Пролог. Додаткові можливості

Створення і декомпозиція термів

У програмах мовою Пролог можна використовувати стандартні терми синтезу термів: *functor, arg, =..*
Розглянемо спочатку інфіксний оператор =..

```
Term =.. List
```

Тут *List* – список, головою якого є головний функтор терма, а хвостом – список аргументів. Наприклад:

```
?- belogs(f,L) =.. List   ==>   List = [belongs, f, L]

?- person(bill, smith, date(13,9,1991),
          work(symphonySolution, 50000)) =.. Decomposition   ==>
Decomposition = [person | bill, smith, date(13,9,1991),
                          work(symphonySolution, 50000)]

?- R =.. [rectangle, 3, 4]   ==>   R = rectangle( 3, 4)
```

Можливість виконувати декомпозицію та синтез термів надає програмам додаткової гнучкості.

Приклад 1. Деяка програма маніпулює плоскими геометричними фігурами: прямокутниками, трикутниками, кругами. Для цього у ній використовують відношення вигляду

```
rectangle(A, B)     % прямокутник задано розмірами сторін
triangle(A, B, C)   % трикутник – трьома сторонами
circle(R)           % круг – радіусом
```

Як реалізувати відношення *збільшити_у_k_разів*? Наприклад, так:

```
%  multiply(Shape, K, NewShape)
multiply(rectangle(A, B), K, rectangle(C, D)) :-
     C is A * K,  D is B * K.
multiply(circle(R), K, circle(P)) :- P is R * K.
```

Це працюватиме, але процедура *multiply* далека від досконалості: потрібно оголошувати черговий варіант для кожного типу фігури. Наприклад, бракує ще рядочка для *triangle*. А якщо до програми додаватимуть нові типи фігур, то виникне проблема пошуку (в тексті) та оновлення *multiply*.

Можна запропонувати універсальне рішення:

```
multiply(Shape, K, NewShape) :-
   Shape =.. [ Kind | Arguments ],
   multList(Arguments, K, NewArguments),
   NewShape =.. [ Kind | NewArguments].

multList([], _, []).
multList([ X | T ], K, [Y | Tail]) :-
   Y is X * K, multList(T, K, Tail).
```

Приклад 2. Перетворення формул часто потребує підстановок – одного виразу замість іншого. Спробуємо запрограмувати відношення *підстановка*.

```
% substitute(Instead, Where, What, Result)
```

Тут терм *Result* отримуємо після заміни всіх входжень підтерма *Instead* в терм *Where* на *What*. Наприклад:

```
?- substitute(t, 2*t*f(t), sin(x), F)   ==>   F=2*sin(x)*f(sin(x))
?- substitute(sin(x), 2*sin(x)*f(sin(x)), t, F)   ==>   F=2*t*f(t)
?- substitute(a+b, f(a, A+B), v, R)   ==>   R=f(a, v)
```

Входженням підтерма P в терм T називатимемо елемент терма T, сумісний з P при перегляді T від головного функтора до аргументів (зверху-вниз). Потрібно проаналізувати декілька випадків:

- якщо *Instead*=*Where* (суміщення), то *What*=*Result*;
- у протилежному випадку, якщо *Where* – структура, то треба виконати підстановку для аргументів, інакше *Where* – атом або число, відмінне від *Instead*, тому нема чого підставляти, і *Result*=*Where*.

Сказане можна описати програмою.

```
% substitute( Instead, Where, What, Result)
%   we get Result after substitution all occurrences of Instead
%   in the Where by the What

% Case 1. Substitution of all Term
substitute( Term, Term, Result, Result) :- !.   % !else

% Case 2. Nothing to substitute if the Term is 'atomic'
substitute( _, Term, _, Term) :-
   atomic(Term), !.    % !else

% Case 3. Substitution in the arguments
substitute( Instead, Where, ByWhat, Result) :-
   Where =.. [Functor | Arguments],         % to get arguments
   subst_list( Instead, Arguments, ByWhat, NewArgs), % substitute args
```

```
   Result =.. [Functor | NewArgs].

atomic(X) :- not( X =.. [_|_]).

subst_list( _, [], _, []).
subst_list( Instead, [Term | Rest], ByWhat, [NewTerm | Tail])  :-
   substitute( Instead, Term, ByWhat, NewTerm),
   subst_list( Instead, Rest, ByWhat, Tail).
```

Створення цілей

Терми, отримані за допомогою предиката =.. можна використовувати як цілі. Це дає можливість програмі самій породжувати та обчислювати цілі, структура яких не обов'язково відома на момент написання програми: програми на Пролог можуть самонавчатися. Схематично процес побудови нової цілі можна зобразити так:

```
decide(Functor),
calculate(ListOfArguments),
Goal =.. [ Functor | Arguments],
Goal    % call(Goal)
```

Приклад використання цієї техніки демонструє, як виконати довільне перетворення елементів списку

```
% --- manageList(oldList, binaryPredicate, newList)
manageList([], _, []).
manageList([ X | Tail], F, [ Y | MTail]) :-
   G =.. [ F, X, Y], call(G),
   manageList(Tail, F, MTail).

double(X,Y) :- Y is X * 2.

?-manageList([1,3,2,4,5], double, L), write(L), nl.
```

Такий підхід можна легко узагальнити на опрацювання декількох списків, оскільки F може позначати відношення довільної арності.

Інші способи декомпозиції

Дізнатися головний функтор і арність довільного терма можна відношенням

```
functor(Term, Functor, Arity)
```

Наприклад

```
?- P = person(bill,smith,date(13,9,1991),work(symphonySolution,50000)),
   functor(P, F, N).   ==>   F = person, N = 4
```

Запит

```
?- functor(D, data, 3)   ==>   D = data(_1, _2, _3)
```

створює «узагальнений» терм з заданим функтором і трьома аргументами – автоматично згенерованими змінними.

Дізнатися аргумент терма за його порядковим номером можна відношенням

```
arg(N, Term, Argument)    % не реалізовано в Strawberry Prolog !!!
```

Наприклад

```
?- arg(1, date(13, sep, 2019), A).   ==>   A = 13

?- functor(D, data, 3),
      arg(1, D, 13), arg(2, D, sep), arg(3, D, 2019).
==>  D = data(13, sep, 2019)

% оголосіть відношення arg !
```

Синтез і декомпозицію атомів виконують відношенням

```
name(Atom, ListOfLetters)    % не реалізовано в Strawberry Prolog !!!
```

Воно ставить у відповідність ім'я та список ASCII-кодів його літер. Наприклад

```
?- name(date, L).   ==>   L = [100, 97, 116, 101]
% як це зробити в Strawberry Prolog ?
```

Різні види рівності

- $X = Y$ означає X та Y сумісні, виконує суміщення в разі успіху
- $X \mathrel{?}= Y$ означає X та Y суміщувані, перевіряє можливість суміщення
- $X \backslash= Y$ обернене до попереднього
- *X is Expression* означає X сумісний зі значенням арифметичного виразу *Expression*, виконує обчислення
- *Expr_1* $=:=$ *Expr_2* перевіряє рівність двох виразів (числових чи рядкових), виконує обчислення
- *Expr_1* $=\backslash=$ *Expr_2* перевіряє нерівність двох виразів (числових чи рядкових), виконує обчислення
- *Term_1* $==$ *Term_2* перевіряє тотожність двох термів
- *Term_1* $\backslash==$ *Term_2* перевіряє нетотожність двох термів

База знань (фактів)

Ми вже знаємо, що програму мовою Пролог можна трактувати як базу даних. А всяку базу можна модифікувати. Виявляється, що програму – також. Після компіляції Пролог-програми машина виведення зберігає в пам'яті множину фактів і правил – базу знань. Цю базу можна модифікувати під час виконання програми: дописувати в неї нові (побудовані чи виведені) факти і правила, за потреби вилучати окремі факти чи правила.
Реалізація Strawberry Prolog має також предикати для дописування рядків коду в текст програми (до файла).

```
assert(Fact).  % дописує терм в кінець бази
asserta(Fact). % дописує терм на початок бази

% дописує терм в кінець бази та файла програми Strawberry
assert_in(Fact).
% дописує терм на початок бази та файла програми Strawberry
asserta_in(Fact).

retract(Term). % вилучає з бази перше входження Term
retract_in(Term) % вилучає Term з бази і з програми Strawberry
retractall(Term) % вилучає з бази Strawberry всі входження Term

% додавання правил має свої особливості:
% правило беруть у дужки і не ставлять крапку
% Strawberry дозволяє обходитись без додаткових дужок
assert((Rule)).
asserta((Rule)).
```

Наприклад

```
good_weather :- sunny, not(rain).
unusual_weather :- sunny, rain.
bad_weather :- rain, fog.
rain.
fog.
?- good_weather, write(good).       % fail
?- bad_weather, write(bad).         % bad Yes
?- retract(fog).                    % Yes
?- bad_weather, write(bad).         % fail
?- assert(sunny).                   % Yes
?- unusual_weather, write(unusual). % unusual Yes
?- retract(rain).                   % Yes
?- good_weather, write(good).       % good Yes
```

Відповідь про стан погоди змінюється в ході діалогу.
Додавати-вилучати можна також правила

```
fast(ann).
fast(pit).
slow(tom).
slow(pat).
slow(jim).
speed(X):-fast(X);slow(X).

?- assert((faster(X,Y) :- fast(X),slow(Y))),
   faster(A,B), write([A,faster,B]),nl.
```

Відношення *faster* не було в програмі, але після виконання *assert* запит дасть очікувану відповідь.

За допомогою *assert* та *retract* можна власноруч реалізувати предикат, який збирає всі можливі розв'язки. Логіка дій:

- знайдений розв'язок можна дописати до бази знань;
- щоб він був упізнаваним, можна позначити його спеціальним способом, наприклад, вказати, що він має властивість *solution_* – тут підкреслення спеціально для того, щоб функтор відрізнявся від тих, які міг би використовувати користувач у своїй програмі;
- *fail* заставить знайти всі розв'язки;
- щоб запит завершився успіхом, предикат матиме альтернативу без *fail*; її завдання – завершити послідовність розв'язків термінальним елементом, наприклад, атомом *b0tt0m* (нулі заради унікальності);
- завершальний крок – вилучити розв'язки по одному з бази знань і зібрати їх в список, збирання завершується тоді, коли вилучити термінальний елемент (перевірка на тотожність!).

Програма за цим планом:

```
findAll( X, Goal, Xlist)  :-
  call( Goal),                          % find a solution
  assert( solution_(X) ),               % put it yo the DB
  fail;                                 % look for another solution
  assert( solution_(b0tt0m) ),          % mark the end of solution
  collect( Xlist).                      % gather all solutions

collect( L)  :-
  retract( solution_(X) ), !,           % get a solution from the DB
  ( X == b0tt0m, !, L = []              % is it the last one?
    ;
    L = [X | Rest], collect( Rest) ).   % if not - gather the rest

?- findAll(X, speed(X), L), write(L).
```

Стиль програмування

Наступні міркування є досить загальними і можуть бути застосовані для написання програм різними мовами, не тільки мовою Пролог.

Що таке хороша програма?

На практиці застосовують різні критерії якості програм. Перерахуємо загальноприйняті

Правильність – хороша програма повинна виконувати те, для чого її написали. Може видатися дивним, але цей очевидний критерій часто порушується через боротьбу за ефективність, чи тому, що програму написали ще до того, як остаточно зрозуміли постановку задачі.

Ефективність. Хороша програма не витрачає зайвої пам'яті чи процесорного часу. На Пролог легко писати неефективні програми, особливо, якщо концентруватися на декларативному сенсі і забувати про процедурний. Ми покажемо декілька способів підвищення ефективності.

Простота, читабельність. Хорошу програму легко читати і розуміти. Вона не мала б бути складною понад потребу: не варто застосовувати різноманітні трюки програмування, що затіняють сенс програми. Подання тексту програми має полегшувати її розуміння. Процедури відокремлюють одна від одної одним або декількома порожніми рядками. Цілі у тілі правила варто розташовувати кожну в окремому рядку з однаковими відступами від початку. Важливе також використання осмислених імен, що відповідають призначенню предикатів, змінних тощо.

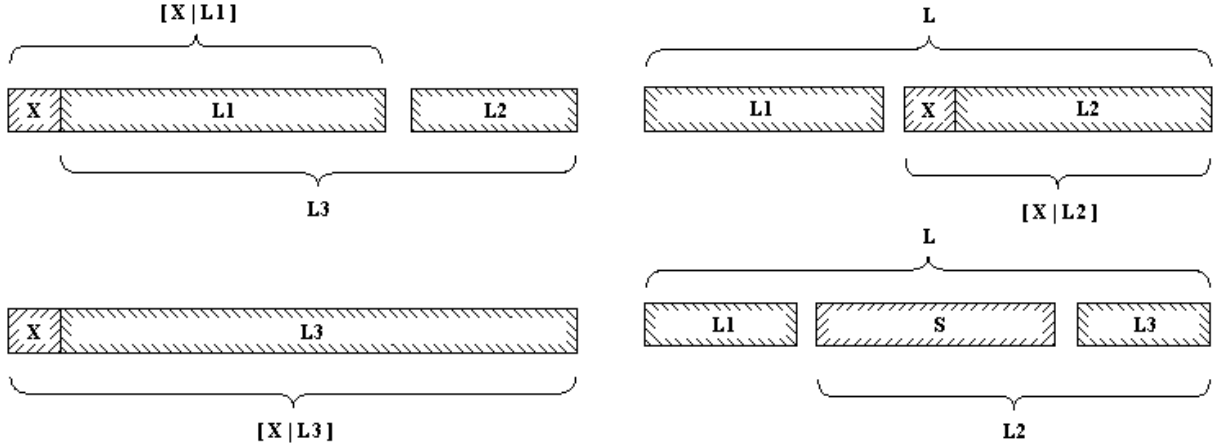
Зручність модифікації. Простота та модульна організація програми полегшують внесення змін і доповнень, а хороша програма мала б бути придатною до розширення.

Живучість. Хороша програма не повинна виходити з ладу відразу, коли користувач введе неправильні чи неочікувані дані.

Документованість. Чи не найскладніший критерій. Ніхто не хоче писати коментарі, проте хороша програма мала б містити докладні коментарі до всіх частин коду. Процедури варто ілюструвати прикладами використання, прикладами вхідних даних та очікуваних результатів, поясненнями щодо призначення процедури. Не зайвим буде вказання відношень найвищого рівня загальності.

Одним з методів побудови *правильних* програм є стратегія покрокової деталізації зверху вниз. Вона відома ще з часів структурного програмування,

але і для написання Пролог програм цілком підходить. Згідно з цією стратегією програму отримують поступово як результат низки перетворень загальних, абстрактних способів отримання розв'язку на більш конкретні й деталізовані. Починають з побудови «розв'язку верхнього рівня (абстракції)». Зазвичай його формулюють у термінах предметної області, звичайною мовою, використовують рисунки та схеми. Згадайте хоча б схеми, що ілюструють відношення «конкатенація» та «підсписок».



Розв'язок, записаний у термінах задачі, швидше всього буде правильним. Далі його деталізують, уточнюючи окремі частини. Кожен крок деталізації мав би бути достатньо невеликим, щоб зробити можливим перехід до понять менш загальних, і перехід до уточненого розв'язку відбувся без помилок. Деталізацію завершують тоді, коли отримують програму, записану мовою програмування.

При програмуванні мовою Пролог говорять про деталізацію *відношень*. Якщо для конкретної задачі важливішим виявиться процедурний аспект, можна говорити про деталізацію *алгоритмів*.

Загальні принципи написання пролог-програм

Використання рекурсії

Можливі випадки формулювання задачі треба розбити на дві групи:

1. Тривіальні чи «граничні» випадки.
2. Загальні випадки, розв'язки яких можна отримати з простіших (менших за розміром) варіантів початкової задачі.

Ілюстрації – більшість процедур опрацювання списків.

Узагальнення

Часто буває корисно переформулювати конкретну задачу для загального випадку так, щоб він допускав побудову рекурсивного розв'язку. Тоді вихідна задача стає частковим випадком загального варіанту, і її розв'язок отримується автоматично.

Ілюстрація – задача про вісім ферзів. При її розв'язуванні ми переходили до задачі про N ферзів з тривіальним випадком $N = 0$.
Використання схем.
Для відшукання ідеї побудови розв'язку варто використовувати графічне подання задачі. За допомогою схем легко зображати відношення між сутностями предметної області, а Пролог – природній засію для програмного відображення відношень. Часто програма отримання розв'язку задачі може бути описом того, як ми *бачимо* задачу.

Стиль програмування

Ми вже згадували про домовленості щодо запису тексту та вибору імен. Важко переоцінити важливість їх дотримання. Нагадаємо також про використання відсікань – використовуйте їх відповідально:

- якщо можна обійтися без відсікань, обходьтеся
- завжди віддавайте перевагу «зеленим» відсіканням, а не «червоним»
- якщо можна замість відсікання використати оператор *not*, використовуйте
- пам'ятайте, що успіх цілі *not(human(sokrat))* означає лише «у програми нема даних про те, чи є Сократ людиною»
- якщо можна довге твердження з відсіканнями розділити на декілька простих, розділяйте.

Наведемо приклад правильного та неправильного оголошення процедури злиття двох впорядкованих списків. Почнемо з невдалого, записаного «в імперативному стилі».

```
merge(ListA, ListB, ResultList) :-
   ListA = [], !, ResultList = ListB;   % if first list is empty
   ListB = [], !, ResultList = ListA;   % if second list is empty
   ListA = [ X | RestA ],
   ListB = [ Y | RestB ],
   (X < Y, !,
      Z = X,              % Z is the head of result list
      merge(RestA, ListB, ResultRest);
      Z = Y,
      merge(ListA, RestB, ResultRest)),
   ResultList = [ Z | ResultRest].
```

Порівняйте з декларативним оголошенням.

```
merge([], List, List).   % result of merging List with empty list
merge(List, [], List).   % the List
merge([X | RestA], [Y | RestB], [X | RestResult]) :-
   X < Y, !,             % if first list starts with a less element
```

```
    merge(RestA, [Y | RestB], RestResult).
merge(ListA, [Y | RestB], [Y | RestResult]) :-    % else
    merge(ListA, RestB, RestResult).
```

Ефективність

Тривалість виконання стає суттєвою, якщо програму постійно використовують для розв'язування класу задач. Сучасні комп'ютери не дуже добре пристосовані до алгоритмів роботи машини виводу Пролог, тому не зайвим буде розглянути декілька прийомів пришвидшення її роботи. Можна говорити про покращення двох видів:

- зміна алгоритму з тим, щоб якнайшвидше відтинати ті гілки дерева рішень, які не ведуть до розв'язку, тим самим мінімізувати обчислення хибних варіантів; або запам'ятовувати знайдені вдалі рішення;
- удосконалення структур даних програми, які моделюють об'єкти предметної області, з тим, щоб операції над ними виконувалися швидше.

Зміна структури в задачі про 8 ферзів
Для перебору ординат ми використовували відношення

```
belongs(Y, [1,2,3,4,5,6,7,8])
```

Саме воно постачало програмі варіанти значень для Y. Але заданий у списку порядок координат дуже невдалий для задачі. Він поміщає всі ферзі на одну діагональ, що завідомо неправильно. Набагато розумніше (і ефективніше) було б використати щось таке:

```
belongs(Y, [1,5,2,6,3,7,4,8])
```

У задачах пошуку на графі порядок опису графа також має значення. Якщо Ви описуєте граф послідовністю фактів, чи елементами списку, першими розташовуйте інформацію про ті вершини, які мають найбільше сусідів, інформацію про сусідні вершини і в програмі розташовуйте поруч.

Додавання фактів до бази знань
У цій лекції вже було наведено приклад використання предиката *asserta* для суттєвого підвищення ефективності алгоритму обчислення послідовності Фібоначчі – з експотенційного за складністю до лінійного.

Підвищення ефективності конкатенації списків через удосконалення структури

Пригадаємо собі оголошене раніше відношення

```
concatenate([], List, List).
concatenate([X | Tail], List, [X | NewTail]) :-
   concatenate(Tail, List, NewTail).
```

Воно очевидне в декларативному сенсі й доволі затратне в процедурному, адже за кожного виконання «розбирає» перший список на окремі елементи. Виявляється, лінійний за складністю алгоритм можна зробити константним за рахунок зміни способу подання списку.

Всякий список можна позначити парою списків так, щоб знати, де розташовано його закінчення. Наприклад, список [a, b, c] можна зобразити [a, b, c | [d, e]] : [d, e], або [a, b, c] : [ ], або [a, b, c | T] : T, де T – довільний список. За такого подання списків відношення конкатенації можна записати простим фактом:

```
concat(A:B, B:C, A:C).
```

Тоді запит

```
?- concat([a,b,c,d|T]:T,[e,f,g|L]:L,R), write(R).
```

миттєво видає відповідь після одного єдиного суміщення:

```
[a, b, c, d, e, f, g] : []Yes.
```

Це далеко не вичерпний перелік. Сподіваємося, читач відшукає і запропонує свої способи та приклади підвищення ефективності.

Запитання : завдання

1. Напишіть програму для спрощення алгебричних формул, наприклад, “$a + 0 \rightarrow a$”, “$a * 0 \rightarrow 0$”, “$a * 1 \rightarrow a$”.
2. Напишіть інтерактивну програму перетворення елементів списку за допомогою *manageList*, описаного в лекції, так, щоб користувач міг ввести з клавіатури функтор чи правило, яке потрібно застосувати до кожного елемента списку.
3. Визначте відношення *arg*, описане в лекції, наведіть приклади використання.
4. Наведіть приклад програмного створення та обчислення цілей – програму, шлях і спосіб виконання якої залежить від попередньо обчислених даних, побудованих функторів.

5. Наведіть змістовний приклад програми, яка додає до бази знань нові факти та правила, використовує їх у подальших обчисленнях.
6. Випробуйте дію описаного в лекції *findAll*, порівняйте його з вбудованим аналогом. Який ефективніший? Чи можна замість *solution_* та *b0tt0m* використовувати якісь інші імена?
7. Наведіть приклад, можливо з власного досвіду, написання складних відношень у імперативному та декларативному стилях.
8. Наведіть приклади використання списків, зображених парою $List : T$. Чи можна запропонувати ефективні предикати, що описують взаємне перетворення класичних списків у вдосконалені та навпаки? У яких задачах «пара списків» буде особливо доречною.
9. Наведіть власні приклади підвищення ефективності пролог-програм.