Dynamic Fact & Meta Predicates

عملي مشترك
محتوى مجاني غير مخصص للبيع التجاري

04/06/2023

RB Informatics;

مبادئ الذكاء الصنعي

تعاملنا سابقاً مع الحقائق ضمن البرولوغ حيث نقوم بكتابتها ضمن ملف وقراءة هذا الملف عند تشغيل واجهة الاستعلامات، وعند التعديل على هذه الحقائق ضمن الملف الموجود فيه يجب علينا تحديث القراءة مجدداً ضمن واجهة الاستعلامات وهذه الحقائق تدعى بالحقائق الثابتة، في حين أردنا إضافة حقيقة ما أثناء عمل البرنامج سوف يتوجب علينا التعامل مع الحقائق الديناميكية.

الحقائق الديناميكية:

هي حقائق يتم تعريفها خلال تشغيل البرنامج "Run Time" يمكن إضافتها في أي وقت، كما تسمح الحقائق الديناميكية بتعديل قاعدة البيانات الخاصة بالبرنامج وإجراء تغيرات فيها بناء على المتغيرات الداخلية للبرنامج.

- التوابع التي سنستخدمها لتخزين الحقائق الديناميكية:
  - assert (fact_name(value)): يخزن الحقيقة في النهاية.
  - asserta (fact_name(value)): يخزن الحقيقة في البداية.
  - retract (fact_name(value)): يحذف الحقيقة.
  - retractall (fact_name(_)): يحذف جميع الحقائق.

لتعريف حقيقة ما على أنها ديناميكية في ملف الـ knowledge base نكتبها بالشكل التالي:

```
:- dynamic fact_name/n
```

حيث: fact_name: اسم القاعدة، n: عدد البارامترات.

ملاحظة:

يجب أن تكون القيم التي نقوم بتخزينها معلومة، أي لا تستطيع كتابة assert (fact(X,Y)) وقيم X,Y غير معلومة حيث يجب علينا بداية إسناد قيم للمتحولين ومن ثم تخزين هذه الحقيقة.

مثال: اكتب قاعدة فيبوناتشي حيث يتم تخزين قيم فيبوناتشي أثناء الاستدعاء.

لقد تعرفنا سابقاً على قاعدة فيبوناتشي وكانت بالشكل:

```
fib(0,0).
fib(1,1).
fib(X,Res) :- X>1, X1 is X-1, X2 is X-2, fib(X1,Res1), fib(X2, Res2), Res is Res1 + Res2.
```

- من أجل تخزين القيم أثناء الاستدعاء:
  - نعرف القاعدة على أنها ديناميكية.
  - في نهاية القاعدة العامة نخزن الحقيقة ديناميكياً في البداية من خلال التابع asserta(fib(X,Res))، لأن التخزين في النهاية لن يفيد في سرعة الوصول للحقيقة المخزنة، فعند السؤال عن قيمة ما سيبدأ بالمطابقة من البداية فإن قابل القيمة مخزنة يرجعها فوراً، وإلا سيتابع ليصل إلى القاعدة فيعيد حسابها.

```
:- dynamic fib/2.
fib(0,0).          fib(1,1).
fib(X,Res):-  X>1, X1 is X-1, X2 is X-2, fib(X1,Res1), fib(X2, Res2), Res is Res1 + Res2,
              asserta(fib(X,Res)).
```

ونقوم بالاستدعاء التالي:

```
?- fib(10,Res).          →      Res = 55.
```

هنا أصبحت القيمة fib(10,55) مخزنة، فعند السؤال مرة ثانية عن القيمة ذاتها لن يعيد حسابها بل ستكون محفوظة فيرجعها فوراً، فأصبح الاستدعاء يستغرق أجزاء من الثواني بدل من عدة ثواني.

Meta Predicates: find all Vs bag of Vs set of

ليكن لدينا الحقائق التالية:

```
parent( marry, john).
parent( noah, john).
parent( john, bob).
parent( bob, sophia).
parent( sophia, rose).
ancestor( X, Y):- parent( X, Y).
ancestor( X, Y):- parent( X, Z), ancestor( Z, Y).
```

عند القيام بالـ query التالي:

```
?- parent( X, john).
     X= marry;   X=noah.
```

وهنا كما تكلمنا سابقاً أنه سيظهر أول إجابة صحيحة وعند استخدام (;) يظهر ثاني إجابة صحيحة.

ولكن ماذا لو أردت أن يظهر لي جميع النتائج الصحيحة ضمن سلسلة؟!

هنا نستخدم "find all" تقوم هذه الدالة بجمع جميع الحلول الممكنة وتخزينها في السلسلة.

تأخذ ثلاث بارامترات findall(Object, Goal, L):

- Object: هو الهدف الذي يتم البحث عنه (المتحول).
- Goal: الاستفسار الذي نقوم به.
- L: القائمة التي يتم تخزين الحلول فيها.

```
?- findall( X, parent( X, john), Res ).
        Res = [marry, noah].
```

لو أردنا جميع أسلاف rose:

```
?- findall( X, ancestor( X,  rose), Res ).
        Res = [sophia, marry, noah, john, bob].
```

لو أردنا جميع أبناء وأحفاد rose:

```
?- findall( X, ancestor( rose, X), Res ).
        Res = [ ].
```

انتبه find all عند عدم وجود جواب يرجع سلسلة فارغة.

لو أردنا الاستفسار عن متحولين معاً، مثلاً أريده أن يرجع لي كل شخص مع أسلافه، إذا أردنا تنفيذها باستخدام find all:

```
?- findall( (X, Y), ancestor(X,Y), Res ).
        Res = [(marry, john), (noah, john), (john, bob), (bob, sophia), (sophia, rose),
               (marry, bob), (marry, sophia), (marry, rose), ...].
```

لاحظ أن الترتيب هكذا صعب التعامل معه نوعاً ما، كما أنه عشوائي، لذلك يوجد تعليمة bag of تقوم بتجميع جميع النتائج كسلاسل، مثلاً من أجل X تساوي john تضع جميع قيم Y في سلسلة، ثم من أجل جميع قيم X تساوي bob تضع قيم Y في سلسلة وهكذا، وشكل التعليمة وبارامتراتها مطابق تماماً لـ find all.

فإذا أردنا تكرار الاستفسار السابق باستخدام bag of:

```
?- bagof(Y, ancestor(X,Y), Res ).
        X = bob,        Res = [sophia, rose] ;
        X = john,       Res = [bob, sophia, rose] ;
        X = marry,      Res = [john, bob, sophia, rose] ;
        X = noah,       Res = [john, bob, sophia, rose] ;
        X = sophia,     Res = [rose].
```

لاحظ بعد كل نتيجة نستخدم فاصلة منقوطة لإظهار النتيجة التالية،

ماذا لو أردنا إرجاع سلسلة واحدة تحوي جميع هذه السلاسل؟

بالتأكيد سنستخدم find all، لكن هنا المتحول الذي أريد جمع نتائجه هو ناتج عملية الـ bag of أي متحول الـ Res، والاستفسار الذي أريد القيام به ضمن الـ find all هو تعليمة الـ bag of، وسنخزن الناتج في Res1 مثلاً:

```
?- findall( L, bagof(Y, ancestor(X,Y), L ), Res).
        Res = [[sophia, rose], [bob, sophia, rose], [john, bob, sophia, rose],
               [john, bob, sophia, rose], [rose]].
```

لاحظ أن الناتج هو عبارة عن مجموع السلاسل السابقة.

يمكننا جعل تعليمة bag of تهمل قيمة متحول ما وترجع النتائج دون فرزهم حسب ذلك المتحول وذلك من خلال كتابة اسم المتحول بعده علامة ^ قبل الاستفسار في الـ bag of.

```
-? bagof(Y, X^ancestor(X,Y), Res ).
        Res = [john, john, bob, sophia, rose, bob, sophia, rose, bob, ...].
```

لاحظ أنها أصبحت مثل الـ find all، حتى أننا لو قمنا بذات التعليمة باستخدام find all لأرجعت نفس الناتج:

```
?- findall(Y, ancestor(X,Y), Res ).
        Res = [john, john, bob, sophia, rose, bob, sophia, rose, bob, ...].
```

لنقم بالاستفسار التالي:

```
?- bagof( Y, ancestor( rose, Y), Res ).  →   false.
```

انتبه bag of عند عدم وجود جواب يرجع false.

هنالك تعليمة ثالثة set of مطابقة تماماً لـ bag of تختلف عنها فقط بأنها ترتب سلسلة النتيجة وتحذف التكرارات.

```
?- bagof(Y, ancestor(X,Y), Res ).
        X = bob,        Res = [sophia, rose] ;
        X = john,       Res = [bob, sophia, rose] ;
        X = marry,      Res = [john, bob, sophia, rose] ;
        X = noah,       Res = [john, bob, sophia, rose] ;
        X = sophia,     Res = [rose].
```

```
?- setof(Y, ancestor(X,Y), Res ).
        X = bob,        Res = [rose, sophia] ;
        X = john,       Res = [bob, rose, sophia] ;
        X = marry,      Res = [bob, john, rose, sophia] ;
        X = noah,       Res = [bob, john, rose, sophia] ;
        X = sophia,     Res = [rose].
```

لنقم بالاستفسار التالي:

```
-?  setof(Y, ancestor(rose,Y), Res ).   →   false.
```

انتبه set of عند عدم وجود جواب يرجع false.

تمرين (1): نريد تابع يعيد لي جميع الأشخاص الذين يقطنون في مدينة معينة.

بفرض لدينا المعلومات:

```
lives_in_city( john, new_york).
lives_in_city( mike, new_york).
lives_in_city( sarah, los_angelos).
lives_in_city( john, chicago).
```

فيكون التابع بالشكل:

```
people_in_city( City, People):- findall( Person, lives_in_city( Person, City), People).
```

وإذا قمنا بالـ query التالي:

```
?- people_in_city(new_york, People).        →       People = [john, mike].
```

كان بإمكاننا استخدام bag of أو set of لأنهم جميعاً سينفذون نفس المهمة.

تمرين (2): اكتب تابع يقوم بحساب مجموع عددين add_tow_numbers /3 باستخدام الحقائق الديناميكية.

❖ أولاً نعرف الدالة add_tow_numbers لجمع عددين وإرجاع الناتج:

```
add_tow_numbers(X,Y, Res) :- Res is X+Y.
```

❖ نقوم بتعريف قاعدة ديناميكية res_add_tow_numbers لتخزين النتائج السابقة:

```
:-dynamic res_add_tow_numbers/3.
```

❖ نقوم بتعديل القاعدة الأساسية لتقوم بحفظ الناتج في res_add_tow_numbers:

```
add_tow_numbers(X,Y, Res) :- Res is X+Y, assert( res_add_tow_numbers(X,Y, Res) ).
```

❖ أخيراً نضيف قاعدة تفحص هل هذا الناتج محسوب من قبل، إذا كان محسوب يرجعه ويقوم بالتوقف (!):

```
add_tow_numbers(X,Y, Res) :- res_add_tow_numbers(X,Y, Res), !.
```

❖ فيصبح البرنامج كاملاً:

```
:-dynamic res_add_tow_numbers/3.
add_tow_numbers(X,Y, Res) :- res_add_tow_numbers(X,Y, Res), !.
add_tow_numbers(X,Y, Res) :- Res is X+Y, assert( res_add_tow_numbers(X,Y, Res) ).
```

طبعاً كان يمكننا حل التمرين بالطريقة التي استخدمناها سابقاً في فيبوناتشي من خلال تحويل القاعدة add_tow_numbers إلى قاعدة ديناميكية بدلاً من تعريف قاعدة جديدة، ولكن عندئذ نستخدم asserta.

تمرين (3): اكتب تابع يقوم بإضافة أشخاص وإرجاع جميع أسماء الأشخاص وأعمارهم المدخلة من قبل، وتعليمة أخرى تقوم بإرجاع جميع الأشخاص الذين في عمر محدد مع ترتيبهم أبجدياً، باستخدام حقائق ديناميكية.

```
:- dynamic person/2.
add_person( Name, Age):- assert( person( Name, Age) ).
get_people( People):- findall( (Name, Age), person(Name, Age), People).
get_people( Age, People):- setof( Name, person(Name, Age), People).
```

- أولاً نعرف الحقيقة الديناميكية person/2.
- ثم نعرف دالة لإضافة شخص جديد إلى قاعدة الحقائق الديناميكية حيث نقوم بإدخال اسم وعمر الشخص.
- نعرف دالة لإرجاع جميع الأشخاص المضافين إلى قاعدة الحقائق الديناميكية باستخدام findall، لاحظ أن المتحول الذي أريده هو Name مع Age لذلك في مكان المتحول المطلوب نضع المتحولين معاً لكن بين قوسين ليتعامل معهم ككيان واحد.
- ثم أخيراً نعرف دالة لإرجاع جميع الأشخاص المضافين إلى قاعدة الحقائق الديناميكية باستخدام set of مع تحديد العمر المطلوب.

لنقم بالـ query التالي:

```
?- add_person(a,10).          →   True.
?- add_person(b,10).          →   True.
?- add_person(c,10).          →   True.
?- get_people(People).        →   People = [ (a,10), (b,10), (c,10) ].
?- get_people(10,People).     →   People = [a, b, c].
```

لاحظ في الاستفسار الأول لا يوجد فائدة من استخدام bag of أو set of لأن الشخص الواحد لديه نتيجة واحدة.

بينما في الاستفسار الثاني فاستخدام set of أفضل وأسرع لأنها تقوم بترتيب النتيجة لوحدها.

تمرين (4): اكتب برنامج يقوم بإضافة وحذف والبحث وعرض الكتب المتوفرة في المكتبة.

كل كتاب يحوي اسم الكتاب واسم الكاتب.

```
:- dynamic book/2.

add_book( Name, Author) :- assertz( book( Name, Author) ), write('تم إضافة الكتاب بنجاح').
delete_book( Name, Author) :- retract( book( Name, Author) ), write('تم حذف الكتاب بنجاح').
search_book( Name, Author) :-  book( Name, Author) , write('الكتاب متوفر في المكتبة').
all_book( Books ) :- findall( (Name, Author), book( Name, Author), Books ).
```

- أولاً نعرف الحقيقة الديناميكية book والتي تأخذ بارامترين هما Name, Author.
- نعرف الدالة add_book لإضافة الكتب المرادة في قاعدة البيانات الديناميكية من خلال التابع assert وبعدها طباعة "تمت إضافة الكتاب بنجاح".

❖ نعرف الدالة delete_book لحذف الكتب المرادة من قاعدة البيانات الديناميكية من خلال التابع retract وبعدها طباعة "تم حذف الكتاب بنجاح".

❖ نعرف الدالة search_book للبحث عن كتاب معين في المكتبة من خلال قاعدة البيانات الديناميكية book والتي يوجد فيها كل الكتب المضافة وبعدها طباعة "الكتاب متوفر في المكتبة".

❖ وأخيراً نعرف الدالة all_book من أجل عرض جميع الكتب الموجودة في المكتبة من خلال الدالة find all.

لنقم الآن بالـ query:

```
?- add_book('11.11', 'يوسف جاسم رمضان').
تم إضافة الكتاب بنجاح
true.

?- add_book('Mrs.Dalloway', 'Virginia Woolf').
تم إضافة الكتاب بنجاح
true.

?- search_book('Mrs.Dalloway', Author).
الكتاب متوفر في المكتبة
Author = 'Virginia Woolf'.

?- search_book('11.11', 'يوسف جاسم رمضان').
الكتاب متوفر في المكتبة
true.

?- all_book(Book).
Book = [('11.11', 'يوسف جاسم رمضان'), ('Mrs.Dalloway', 'Virginia Woolf')].

?- delete_book('Mrs.Dalloway', 'Virginia Woolf').
تم حذف الكتاب بنجاح
true.

?- all_book(Book).
Book = [('11.11', 'يوسف جاسم رمضان')].
```

The universe is under no obligation to make sense to you

Space is an inspirational concept that allows you to dream big.

تمرين (5): اكتب تابع يقوم بالمرور على مصفوفة وطباعة * مكان كل عنصر.

أولاً سنقوم بتحديد حجم المصفوفة من خلال تعريفه كحقيقة:

```
grid_size(4,6).
```

❖ لطباعة المصفوفة يجب المرور على جميع الأسطر واحد تلو الآخر ومعالجته، لذلك نحن بحاجة للمرور باستخدام التعليمة between على جميع القيم بين 1 وعدد الأسطر N، ومن أجل كل قيمة نقوم باستدعاء تابع يعالج الأعمدة، ومن ثم التراجع وتجريب سطر آخر وذلك من خلال تعليمة fail:

```
get_row():- grid_size(N,_), between(1,N,_),  \+ get_col(), nl , fail.
```

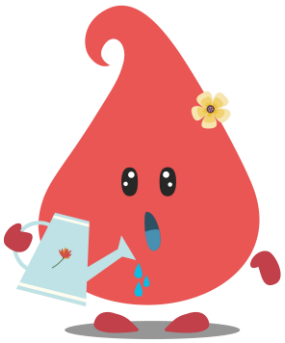
❖ من أجل كل سطر يجب المرور على جميع الأعمدة، أي سنستخدم التعليمة between للمرور على جميع القيم بين 1 وعدد الأعمدة M، ومن أجل كل عنصر سنقوم بطباعة * يليها فراغ، ومن ثم نقوم بالتراجع من خلال fail لطباعة العنصر التالي:

```
get_col():- grid_size(_,M), between(1,M,_), write('* '), fail.
```

❖ أخيراً نكتب التعليمة الأساسية print_grid والتي تقوم باستدعاء get_row، لكن لاحظ أن get_row دائماً سيرجع false لذلك سنضع قبله \+ لتصبح True، وكذلك الأمر في get_row نضع قبل get_col نفي \+ لتصبح True.

فيصبح لدينا البرنامج كاملاً:

```
print_grid( ):- \+get_row().
get_row():- grid_size(N,_), between(1,N,_), \+ get_col(), nl , fail.
get_col():- grid_size(_,M), between(1,M,_), write('* '), fail.
```



عند القيام بالـ query:

```
?- print_grid.
* * * * * *
* * * * * *
* * * * * *
* * * * * *
true.
```

تمرين (6): اكتب تابع يقوم بالمرور على مصفوفة وطباعة عناصرها.

بفرض لدينا:

```
grid_size(4,4).
cell( 1, 1, a).   cell( 1, 2, b).   cell( 1, 3, c).   cell( 1, 4, d).
cell( 2, 1, a).   cell( 2, 2, b).   cell( 2, 3, c).   cell( 2, 4, d).
cell( 3, 1, a).   cell( 3, 2, b).   cell( 3, 3, c).   cell( 3, 4, d).
cell( 4, 1, a).   cell( 4, 2, b).   cell( 4, 3, c).   cell( 4, 4, d).
```

❖ بنفس الطريقة السابقة، سنمر على الأسطر، لكن قيمة السطر التي كان يرجعها تابع الـ between لن نهملها بل سنخزنها في متحول X مثلاً، ونرسلها إلى تابع طباعة الأعمدة.

❖ عند معالجة الأعمدة بنفس الطريقة السابقة لكن قيمة العمود التي كان يرجعها تابع الـ between لن نهملها بل سنخزنها في متحول Y مثلاً، الآن أصبح لدينا رقم السطر والعمود للخلية التي نريد طباعتها، سنقوم بجلب قيمتها وتخزينها بمتحول C مثلاً، ومن ثم نطبع قيمة المتحول C.

فيصبح البرنامج:

```
print_grid( ):- \+get_row( ).
get_row( ):- grid_size(N,_), between(1,N,X), \+ get_col(X), nl , fail.
get_col(X):- grid_size(_,M), between(1,M,Y), cell(X,Y,C),write(C), write(' '), fail.
```

عند القيام بالـ query:

```
?- print_grid.
a b c d
a b c d
a b c d
a b c d
true
```

ملاحظة:

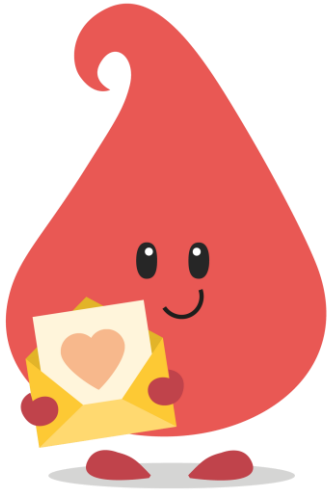
يمكن كتابة تعليمة الـ if بشكل مختصر في البرولوغ بالشكل:

```
((condition) -> (if_true) ; (if_false))
```

مثلاً إذا أردنا كتابة تابع الـ max بشكل مختصر:

```
maxx(X,Y,Z):- (X=<Y -> Z=Y ; Z=X).
```

... The End ...



ومع نهاية هذه المحاضرة نكون قد أتممنا بإذن الله
مقرر العملي لمادة الذكاء
ونختم معكم رحلتنا الشيقة، متمنين لكم دوام الصحة
وأعلى العلامات
إلى اللقاء يا أصدقاء..