Logic Programming

Section #3

Eman Safwat

24/10/2009

Assignment Solution:

Use Recursion to get the power function, which takes 3 arguments:

X: Base

Y: Power

Z: Final Result

X ^ 0 =1 (Base Step)

X ^ Y = X * (X ^ Y-1) (Recursive Step)

Solution:

Predicates

power (X, 0, 1) :- !.

power (X, Y, Z) :-

Y1 = Y – 1,

power (X, Y1, Z1),

Z = Z1 * X.

هوا الحل ده زي حل مسألة ال Factorial بتاعت السيكشن اللى فات (آخر مسألة)، فروح إفهمها كويس .. هتلاقى ديه سهلة إن شاء الله.

There are two types of Recursion:

Tail Recursion – Non Tail Recursion.

اللي درسناه في المثالين ده ال Non-Tail دلوقتي هنحل مسألة ال Power تاني بال Tail Recursion.

By C#:

```
int power (int x, int y)
{
        int p = 1;
        for (int i = 0; i < y; i++)
                p = p * x;
        return p;
}
```

By Tail Recursion:

Predicates

power_tail (integer, integer, integer, integer).

Clauses

power_tail (_, 0, P, P):- !.

power_tail (X, Y, T, Z) :-

NT = T * X,

NY = Y -1,

power_tail (X, NY, NT, Z).

Goal

power_tail (5, 3, 1, What).

What = 125

نشرح بقى:

power_tail (integer, integer, integer, integer).

ال Predicate **دي بتاخد** 4 arguments **أول اتنين** inputs **وواحد** Temporary Value (input) **وآخر واحد** output.

power_tail (X, Y, T, Z) :-

NT = T * X,

NY = Y -1,

power_tail (X, NY, NT, Z).

هنشرح ال Rule **دي الأول، ال** X **بتاخد ال** Base number **وال** Y **ال** Power**، وال** T **ال** Temporary Value **وده بيشيل ال** Value **بتاعت ال** X **لحد ما ال** Recursion **يخلص (**Y = 0**) وال** Z **دي ال** output.

اللي بيحصل إنه ال X, Y, T **بيدخلوا** Binded**، وال** NT **ده** New Temp **بيشيل قيمة ال** T, X **مضروبين في بعض.. وال** NY **نفس الحكاية بيشيل قيمة ال** Y – 1. **وبعدين بنده عال** Function **تاني بال** Values **الجديدة وهكذا ال** Y **بتقل لحد ماتوصل لل** Zero **وتخش في ال** Base Step:

power_tail (_, 0, P, P):- !.

أول ما ال Y بتوصل Zero بيدخل في ال Rule دي ويرجع ال Z باخر قيمة كانت لل T (اللي هيا هنا اسمها P).

عملنا ال X ب underscore عشان مش مهم ال Value بتاعته هتبقى بكام.

Goal

power_tail (5, 3, 1, What).

Tracing:

X = 5, Y = 3, T = 1

NT = 5 * 1 = 5

NY = 3 – 1 = 2

power_tail (5, 2, 5, Z)

X = 5, Y = 2, T = 5

NT = 5 * 5 = 25

NY = 1

power_tail (5, 1, 25, Z)

X = 5, Y = 1, T = 25

NT = 5 * 25 = 125

NY = 0

power_tail (5, 0, 125, Z)

P = 125 (Third argument - binded)

P = 125 (Fourth argument)

Z =125

Backtrack

قيمة ال Z هيتعملها Assignment لل Z في ال Level اللي وبعدين ال Level اللي قبله لحد مانوصل لأول مرة اتندهت فيها ال Function وتبقى ال What = 125.

إيه الفرق بقى بين ال Tail Recursion وال Non-Tail Recursion؟ وليه أصلا فيه Tail؟ (ما احنا كنا شاغلين بال Non-Tail كويس :p)

في ال Tail(اللي احنا لسة حالين بيها) ال Prolog مش بتعرّف New Variables لما تنادي على ال Function كل مرة في ال Recursion لأ، هيا بتعمل Overriding لل Variables بس، فكده بتوفر Memory..

مش زي في ال Non-Tail كل مايندہ Function ويروح ل Level تاني بيعرّف New Variables فبياخد Memory أكتر، عشان كل Level معتمد على ال Level اللي قبله في قيمة ال Variables بتاعته (ال Tail لأ).

So Tail Recursion is more efficient than Non-Tail Recursion.

Lists

Lists are powerful data structures for holding and manipulating groups of things. In Prolog, a list is simply a collection of terms. The terms can be any Prolog data types, including structures and other lists. Syntactically, a list is denoted by square brackets with the terms separated by commas.

يعني ال List دي عبارة عن Type جديد بعرّفه عشان استخدمه في البرنامج (زي ال Structs).

Defining Lists:

بس ال Visual اللي احنا بنستعمله مفهوش Lists جاهزة (Bulit-in) فبعمل Region جديدة اسمها Domains بعرّف فيها كل ال Lists اللي أنا عايزها، بتكتب قبل ال Predicates.

Domains

ilist = integer *

rlist = real *

كده أنا عرّفت ilist دي List of integers و rlist دي List of Real Numbers.

iilist = ilist *

دي List of Lists، دايما ال List بتتعرّف Dynamic (زي X = new int في ال C++).

Matching Lists:

L = [1, 2, 3, 4, 5]

The brackets are the beginning and the end of the list, and the commas separate the various elements. Here all the elements are atoms, but a list can contain all sorts of elements, and different types of element can occur in the same list

دي كده List عادية بس ممكن تتكتب كده:

Most prolog code will not define a list explicitly, like these examples, but rather handle lists of any length, with many possible elements. To handle lists without knowing what's inside them or how long they are, you use the bar notation:

L = [H | T]

Head represents the leftmost element of the list and the variable **Tail** represents the rest of the list represented **as another list**. A head can be anything, from a predicate to another list, but the **tail** is always **another list**.

يبقى الشكل ده لل List **معناه إنه** At Least **فيها** One Element **اللي هو ال** Head**، ال** Tail **ممكن تبقى** empty List [ ] **بس ال** Head **مينفعش تبقى** empty .

لو ال List من نوع integer **يبقى ال** Head **وال** Tail **من نوع** integer **برضو وهكذا..**

L = [X]

معناها إنه ال List **لازم يكون فيها** element **1 على الاقل غير كده مش هيعمل** Matching.

[1, 2, 3, 4] = [H | T]

هيعمل Matching **وبما إن ال** H **وال** T **الاتنين** Free Variables **هيعمل** Assignment:

H = 1, T = [2, 3, 4]

[1, 2, 3, 4] = [H1, H2 | T]

H1 = 1, H2 = 2, T = [3, 4]

[5, 6] = [H1, H2 | T]

H1 = 5, H2 = 6, T = [ ]

[5, 6] = [H1, H2, T]

مش هتعمل Matching **عشان ال** List **هنا فيها** 3 elements**، مش** Head **و** Tail.

Note:

الحاجات اللي بال English **دي أنا جايبها من** Wiki **أو مواقع تانية، عشان محدش يفتكر إن دكتور. عمرو ادي سيكشني حاجات زيادة لا سمح الله!**

Example:

Domains

ilist = integer *

Predicates

count (ilist, integer).

Clauses

count ([ ], 0). (Base step)

count ([H | T], C) :-

count (T, CT), C = CT +1.

Goal

Count ([7, 6, 8, 11], What).

What = 4

Count بتعمل Counting للـ elements بتاعت الـ List وترجع عددهم.

أول مرة بتقسّم الـ List لـ Head و Tail وبعدين تنده على نفسها تاني وتقسم الـ Tail لـ Head و Tail تاني وهكذا لحد مال List تبقى empty وساعتها هتدخل في الـ Base step وترجع 0. وبعدين Backtracking وتزود 1 في كل مرة بترجع فيها لحد ماترجع لأول Function اتندهت وتزود فيها آخر element اللي هوا الـ Head الاصلية.

ليه مثلا محطيتش ! في آخر الـ Base step زي ما احنا متعودين في الـ Non-Tail Recursion؟

عشان هوا هنا لما يوصل للـ Base step هيدخل فيها، بعدين هيجي يدخل في الـ Rule التانية اللي فيها الـ Head والـ Tail مش هيدخل فيها عشان مفيش Matching حيحصل:

[H | T] ≠ [ ]

Note:

لو عاوزة اخليها Sum مش Count هاغير حاجة واحدة اللي هيا:

C = CT + H

هيجمع كل Value للـ Head ويرجع مجموعهم.

Example:

get_first ([1, 2, 3], What).

What = 1

How??

get_first ([H | T], H).

هنا بيرجع ال Head على طول اللي هيا First element.

get_last ([5, 6, 7], What).

What = 7

How??

get_last ([H], H).

get_last ([H |T], X) :-

get_last (T, X).

كل مرة بقسّم ال List ل Head و Tail وكل مرة اديله ال Tail لحد مايوصل إن Tail مفهوش غير element واحد هيخش في ال Base step وهيبقى هوا آخر element في ال List.

لو خليتها:

Goal

get_last ([ ], What).

What = no solution.

Note:

احنا طبعا ال Recursion اللي استعملناه هنا Non-Tail، ولما باجي أفكر في مسألة احلها بيه إزاي؟ دايما أبدأ بال Base step اللي هيا بتكون أبسط Case لل Condition بتاعي ولما اجيبلها ال Recursive step هتكون الحاجة اللي بكررها كتير عشان اوصل لل Base.

---

add_to_head (3, [1, 2], What).

What = [3, 2, 1]

How??

Clauses

add_to_head (H, L, [H | L]).

سهلة ولا إيه؟ ;p

add_to_tail (3, [1, 2], What)

What = [1, 2, 3]

How??

add_to_tail (X, [ ], [X]).

add_to_tail (X, [H | T], [H | NT]) :-

add_to_tail (X, T, NT).

هنا أنا عاوزة اضيف element لاخر ال List، فال Base step عندي إنه ال List تبقى فاضية فضيف فيها وأرجع اضيف باقي ال elements قبلها.. ال Recursive step كل مرة بأبعت ال Tail عشان يتقسم ل Head و Tail ويرجعلي ال Tail الجديد (NT) لحد ماوصل لل Base step واضيف ال X وبعدين Backtracking وترجع ال X في NT و Backtracking لحد مرجع لأول مرة اتندهت فيها ال **Function**

فأرجع ال List وفيها ال X متضافة.

لو مفهمتوش ال Tracing أوي ال Conc هتفهمكم إن شاء الله.

لو عملت كده:

add_to_tail (X, [H | T], NT) :-

add_to_tail (X, T, NT).

هيرجع ال New Tail بس في الآخر من غير أول Head.

---

conc ([3, 4, 5], [11, 12], What).

What = [3, 4, 5, 11, 12]

How??

Clauses

conc ([ ], L, L).

conc ([H | T], L, [H | TL]) :-

conc (T, L, TL).

Goal

conc ([3, 4], [5, 6], What).

Tracing:

conc ([3, 4], [5, 6], What).

حيخش في أول Rule مش هات Match هيروح عالتانية:

H = 3, T = [4], L = [5, 6], What = [3 | TL] → Level 1

هتنده Conc اللي جواها:

conc ([4], [5, 6], TL)

H = 4, T = [ ], L = [5, 6], TL = [4 | TL2] → Level 2

كده ال T بقت empty يبقى وصلنا لل Base step:

conc ([ ], [5, 6], TL2)

[ ] = [ ] (Matched)

L = [5, 6]

TL2 = [5, 6] → Level 3

Backtarcking

Level 2:

TL = [4, 5, 6]

Backtracking

Level 1:

What = [3, 4, 5, 6]

لو اداني مثلا:

Goal

conc ([3, 4], [1, 2], [3, 4, 1, 2]).

yes

conc ([3, 4], What, [3, 4, 1, 2]).

What = [1, 2]

conc (What, [2], [3, 4, 1, 2]).

What = [3, 4, 1]

conc (L1, L2, [3, 4, 1, 2])

L1=[], L2=[3,4,1,2]

L1=[3], L2=[4,1,2]

L1=[3,4], L2=[1,2]

L1=[3,4,1], L2=[2]

L1=[3,4,1,2], L2 = []

5 Solutions

كل ال Combinations بين L1 و L2.

conc (L1, [1], [3, 4, 1, 2]).

No Solution

يبقى Conc بتقسم أو بتجمع Lists 2 على بعض على حسب ال Inputs.

Example:

Clauses

```
even_sum ([ ], 0).

even_sum ([H | T], S) :-
        H mod 2 = 0,
        even_sum (T, ST),
        S = ST + H.
```

H mod 2 = 0 دي هتعمل Comparison (عشان ال H و 0 دول Binded).

كل مرة لو ال Condition اتحقق هيبقى الرقم Even وهجمع ال Value بتاعته.

بس لو سيبت ال Rules كده بس هيجي يلاقي رقم Odd ال Condition مش هيتحقق (H mod 2 = 0) فهيقف ومايكملش على باقي الأرقام من ال List.

فهنعمل Rule تانية:

```
even_sum ([H | T], S) :-
                H mod 2 = 1,
                even_sum (T, S).
```

كده لو أول Rule مااتحققتش هيدخل في التانية وينفذ ال Rule بس الفرق إن هنا مش هيجمع ال Value بتاعت ال Odd.

أو ممكن نعملها كده:

Clauses

```
even_sum ([ ], 0).
even_sum ([H | T], S) :-
                H mod 2 = 0, !.
                even_sum (T, ST),
                S = ST + H.
even_sum ([H | T], S) :-
                even_sum (T, S).
```

كده كل مايجي يعمل Backtracking عشان يروح even_sum التانية يقف ومايكملش فبالتالي مش حيخش ال Rule التانية إلا أما يكون Odd فمش محتاجة ال Condition ← (H mod 2 = 1).

Assignment:

```
get_at ([3, 4, 5, 6], 2, What)
What = 5
```

ترجعلي ال رقم بدلالة ال Index بتاعه (بيبدأ من Zero).

```
even_list ([1, 2, 3, 4, 5], What)
What = [2, 4]
```

ترجع ال even List من كل ال elements الموجودة.

See Also:

-An Introduction to Prolog