УДК 004.8 https://doi.org/10.15407/pp2020.01.003

А.Ф. Кургаев

ОПИСАНИЕ СПИСКОВ И МНОЖЕСТВ В МЕТАЯЗЫКЕ НОРМАЛЬНЫХ ФОРМ ЗНАНИЙ

Предложена формализация списков, предикатов на списках и множествах в метаязыке нормальных форм знаний, базируясь на известных Пролог-формализациях этих понятий, использующих списковый домен. Среди предикатов на списках описаны: добавление элемента, удаление элемента, поиск последнего элемента, поиск соседних элементов, конкатенация списков, реверс и др. Используя списковый домен описаны предикаты на множествах: превращения списка в множество, принадлежности элемента множеству, объединения, пересечения, разности, симметрической разности, совпадения, дополнения множеств.

Ключевые слова: метаязык, список, множество, предикат, рекурсия, определение.

Введение

Список – один из самых простых и полезных типов структур составных объектов логического программирования, позволяющий во многих случаях улучшить «читабельность» программ. Запись в виде списка свободна по форме и одновременно достаточно точна и понятна [1–3].

Списки можно использовать для представления всевозможных знаний. Также они позволяют представить практически любые неоднородные и/или иерархические структуры данных, возможные в символьных вычислениях, поддерживающие функциональный стиль программирования. В виде списков удобно представлять формулы, функции, деревья, графы, множества и многие другие сложные объекты [4–6].

Множество – одна из наиболее важных структур данных, используемых как в математике, так и в программировании. Это набор элементов, подобный списку, отличающийся от него лишь фактом принадлежности элемента множеству [3].

В метаязыке нормальных форм знаний (НФЗ) [7, 8], в отличие от ЛИСПа и Пролога, нет такой встроенной структуры данных, как *список*, эффективность которой обоснована не только теоретически, но и обширной практикой использования этих языков при создании систем искусственного интеллекта. Тем более в метаязыке НФЗ нет такой встроенной структуры данных, как *множество*. Поэтому, для практического использования метаязыка НФЗ важно реализовать понятия списка, множества и предикаты на списках и множествах, опираясь на стандартные домены метаязыка НФЗ.

В качестве базовой структуры используем область 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 и множествах использована нотация метаязыка НФЗ [7, 8]:

1. description = determination (determination) ;

© А.Ф. Кургаев, 2020

2. determination = negativ nameConcept definition bodyDeterm endDeterm;

3. negativ = inversion / true;

4. nameConcept = identifier / integer / chainSigns;

5. identifier = letter (letter/decimalDigit);

6. integer = decimalDigit (decimalDigit);

7. chainSigns = ^metaSign sign (^metaSign sign);

8. bodyDeterm = structure / terminal;

9. terminal = space (space);

10. structure = singleDefinit (separator singleDefinit);

11. singleDefinit = negativ primary mode (concatenate negativ primary mode);

12. primary = iterationSeq / nameConcept / line;

13. iterationSeq = startIterationSymb bodyDeterm endIterationSymb;

14. mode = analysis / traceAnalysis / generation / true;

15. line = quotationMark nameConcept quotationMark;

где definition – разделитель двух частей определения, изображается символом '=';

separator – отношение альтернативного выбора изображается символом '/';

concatenate – отношение конкатенации изображается символом space ' ';

startIterationSymb, endIterationSymb – пара скобок '(' и ')', обрамляющих итерируемый элемент;

inversion – отношение отрицания изображается символом '^';

endDeterm – конец определения изображается символом ';';

quotationMark – текстовая кавычка, изображается символом '"';

analysis – режим анализа изображается символом '?';

traceAnalysis – режим анализа со следом изображается символом '#';

generation – режим порождения изображается символом '!';

letter = 'A' / 'B' / 'C' /…/ 'Z' / 'a' / 'b' / 'c' /... / 'z';

decimalDigit = '0'/'1'/'2'/'3'/'4'/'5'/'6'/'7'/'8'/'9';

sign = '-' / '&' / '%' / '$' / '@' / '~' / ':' / '<' / '>' / … / ',' / '.' / '_';

metaSign = '(' / ')' / space / '/' / '=' / '?' / '#' / '!' / ';' / '"'/ '"' ;

1. Представление задачи

В терминах метаязыка НФЗ любая задача формулируется как доказательство категорического суждения *P*(*x*), предикат *P* которого задан именованной структурой, 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 = number / variable / atom / structure;
structure = atom '(' term (',' term) ')';
variable = letter1 (letter / letter1 / number);
letter1 = A / B / C / ... / Z;
letter = a / b / c / ... / z;
number = numeral (numeral);
numeral = 0/1/2/3/4/5/6/7/8/9;
atom = letter (letter / letter1 / numeral);
E1= element;
E2= element;
E3= element;

Список – это рекурсивная структура данных, поэтому 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 утверждениями: одно определяет, что делать с пустым списком, второе – что делать с обычным списком.

При описании любой задачи необходимо, прежде всего, определиться с аргументами предиката, принять некоторую структуру области данных, описать про-

цесс анализа конкретного значения аргументов, и далее, – процесс вывода. Среди базовых предикатов на списках: формирование, объединение списков; поиск элемента в списке; вставка элемента в список и удаление из списка и др. [9].

2. Предикаты на списках

2.1. 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).
```

/* nl - переход на новую строку */

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? ^write_H nl! viv_write_a_list;
^genL  = RIO L ']'! RIO;
^stepL =  X comma L# ;
^write_H = X nl!;
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.

2.2. 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# ^',';
^writeX = A '=' X ';' nl!;
^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 присоединением 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.

2.3. Удаление элемента. Предикат away(X,L,L1) удаления элемента оперирует тремя аргументами: L1 – это список L, из которого изъят элемент X. В Прологе записывается 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!;
analysX = RB X! RB / ^RB;
Y = term;
```

2.4. Принадлежность элемента списка. 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 так: во-первых, конечно, для проверки, есть ли в списке конкретное значение, во-вторых, – получить элементы заданного списка.

2.5. Конкатенация списков. У предиката conc(L1,L2,L3) конкатенации списков три аргумента; первые два L1 и L2 – объединяемые списки, а третий – список L3 результат конкатенации первых двух. За основу определения этого термина берут рекурсию по первому списку, а за базис – факт, утверждающий, что присоеди-

нение 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   ^genX    ^takeL1   ^genL3
analysC;
^headTail3  = X ',' L3#;
analysC = analysL1_L2 ^',' ^writeL1 /
^headTail3   ^genL1X   ^takeL1   ^genL3
analysC;
^genL1X = 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 = ^genX  ^takeL1 ^writePair (^genL2
^headTail2 ^genL1X ^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 ^genL1X 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 первые 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 первый 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.

2.6. Последний элемент списка. В Прологе предикат last(L,X) определяется как последний элемент одноэлементного списка являясь этим элементом, а послед-

ний элемент обычного списка – это последний элемент его хвоста:

last ([X],X).
last ([_|L],X):- last (L,X).

В метаязыке этот предикат можно определить, используя итерацию:

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

Здесь итерация (Y ',') выбирает первые *n*-1 элементов списка L , последний элемент которого связывает переменную X#, чье значение далее составляет результат решения.

2.7. 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 / ^stepL
viv_neighbors1;
neighbors2='(' X# ','? Y# ','? A# ')' ^writeXYS;
neighbors3 = '(' X# ','? A# ',' '['? L# ']' ')'
viv_neighbors3;
viv_neighbors3 = ^genL analysX? Y# ^writeY / ^stepL viv_neighbors3; ;
neighbors4='(' A# ','? Y# ',' '['? L# ']' ')'
viv_neighbors4;
viv_neighbors4 = ^genL? X# analysY ^writeX / ^stepL viv_neighbors4;
neighbors5='(' A# ','? B# ',' '['? L# ']' ')' (^genL?
X comma Y# ^writeXY ^genL ^stepL);
analysXY = RB X ',' Y! RB / ^RB;
^writeXYS = A '=' '[' X ',' Y ']' nl!;
^writeY = A '=' Y ';' nl!;
^writeXY = A '=' X ',' B '=' Y ';' nl!.

2.8. 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# ']' ')' v_reverse
^genL1 analysL3;
viv_reverse = ^genL2 ^headTail2 ^genX
^takeL1 (^genL2 ^headTail2 ^genL1X
^takeL1);
analysL3 = RB L3! RB / ^RB.

2.9. 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# ']' ')' viv_reverse ^genL1 analysL2.

2.10. 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 для неизвестного результирующего списка (заданного именем A переменной) включает две рекурсивные альтернативы: удаление головы текущего списка L2 в случае ее тождественности заданному элементу; иначе присоединить голову текущего списка к текущему составу результирующего списка L3.

```
delete_all = '(' Y# ',' '[' L2# ']' ',' '[' A L3# ']' ')'
            viv_del_all;
viv_del_all = ^genL2 ^listL2 viv_del_all /
^takeXL2 ^upbuildingL3 ^takeL3 viv_del_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_del_one;
viv_del_one = ^genL2 ^listL2 ^writeL3_L2 /
          ^takeXL2           ^upbuildingL3
^takeL3
          viv_del_one/^writeL3;
^writeL3_L2 = A '=' '[' L3 ',' L2 ']' nl!.
```

3. Понятие множества

В метаязыке НФЗ, также как в Прологе и ЛИСПе нет такой встроенной структуры данных, как *множество*. Поэтому, придется реализовывать это понятие, используя имеющиеся стандартные домены. В качестве базового примем списковый домен, приняв:

Определение 1. Множество – список без элементов повтора.

Фактически, нет никакой реализации понятия множество, которое бы точно отвечало этому математическому объекту. Принятое определение – лишь приближение к "истинному" объекту множество.

Рассмотрим предикаты, реализующие основные теоретико-множественные отношения.

3.1. Предикат превращения списка в множество. Эту функцию выполняет предикат, удаляющий из списка все повторные вхождения элементов. Аргументами предиката unik(L1,L2) являются два списка: произвольный начальный – L1 и результирующий – L2 без повторяемых элементов. Его семантика простая: если элемент X головы списка L1 входит в хвост этого списка L1 (проверяется предикатом member), то он не записывается в результирующий список L2, иначе этот элемент X списка L1 вставляется в результирующий список L2 в качестве его головы.

Пролог описание этого предиката имеет вид.

```
unik([ ],[ ]).
unik([H|T], L):– member(H,T), unik(T,L).
unik([H|T], [H|L]):– unik(T,L).
```

Подобное метаязыковое определение включает две альтернативы: первая фиксирует тот факт, что пустой список остается пустым безусловно; вторая альтернатива определяет процесс вывода результирующего списка для общего случая в форме итерации двух вложенных альтернатив. Первая из них определяет наличие повторений в текущем исследовании списка, а вторая – процесс пошагового накопления результирующего списка с использованием термина ^upbuildingL1.

```
unik = '(' '[' ']' ',' '[' ']' ')' / '(' '[' L# ']' ','? A L1#
   ')' viv_unik ^writeL1;
viv_unik=(^genL  ^headTail2        member1
^genL ^stepL/ ^genL ^stepL ^upbuildingL1
^takeL1);
member1 = ^genL2  analysX / ^stepL2
              member1;
^upbuildingL1 = ^genL1 ^nullX ^genXL1 /
                    ^genX;
^genL1  = RIO L1 ']'! RIO;
^stepL2 = Y ','? L2# ']';
```

Далее рассмотрим метаязыковую реализацию основных теоретико-множественных отношений.

3.2. Принадлежность элемента множеству. В качестве реализации предиката принадлежности элемента *x* множеству A ($x \in A$) можно использовать предикат member(X,L) принадлежности элемента X списку L, где: X – объект того же типа, что и элементы списка L.

3.3. Объединение двух множеств. Объединением двух множеств является множество, чьи элементы принадлежат или первому, или второму множеству. Формальное определение объединения множеств A и B:

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

на рис. 1 обозначено штриховкой.

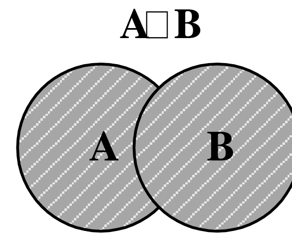

Рис. 1. Объединение множеств A и B

У предиката union(L1,L2,L3) три параметра: первые два L1, L2 – объединяемые множества, третий L3 – результат объединения двух первых множеств. При этом важно, чтобы ни одно значение не входило в итоговое множество повторно.

Пролог описание этого предиката имеет вид.

```
union([ ],S2,S2).
union([H|T],S2,S):– member(H,S2), !,
                      union(T,S2,S).
union([H|T],S2,[H|S]):– union(T,S2,S).
```

В метаязыке предикат union(L1,L2,L3) определим для случаев, когда являются пустыми оба первых множества или один из них, или же оба первых множества составлены из констант, а третье множество является, соответственно, пустым, задано константами или именовано переменной.

```
union = '(' '[' ']' ',' '[' ']' ',' analysA1 ')' /
      '(' '[' L1# ']' ',' '[' ']' ','? A# ')' ^writeL1 /
      '(' '[' ']' ',' '[? L1# ']' ','? A# ')' ^writeL1 /
      '(' '[? L# ']' ',' '[' ']' ',' '[? L1# ']' ')'
       equivalentLL1 / '(' '[' ']' ',' '[? L# ']' ',' '[?
       L1# ']' ')' equivalentLL1 / '(' '[? L# ']' ','
  '[? L1# ']' ',' A# ')' viv_union_2 ^writeL_L1 /
      '(' '[? L# ']' ',' '[? L1# ']' ',' '[? L4# ']' ')'
      viv_union_3 / '(' '[? L1# ']' ','? A# ',' '[?
      L2# ']' ')' viv_minus_L2L1 ^writeL /
       '(? A# ',' '[? L1# ']' ',' '[? L2# ']' ')'
  viv_minus_L2L1 ^writeL / '(? A L1# ','? B#
',' '[' L3# ']' ')' ^genL3 ^takeL2 ^unionAnswer;
^unionAnswer = ^writePair ^headTail2
         ^hypothes1 ^takeL1 ^writePair below;
below = (^genL2 ^headTail2 ^hypothes2
            ^takeL1 ^writePair);
equivalentLL1 = ^genL (^headTail2 ^genL1
      ^takeL3 member2 ^genL2) nullX
      ^genL1 (^headTail2 ^genL ^takeL3
```

```
        member2 ^genL2) nullX;
member2 = ^genL3 analysX / ^stepL3
        member2;
viv_ union_2 = ^genL (^headTail2 ^genL1
             ^takeL31 member3 ^genL2) nullX;
member3 = (^genL3  analysX ^takeL3 /
             ^stepL3 ^genYL1 ^takeL1);
viv_ union_3 = viv_ union_2 ^genLL1? L#
             ^genL4?  L1# equivalentLL1;
^writeL_L1 = A '=' '[' L ',' L1 ']' nl!;
^takeL31 = L3 L1#;
^stepL3 = Y ',' L3#;
^genYL1 = RIO Y ',' L1! RIO;
^genLL1  = RIO L ',' L1 ']'! RIO;
^genL4  = RIO L4 ']'! RIO.
```

Интерпретация термина analysA1 вывода результата для пустых объединяемых множеств состоит в порождении пустого множества, как значения переменной A, или порождении значения истинности истина, если результирующее множество задано формой пустого множества.

Вторая и третья альтернативы завершаются печатью writeL1 значения известного множества в качестве значения результирующего множества.

Четвертая и пятая альтернативы завершаются значением истинности истина термина viv_union_1, если эквивалентны результирующее и не пустое заданное множество.

Шестая альтернатива завершается печатью writeL_L1 значения объединенного множества после удаления из множества L1 элементов множества L в процессе интерпретации термина viv_union_2.

Седьмая альтернатива в процессе вывода термина viv_union_3 состоит вначале в синтезе объединения множеств L и L1 (используя предикат viv_union_2), далее переназначение переменных и, в завершение (используя предикат equivalentLL1), вывод эквивалентности синтезированного множества заданному значению результирующего множества.

Восьмая и девятая альтернативы вычисляют и печатают неизвестное множество, используя термин viv_minus_L2L1 вычитания над известными (результирующим и одним из компонентных) множествами.

Десятая альтернатива состоит в порождении перераспределения заданного значения результирующего множества на два составляющих множества. Семантика этого процесса совпадает с семантикой термина conc_4 порождения перераспределения заданного значения результирующего списка на два подсписка.

3.4. Пересечение двух множеств. Пересечение двух множеств – это множество из элементов, принадлежащих и первому, и второму множествам. Формальное определение пересечения множеств A и B:

$$A \cap B = \{x | x \in A \ \& \ x \in B\}$$

на рис. 2 обозначено штриховкой.

У предиката, реализующего эту операцию, три параметра: первые два L1, L2 – исходные множества, третье множество L3 – результат пересечения двух первых.

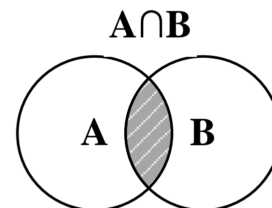

Рис. 2. Пересечение множеств A и B

В итоговое множество входят те элементы, которые есть и в первом, и во втором множестве одновременно. Пролог описание этого предиката имеет вид.

```
intersection([],_,[]).
intersection([H|T1],S2,[H|T]):–
        member(H,S2),!,
intersection(T1,S2,T).
intersection([_|T],S2,S):–
intersection(T,S2,S).
```

Базис рекурсии: пересечение пустого множества с любым множеством будет пустым множеством.

Шаг рекурсии включает два случая: если голова первого множества является элементом второго множества, она становится головой результирующего множества, чей хвост образуется как пересечение хвоста первого множества со вторым множеством; иначе, результирующее множе-

ство образуется пересечением хвоста первого множества со вторым множеством.

В метаязыке предикат intersection(L1,L2,L3) определим для случаев, когда пустыми являются оба первых множества или одно из них, или же оба первых множества составлены из констант, а третье множество является, соответственно, пустым, задано константами или именовано переменной.

```
intersection = intersectionEmpty ',' analysA1
        ')' / '(' '['? L1# ']' ',' '['? L2# ']' ','? A L# ')'
        viv_intersect_1 ^writeL / '(' '['? L1# ']' ','
     '['? L2# ']' ',' '['? L4# ']' ')' viv_intersect_2;
intersectionEmpty = '(' '[' ']' ',' '[' ']' /
            '(' '[' L1 ']' ',' '[' ']' / '(' '[' ']' ',' '[' L1 ']';
viv_intersect_1 = (^genL2 ^headTail2 ^genL1
          ^takeL3 member4) nullX;
member4 = ^genL3  analysX ^genXL? L# /
           ^stepL3 member4;
viv_intersect_2 = viv_intersect_1 ^genL4?
             L1# equivalentLL1;
^genXL = RIO X ',' L! RIO.
```

Интерпретация термина analysA1 вывода результата при наличии пустых множеств состоит в порождении пустого множества, как значения переменной A, или порождении значения истинности истина, если результирующее множество задано формой пустого множества.

Вторая альтернатива завершается печатью writeL множества, являющегося пересечением заданных множеств.

Третья альтернатива описывает последовательность действий: вначале, синтез пересечения двух заданных множеств, далее переназначение переменных и, в завершение, вывод эквивалентности синтезированного множества заданному значению результирующего множества.

3.5. Разность двух множеств. Разность двух множеств – это множество, составленное из элементов первого множества, не принадлежащих второму. На рис. 3 штриховкой обозначена формальная запись двух вариантов разности множеств A и B:

$$A\backslash B=\{x|x\in A\ \&\ x\notin B\},$$

$$B\backslash A=\{x|x\in B\ \&\ x\notin A\}.$$

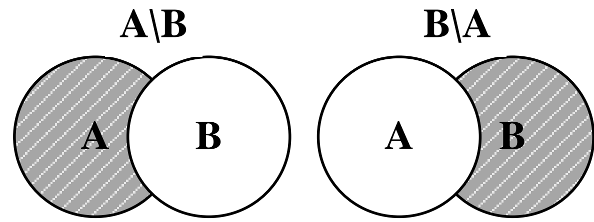

Рис. 3. Разность множеств A и B

У предиката minus(L1,L2,L3), реализующего разность, три аргумента: первый L1 – уменьшаемое множество, второй L2 – вычитаемое множество, третий L3 – результат вычитания второго множества из первого. Третье множество содержит элементы первого множества без элементов второго. Пролог-описание этого предиката имеет вид.

```
minus([],_,[]).
minus([H|T],S2,S):– member(H,S2),!,
            minus(T,S2,S).
minus([H|T],S2,[H|S]):– minus(T,S2,S).
```

Базис рекурсии: вычитание произвольного множества из пустого множества ничего, кроме пустого множества, дать не может. Шаг рекурсии: если голова первого множества входит во второе множество, то разность множеств образуется вычитанием второго множества из хвоста первого; иначе, разность множеств является конкатенацией головы первого множества и результата вычитания второго множества из хвоста первого множества.

В метаязыке предикат minus(L1,L2,L3) определим для случаев, когда: пустыми являются оба первых множества или только уменьшаемое; пустым является вычитаемое множество, а результирующее неизвестно; пустым является вычитаемое множество, а уменьшаемое и результирующее множества заданы; заданы уменьшаемое и вычитаемое множества, а результирующее неизвестно; все три множества заданы или же заданы вычитаемое и результирующее множества, а уменьшаемое неизвестно.

```
minus ='(' '[' ']' ',' '['? L1# ']' ',' analysA1 ')' /
        '(' '[' L1# ']' ',' '[' ']' ','? A# ')' ^writeL1 /
        '(' '[' L# ']' ',' '[' ']' ','? L1# ')'
```

```
        equivalentLL1 /
        '(' '['? L2# ']' ',' '['? L1# ']' ',' A L # ')'
        viv_minus_L2L1 ^writeL /
        '(' '['? L2# ']' ',' '['? L1# ']' ',' '['? L4# ']' ')'
        viv_minus /
        '(' '['? A# ']' ',' '['? L# ']' ',' '['? L1# ']' ')'
        ^writeL_L1 /
        '(' '['? L2# ']' ',' A L # ',' '['? L1# ']' ')'
        viv_minus_L2L1 ^writeL;
viv_minus_L2L1 = (^genL2 ^headTail2
        stepMinus);
stepMinus = ^genL1 ^takeL3 ^member5
        ^genXL? L# / true;
member5 = ^genL3 analysX/^stepL3
        member5;
viv_minus = viv_minus_L2L1 ^genL4? L1#
        equivalentLL1.
```

Интерпретация термина analysA1 вывода результата при наличии пустых множеств состоит в порождении пустого множества, как значения переменной A, или порождении значения истинности истина, если результирующее множество задано формой пустого множества.

Вторая альтернатива завершается печатью writeL уменьшаемого множества, как значения результирующего множества.

Третья альтернатива описывает условие эквивалентности уменьшаемого и результирующего множеств при пустом вычитаемом множестве.

Четвертая альтернатива описывает последовательность действий по формированию и печати разности заданных множеств: уменьшаемого и вычитаемого.

Пятая альтернатива описывает вычисление разности заданных множеств, далее переназначение переменных и, в завершение, вывод эквивалентности синтезированного множества заданному значению результирующего множества.

Шестая альтернатива завершается печатью writeL_L1 конкатенации вычитаемого и результирующего множеств.

Седьмая альтернатива описывает последовательность действий по формированию и печати разности заданных множеств: уменьшаемого и результирующего.

3.6. Отношение подмножества. Одно множество содержится в другом, если каждый элемент первого множества принадлежит второму множеству. Формальное определение отношения подмножества (AKO – a kind of) множеств A и B:

$$A \subseteq B \Leftrightarrow \forall x (x \in A \Rightarrow x \in B).$$

На рис. 4 обозначено штриховкой.

У предиката subset(L1,L2), реализующего это отношение, два параметра: L1 – множество, являющееся подмножеством другого L2 множества.

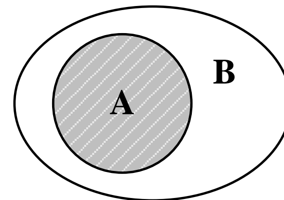

Рис. 4. Множество A является подмножеством B

Базис рекурсии: представлен фактом, утверждающим, что пустое множество является подмножеством любого множества.

Шаг рекурсии: первое множество является подмножеством второго, если первый элемент первого множества принадлежит второму множеству, а его хвост является подмножеством второго множества.

Пролог описание этого предиката имеет вид.

```
subset([],_).
subset([H|T],S):– member(H,S), subset(T,S).
```

В метаязыке предикат subset(L1,L2) определим для трех случаев, когда: пустым является первое L1 множество, а второе L2 – любым; оба множества L1, L2 – произвольны; первое L1 множество задано переменной, а второе – константное.

```
subset = '(' '[' ']' ',' '[' L1 ']' ')' / '(' '[' L2# ']' ',' '['
        L1# ']' ')' viv_Subset / '(' A L1# ','?
        '[' L# ']' ')' ^writeL1 ^genL ^headTail2
   ^hypothes1 ^takeL1 ^writeL1 viv_Subset1;
viv_Subset = (^genL2 ^headTail2 stepSubset)
        nullX;
stepSubset = ^genL1 ^takeL3 ^member5;
```

```
viv_Subset1 = (^genL2 ^headTail2
        ^hypothes2  ^takeL1 ^writeL1).
```

Первая альтернатива определения термина subset фиксирует тот факт, что пустое множество является подмножеством любого множества.

Вторая альтернатива состоит в поочередной проверке элементов первого множества L2 со всеми элементами второго множества L1.

Третья альтернатива описывает процесс последовательного порождения всех (начиная с пустого) подмножеств заданного множества L2.

3.7. Совпадение двух множеств. Два множества A и B называются равными, если одновременно выполнено $A\subseteq B$ и $B\subseteq A$, т. е. множество A содержится в множестве B и множество B в множестве A. Иначе говоря, два множества равны, если все элементы первого множества содержатся во втором множестве, и наоборот. Пролог-описание предиката, описывающего отношение эквиваленции двух множеств, имеет вид.

```
equal(A,B):– subset(A,B), subset(B,A).
```

В метаязыке предикат equal(L,L1) определяется последовательностью итераций: первая проверяет вхождение всех элементов первого множества L во вторoе множество L1, а вторая итерация проверяет вхождение всех элементов второго множества L1 в первое множество L.

```
equal = '(' '['? L# ']' ',' '['? L1# ']' ')' subsetLL1
        subsetL1L;
subsetLL1= ^genL (^headTail2 ^genL1
        ^takeL3 member2 ^genL2) nullX;
subsetL1L = ^genL1 (^headTail2 ^genL
        ^takeL3 member2 ^genL2) nullX.
```

3.8. Отношение собственного подмножества. Если множество B содержит множество A и другие элементы, кроме элементов множества A, то A – собственное подмножество множества B: $A\subset B$ (рис. 4) на Прологе имеет вид.

```
prop_subset(A,B):– subset(A,B),
                   not(equal(A,B)).
```

Подобным же образом этот предикат prop_subset(L,L1) определяется и в метаязыке:

```
prop_subset = '(' '['? L# ']' ',' '['? L1# ']' ')'
              subsetLL1 ^subsetL1L.
```

3.9. Симметрическая разность. В отличие от обычной разности, симметрическая разность не зависит от порядка ее аргументов. Симметрической разностью двух множеств называется множество, чьи элементы либо принадлежат первому и не принадлежат второму множеству, либо принадлежат второму и не принадлежат первому множеству. Формальное определение симметрической разности множеств A и B:

$$A\Delta B=\{x|(x\in A \,\&\, x\notin B) \vee (x\in B \,\&\, x\notin A)\}.$$

На рис. 5 обозначено штриховкой.

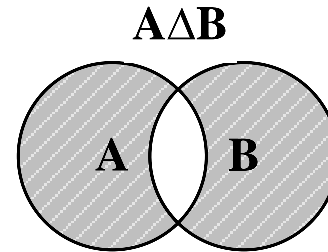

Рис. 5. Симметрическая разность множеств A и B

Симметрическую разность можно выразить через уже реализованные операции:

$$A\Delta B=(A\backslash B)\cup(B\backslash A).$$

Эти соотношения на Прологе получат вид.

```
sim_minus(A,B,SM):– minus(A,B,A_B),
        minus(B,A,B_A),
union(A_B,B_A,SM).
```

В метаязыке предикат sim_minus(L1,L2,L3) также можно выразить через уже определенные предикаты minus(L1,L2,L3) и union(L1,L2,L3) в следующей форме.

```
sim_minus = '(' E1 ',' E2 ',' E3# ')'
```

```
    ^genE1E2AB minus takeAB
    ^genE2E1BA minus takeBA
   ^genABBASM union;
^genE1E2AB = '(' E1 ',' E2 ',' 'AB' ')'!;
takeAB = 'AB' '=' '['? A_B#;
^genE2E1BA = '(' E2 ',' E1 ',' 'BA' ')'!;
takeBA = 'BA' '=' '['? B_A#;
^genABBASM = '(' A_B ',' B_A ',' 'SM' ')'!.
```

В этом выражении термины ^genE1E2AB, ^genE2E1BA и ^genABBASM формируют многоместные аргументы соответствующих предикатов minus и union, а takeAB и takeBA – принимают вычисленные значения переменных.

3.10. Дополнение. Дополнением DA множества A является подмножество элементов некоторого определенного универсума U, не принадлежащих A. Формальное определение отношения supp(A,DA) дополнения:

$$DA = U \setminus A = \{x | x \in U \ \& \ x \notin A\}.$$

На рис. 6 обозначено штриховкой.

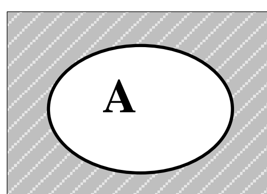

Рис. 6. Отношение дополнения множества A

Множество DA вычислимо лишь тогда, когда определены множества U и A. Для этого случая семантика предиката supp(U,A,DA) совпадает с семантикой предиката minus(U,A,DA), что на Прологе представляется в форме.

```
supp(U,A,DA):– minus(U,A,DA).
```

Предикат supp(L1,L2,L3) можно выразить и в метаязыке через уже определенный предикат minus(L1,L2,L3) в следующей форме.

```
supp = '(' E1 ',' E2 ',' E3# ')' ^genE1E2E3
          minus;
^genE1E2E3 = '(' E1 ',' E2 ',' E3 ')'!.
```

Здесь E1 – универсальное множество U, E2 и E3 – взаимно дополнительные множества A и DA.

Выводы

Базируясь на известных описаниях на Прологе, использующих списковый домен, впервые разработана в метаязыке НФЗ новая практически пригодная формализация *списков, предикатов на списках и множествах*.

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

Литература

1. Братко, Иван. Алгоритмы искусственного интеллекта на языке PROLOG. 3-е издание. Пер. с англ. М.: Издательский дом "Вильямс", 2004. 640 с.
2. Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. СПб.: БХВ-Петербург, 2003. 992 с.
3. Клоксин У., Меллиш К. Программирование на языке Пролог. М.: Мир, 1987. 334 с.
4. Abelson, Harold. Structure and interpretation of computer programs. Harold Abelson and Gerald Jay Sussman, with Julie Sussman. 2nd ed. The MIT Press Cambridge, Massachusetts London, England © 1996 by The Massachusetts. 576 p.
5. Haskell 98. Language and Libraries. The Revised Report. Editor by Simon Peyton Jones. Cambridge Academ, 2003. 277 p.
6. Seibel, Peter. Practical Common Lisp. Apress, 2005. 501 p. DOI 10.1007/978-1-4302-0017-8.
7. Кургаев А.Ф., Григорьев С.Н. Нормальные формы знаний. Доп. НАН України. 2015. № 11. С. 36–43.
8. Кургаев А.Ф., Григорьев С.Н. Метаязык нормальных форм знаний. *Кибернетика и системный анализ*. 2016. Том 52. № 6. С. 11–20.
9. Кургаев А.Ф. Формализация списков в метаязыке нормальных форм знаний. Доп. НАН України. 2017. № 10. С. 18–27. doi: https://doi.org/10.15407/dopovidi2017.10.018

References

1. Bratko, Ivan. Prolog Programming for Artificial Intelligence / Third Edition. – Addison-Wesley, 2012. – 696 p.
2. Adamenko A.N., Kuchukov A.M. Logical Programming and Visual Prolog. - St. Petersburg: BHV-Petersburg, 2003. - 992 p. (in Russian).
3. Clocksin, William., Mellish, Christopher S. Programming in Prolog: Using the ISO Standard 5th Edition. – Berlin, Heidelberg: Springer-Verlag, 2003. – 300 p. DOI 10.1007/978-3-642-55481-0
4. Abelson, Harold. Structure and interpretation of computer programs / Harold Abelson and Gerald Jay Sussman, with Julie Sussman. – 2nd ed. / The MIT Press Cambridge, Massachusetts London, England © 1996 by The Massachusetts. – 576 p.
5. Haskell 98. Language and Libraries. The Revised Report / Editor by Simon Peyton Jones. – Cambridge Academ, 2003. – 277 p.
6. Seibel, Peter. Practical Common Lisp. – Apress, 2005. – 501 p. DOI 10.1007/978-1-4302-0017-8
7. Kurgaev, A. The normal forms of knowledge / A.Kurgaev, S.Grygoryev. – Dopov. NAN Ukraine, 2015, № 11. – P. 36-43 (in Russian).
8. Kurgaev, A. Metalanguage of Normal Forms of Knowledge / A.Kurgaev, S.Grygoryev. – Cybernetics and Systems Analysis. – November 2016, 52(6), 839-848. DOI 10.1007/s10559-016-9885-3
9. Kurgaev, A. The Formalization of Lists in the Meta-Language of Normal Forms of Knowledge / Dopov. NAN Ukraine. – 2017. – №10. – P. 18 – 27. (in Russian). doi: https://doi.org/10.15407/dopovidi2017.10.018

Получено 27.01.2020

Об авторе:

Кургаев Александр Филиппович,
доктор технических наук,
профессор, ведущий научный сотрудник.
Количество научных публикаций в украинских изданиях – более 240.
Количество научных публикаций в зарубежных индексированных изданиях – 24.
Индекс Scopus: 2,
h-индекс (Google Scholar): 6
http://orcid.org/0000-0001-5348-2734.

Место работы автора:

Институт кибернетики имени
В.М. Глушкова НАН Украины,
03187, Киев-187,
проспект Академика Глушкова, 40.
Тел.: 8 050 881 62 18.

E-mail: afkurgaev@ukr.net