление известного элемента X из известного списка L с получением неизвестного списка L1; выяснение, действительно ли известный список L1 получен удалением известного элемента X из известного списка L; выяснение неизвестного списка L, из которого удален известный элемент X с получением известного списка L1; выяснение неизвестного элемента X, удаленного из известного списка L с получением известного списка L1) вывода предиката away(X,L,L1) в метаязыке определим в форме соответствующих четырех альтернатив.

```
away = away1 / away2 / away3 / away4;
away1='(' X# ',' '['? T# ']' ','? A L# ')' viv_away1 writeL;
viv_away1 = ^genT analysX ','? T# ^form_result / ^current_result viv_away1;
away2='(' X# ',' '['? T# ']' ',' '['? L1 L# ']' ')' viv_away1 ^genL analysL1;
away3='(' X# ','? A# ',' '['? L1# ']' ')' ^genXL1? L# writeL;
away4='(' A# ',' '['? T# ']' ',' '['? L# ']' ')' viv_away2;
viv_away2 = ^genT ^headTail1 ^genL searchMatches viv_away2 / writeX;
searchMatches = analysX / Y comma searchMatches;
 ^headTail1 = X ',' T#;
^genXL1 = RIO X ',' L1! RIO;
^form_result = ^genL variantsL? L#;
variantsL = ^T? ^genT? / ^concatL_T;
```

```
^concatL_T = RIO L ',' T RIO!;
^current_result = Y ',' T# ^genL currentL? L#;
currentL = ^T? ^genY / ^concatL_Y;
^concatL_Y = RIO L ',' Y RIO!;
^genT = RIO T ']'! RIO;
^genY  = RIO Y ']'! RIO;
writeL = A '=' '[' L ']' nl!;
writeX = A '=' X ';' nl!;
analysX = RB X! RB / ^RB;
Y = term;
```

5. Принадлежность элемента списку. У предиката member(X,L) принадлежности элемента X списку L два аргумента: L — некоторый список и X — объект того же типа, что и элементы списка L. Его определение основывается на следующем: X — или голова списка L, или X принадлежит его хвосту. В Прологе это записывают в виде двух утверждений, первое из них — факт, завершающий вычисление, второй — рекурсивное правило. Если же список пуст, то предикат ложен: у пустого списка нет элементов:

```
member(X, [X| _ ]).
member(X, [ _ | T ]):- member(X,T).
```

В метаязыке предикат member(X,L) можно определить рекурсией: если X совпадает с головой списка, то, независимо от значения его хвоста, получим результирующее значение истинности “истина”; иначе, делается очередной шаг исследования следующего элемента списка L после удаления головы текущего списка.

```
member = '(' X# ',' '[' T# ']' ')' viv_member / '(' A# ',' '['? X# writeX (',' X# writeX) ']' ')';
viv_member = ^genT  analysX / ^step2 viv_member;
^step2 = Y comma? T#.
```

Этот предикат можно использовать двояко: во-первых, конечно, для проверки, есть ли в списке конкретное значение, и, во-вторых, — получить элементы заданного списка.

6. Конкатенация списков. У предиката conc(L1,L2,L3) конкатенации списков три аргумента; первые два L1 и L2 — объединяемые списки, а третий — список L3 является результатом конкатенации первых двух. За основу определения этого термина берут рекурсию по первому списку, а за базис — факт, утверждающий, что присоединение произвольного списка к пустому списку дает тот самый произвольный список.

```
conc([ ], L, L).
conc([H|T], L, [H|T1]) :- conc(T,L,T1).
```

Этот предикат можно применять для решения разных задач: для получения списка, являющегося конкатенацией двух заданных; для проверки, является ли третий список конкатенацией двух первых; для разбивки третьего списка на подсписки; для поиска одного из первых списков по заданным двум другим и др. Для этих вариантов использования предикат конкатенации conc(L1,L2,L3) определим в метаязыке так:

```
conc = conc_1 / conc_2 / conc_3 / conc_4 / conc_5;
conc_1 = '(' '[' ']' ',' '[' ']' ',' analysA1 ')' / '(' '[' L1# ']' ',' '[' ']' ',' analysA2 ')'
       / '(' '[' ']' ',' '['? L2# ']' ',' analysA3 ')' / '(' '['? L1# ']' ',' '['? L2# ']' ',' analysA4 ')';
analysA1 = A# writeEmpty / '[' ']';
```

```
writeEmpty = A '=' '[' ']' nl!;
analysA2 = A# writeL1 / analysL1 ^',';
writeL1 = A '=' '[' L1 ']' nl!;
analysA3 = A# writeL2 / analysL2 ^',';
writeL2 = A '=' '[' L2 ']' nl!;
analysL2 = RB L2! RB / ^RB;
analysA4 = A# writeL1_L2 / analysL1_L2 ^',';
writeL1_L2 = A '=' '[' L1 ',' L2 ']' nl!;
analysL1_L2 = RB L1 ',' L2! RB / ^RB;
conc_2 = '('? A# ',' '['? L2# ']' ',' '['? L3# ']' ')'  ^genL3  analysCon6;
^genL3  = RIO L3 ']'! RIO;
analysCon6 = analysL2 ^',' writeEmpty / ^headTail3 ^hypothes1 ^takeL1 ^genL3 analysC;
^headTail3  = X ',' L3#;
^hypothes1 = RIO X ']'! RIO;
analysC = analysL1_L2 ^',' writeL1 / ^headTail3 ^hypothes2 ^takeL1 ^genL3 analysC;
^hypothes2 = RIO L1 ',' X ']'! RIO;
conc_3 = '(' '['? L1# ']' ','? A# ',' '['? L3# ']' ')' ^genL3  analysL1 writeConc5;
writeConc5 = ^',' writeEmpty / ',' L2# writeL2;
conc_4 = '('? A L1# ','? B# ',' '[' L3# ']' ')' ^genL3 ^takeL2 writePair ^headTail2 below;
below = ^hypothes1 ^takeL1 writePair (^genL2 ^headTail2 ^hypothes2 ^takeL1 writePair);
^takeL2= L2#;
^headTail2  = X comma L2#;
^takeL1 = L1#;
writePair = A '=' '[' L1 ']' ',' B '=' '[' L2 ']' ';' nl!;
conc_5 = '('? A L1# ',' '['? Y ',' B# ']' ',' '[' zeroPair nextPair;
zeroPair = T3# ']' ')' ^genT3  ^listL2? writePair / ^takeXL2 ^genX;
nextPair = ^takeL1 ^genL2 ^listL2 writePair / ^takeXL2 ^hypothes2 nextPair;
^listL2= analysY comma? L2#;
analysY = RB Y! RB / ^RB;
^takeXL2 = X comma L2#;
^genT3  = RIO T3 ']'! RIO;
^genX  = RIO X ']'! RIO;
L3 = term (',' term) / true;
L4 = term (',' term) / true;
T1 = term (',' term);
T2 = term (',' term);
T3 = term (',' term);
T4 = term (',' term);
```

Первая альтернатива conc_1 определяет варианты формирования результата, если первых два списка составлены из констант или один из них или оба являются пустыми, а третий список задан константами или именован переменной. Вторая альтернатива conc_2 определяет вариант поиска первого списка, если второй и третий списки составлены из констант. Третья альтернатива conc_3 определяет вариант поиска второго списка, если пер-

вый и третий списки составлены из констант. Четвертая альтернатива conc_4 определяет вариант поиска первого и второго списков, если третий список составлен из констант. Пятая альтернатива conc_5 определяет вариант поиска первого A и второго B списков, находящихся соответственно левее и правее заданного элемента Y третьего списка ′[′ T3 ′]′, составленного из констант.

7. Поиск последнего элемента списка. В Прологе предикат last (L,X) определяется так:

last ([X],X). /* последний элемент одноэлементного списка является этим элементом */

last ([_|L],X):- last (L,X). /*последний элемент списка — это последний элемент его хвоста*/

В метаязыке этот предикат можно определить, используя итерацию:

last = ′(′ ′[′ (Y ′,′)? X# ′]′ ′,′? A# ′)′ writeX.

В этом определении итерация (Y ′,′) выбирает первые n–1 элементов списка L , последний элемент которого связывает переменную X#, чье значение далее составляет результат решения.

8. Два соседних элемента списка. У этого предиката neighbors(X,Y,L) три параметра: первые два X, Y — элементы, третий L — список элементов такого же типа. Для случая, когда порядок размещения элементов X,Y в списке L важен, в Прологе этот предикат определяют с использованием предиката конкатенации:

neighbors(X,Y,L):- conc(_,[X,Y|_],L).

Все пять вариантов (выяснение, действительно ли известные два элемента X, Y соседствуют в известном списке L; выяснение, какой неизвестный элемент X является соседним слева от известного элемента Y в известном списке L; выяснение, какой элемент Y является соседним справа от известного элемента X в известном списке L; выяснение, какие пары элементов X, Y соседствуют в известном списке L; сконструировать новый список из двух известных элементов) вывода предиката neighbors(X,Y,L) в метаязыке определим в форме соответствующих альтернатив.

В метаязыке предикат neighbors(X,Y,L) можно определить так:

```
neighbors = neighbors1 /  neighbors2 /  neighbors3 / neighbors4 / neighbors5;
neighbors1 = ′(′ X# ′,′ Y# ′,′ ′[′ L# ′]′ ′)′ viv_neighbors1;
viv_neighbors1 = ^genL analysXY / ^step1 viv_neighbors1;
neighbors2=′(′ X# ′,′? Y# ′,′ ′[′? A# ′]′ ′)′ writeXYS;
neighbors3 = ′(′ X# ′,′? A# ′,′ ′[′? L# ′]′  ′)′ viv_neighbors3;
viv_neighbors3 = ^genL  analysX? Y# writeY / ^step1 viv_neighbors3;
neighbors4=′(′ A# ′,′? Y# ′,′ ′[′? L# ′]′ ′)′ viv_neighbors4;
viv_neighbors4 = ^genL?  X# analysY writeX / ^step1 viv_neighbors4;
neighbors5=′(′ A# ′,′? B# ′,′ ′[′? L# ′]′ ′)′ (^genL? X comma Y# writeXY ^genL ^step1);
analysXY = RB X ′,′ Y! RB / ^RB;
writeY = A ′=′ Y ′;′ nl!;
writeXY = A ′=′ X ′,′ B ′=′ Y ′;′ nl!;
```

9. Реверс списка. У этого предиката reverse(L1, L2) два параметра: первый — исходный список L1, второй — обращенный список L2. За базис определения этого термина принимают факт, утверждающий, что реверс пустого списка дает пустой список. Шаг рекурсии состоит в конкатенации обращенного хвоста списка с первым элементом исходного списка:

```
reverse([ ],[ ]).
reverse([X|T],Z):— reverse(T,S), conc(S,[X],Z).
```

В метаязыке предикат reverse(L1, L2) можно определить так:

```
reverse = '(' '[' ']' ',' A# ')' writeEmpty / '(' '[' L2# ']' ',' A# ')' viv_reverse writeL1
                                / '(' '[' L2# ']' ',' '['? L3# ']' ')' viv_reverse ^genL1  analysL3;
viv_reverse = ^genL2  ^headTail2  ^genX  ^takeL1 further
further = (^genL2  ^headTail2 ^hypothes2 ^takeL1);
analysL3 = RB L3! RB / ^RB.
```

10. Палиндром. Палиндромом называют список, совпадающий со своим обращением. В Прологе предикат palindrom(L) определяют через предикат reverse (L,L) с двумя одинаковыми аргументами:

```
palindrom(L):- reverse (L,L).
```

Подобное метаязыковое определение состоит из двух последовательных преобразований: обращение заданного списка и проверка, совпадает ли результат с начальным списком:

```
palindrom = '(' '[' L2# ']' ')' ^genL2  ^headTail2  viv_reverse ^genL1  analysL2;
viv_reverse = ^genX  ^takeL1 (^genL2  ^headTail2 ^hypothes2 ^takeL1).
```

11. Удаление из списка всех вхождений заданного значения. Этот предикат delete_all(X,[L],[L1]) имеет три параметра: первый X — удаляемый элемент, второй L — начальный список, а третий L1 — результат удаления из начального списка L всех вхождений заданного элемента X.

Определение на Прологе включает три утверждения: первое является базовым фактом — пустой список не уменьшается; второе утверждает, что после удаления заданного элемента из головы начального списка L результирующий список является хвостом списка L без вхождений заданного элемента X; третье утверждает, что результирующий список является конкатенацией последовательности элементов без удаляемого и результата удаления заданного элемента из хвоста начального списка.

```
delete_all(_,[],[]).
delete_all(X,[X|L],L1):- delete_all (X,L,L1).
delete_all (X,[Y|L],[Y|L1]):- X<>Y, delete_all (X,L,L1).
```

Метаязыковое определение включает две рекурсивные альтернативы: удаление головы текущего списка в случае ее тождественности заданному элементу; иначе присоединить голову текущего списка к текущему составу результирующего списка.

```
delete_all = '(' Y# ',' '[' L2# ']' ','  '[' A L3# ']' ')' viv_delete_all;
viv_delete_all=^genL2 ^listL2 viv_delete_all / ^takeXL2 ^upbuildingL3 ^takeL3 viv_
delete_all / writeL3;
^genL3X  = RIO L3 ',' X ']'! RIO;
^upbuildingL3 = ^genL3 ^nullX ^genL3X  / ^genX;
^nullX = X;
^takeL3= L3#;
writeL3 = A '=' '[' L3 ']' nl!;
```

Если нужно изъять не все вхождения определенного значения в список, а только первое, то в Прологе используют следующее определение:

```
delete_one(_,[],[]).
delete_one(X,[X|L],L):-!.
delete_one(X,[Y|L],[Y|L1]):- delete_one(X,L,L1).
```

Метаязыковое определение этой семантики получает вид:

```
delete_one = '(' Y# ',' '[' L2# ']' ',' '[' A L3# ']' ')' viv_delete_one;
viv_delete_one = ^genL2 ^listL2 writeL3_L2 / ^takeXL2 ^upbuildingL3 ^takeL3 viv_
delete_one / writeL3;
writeL3_L2 = A '=' '[' L3 ',' L2 ']' nl!;
```

Заключение. Базируясь на известных описаниях на Прологе, использующих списковый домен, впервые разработана в метаязыке НФЗ практически пригодная формализация множества предикатов на списках.

Представляется, что метаязыковая формализация практически пригодна не только в качестве новой реализации задач оперирования списками, но и в качестве спецификации для реализации этих задач в составе других систем.

ЦИТИРОВАННАЯ ЛИТЕРАТУРА

1. Abelson H., Sussman G.J., Sussman J. Structure and interpretation of computer programs. Cambridge: MIT Press, 1996. 576 p.
2. Haskell 98. Language and Libraries. The Revised Report. Ed. S.P. Jones. Cambridge Academ. 2003. 277 p.
3. Seibel P. Practical Common Lisp. Apress, 2005. 501 p. doi: https://doi.org/10.1007/978-1-4302-0017-8
4. Братко И. Алгоритмы искусственного интеллекта на языке PROLOG. 3-е изд. Пер. с англ. Москва: Изд. дом “Вильяме”, 2004. 640 с.
5. Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. СПб.: БХВ-Петербург, 2003. 992 с.
6. Клоксин У., Меллиш К. Программирование на языке Пролог. Москва: Мир, 1987. 334 с.
7. Кургаев А.Ф., Григорьев С.Н. Нормальные формы знаний. *Допов. Нац. акад. наук України.* 2015. № 11. С. 36—43.
8. Кургаев А.Ф., Григорьев С.Н. Метаязык нормальных форм знаний. *Кибернетика и системный анализ.* 2016. **52,** № 6. С. 11—20.

Поступило в редакцию 12.06.2017

REFERENCES

1. Abelson, H., Sussman, G. J. & Sussman, J. (1996). Structure and interpretation of computer programs. Cambridge: MIT Press.
2. Haskell 98. (2003). Language and Libraries. The Revised Report. Ed. S.P. Jones. Cambridge Academ.
3. Seibel, Peter. (2005). Practical Common Lisp. Apress. doi: https://doi.org/10.1007/978-1-4302-0017-8
4. Bratko, Ivan. (2012). Prolog Programming for Artificial Intelligence. Third Edition. Addison-Wesley.
5. Adamenko, A. N. & Kuchukov, A. M. (2003). Logical Programming and Visual Prolog. St. Petersburg: BHV-Petersburg (in Russian).
6. Clocksin, William., Mellish & Christopher, S. (2003). Programming in Prolog: Using the ISO Standard 5th Edition. Berlin etc.: Springer. doi: https://doi.org/10.1007/978-3-642-55481-0
7. Kurgaev, A. & Grygoryev, S. (2015). The normal forms of knowledge. Dopov. Nac. acad. nauk Ukr. No. 11, pp. 36-43 (in Russian).
8. Kurgaev, A. & Grygoryev, S. (2016). Metalanguage of Normal Forms of Knowledge. Cybernetics and Systems Analysis. 52(6), pp. 839-848. doi: https://doi.org/10.1007/s10559-016-9885-3

Received 12.06.2017

О.П. Кургаєв

Інститут кібернетики ім. В.М. Глушкова НАН України, Київ
E-mail: afkurgaev@ukr.net

ФОРМАЛІЗАЦІЯ СПИСКІВ
У МЕТАМОВІ НОРМАЛЬНИХ ФОРМ ЗНАНЬ

Вперше запропоновано формалізацію предикатів на списках у метамові нормальних форм знань, базуючись на відомих описах цих понять на Прологу, які використовують списковий домен. Серед описаних предикатів: додавання елемента в список, видалення елемента, видалення повторів, приналежність елемента списку, пошук останнього елемента списку, пошук сусідніх елементів списку, конкатенація списків, реверс й ін.

Ключові слова: *метамова нормальних форм знань, список, предикат, визначення, рекурсія.*

A. F. Kurgaev

V.M. Glushkov Institute of Cybernetics of the NAS of Ukraine, Kiev
E-mail: afkurgaev@ukr.net

THE FORMALIZATION OF LISTS IN THE META-LANGUAGE
OF NORMAL FORMS OF KNOWLEDGE

The formalization of list-based predicates in the meta-language of normal forms of knowledge is presented for the first time, based on the known descriptions of these concepts in Prolog, which use a list-domain. Among the predicates described are the following: adding an element to the list, removing an element, removing duplicates, checking if an element is in a list, finding the last element of a list, finding adjacent elements in a list, concatenation of lists, reversing a list, etc.

Keywords: *meta-language of normal forms of knowledge, list, predicate, definition, recursion.*