قمنا سابقا بالتعرف على بناء النظم الخبيرة باستخدام مكتبة experta والتي توفر لنا اساسيات بناء اي نظام خبير والتي تحدثنا عنها سابقا (Facts, Rule and Knowledge Engine) والان سنقوم بالتعرف على ادوات واساليب توفرها لنا ايضا مكتبة experta لنبني تركيبات patterns اعقد والتي بدورها تساعدنا لبناء قواعد تصف مشكلتنا بشكل اكثر دقة وكفاءة

سبق وتحدثنا عن ان الRule هو عبارة عن مجموعة من الpatterns والتي نقوم بتركيبها مع بعضها البعض لنوصف شيء معين وندعو هذا الجزء بالLHS والجزء الاخر الRHS هو عبارة عن تابع يتم تعريفه مع هذه الRule يتم تنشيط هذا التابع وتنفيذه في حال تحقق الLHS

سنستعرض الان اهم الطرق والادوات التي تساعدنا في بناء تركيبات patterns والتي توفرها لنا مكتبة experta

Conditional Statements

هي عبارة عن الية لتركيب patterns مختلفة تعد ضرورية لتوصيف حالة معينة يتوجب عند تحققها القيام بامر معين

1. AND Element:

هي عبارة عن conditional element تقوم بانشاء pattern وذلك بتمرير الحقائق المراد تطبيق هذه العلاقة عليها كarguments لها

ولتتحقق هذه القاعدة يجب على كافة الحقائق داخلها ان تتحقق

مثال:

```
@Rule (AND(
      Fact(study = True),
      Fact(helpOther = True)
    )
  )
def success():
 print('^_^')
```

2. OR Element:

هو عبارة عن conditional element ايضا يقوم بانشاء pattern بنفس الطريقة بتمرير الحقائق المراد تطبيق هذه العلاقة عليها كarguments لها

ولتحقق هذه القاعدة يجب ان يتحقق على الاقل حقيقة واحدة من الحقائق المعرفة داخله

```
@Rule (OR(
      Fact(something = True),
      Fact(anotherOne = False)
    )
  )
def function():
  print('while at least one of the facts in the OR element is True, then this will be executed')
```

3. NOT Element:

هو عبارة عن conditional element ايضا يقوم بانشاء pattern بنفس الطريقة بتمرير الحقائق المراد تطبيق هذه العلاقة عليها كargumentsلها
وتتحقق هذه القاعدة عندما لا يتطابق الpattern المعطى مع اي قاعدة من القواعد الموجودة لدينا في الworking memory, اي يعاكس ال pattern المعطى
مثال:

```
@Rule (NOT(
      Fact(something = True)
    )
  )
def function():
  print('while there is no fact represent the pattern in the NOT element, then this will be executed')
```

4. TEST Element:

هو عبارة عن conditional element يقوم بالتحقق من تابع اذا ما تم التنفيذ بشكل صحيح وانتهى سيقوم هذا الelement باعادة قيمة صحيحة True وينتقل للعملية التالية

مثال:

```
from experta import Rule, Fact, TEST, KnowledgeEngine, MATCH

class Number(Fact):
   pass

class Descending(KnowledgeEngine):
```

```
    @Rule(
      Number(MATCH.a),
      Number(MATCH.b),
      TEST(lambda a, b: a > b),
      Number(MATCH.c),
      TEST(lambda b, c: b > c)
    )
    def sort_descending(self, a, b, c):
        print(a,b,c)
        pass

m = Descending()
m.reset()
m.declare(*[Number(x) for x in (12, 33, 42)])
m.run()
```

كما نعلم فان الKnowledgeEngine يعمل عن طريق مطابقة الحقائق الموجودة في الworking memory مع القواعد المعطاة، وعندما يجد الحقائق التي توافق القاعدة المعطاة يتم تنفيذ التابع المعرف مع القاعدة.
لدينا هنا قاعدة تأخذ اولا عددين مختلفين تخزن العدد الاول في المتغير a والثاني مختلف عن الاول في المتغير b
يتم ارسالهم الىTEST Element والتي تقوم بالتحقق من ان العدد الاول اكبر من الثاني
ثم تاخذ القاعدة لدينا عدد ثالث مختلف عن الاثنين السابقين وتخزنه في متغير c
ومرة أخرى يتم ارسال العدد الثاني b مع العدد الاخير c الى الTEST Element ليتحقق من ان العدد الثاني اكبر من الثالث
الان اذا نظرنا الى هذه القاعدة من بعد نرى انها تاخذ ثلاث اعداد وتتحقق من ان العدد الاول اكبر من الثاني ومن ثم ان الثاني اكبر من الثالث، اي انها تعرف pattern لاعداد مرتبة بشكل تنازلي
الان قمنا باضافة ثلاثة حقائق عددية باستخدام الclass الخاص بنا بالتالي اصبحوا في الWorking Memory والان عند تشغيل الEngine فانه سوف يقوم بالبحث في الWorking Memory عن المتغيرات التي تحقق القاعدة المعرفة
فاذا اخذ على سبيل المثال الترتيب المعطى للحقائق 12, 33, 42
سيجد ان الTEST الاولى سوف تفشل لان 12 ليست اكبر من 33

بالتالي القاعدة لم تتحق وسيأخذ هذه الحقائق باشكال مختلفة حتى يأخذهم بالترتيب الذي يؤدي الى تحقق هذه القاعدة وبالتالي يقوم بتنفيذ الRHS وطباعة هذا الترتيب، والذي كما قلنا سابقا هو ترتيب تنازلي اي ستكون نتيجة التنفيذ السابق:

Output:
42 33 12

5. EXISTS Element:

هو عبارة عن conditional element ايضا يقوم بانشاء pattern وذلك بتمرير الحقائق المراد تطبيق هذه العلاقة عليها كargumentsلها
ولتحقق هذه القاعدة يجب ان يتحقق واحد او اكثر من الحقائق مع الpattern المعرف فيه
- تتوقف عن البحث عندما لاتجد اي حقيقة توافق هذا الpattern

مثال:

```
@Rule(
  EXISTS(
    Fact(something=True)
  )
)
def function():
  print('while there is at least one pattern in the EXISTS element matched with the facts in the working memory, then this will be executed')
```

6. FORALL Element:

هو عبارة عن conditional element ايضا توفر الية لتحديد ان مجموعة من الConditional Elements محققة من اجل كل ظهور لمجموعة من الحقائق او الConditional Elements الاخرى
فرضًا أن لدينا قاعدة معرفة تحتوي على حقائق حول الطلاب ونريد التحقق مما إذا كان كل طالب قد اجتاز مواد القراءة والكتابة والحساب فنقوم بمنحه شهادة خاصة به عن اجتياز هذا الفصل.

```
from experta import KnowledgeEngine, Fact, Rule, FORALL, MATCH
```

```
class Student(Fact):
    pass

class Reading(Fact):
    pass

class Writing(Fact):
    pass

class Arithmetic(Fact):
    pass

class MyKnowledgeEngine(KnowledgeEngine):
    @Rule(
        FORALL(
            Student(MATCH.name),
            Reading(MATCH.name),
            Writing(MATCH.name),
            Arithmetic(MATCH.name)
        )
    )
    def check(self):
        print(f'The Student has passed all the exams')

engine = MyKnowledgeEngine()
engine.reset()
engine.declare(
    Student(name='RBCs'),
    Writing(name='RBCs'),
    Arithmetic(name='RBCs'),
    Reading(name='RBCs')
)
engine.run()
```

في هذا المثال، إذا كان لدينا حقائق تفيد بأن الطالب 'RBCs' اجتاز مواد القراءة والكتابة والحساب، سيتم تنفيذ القاعدة وستظهر الرسالة المعطاة.
الآن، يمكنك أن ترى كيف يمكن استخدام عنصر الشرط FORALL للتحقق من تطابق مجموعة من الحقائق مع كل حدوث لعنصر آخر
فهو اخذ اولا حقيقة من النوع Student باسم معين وقام بتخزينه في المتغيرname ثم انتقل ليتحقق اذا كان هناك حقيقة من نوع Reading بذات الاسم المخزن في المتغير name وثم ينتقل للثالث فالرابع وهكذا حتى ينتهي من التحقق من كل الشروط لكل حقيقة موجودة لدينا وعندما تتحق القاعدة يقوم بطباعة جملة الطباعة

Field Constraint

تقوم بتعريف قيود على قيم الحقول في الحقائق والتي يجب تحققها لكي تطابق الحقيقة الpattern وتستخدم لضمان ان الحلول او النتائج ضمن نطاق مقبول او معقول ومن انواع هذه القيود:

1. L:

توفر لنا عملية مساواة دقيقة مع قيمة محددة نقوم نحن بتحديدها تقوم بعمل مشابه لاستخدام اشارة المساواة (==)

انها تضمن بان الحقل سيوافق القيمة المعطاة تماما

مثال:

```
class MyKnowledgeEngine(KnowledgeEngine):
  @Rule(
    Fact(L('cat'))
  )
  def test(self):
    print('cat detected')

engine = MyKnowledgeEngine()
engine.reset()
engine.declare(
  Fact('cat'),
  Fact('dog'),
  Fact('fish')
```

```
)
engine.run()
```

2. W:

هذا العنصر يعمل كبطاقة جامحة (اي مثل ورقة الجوكر ^_^) تطابق اي قيمة ايا تكن من غير اي قيود على هذه القيمة

مثال:

```
from experta import W

class MyKnowledgeEngine(KnowledgeEngine):
  @Rule(
    Fact('animal', W())
  )
  def test(self):
    print('animal detected')

engine = MyKnowledgeEngine()
engine.reset()
engine.declare(
  Fact('animal', 'cat'),
  Fact('animal', 'dog'),
  Fact('human', 'ahmad'),
  Fact('animal', 'fish')
)
engine.run()
```

هنا اي حقيقة من فصيلة "حيوان" بغض النظر عن نوعه سوف تحقق القاعدة

3. P:

هذا العنصر يستخدم دالة تحكم لتحديد ما إذا كانت القيمة تطابق شرطًا معينًا ويعيد قيمة True في حال تحقق الشرط المعطى

مثال

إذا أردنا قاعدة تطابق حقيقة تحتوي على عدد في المؤشر الأول أكبر من 10 ومن نوع integer سنستخدم:

```
from experta import P

class MyKnowledgeEngine(KnowledgeEngine):
  @Rule(
    Fact(
      P(lambda x: isinstance(x, int) and x > 10),
    )
  )
  def test(self):
    print('the number is integer and greater than 10')


engine = MyKnowledgeEngine()
engine.reset()
engine.declare(
  Fact('RBCs', 21),
  Fact(7),
  Fact(14)
)
engine.run()
```

Composing Field Constraint

هي العمليات المنطقية الاساسية المعروفة and, or and not يمكن تطبيق هذه العمليات باستخدام الرموز ~ and | ,& على الترتيب وذلك للجمع بين الField Constraints المختلفة

1. عملية الand:

باستخدام الرمز "&" استطيع الجمع بين علاقتين حيث يجب تحققهم معا لتكون القاعدة محققة

```
@Rule(Fact(x=P(lambda x: x >= 0) & P(lambda x: x <= 255)))
def _():
    pass
```

2. عملية الor:

باستخدام الرمز "|" استطيع الجمع بين علاقتين حيث تحقق احدهما او كليهما كافي لتكون القاعدة محققة

```
@Rule(Fact(name=L('Alice') | L('Bob')))
def _():
    pass
```

3. عملية الnot:

باستخدام الرمز "~" استطيع نفي علاقة معينة ليكون عدم تحققها هو شرط تحقق القاعدة

```
@Rule(Fact(name=~L('Charlie')))
def _():
    pass
```

MATCH object:

يتم استخدامها لربط قيم من Fact معينة مع متغير معين باسم ما

عندما نستخدم الMATCH فنحن نخبر النظام الخبير ليأخذ القيمة من الfact ويقوم بربطها بمتغير بنفس اسم المتغير الذي نقوم بتحديده

مثال:

```
class MyEngine(KnowledgeEngine):
    @Rule(Fact(MATCH.my_value))
    def rule(self, my_value):
        print(f"The value is: {my_value}")
```

في هذا المثال الحقل my_value من الحقيقة Fact يتم تخزين القيمة الخاصة به في متغير باسم my_value والتي يمكن ان يتم استخدامها لاحقا في القواعد

AS object:

يشابه الى حد كبير MATCH ، لكنه يسمح لنا بتعيين اسم مخصص للمتغير الذي سيحتوي على الحقيقة نفسها بينما في الMATCH كنا هناك نقوم بتخزين قيمة معينة من داخل الحقيقة في المتغير بدلا من الامساك بالحقيقة ذاتها وتخزينها في المتغير. يكون مفيدًا عندما نرغب في إعطاء اسم أكثر دقة لمتغير يمثل حقيقة ما ليتم استخدامه فيما بعد بالRHS أو عندما نريد تجنب تعارض الأسماء.

مثال:

```
class MyEngine(KnowledgeEngine):
    @Rule(AS.my_fact << Fact())
    def rule(self, my_fact):
        print(f"the fact is: {my_fact}")
```

هنا لدينا AS.my_fact تقوم بانشاء متغير باسم my_fact يحتوي على object من Fact والذي يمكن استخدامه لاحقا في الRule كلها

Nested Matching

اذا كان لدينا حقيقة تحوي على dictionary او list ونريد ان نصل الى عناصر معينة منها فيمكننا ذلك عن طريق

كتابة اندرسكور مرتين ومن ثم اسم الkey من الdictionary الذي نريد الوصول اليه

او كتابة اندرسكور مرتين ايضا ومن ثم رقم الindex من الlist للعنصر الذي اريد الوصول اليه

مثال:

```
from experta import *

class Ship(Fact):
    pass

class MyKnowledgeEngine(KnowledgeEngine):
    @DefFacts()
    def intial_data(self):
        yield Ship(data={
                "name": "SmallShip",
                "position": {
                    "x": 300,
                    "y": 200
                },
                "parent": {
```

```
                "name": "BigShip",
                "position": {
                    "x": 150,
                    "y": 300
                }
            }
        }
    )

@Rule(
    Ship(
        data__name=MATCH.name1,
        data__position__x=MATCH.x,
        data__position__y=MATCH.y,
        data__parent__name=MATCH.name2,
        data__parent__position__y=MATCH.y
    )
)
def collision_detected(self, name1, name2):
    print(f'COLLISION! {name1} {name2}')

engine = MyKnowledgeEngine()
engine.reset()
engine.run()
```

Mutable Objects

عادة عندا يتم انشاء حقيقة جديدة كل قيمها يتم نقلهم الى نوع غير قابل للتعديل هذه العملية تتم بشكل داخلي ولذلك لانه لايجب ان يكون من المسموح بشكل افتراضي تعديل وتغير قيم الحقائق اثناء العمل لانها قد تؤدي الى حدوث اخطاء

ولكن بالطبع هناك الية تتيح لنا ان نجعل هذه البيانات قابلة للتعديل عند الحاجة وذلك كالتالي:

```
from experta.utils import unfreeze

class MutableTest(KnowledgeEngine):
    @Rule(Fact(v1=MATCH.v1, v2=MATCH.v2, v3=MATCH.v3))
    def is_immutable(self, v1, v2, v3):
        v2=unfreeze(v2)
```

```
        v2.append(5)
        print(type(v1), "is Immutable!")
        print(type(v2), "is mutable!")
        print(type(v3), "is Immutable!")

ke = MutableTest()
ke.reset()
ke.declare(Fact(v1={"a": 1, "b": 2}, v2=[1, 2, 3], v3={1, 2, 3}))
ke.run()
```

هنا قمنا بارسال قيم حقائق كdictionary و كlist وهي بشكل افتراضي تكون غير قابلة للتعديل ، اذا ما اردنا التعديل عليها يمكننا استخدام تابع unfreez من experta.utils وذلك لكي تصبح قابلة للتعديل

تمرين:
نريد ان نقوم بكتابة مجموعة قواعد لتحضر لنا العنصر الاكبر من بين مجموعة عناصر معطاة مع حذف كل العناصر الاصغر:

```
class Maximum(KnowledgeEngine):
    @Rule(NOT(Fact(max=W())))
    def init(self):
        self.declare(Fact(max=0))

    @Rule(Fact(val=MATCH.val),AS.m <<
Fact(max=MATCH.max),TEST(lambda max, val: val > max))
    def compute_max(self, m, val):
        self.modify(m, max=val)

    @Rule(AS.v <<
Fact(val=MATCH.val),Fact(max=MATCH.max),TEST(lambda max,
val: val <= max))
    def remove_val(self, v):
        self.retract(v)

    @Rule(AS.v << Fact(max=W()),
          NOT(Fact(val=W())))
    def print_max(self, v):
```

```
        print("Max:", v['max'])


m = Maximum()
m.reset()
m.declare(*[Fact(val=x) for x in (12, 33, 100, 55, 11, 75, 34, 42)])
m.run()
```

قمنا اولا بتعريف قاعدة للبحث في الحقائق واذا لم نجد اي حقيقة لديها max ايا تكن قيمته تقوم بتعريف حقيقة فيها المتغيرmax بقيمة ابتدائية 0
ثم قمنا باخذ حقيقة برقم وخزناه بمتغير val واخذنا حقيقة التي فيها القيمة العظمى max الحالية وخزناها في متغير max وقمنا بتخزين الحقيقة باكملها في متغير بالاسم m ثم عن طريق TEST قمنا بالتحقق من اذ ماكانت القيمة التي لدينا اكبر من العظمى يتحقق الLHS لدينا وننتقل الى تنفيذ التابع الRHS والتي فيها نقوم باستخدام المتغيرm الذي خزنا فيه الحقيقة التي تحوي القيمة العظمى الحالية وذلك لتغير قيمتها بقيمة المتغير val
الان لدينا قاعدة الحذف للقيم الاصغر من القيمة العظمى تقوم هذه القاعدة بالعمل بنفس طريقة القاعدة السابقة تقريبا ولكن تقوم باستخدام تابع retract الذي يقوم بحذف القاعدة الغير عظمى اخيرا عندما نصل الى حالة لايوجد اي حقيقة بمتغير عادي val ولكن لدينا حقيقة بمتغير max نكون قد وصلنا للحالة النهائية وبهذا نقوم بعملية الطباعة

تمرين :
في كرة القدم كثيرا ماتكون العاطفة هيي المحرك وراء تقييماتنا ولمثل هذه الظروف وجدت النظم الخبيرة لتحكم لنا بالمنطق والبرهان ومن غير عاطفة لتخبرنا بالفريق الاعظم
ولهذا سنقوم باستخدام نظام خبير على مباراة من دوري ابطال اوروبا ليحكم النظام لنا بالمنطق من هو الفريق الافضل في العالم عموما وفي اوروبا خصوصا
ولتحقيق ذلك ليكن لدينا الحقائق التالية:

```
from experta import KnowledgeEngine, Fact, Rule, DefFacts,
FORALL, MATCH, TEST, W

class Player(Fact):
  pass

class Score(Fact):
  pass
```

```
class Monitor(Fact):
  pass

class Partner(Fact):
        pass


class ChampionsLeague2009(KnowledgeEngine):
  @DefFacts()
  def initial_values(self):
    yield Player(name= 'Steven Gerrard', team= 'Liverpool', role= 'CMF')
    yield Player(name= 'Fernando Torres', team= 'Liverpool', role='CF')
    yield Player(name= 'Gonzalo Higuain', team= 'Real Madred', role=
'CF')
    yield Player(name= 'Sergio Ramos', team= 'Real Madred', role=
'DMF')
    yield Score(name= 'Steven Gerard', degree= 10)
    yield Score(name= 'Fernando Torres', degree= 9)
    yield Score(name= 'Gonzalo Higuain', degree= 1)
    yield Score(name= 'Sergio Ramos', degree= 1)
    yield Monitor(p1= 'Steven Gerard', p2= 'Gonzalo Higuain')
    yield Monitor(p1= 'Fernando Torres', p2= 'Sergio Ramos')
    yield Partner(p1= 'Steven Gerrard', p2= 'Fernando Torres')
    yield Partner(p1= 'Gonzalo Higuain', p2= 'Sergio Ramos')
```

هذه حقائق عن مباراة دور ال16 من دوري ابطال اوروبا 2009 بين فريقي ليفربول وريال مدريد

قمنا اولا بتعرفي كلاسات لكل من اللاعبين وتقييمهم واللاعب من الفريق الاخر الذي يقومون بمراقبته وزميلهم في الفريق وقمنا بالوراثة من الكلاسFact وذلك لتتم معاملة الكلاسات كحقائق من قبل الEngine فلا نستطيع ان ننشئ كلاسات خاصة بنا دور الوراثة من الكلاس Fact وذلك لكي تخزن بالKnowledgeBase وتنقل الى Working Memory عند الحاجة وغيرها من العمليات التي يقوم بها الEngine للحقائق

سنعتمد في هذا النظام الخبير على مقارنة كل لاعب مع اللاعب الذي يراقبه لنرى الافضل بينهما وبناءا على هذا يتبين من الفريق الذي لديه تقييم اعلى بشكل عام وبالتالي من هو الفريق الافضل .

فيكون لدينا التالي:

```
  @Rule(
    Player(name= MATCH.player1, team= MATCH.team, role=W()),
    Score(name= MATCH.player1, degree= MATCH.degree1),
    Monitor(p1= MATCH.player1, p2= MATCH.player2),
    Score(name=MATCH.player2, degree = MATCH.degree2),
    TEST(lambda degree1, degree2: degree1 > degree2),
    Partner(p1= MATCH.player1, p2= MATCH.player3),
    Score(name= MATCH.player3, degree= MATCH.degree3),
    Monitor(p1= MATCH.player3, p2= MATCH.player4),
    Score(name= MATCH.player4, degree= MATCH.degree4),
    TEST(lambda degree3, degree4: degree3 > degree4)
  )
  def best_team_detection_^_^(self, team):
    print(f'The Best Team is: {team}')

engine = ChampionsLeague2009()
engine.reset()
engine.run()
```

قمنا اولا باخذ لاعب ما وحفظنا اسمه في المتغير player1 من فريق ما team بغض النظر عن مركز هذا اللاعب ايا يكن

ثم قمنا باستخدام الكلاس Score لنجلب تقييم هذا اللاعب عن طريق استخدام المتغير الذي يخزن اسمه player فحصلنا على تقييمه degree1

ثم قمنا باستعمال الكلاس Monitor لنعلم اللاعب من الفريق الاخر الذي يراقبه هذا اللاعب عن طريق استخدام المتغير player1 الذي يخزن اسمه وقمنا بتخزين اسم اللاعب الخصم في المتغير player2

ثم بنفس الطريقة السابقة قمنا باستعمال الكلاس Score لناخذ تقييم اللاعب الخصم وخزناه في المتغير degree2

الان بات اصبح لدينا تقييم كل من اللاعبين فسنقوم باستعمال TEST Element وذلك للمقارنة بين تقييمي كل من اللاعبين

والان عندما يكون اللاعب الاول له التقييم الاعلى سننتقل الى القاعدة التالية وهيي Partner والتي تقوم باخبارنا بزميل هذا اللاعب لكي نقارن ايضا بينه وبين اللاعب الذي يراقبه

والان بتكرار نفس الطريقة السابقة في المقارنة بين لاعبين مع الانتباه الى اسماء المتغيرات التي نستخدمها لتخزين اسماء اللاعبين والنتيجة الخاصة بهم

نظرة عامة عن الطريقة المتبعة في الحل:
اي لاعب يكون تقييمه اعلى من لاعب الخصم الذي يراقبه ولديه ايضا زميل في الفريق تقييمه ايضا اعلى من لاعب فريق الخصم الذي يراقبه بالتالي هذا الفريق ككل هو الافضل
فيكون لدينا نتيجة التنفيذ لمعرفة الفريق الافضل كالتالي:

Output:
The Best Team is: Liverpool

هذا تمرين اوسع قليلا عن المعطى والذي كان من دون اضافة اللاعب الزميل في الفريق وكان بتقييد الفريق
اي كان الطلب ان نقوم بتثبيت فريق معين ونرى ان كان افضل بدلا عن البحث من غير تحديد عن من الافضل
ولتنفيذ ذلك نقوم بالتالي:

```
  @Rule(
     FORALL(
       Player(name= MATCH.player1, team= "Liverpool", role=W()),
       Score(name= MATCH.player1, degree= MATCH.degree1),
       Monitor(p1= MATCH.player1, p2= MATCH.player2),
       Score(name=MATCH.player2, degree = MATCH.degree2),
       TEST(lambda degree1, degree2: degree1 > degree2)
     )
  )
  def best_team_detection(self):
    print(f'The Best Team is: Liverpool')

engine = ChampionsLeague2009()
engine.reset()
engine.run()
```

بشكل مشابه جدا للسابق لكن الان المهمة ان نضع فريق معين اما اعيننا (وليكن ليفربول) ونرى ان كان هو الفريق الافضل ام لا
فقمنا بتثبيت الفريق في القاعدة الاولى وبنفس الطريقة جلبنا تقييم اللاعب واللاعب الذي يراقبه وتقييمه هو الاخر وقمنا بالمقارنة بينهما مع استخدام FORALL هنا وذلك لنتحقق ان كل لاعبي هذا الفريق هم بالفعل يحققون هذا الشرط فيكون الخرج ايضا:

Output:
The Best Team is: Liverpool

ملاحظة اربيسيزية:
يبدو انها حقيقة غير قابلة للشك