Лабораторні роботи №5-6.

Основи програмування на мові Пролог

Вступ

Пролог є логічним і декларативна мова програмування. Ім'я безпосередньо, Пролог, короткий для Програмування в Логіці. Спадщина Prolog' включає дослідження на теорема provers і інші автоматизовані системи віднімання розвивалися в 1960-х і 1970-х. Механізм виведення Прологу заснований на Robinson' принципі (1965) резолюції разом з механізмами для розпаковування відповідей, запропонованих Зеленню (1968). Ці ідеї прийшли разом дійсно з прибуттям процедур лінійної резолюції.
Явний мета-направлено процедури лінійної резолюції, як наприклад ті з Kowalski і Kuehner (1971) і Kowalski (1974), надали поштовх розвитку загальної логіки мети, програмування системи. Пролог "Спочатку" був "марсельним Прологом", заснованим на роботі Colmerauer (1970). Перший детальний опис мови Прологу був керівництвом для марсельного перекладача Прологу (Roussel, 1975). Інший головний вплив на природу цього першого Прологу був, що це проектується, щоб полегшити обробку розмовної мови.

1. Як Управляти Прологом

Приклади в цій Консультації Прологу розвивалися, використовуючи або Пролог Quintus, бігом на Цифровій Корпорації Устаткування MicroVAXes (древня історія), або, використовуючи Пролог SWI на будь-яких Сонячних Іскрах (давно), в Windows на ПК (недавно), або (недавно) під ОС X операційна система на Mac.
Щоб запустити діалогову сесію SWI-prolog під Unix, відкривають термінальне вікно і друкують apprоprite команду (вказано в інсталяційних інструкціях). Наприклад, на нашому Mac це є ...

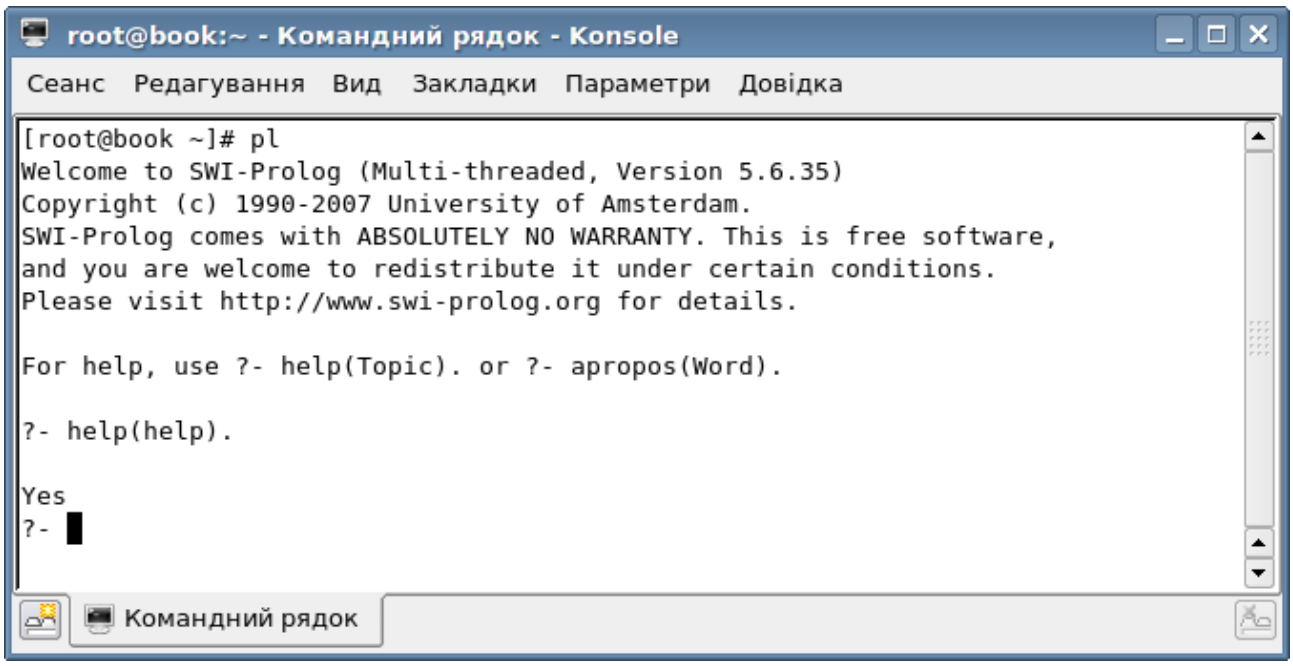
```
$ /opt/local/bin/swipl
```

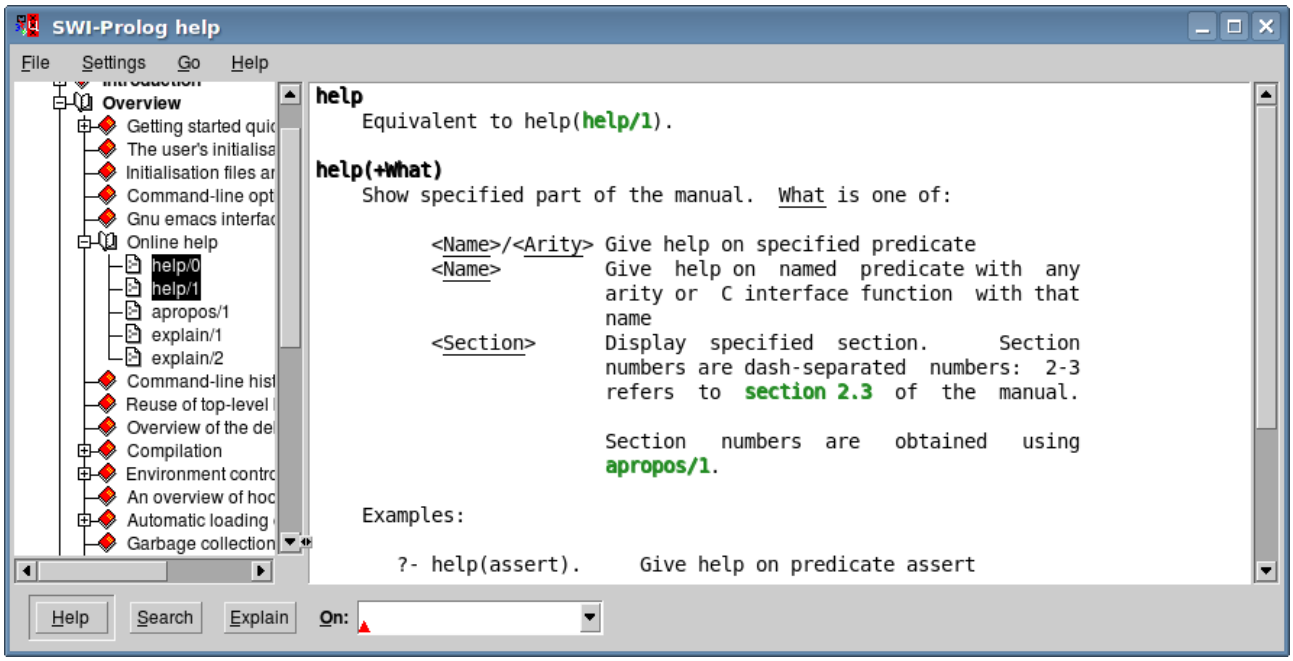
{Ми ніколи не друкуємо цю останню лінію: ми використовуємо Unix файли джерела, щоб запустити SWI-prolog, використовуючи додаткові аргументи командного рядка і/або переключає для цілей спеца. Студент міг дослідити цю можливість скоро після вивчення більше основ прологу.}
Під Windows, SWI-prolog встановлює стартову ікону, яка може бути таким, що клацнув двічі, щоб ініціювати перекладача. Перекладач потім починається в його власному командному вікні. Повідомлення запуску або прапор, можливо, з'являється, і це скоро буде завершено goal негайно, виглядаючи подібним до наступного

```
?- _
```

Діалогові цілі в Пролозі вводить користувач, наступний за "?- " негайно. Багато Прологів мають інформацію допомоги командного рядка. Пролог SWI має обширну інформацію допомоги. Ця допомога індексована і веде користувача. Щоб дізнатися більше про це, спроба

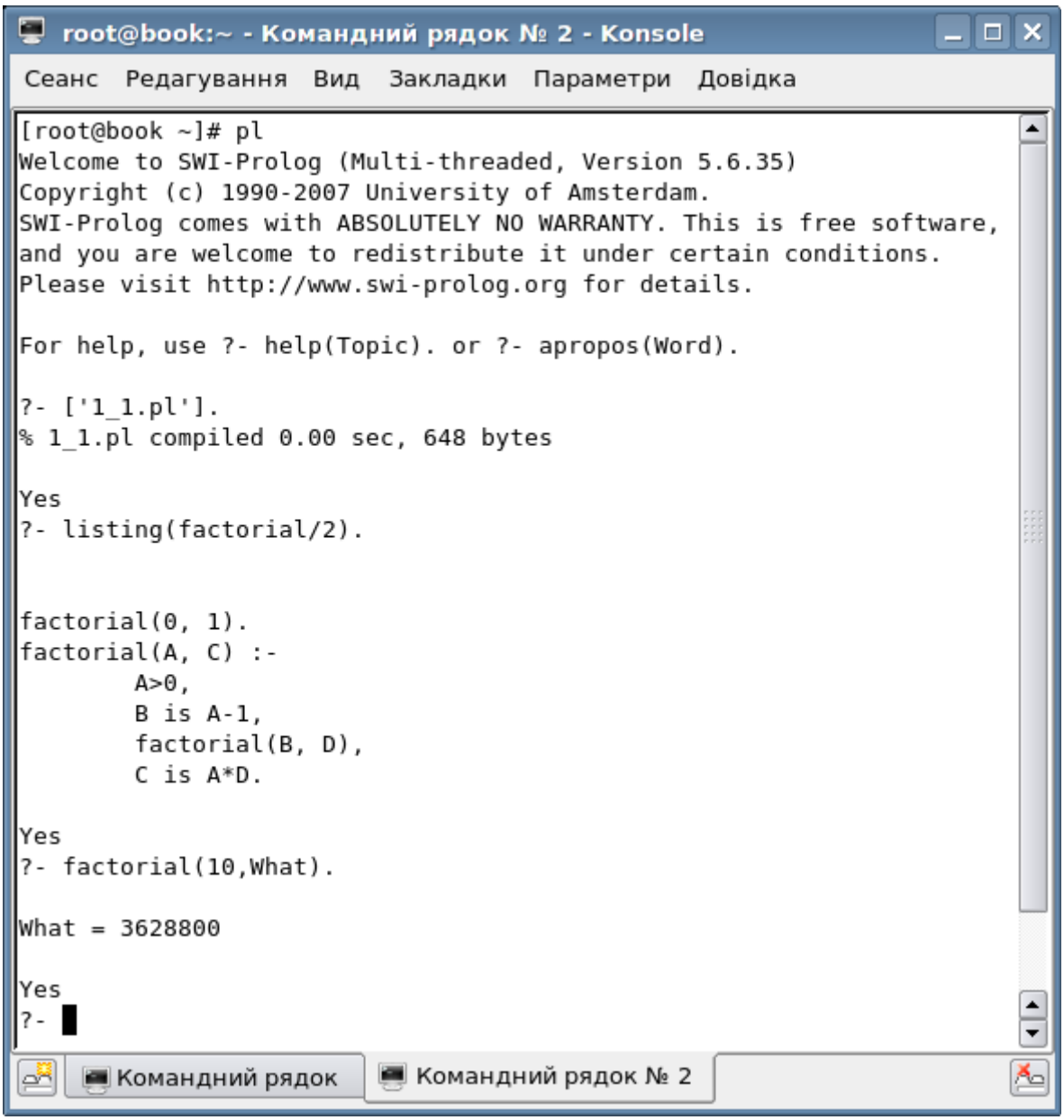
```
?- help(help).
```





Відмітьте, що всі показані символи потрібно надрукувати, завершено поверненням вагону. Щоб ілюструвати деякі специфічні взаємодії з прологом, розглядають наступну типову сесію. Кожен файл послався передбачається, щоб бути місцевим файлом на user' рахунку, який також створив користувач, отриманий копіюючи безпосередньо від деякого іншого суспільного джерела, або отриманий зберігаючи текстовий файл, використовуючи browser. Дорога досягти останнього - або слідувати за URL до вихідного файлу а потім збереження, або вибирати текст у веб-сторінці Консультації Прологу, копіюють це, наклеюють у вікні текстового редактора а потім зберігають до файлу. Коментарі /* ... */ поряд з цілями прямують в примітках, наступних за сесією.

```
?- ['2_1.pl'].           /* 1. Load a program from a local
file*/
yes
?- listing(factorial/2). /* 2. List program to the screen*/

factorial(0,1).

factorial(A,B) :-
          A > 0,
          C is A-1,
          factorial(C,D),
          B is A*D.
yes

?- factorial(10,What).     /* 3. Compute factorial of 10 */
What=3628800

?- ['2_7.pl'].            /* 4. Load another program */

?- listing(takeout).

takeout(A,[A|B],B).
takeout(A,[B|C],[B|D]) :-
         takeout(A,C,D).
yes

?- takeout(X,[1,2,3,4],Y).  /* 5. Take X out of list
[1,2,3,4] */
X=1  Y=[2,3,4] ;              Prolog waits ... User types ';'
and Enter
X=2  Y=[1,3,4] ;               again ...
X=3  Y=[1,2,4] ;               again ...
X=4  Y=[1,2,3] ;               again ...
no                             Means: No more answers.

?- takeout(X,[1,2,3,4],_), X>3.  /* 6.  Conjunction of goals
*/
X=4 ;
no

?- halt.                         /* 7. Return to OS */
```


root@book:~ - Командний рядок № 2 - Konsole
Сеанс Редагування Вид Закладки Параметри Довідка
[root@book ~]# pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.35)
Copyright (c) 1990-2007 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).
?- ['1_1.pl'].
% 1_1.pl compiled 0.00 sec, 648 bytes
Yes
?- listing(factorial/2).
factorial(0, 1).
factorial(A, C) :-
A>0,
B is A-1,
factorial(B, D),
C is A*D.
Yes
?- factorial(10,What).
What = 3628800
Yes
?-
Командний рядок
Командний рядок № 2

2. Програми Типового Прологу

2.1 Забарвлення карти

Знаменита проблема в математиці стосується забарвлення сусідніх плоских регіонів. Подібно до картографічних карт, це потрібно, щоб, що б кольори не були фактично used, ні два сусідні регіони, можливо, не мають того ж кольору. Два регіони - даний adjacent, забезпечений, вони розділяють деякий сегмент пограничної лінії. Розглядайте наступну карту.

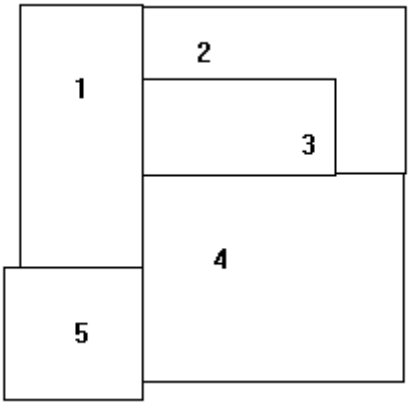


Fig. 2.1.1.

Ми надали числові імена регіонам. Щоб представити який регіони сусідні, розглядають також наступного графа.

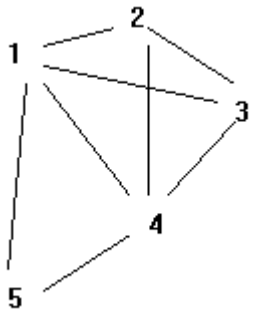


Fig. 2.1.2.

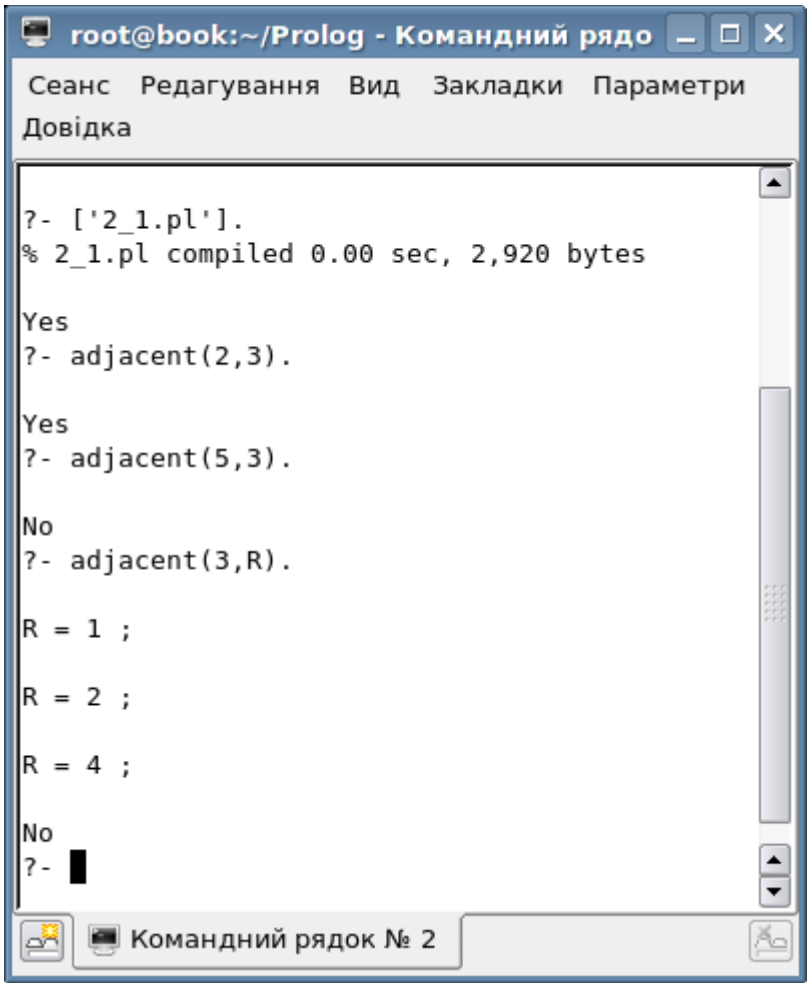
Тут ми стерли оригінальні кордони і замість відтягнули дугу між іменами двох регіонів, забезпечив, вони сусідні в оригінальному малюнку. Фактично, граф суміжності супроводжуватиме всю оригінальну інформацію суміжності. Представлення Прологу для інформації суміжності могли представити наступні пропозиції одиниці, або факти.

```
adjacent(1,2). adjacent(2,1). adjacent(1,3). adjacent(3,1).
adjacent(1,4). adjacent(4,1). adjacent(1,5). adjacent(5,1).
adjacent(2,3). adjacent(3,2). adjacent(2,4). adjacent(4,2).
adjacent(3,4). adjacent(4,3). adjacent(4,5). adjacent(5,4).
```

Якщо ці пропозиції були завантажені в Пролозі, ми могли спостерігати наступну поведінку для деяких цілей.

```
?- adjacent(2,3).
```

```
yes
?- adjacent(5,3).
no
?- adjacent(3,R).
R = 1 ;
R = 2 ;
R = 4 ;
no
```



Один міг оголосити забарвлення для регіонів в Пролозі також, використовуючи пропозиції одиниці.

```
color(1,red,a). color(1,red,b). color(2,blue,a).
color(2,blue,b). color(3,green,a). color(3,green,b).
color(4,yellow,a). color(4,blue,b). color(5,blue,a).
color(5,green,b).
```

Тут ми кодували "a" і "b" забарвлення. Ми хочемо написати визначення Прологу суперечливого забарвлення, означаючи, що два сусідні регіони мають той же колір. Наприклад, тут є пропозиція Прологу, або правило, до цього ефекту.

```
conflict(Coloring) :- adjacent(X,Y), color(X,Color,Coloring),
                      color(Y,Color,Coloring).
```

Наприклад

```
?- conflict(a).
no
?- conflict(b).
yes
?- conflict(Which).
Which = b
```

Ось - інша версія "conflict", який має логічніші параметри.

```
conflict(R1,R2,Coloring) :- adjacent(R1,R2),
                            color(R1,Color,Coloring),
                            color(R2,Color,Coloring).
```

Пролог дозволяє і відрізняє два визначення "conflict"; один має один логічний параметр ("conflict/1") і інший має три ("conflict/3"). Зараз ми маємо

```
?- conflict(R1,R2,b).
R1 = 2 R2 = 4
?- conflict(R1,R2,b),color(R1,C,b).
R1 = 2 R2 = 4 C = blue
```

Останні goal засоби, що регіони 2 і 4 сусідні і обидва блакитні. Подібно до "conflict(2,4,b) " говорять, що грунтовані зразки є наслідками програми Пролог. Єдина дорога продемонструвати такий наслідок - відтягнути дерево програмної пропозиції, що має наслідок як корінь дерева, використовують пропозиції програми, щоб відгалузитися дерево, і кінець кінцем виробляють обмежений tree має всі дійсні листи. Наприклад, наступне дерево пропозиції може бути сконструйоване, використовуючи повністю грунтовані зразки (немає змінних) пропозицій програми.

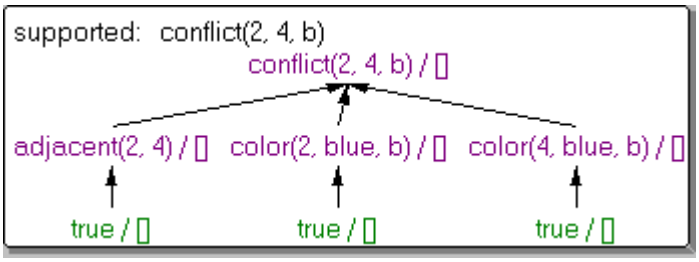


Fig. 2.1.3

Щоб дізнатися більше про візуальний логічний інструмент звикся автоматично робити digrams подібно до того в попередньому показі. Нижня крайня зліва гілка, відтягнута в дереві, відповідає пропозиції одиниці

```
adjacent(2,4).
```

який еквівалентний в Пролозі до пропозиції

```
adjacent(2,4) :- true.
```

Зараз, з іншого боку, "conflict(1,3,b) " - не наслідок програми Пролог, тому що не можливо сконструювати обмежене обмежене дерево пропозиції, використовуючи грунтовані пропозиції P, що містить всі "true" листи. Також, "conflict(a) " - не наслідок програми, оскільки один чекав би. Нам доводитиметься більше говорити про дерева програмної пропозиції в подальших секціях.
Ми знову відвідаємо фарбувальну проблему знову в Секції 2.9, де ми розвиватимемо

програму Прологу, яка може обчислити всі можливі забарвлення (дані кольори, щоб фарбувати). Знаменитий Чотири Кольорова Здогадка була, що жодна плоска карта не вимагає більше чотирьох різних кольорів. Це довів Appel і Haken (1976). Рішення використовувало комп'ютерну програму (не Пролог), щоб перевірити на багатьох специфічних випадках плоских карт, для того, щоб виключити можливі випадки, занепокоєння яких заподіює. Карта у в Мал. 2.1.1 вимагає як мінімум чотири кольори; наприклад ...

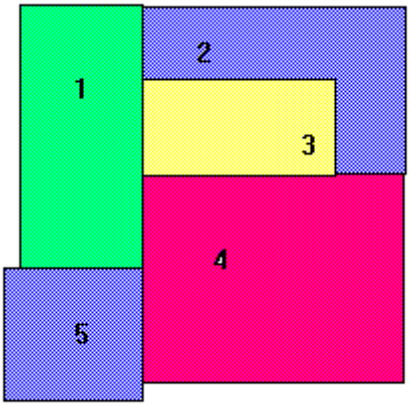


Fig. 2.1.4

Завдання 2.1, Якщо карта має регіони N, то оцінюють, скільки обчислень, можливо, доведеться зробити для того, щоб визначити чи або не забарвлення знаходиться в конфлікті. Аргументуйте використання дерев програмної пропозиції.

2.2. Два визначення факторіалу

Два затверджують визначення, які обчислюють фабричну функцію, знаходяться у файл 2_2.pl. Перший з цих визначень є:

```
factorial(0,1).
factorial(N,F) :-   N>0,
                    N1 is N-1,
                    factorial(N1,F1),
                    F is N * F1.
```

Це програмні склади двох пропозицій. Перша пропозиція - пропозиція одиниці, не маючи жодного тіла. Другий - правило, тому що це має тіло. Тіло другої пропозиції - справа сторона ":-", який може бути вичитаний нібито". Склади тіла літералів, відокремлених коми ",", кожен з якого може бути вичитаний, як "і". Глава пропозиції - ціла пропозиція, якщо пропозиція - пропозиція одиниці, інакше глава пропозиції - частина, що з'являється ліворуч від двокрапки в ":-". Декларативне свідчення першої (одиниця) пропозиції говорить, що "факторіал 0 складає 1" і друга пропозиція оголошує, що "факторіал N - F, якщо N>0 і N1 - N-1 і факторіал N1 - F1 і F — N*F1".

Мета Прологу, щоб обчислити факторіал номера 3 реагує із значенням для W, goal змінна:

```
?- factorial(3,W).
W=6
```

Вважайте, наступне дерево пропозиції сконструювало для буквального "factorial(3,W) ". Як пояснено в попередній секції, дерево пропозиції не містить жодних довільних змінних, але замість має моменти (значення) змінних. Кожен перехід під вершиною визначає пропозиція в оригінальній програмі, використовуючи доречні моменти змінних; вершину визначає деякий момент глави пропозиції і літерали тіла пропозиції визначають дітей вершини в дереві пропозиції.

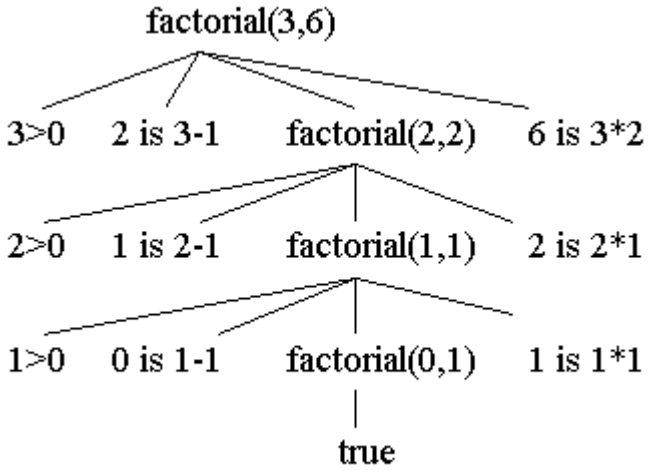


Fig. 2.2

Всі арифметичні листи вірні оцінкою (під інтерпретацією, що призначається), і найнижчий зв'язок в дереві відповідає дуже першій пропозиції програми для факторіалу. Що перша пропозиція могла бути написане

```
factorial(0,1) :- true.
```

і, фактично ?- достеменний - мета Прологу, яка завжди має успіх (достеменний вбудований). Ради стислості, ми не відтягнули "true" вирушає під дійсними арифметичними літералами. Дерево програмної пропозиції забезпечує значення програми для мети в корені дерева.

Тобто, "factorial(3,6) " - наслідок програми Пролог, тому що є дерево пропозиції, упроваджене в "factorial(3,6) ", всі чиї листи вірні. Буквальний "factorial(5,2) " є, з іншого боку, не наслідок програми, тому що немає жодного дерева пропозиції, упровадженого в "factorial(5,2) ", що має всі дійсні листи. Тому значення програми для буквального "factorial(5,2) " є, що це - брехня. Фактично

```
?- factorial(3,6).
yes
?- factorial(5,2).
no
```

як очікується. Дерева пропозиції - так звані і-дерева, починаючи з, для того, щоб корінь був наслідком програми, кожне з його піддерев має також бути упроваджене в літерали, які є собі наслідками програми. Нам доводитиметься більше говорити про дерева пропозиції пізніше. Ми вказали, що дерева пропозиції забезпечують значення або семантику для програм. Ми бачитимемо інший підхід до програмної семантики в Главі 6. Дерева пропозиції забезпечують інтуїтивний, також як і правильний, наближаються до програмної семантики. Нам потрібно буде розрізнити між деревами програмної пропозиції і так що-звернувся до дерев виведення Прологу. Дерева пропозиції "статичні" і можуть бути відтягнуті для програми і мети не дивлячись на особливий процедурний механізм задоволення мети. Грубо, кажучи, дерева пропозиції відповідають декларативному свідченню програми. Дерева виводу, з іншого боку, беруть до уваги обов'язковий для змінної механізм Прологу і замовлення, що підцілі розглядаються. Дерева виводу обговорюються в Секції 3.1 (але бачать натхнення нижче).
Слід виконання Прологу також показує, як змінні зв'язані для того, щоб задовольнити цілі. Наступні типові покази, як типовий дослідник Прологу повернений від випадку до випадку.

```
?- trace.
% The debugger will first creep -- showing everything
(trace).

yes
[trace]
?- factorial(3,X).
  (1) 0 Call: factorial(3,_8140) ?
  (1) 1 Head [2]: factorial(3,_8140) ?
  (2) 1 Call (built-in): 3>0 ?
  (2) 1 Done (built-in): 3>0 ?
  (3) 1 Call (built-in): _8256 is 3-1 ?
  (3) 1 Done (built-in): 2 is 3-1 ?
  (4) 1 Call: factorial(2, _8270) ?
   ...
  (1) 0 Exit: factorial(3,6) ?
X=6
[trace]
?- notrace.
% The debugger is switched off

yes
```

Заголовок цієї секції послався на два фабричні визначення. Ось - інший один, з тим же ім'ям предиката, але використовуючи три змінні.

```
factorial(0,F,F).

factorial(N,A,F) :-
```

```
    N > 0,
    A1 is N*A,
    N1 is N -1,
    factorial(N1,A1,F).
```

Для цієї версії, використовують наступний вигляд мети:

```
?- factorial(5,1,F).
F=120
```

Другий параметр у визначенні є так званим параметр, що накопичується. Ця версія належним чином хвостова рекурсивний. Дуже поважно для студента завершити наступні вправи.

Вправа 2.2.1, Використовуючи першу фабричну програму, показують явно, що не може можливо бути дерева пропозиції, упровадженого в "factorial(5,2) ", що має всі дійсні листи.

Вправа 2.2.2 Зволікають дерево пропозиції для goal "factorial(3,1,6) ", що має всі дійсні листи, за модою, подібною до цього, зробленому для факторіалу(3,6) заздалегідь. Як дві програми відрізняються відносно того, як вони обчислюють факторіал? Також, стежте за goal "factorial(3,1,6) ", використовуючи Пролог.

2.3 Башти ханойської головоломки

Цей предмет цієї знаменитої головоломки - перемістити диски N від орієнтиру лівої сторони управо peg, використовуючи center орієнтир як допоміжний орієнтир, що тримається. Ніколи не може більший диск бути розміщеним на більш маленькому диску. Наступна діаграма змальовує стартову установку для дисків N=3.

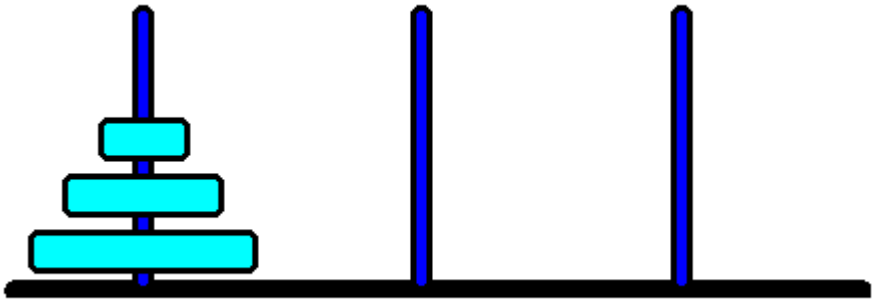


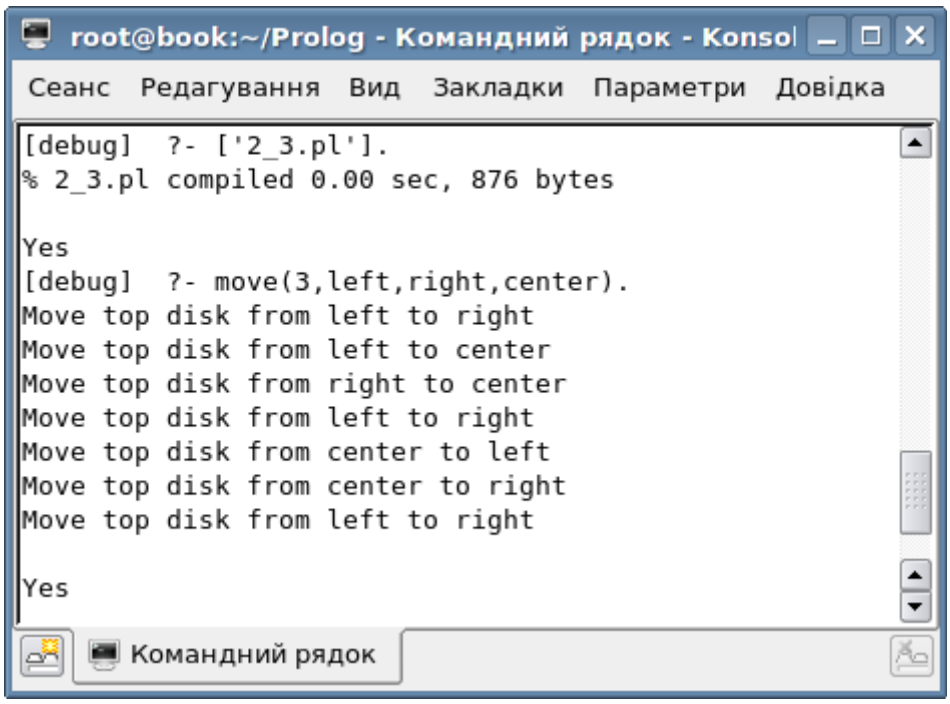
Fig. 2.3

Ось - рекурсивна програма Прологу, яка вирішує головоломку. Це склади двох пропозицій.

```
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl.
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```

Змінні заповнили "_" (або будь-які змінні, що починаються з underscore) - "don"t-care" змінні. Пролог дозволяє цим змінним вільно відповідати будь-яку структуру, але жодні мінливі обов'язкові результати, від цієї безкоштовної відповідності.
Тут - те, що трапляється, коли Пролог вирішує випадок N=3.

```
?-  move(3,left,right,center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right

yes
```



Перша пропозиція в програмі описує пересування єдиного диска. Друга пропозиція оголошує, як рішення могло бути отримане, рекурсивно. Наприклад, декларативне свідчення другої пропозиції для N=3, X=left, Y=right, і кількості Z=center, до наступного:

```
move(3,left,right,center) if
     move(2,left,center,right) and ] *
     move(1,left,right,center) and
     move(2,center,right,left). ] **
```

Це декларативне свідчення пропозиції очевидно правильне. Процедурне свідчення близько пов'язане з декларативною інтерпретацією рекурсивної пропозиції. Процедурна інтерпретація go би що-небудь подібно до цього:

```
In order to satisfy the goal ?- move(3,left,right,center) do
this :
     satisfy the goal ?-move(2,left,center,right), and then
     satisfy the goal ?-move(1,left,right,center), and then
     satisfy the goal ?-move(2,center,right,left).
```

Також, ми могли написати декларативні свідчення для N=2:

```
move(2,left,center,right) if ] *
     move(1,left,right,center) and
     move(1,left,center,right) and
     move(1,right,center,left).
move(2,center,right,left) if ] **
     move(1,center,left,right) and
     move(1,center,right,left) and
     move(1,left,right,center).
```

Зараз substitute bodies їх останніх двох значень для голів і один може "бачити" рішення, яке мета прологу виробляє.

```
move(3,left,right,center) if
     move(1,left,right,center) and
     move(1,left,center,right) and *
     move(1,right,center,left) and
     --------------------------
     move(1,left,right,center) and
     --------------------------
     move(1,center,left,right) and
     move(1,center,right,left) and **
     move(1,left,right,center).
```

Процедурне свідчення для цього останнього великого значення має бути очевидне. Цей приклад пояснює добре три великі дії Прологу:

1) Цілі погоджені проти глави правила, і
2) тіло правила (із змінними відповідно кордон) стає новою послідовністю цілей, неодноразово,
поки
3) деяка базова мета або умова задоволена, або деяка проста акція узята (подібно до друкування чого-небудь).

Мінливий відповідний процес - об'єднання, що звертається.
Вправа 2.3.1 Зволікають дерево програмної пропозиції для goal "move(3,left,right,center) " показують, що це - наслідок програми. Як дерево цієї пропозиції пов'язане з підстановкою обробляють пояснюється вище?
Вправа 2.3.2 Пробують мету Прологу ?-move(3,left,right,left). What' неправильно? Запропонуйте дорогу виправити це і керуватися, щоб бачити, що fix роботи.

2.4 Завантаження програми, редагування програм

Стандартний Пролог стверджує для навантаження програм - "consult", "reconsult", і зображення навантажувача дужки знаками "[...]". Наприклад, мета

2.5 Заперечення як невдача

"not" предикат заперечення-як-невдачі міг бути визначений в пролозі як вказано нижче:

```
not(P) :- call(P), !, fail.
not(P).
```

Мета ?-call(P) вимушує спробу задовольнити перекладачів Більш всього Прологу goal P. мають вбудовану версію цього предиката. Quintus, SWI, і інші прологи використовують "\+" замість "not".
Секція 3.2 має обговорення Прологу відрізаний предикат "!".
Іншим способом один може написати, визначення "not" використовує оператор значення Прологу "->" :

```
not(P) :- (call(P) -> fail ; true)
```

Тіло може бути вичитане, "якщо виклик(P) має (тобто, якщо P має успіх як мета) потім успіх несправності, інакше succeed". Крапка з комою ";" вигляд тут має значення exclusive логічний-або.
Будуть багаточисельні використання "not" в послідовних програмах в главі.
Поважно реалізувати це в меті ?-not(g), якщо г має змінні, і деякий момент цих змінних, ведуть до задоволення г, то ?-not(g) несправності. Наприклад, розглядайте наступну "bachelor програму:

```
bachelor(P) :- male(P), not(married(P)).

male(henry).
male(tom).

married(tom).
```

Потім

```
?- bachelor(henry).
yes
?- bachelor(tom).
no
?- bachelor(Who).
Who= henry ;
no
?- not(married(Who)).
no.
```

Перший три відповіді правильні і як очікується. Відповідь до четвертого запиту, можливо, була б несподівана спочатку. Але вважають, що мета ?-not(одружено(Хто)) терпить невдачу, тому що для мінливого обов'язкового Who=tom, одружений(Хто) має успіх, і так негативні goal несправності. Тому, негативні цілі ?-not(g) із змінними не може очікуватися виробляти скріплення змінних, для яких несправності goal г.
Визначення дерева програмної пропозиції в попередніх секціях призначалося для програм без заперечення як невдача. Для програм, що мають заперечення як невдачу в bodies програмних пропозицій, визначення дерева програмної пропозиції, і визначення наслідку

заснувало на цих деревах потрібно бути ретельно сформульованим. Ми мотивуємо це з ретельно вибраними прикладами тут, і покидають більш теоретичні розгляди, щоб Ділити на частини 8.8.
Для прикладу, розглядають програму

```
p(X) :- q(X), not(r(X)).
r(X) :- w(X), not(s(X)).
q(a).
q(b).
q(c).
s(a).
s(c).
w(a).
w(b).
```

Вважайте наступне трьома наборами дерев пропозиції.

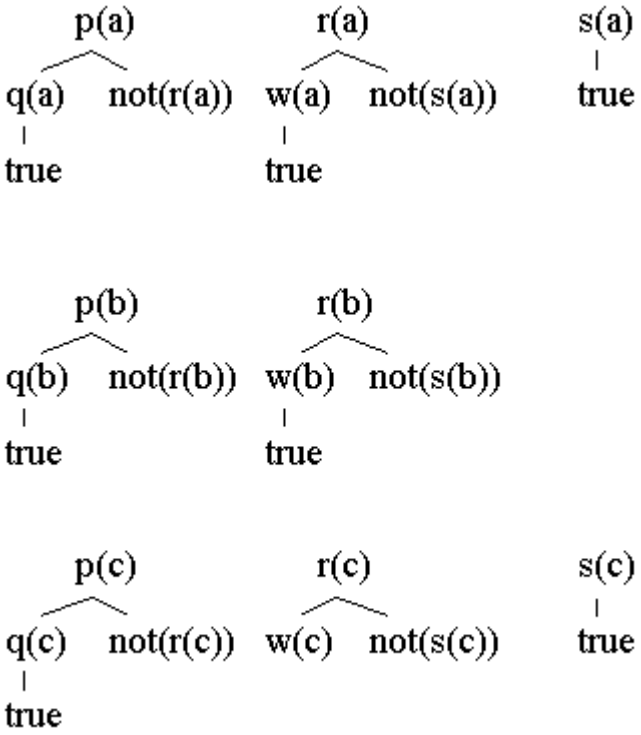


Fig. 2.5

Зараз, перший набір (через) використаний, щоб показати, що p(a) - наслідок програми, другий набір може бути використаний, щоб показати, що p(b) - не наслідок програма, і третій набір показує, що p(c) - наслідок програми. Відмітьте, що дерева пропозиції зараз відтягнуті таким чином, що негативний вершини "not(g) " - листи, і нові дерева, упроваджені в r, розслідувані.
Вправа 2.5.1 Завантажують приклад програми в Пролозі і перевіряють, що Пролог обчислить відповіді відповідно до нашого визначення наслідків.
Вправа 2.5.2 Розглядають наступну зміну прикладу програми.

```
p(X) :- q(X), not(r(X)).
r(X) :- w(x), not(s(X)).
q(a). q(b). q(c).
s(a) :- p(a). s(c).
```

```
w(a). w(b).
```

Що трапляється зараз, якщо один пробує сперечатися чи або не p(a) - наслідок цієї програми? Це показує, що з труднощами стикається, якщо там recursion через заперечення. Що трапляється, якщо ми дозволяємо Прологу пробувати мету `?-p(a).` ?
Вправа 2.5.3 Розглядають наступну програму.

```
u(X) :- not(s(X)).
s(X) :- s(f(X)).
```

Що трапляється з цим прикладом, якщо один пробує визначити наслідки, засновані на деревах пропозиції? Що Пролог робить, коли отримали мету ?-u(1). ? - s(1) наслідок програми?
Останній дві вправи показують, що програми із запереченням, оскільки невдача може бути проблематична, якщо там є або рекурсія через заперечення, або, якщо заперечення деякого безмежного літерала розглядається. Погляньте Секцію 8.8 для подальшого обговорення цієї теми.

2.6 Tree дані і стосунки

Розглядають наступну tree діаграму.

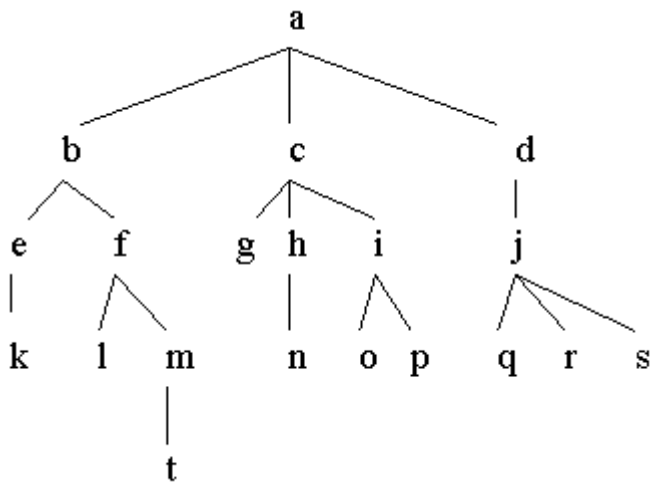


Fig. 2.6

Файл 2_6.pl має виставу для цього дерева і затверджують визначення, щоб зробити деяку обробку дерева. Відзначте використання операторів Прологу в деяких з визначень.

```
/* The tree database */

:- op(500,xfx,'is_parent').

a is_parent b.    c is_parent g.     f is_parent l.     j
is_parent q.
a is_parent c.    c is_parent h.     f is_parent m.     j
is_parent r.
a is_parent d.    c is_parent i.     h is_parent n.     j
is_parent s.
b is_parent e.    d is_parent j.     i is_parent o.     m
is_parent t.
b is_parent f.    e is_parent k.     i is_parent p.

 /* X and  Y are siblings  */
 :- op(500,xfx,'is_sibling_of').
 X is_sibling_of Y :- Z is_parent X,
                      Z is_parent Y,
                      X \== Y.

/* X and Y are on the same level in the tree. */
 :-op(500,xfx,'is_same_level_as').

 X is_same_level_as  X .
 X is_same_level_as  Y :- W is_parent X,
                          Z is_parent Y,
                          W is_same_level_as Z.
```

```
/* Depth of node in the tree. */
 :- op(500,xfx,'has_depth').

 a has_depth 0 :- !.
 Node has_depth D :- Mother is_parent Node,
                     Mother has_depth D1,
                     D is D1 + 1.

/* Locate node by finding a path from root down to the node.
*/
 locate(Node) :- path(Node),
                 write(Node),
                 nl.
 path(a).                              /* Can start at a.
*/
 path(Node) :- Mother is_parent Node,  /* Choose parent,
*/
               path(Mother),           /*  find path and then
*/
               write(Mother),
               write(' --> ').

/*  Calculate the height of a node, length of longest  path
to
    a leaf under the node.   */

 height(N,H) :- setof(Z,ht(N,Z),Set),  /* See section 2.8 for
'setof'.  */
                max(Set,0,H).

 ht(Node,0) :- leaf(Node), !.
 ht(Node,H) :- Node is_parent Child,
               ht(Child,H1),
               H is H1 +1.

 leaf(Node) :- not(is_parent(Node,Child)). /* Node grounded
*/

 max([],M,M).
 max([X|R],M,A) :- (X > M -> max(R,X,A) ; max(R,M,A)).
```

Взаємовідношення "is_sibling_of" перевіряє чи два вузли мають загального батька в дереві. Наприклад

```
?- h is_sibling_of  S.
S=g ;
S=i ;
no
```

Відзначте використання друкарської помилки X \==Y, який має успіх на всяк випадок X і Y - не cobound (зв'яжіть до того ж значення).
Взаємовідношення "is_same_level_as" перевіряє чи двох вузлів на тому ж рівні в дереві.

Предикат "depth" обчислює глибину вузла в дереві (скількох країв від кореня). Наприклад

```
?- t has_depth D.
D=4
```

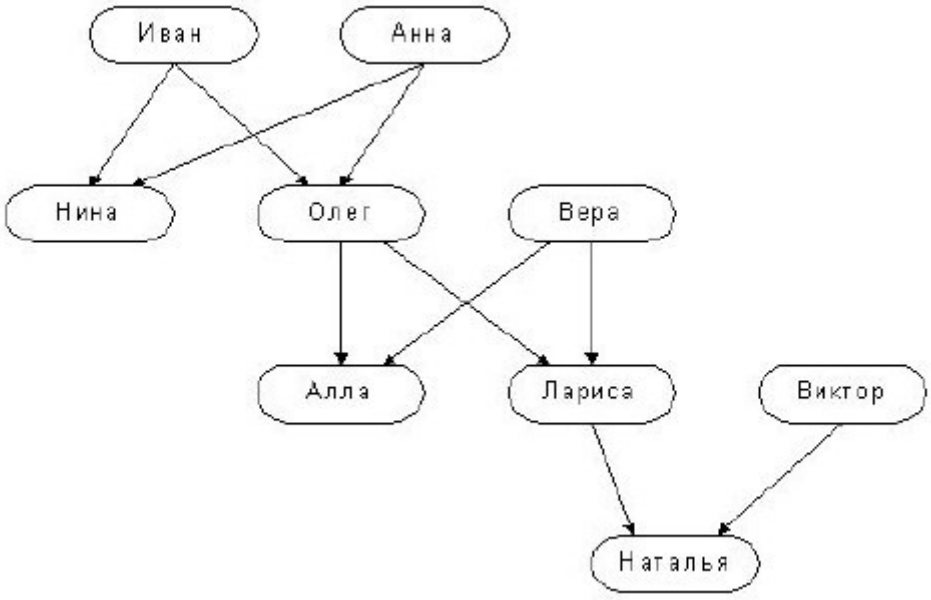
Лабораторна робота 5.

НАПИСАННЯ ПРОСТОЇ ПРОГРАМИ НА МОВІ GNU-PROLOG

Мета роботи: отримання практичних навичок складання, доопрацювання та виконання простої програми в системі програмування GNU-PROLOG.

Завдання:

1. Проінсталювати на власному комп'ютері систему програмування GNUPROLOG та систему редагування текстів программ SciTE (Science Text Editor).
2. Скласти на мові Prolog дерево родинних відношень, використовуючи предикат roditel з двома параметрами: ім'я одного з батьків та ім'я дитини.
3. Написати на мові Prolog та запустити наступні запити:
   •“Хто є і батьками, і має батьків”
   •“Хто не має дітей”
   Наприклад: для схеми родинних зв'язків



програма буде мати вигляд:

```
roditel(`иван`,`нина`).
roditel(`иван`,`олег`).
roditel(`анна`,`нина`).
roditel(`анна`,`олег`).
roditel(`олег`,`лариса`).
roditel(`олег`,`алла`).
roditel(`вера`,`алла`).
roditel(`вера`,`лариса`).
roditel(`лариса`,`наталья`).
roditel(`виктор`,`наталья`).
```

3. Склад звіту про виконання лабораторної роботи:

• Назва, мета та завдання лабораторної роботи

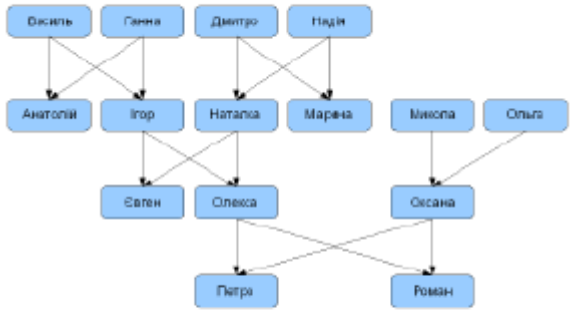
• Зміст індивідуального завдання

• Графічне зображення родинних зв'язків з предикатом

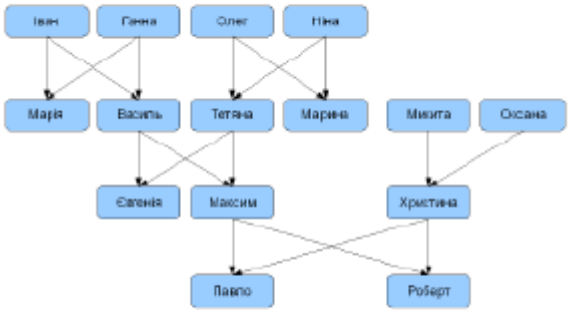
• Програма на мові Prolog

• Скріншоти виконання програми та запитів на завантажених правилах

Індивідуальні завдання до лабораторної роботи 1.

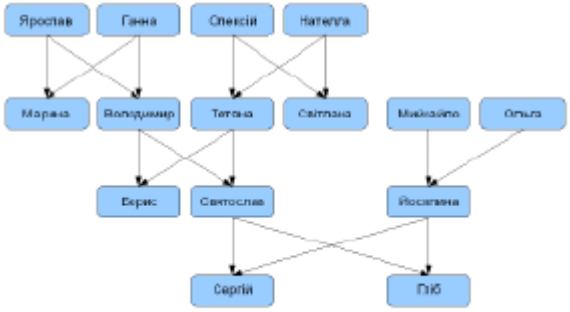


Варіант 1



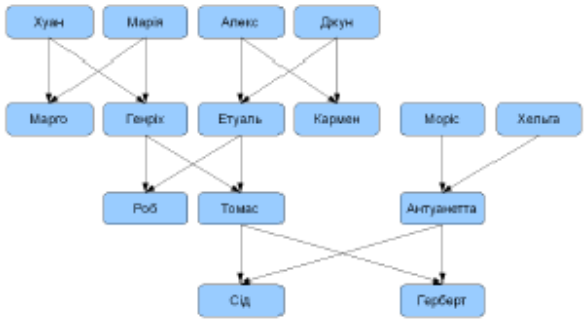
Варіант 2

Варіант 3

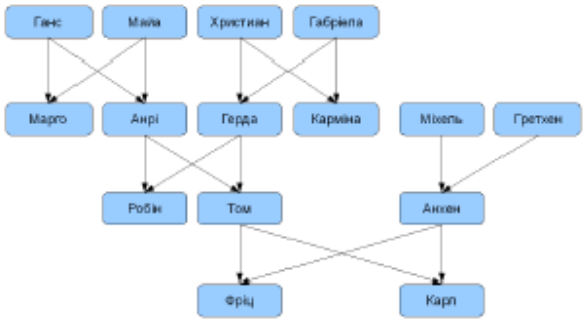


Варіант 4

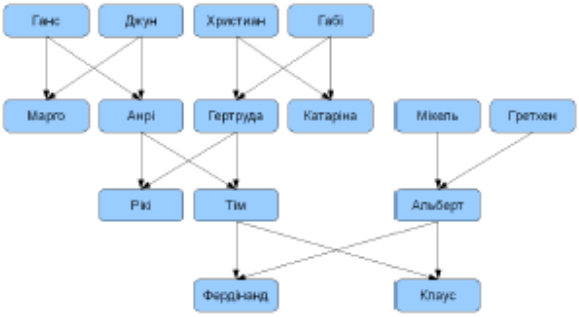
Варіант 5



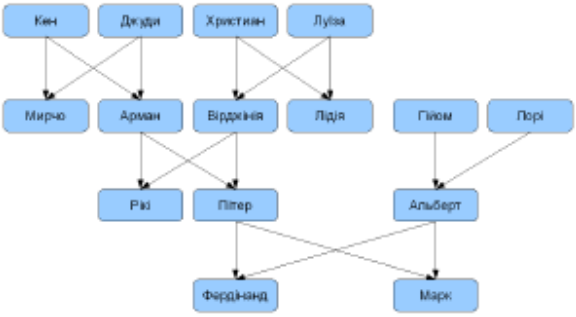
Варіант 6



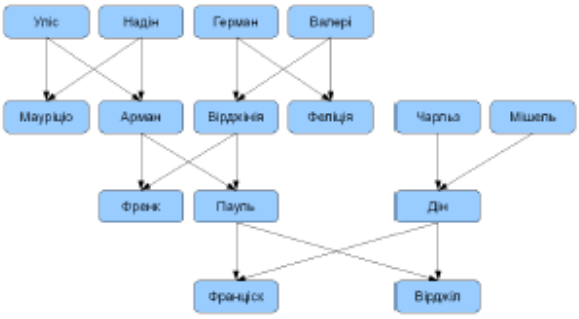
Варіант 7



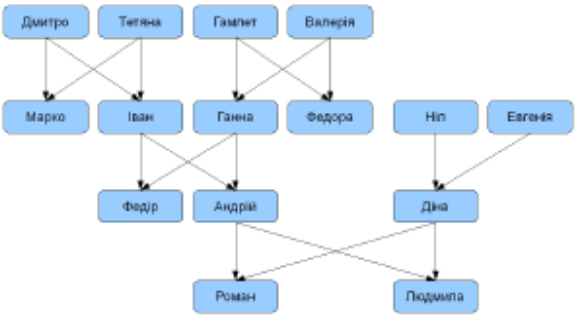
Варіант 8



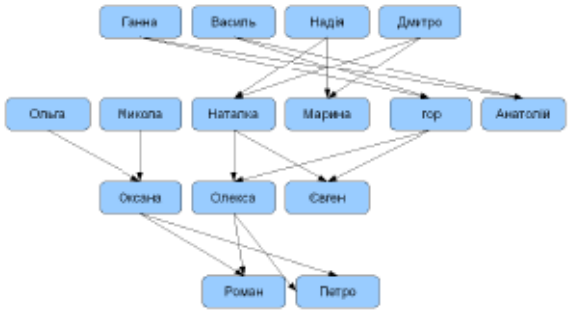
Варіант 9



Варіант 10

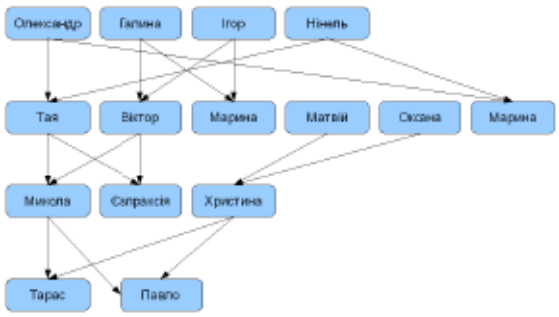


Варіант 11

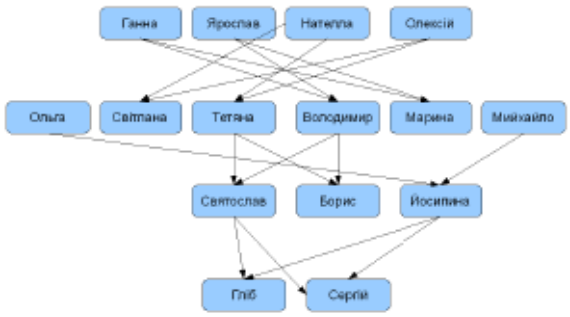


Варіант 12

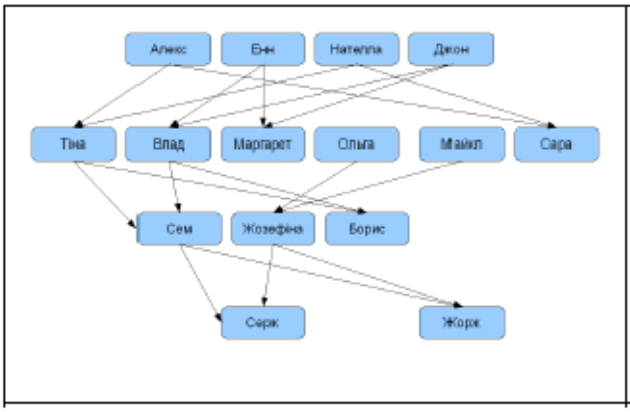
Варіант 13



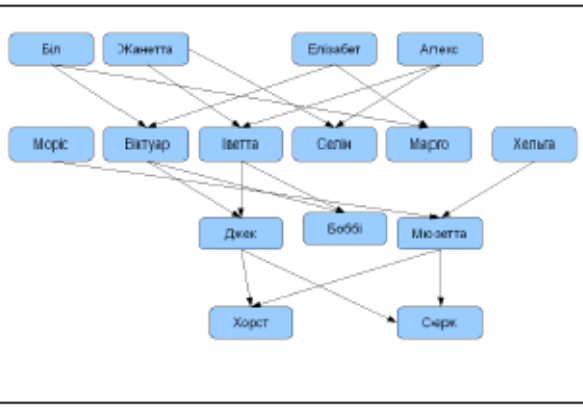
Варіант 14



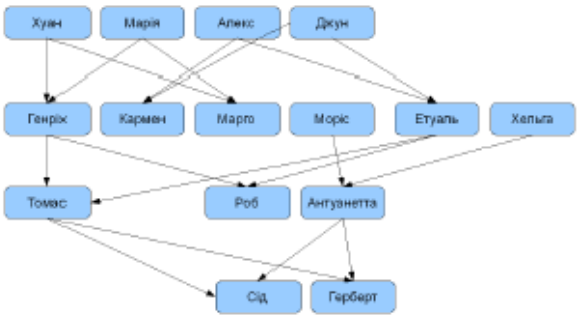
Варіант 15



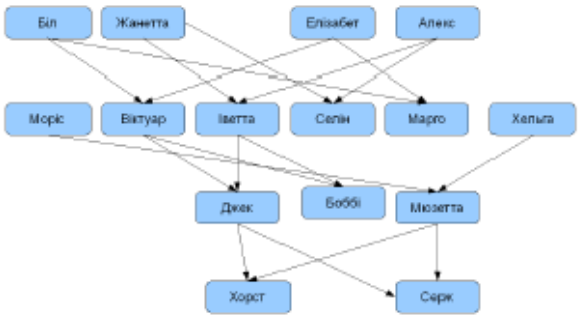
Варіант 16



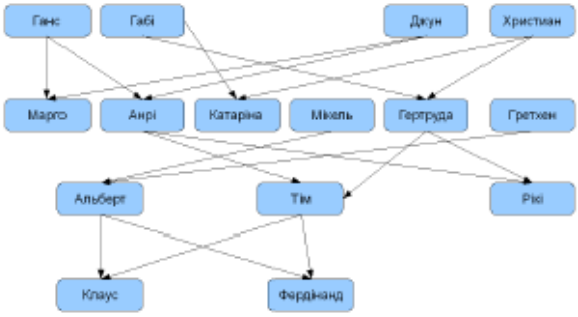
Варіант 17



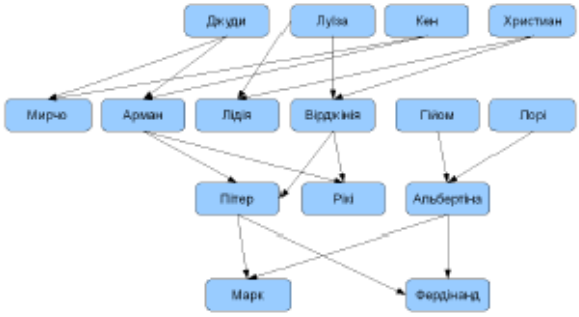
Варіант 18



Варіант 19

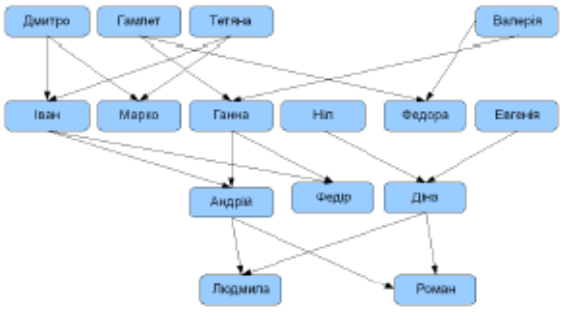


Варіант 20



Варіант 21

Варіант 22



Лабораторна робота 6.
ФОРМУВАННЯ ПРАВИЛ

Мета роботи: отримання практичних навичок складення правил та використання їх в програмі в системі програмування GNU-PROLOG.

Завдання:

1. Програму з лабораторної роботи 1 доповнити новими фактами, що дозволяють побудувати правила для визначення наступних цілей-предикатів:

• батько
• мати
• син
• дочка
• брат
• сестра
• дядько
• тітка
• дід
• баба
• онук
• онучка
• небіж
• небога

2. Склад звіту про виконання лабораторної роботи:

•Назва, мета та завдання лабораторної роботи
•Зміст індивідуального завдання
•Графічне зображення родинних зв'язків з відповідними предикатами
•Програма на мові Prolog
•Скріншоти виконання програми та запитів на завантажених правилах