

Міністерство освіти і науки України
Національний технічний університет України «КПІ ім. Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

З лабораторної роботи №5

Виконав студент

ІП-01 Смыслов Данил _____
(шифр, прізвище, ім'я, по батькові)

Перевірів

ас. Очеретяний О.К. _____
(прізвище, ім'я, по батькові)

Київ 2022

1. Завдання лабораторної роботи

Exercise 2.1 If a map has N regions, then estimate how many computations may have to be done in order to determine whether or not the coloring is in conflict. Argue using program clause trees.

Exercise 2.2.1 Using the first factorial program, show explicitly that there cannot possibly be a clause tree rooted at 'factorial(5,2)' having all true leaves.

Exercise 2.2.2 Draw a clause tree for the goal 'factorial(3,1,6)' having all true leaves, in a fashion similar to that done for factorial(3,6) previously. How do the two programs differ with regard to how they compute factorial? Also, trace the goal 'factorial(3,1,6)' using Prolog.

Exercise 2.3.1 Draw a program clause tree for the goal 'move(3,left,right,center)' show that it is a consequence of the program. How is this clause tree related to the substitution process explained above?

Exercise 2.3.2 Try the Prolog goal `?-move(3,left,right,left)`. What's wrong? Suggest a way to fix this and follow through to see that the fix works.

2. Розв'язання

Exercise 2.1

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

Якщо ми подивимось на функцію для перевірки конфлікту в розфарбуванні то ми можемо побачити, що для перевірки чи маємо ми конфлікт між 2 вершинами нам потрібно 3 операції. Всього у нас вершин n , тобто у найгіршому випадку, коли кожен регіон суміжний з усіма іншими, у нас буде кількість перевірок це k -сть комбінацій з n по 2, тобто $n(n-1)/2$. Оскільки ми маємо 3 операції для кожної перевірки, то складність буде $3n(n-1)/2$.

Для доведення давайте перевіримо конфлікт кольорів для 3 вершин та зробимо трасування для підрахунку операцій.

```

adjacent(1,2).      adjacent(2,1).
adjacent(1,3).      adjacent(3,1).
adjacent(3,2).      adjacent(2,3).

```

```

color(1,red,a).
color(2,blue,a).
color(3,green,a).

```

```

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

```

Трасування:

```

trace, conflict(a)

Call: conflict(a)
Call: adjacent(_666,_668)
Exit: adjacent(1,2)
Call: color(1,_670,a)
Exit: color(1,red,a)
Call: color(2,red,a)
Fail: color(2,red,a)
Redo: adjacent(_666,_668)
Exit: adjacent(2,1)
Call: color(2,_670,a)
Exit: color(2,blue,a)
Call: color(1,blue,a)
Fail: color(1,blue,a)
Redo: adjacent(_666,_668)
Exit: adjacent(1,3)
Call: color(1,_670,a)
Exit: color(1,red,a)
Call: color(3,red,a)
Fail: color(3,red,a)
Redo: adjacent(_666,_668)
Exit: adjacent(3,1)
Call: color(3,_670,a)
Exit: color(3,green,a)
Call: color(1,green,a)
Fail: color(1,green,a)
Redo: adjacent(_666,_668)
Exit: adjacent(3,2)
Call: color(3,_670,a)
Exit: color(3,green,a)
Call: color(2,green,a)
Fail: color(2,green,a)
Redo: adjacent(_666,_668)
Exit: adjacent(2,3)
Call: color(2,_670,a)
Exit: color(2,blue,a)
Call: color(3,blue,a)
Fail: color(3,blue,a)
Fail: conflict(a)

false

```


Як бачимо, к-сть операцій 18. Це число пояснюється тим, що Prolog буде перевіряти сумісність кольорів спочатку для (1;2), а потім для (2;1). Тобто для кожної вершини по 2 рази, тому можна домножити формулу на 2 і отримаємо $2n(n-1)$. Підставивши в формулу, отримаємо $3*3*2 = 18$, що відповідає дійсності.

Exercise 2.2.1

```
factorial(0,1).
```

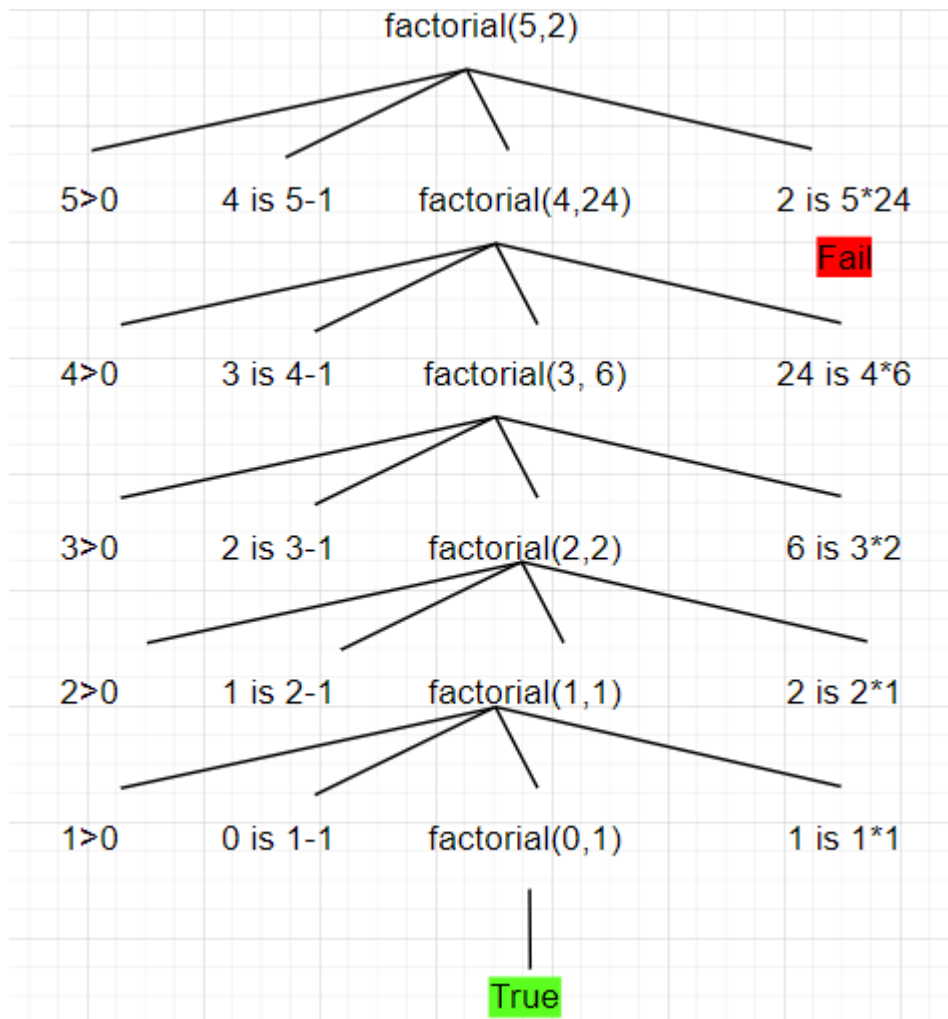
```
factorial(N,F) :-  
    N>0,  
    N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.
```

Виконаємо трасування factorial(5,2)

 trace, factorial(5,2)

```
Call: factorial(5,2)  
Call: 5>0  
Exit: 5>0  
Call: _630 is 5 + -1  
Exit: 4 is 5 + -1  
Call: factorial(4,_632)  
Call: 4>0  
Exit: 4>0  
Call: _640 is 4 + -1  
Exit: 3 is 4 + -1  
Call: factorial(3,_642)  
Call: 3>0  
Exit: 3>0  
Call: _650 is 3 + -1  
Exit: 2 is 3 + -1  
Call: factorial(2,_652)  
Call: 2>0  
Exit: 2>0  
Call: _660 is 2 + -1  
Exit: 1 is 2 + -1  
Call: factorial(1,_662)  
Call: 1>0  
Exit: 1>0  
Call: _670 is 1 + -1  
Exit: 0 is 1 + -1  
Call: factorial(0,_672)  
Exit: factorial(0,1)  
Call: _662 is 1*1  
Exit: 1 is 1*1  
Exit: factorial(1,1)  
Call: _634 is 2*1  
Exit: 2 is 2*1  
Exit: factorial(2,2)  
Call: _630 is 3*2  
Exit: 6 is 3*2  
Exit: factorial(3,6)  
Call: _626 is 4*6  
Exit: 24 is 4*6  
Exit: factorial(4,24)  
Call: 2 is 5*24  
Fail: 2 is 5*24
```

Програма видає результат «Fail» при перевірці $2 \text{ is } 5 \cdot 24$, тому весь виклик цієї функції повертає False. Давайте побудуємо дерево, що більш наглядно зрозуміти це.




Exercise 2.2.2

`factorial(0,F,F).`

```

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

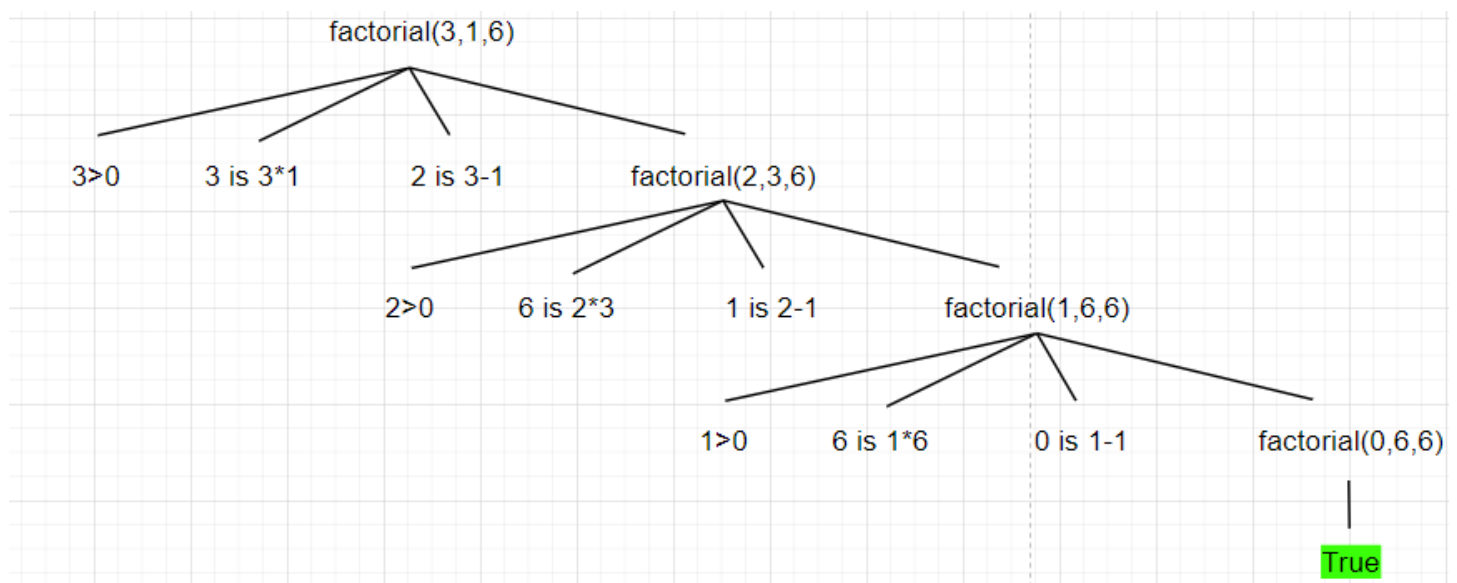
Давайте виконаємо трасування.

 `trace, factorial(3,1,6)`

```
Call: factorial(3,1,6)
Call: 3>0
Exit: 3>0
Call: _630 is 3*1
Exit: 3 is 3*1
Call: _644 is 3 + -1
Exit: 2 is 3 + -1
Call: factorial(2,3,6)
Call: 2>0
Exit: 2>0
Call: _646 is 2*3
Exit: 6 is 2*3
Call: _660 is 2 + -1
Exit: 1 is 2 + -1
Call: factorial(1,6,6)
Call: 1>0
Exit: 1>0
Call: _662 is 1*6
Exit: 6 is 1*6
Call: _676 is 1 + -1
Exit: 0 is 1 + -1
Call: factorial(0,6,6)
Exit: factorial(0,6,6)
Exit: factorial(1,6,6)
Exit: factorial(2,3,6)
Exit: factorial(3,1,6)
```

true


Тепер побудуємо дерево.



Різниця між цими двома програмами в тому, що другий варіант використовує не звичайну, а хвостову рекурсію. Він має певний «акумулятор», що зберігає в собі поточне значення функції. Перша програма порівнювала значення факторіала та задане значення зверху, біля самого кореня. А друга програма – навпаки перевіряє в самому низу, на найбільшій глибині і вже, відповідно, повертає результат. В нашому випадку – True.

Exercise 2.3.1

```
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).
```

 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
true

Тепер давайте виконаємо трасування.

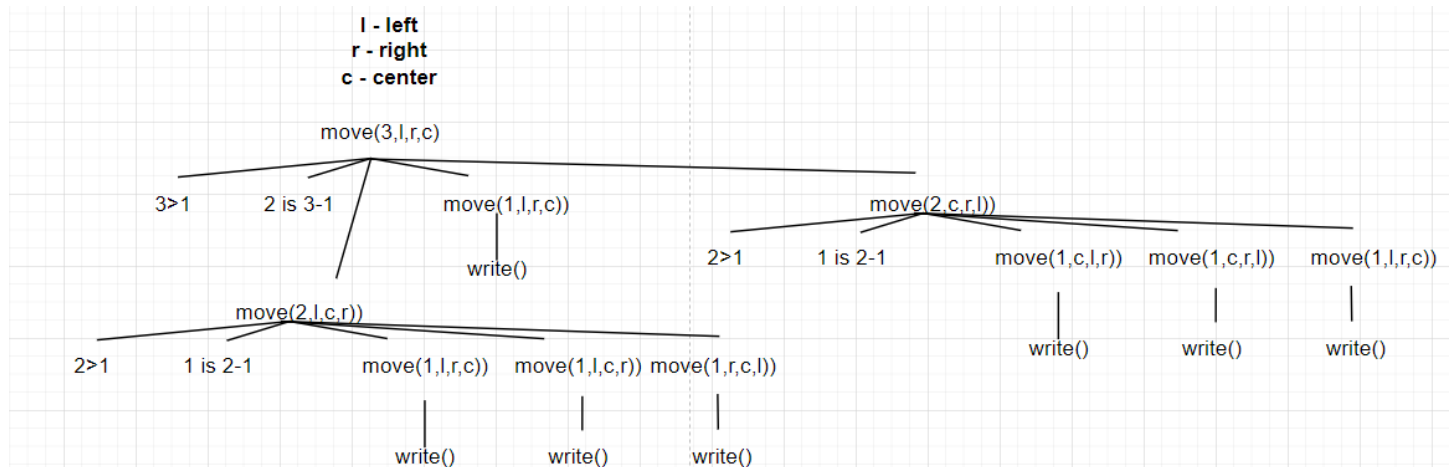
```

trace.move(3,left,right,center)

Call: move(3,left,right,center)
Call: 3>1
Exit: 3>1
Call: _664 /s 3 + -1
Exit: 2 is 3 + -1
Call: move(2,left,center,right)
Call: 2>1
Exit: 2>1
Call: _672 is 2 + -1
Exit: 1 is 2 + -1
Call: move(1,left,right,center)
Call: write("Move top disk from ")
Move top disk from
Exit: write("Move top disk from ")
Call: write(left)
left
Exit: write(left)
Call: write(" to ")
to
Exit: write(" to ")
Call: write(right)
right
Exit: write(right)
Call: nl
Exit: nl
Exit: move(1,left,right,center)
Call: move(1,left,center,_4764)
Call: write("Move top disk from ")
Move top disk from
Exit: write("Move top disk from ")
Call: write(left)
left
Exit: write(left)
Call: write(" to ")
to
Exit: write(" to ")
Call: write(right)
right
Exit: write(right)
Call: nl
Exit: nl
Exit: move(1,center,left,right)
Call: move(1,center,right,_20596)
Call: write("Move top disk from ")
Move top disk from
Exit: write("Move top disk from ")
Call: write(center)
center
Exit: write(center)
Call: write(" to ")
to
Exit: write(" to ")
Call: write(right)
right
Exit: write(right)
Call: nl
Exit: nl
Exit: move(1,center,right,_24552)
Call: move(1,left,right,center)
Call: write("Move top disk from ")
Move top disk from
Exit: write("Move top disk from ")
Call: write(left)
left
Exit: write(left)
Call: write(" to ")
to
Exit: write(" to ")
Call: write(right)
right
Exit: write(right)
Call: nl
Exit: nl
Exit: move(1,left,right,center)
Exit: move(2,center,right,left)
Exit: move(3,left,right,center)
true

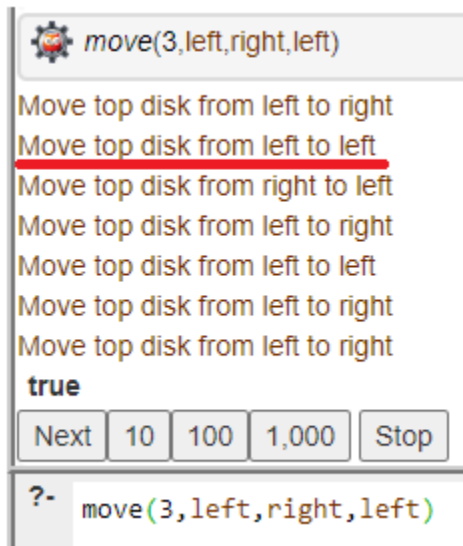
```


Тепер давайте побудуємо частину дерева, використовуючи певні скорочення для позначення. “write()” відповідає за виведення даних в консоль.



Отже, видно, що маємо програмну послідовність. Побудоване дерево збігається з процесом підстановки, описаним вище.

Exercise 2.3.2



Я підкреслив рядок, з якого ми бачимо, що програма пропонує перемістити диск з лівої осі на ліву, через це є певна неоднозначність. Давайте додамо перевірку осей, щоб виправити цю проблему.

```

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

```

Тепер давайте ще раз спробуємо виконати `move(3,left,right,left)`.

```

⚙️ move(3,left,right,left)
false

```

Отже, виправлення спрацювало.