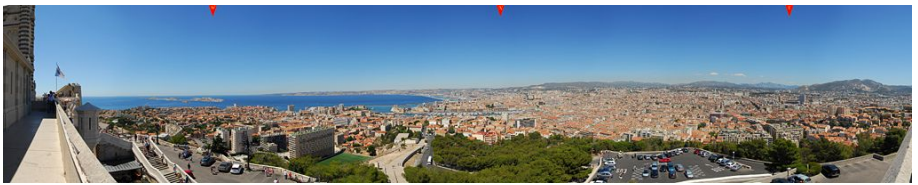
نیما جوهری

زبان برنامه‌نویسی PROLOG
ارائه برای درس زبان‌های برنامه‌سازی
آذر ۱۳۹۱

۱ پرولوگ چیست؟

- زبان برنامه‌نویسی: حاصل از پژوهش‌ها در زمینه‌ی «برنامه نویسی بر پایه منطقی ریاضی»
- طراحی شده در دهه‌ی ۷۰ میلادی توسط «آلین کولمراور [۱] » در مارسی [۲]، فرانسه.

اولین سیستم پرولوگ در سال ۱۹۷۲ توسط «کولمراور» و «فیلیپ راسل [۳] » نوشته شد.



شکل ۱: پانورامایی از شهر مارسی

---

[۱] Alain Colmerauer
[۲] Marseille
[۳] Philippe Roussel

پرولوگ یکی از رایج‌ترین زبان‌ها در عرصه‌ی هوش مصنوعی‌ست. بر خلاف‌زبان‌های imperative مانند c یا Java، پرولوگ یک زبان declarative است.

بدین معنی که برای حل مساله در پرولوگ، صرفا شرایط را توصیف می‌کنیم و در مورد نحوه رسیدن به راه حل تصمیم نمی‌گیریم. اما در زبان‌های imperative، قدم به قدم برنامه را در رسیدن به راه حل پیش می‌بریم.

پرولوگ برای برخی زمینه‌های برنامه نویسی، مانند هوش مصنوعی و پردازش زبان طبیعی کارگشا و برای برخی عرصه‌ها، مانند برنامه‌نویسی گرافیکی یا محاسبات عددی زبانی نامناسب است.

برای حل مساله در پرولوگ، ابتدا جهان را بوسیله‌ی رابطه‌ها، مدل می‌کنیم. برای مثال، یک باغ‌وحش را در نظر بگیرید. در باغ‌وحش داریم:

```
bigger(elephant,dog).
```

به قطعه کد بالا که با «نقطه» به پایان رسیده، فرض درست (حقیقت) [۴] گفته می‌شود. کد بالا بطور شهودی بیان می‌کند فیل از سگ بزرگ‌تر است.

با اضافه کردن چند فرض دیگر به برنامه، جهان را کامل تر می‌کنیم:

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

[۴] Fact

حال می‌توانیم پس از فراخوانی فایل مبدا، از مفسر پرولوگ در مورد جهان پرسش نماییم: [۵]

```
$ swipl -f zoo.prolog
?- bigger(donkey, dog).
Yes
?- bigger(monkey, elephant).
No
```

حال سوال زیر را مطرح می‌کنیم:

```
?- bigger(elephant, monkey).
No
```

جواب داده شده با شهود ما از رابطه‌ی بزرگتری در باغ‌وحش فرضی‌مان سازگار نیست. این بدین خاطر است که در مدلی که ما از جهان ارائه کردیم صحبتی از تعدی بودن رابطه‌ی bigger نشده، با اینکه فیل از اسب از خر از میمون بزرگ‌تر است. پس ایراد در مدل ناقصی‌ست که ما از جهان ارائه کردیم.

[۵] به این عمل در اصطلاح، «query» گرفتن می‌گویند.

قبل از تعریف رابطه‌ی ذکر شده بصورت یک رابطه‌ی تعدی، بطور دقیق‌تر به ساختار زبانی برنامه‌های پرولوگ می‌پردازیم.

۲ Syntax زبان پرولوگ

۱.۲ ترم ها

ساختمان داده‌ی اصلی زبان پرولوگ، ترم‌ها به ۴ دسته اصلی تقسیم می‌شوند:
اتم‌ها: رشته‌های پیوسته (بدون فاصله) از کاراکترها که با حرف کوچک شروع می‌شوند. همچنین رشته‌ای از علامات هم در رده‌ی ترم‌های اتمی قرار می‌گیرند. عبارات با فاصله نیز، در صورت قرار گرفتن بین «'» یک ترم اتمی درنظر گرفته می‌شوند.

```
term *** a_long_term + <---> 'a long term with space'
```

اعداد: رشته‌ای از کاراکترهای عددی

```
0 4.2 -3 3.0e4
```

متغیر: رشته‌ای بی‌فاصله از کاراکترها که با حرف بزرگ یا «_» شروع می‌شند.

```
Animal _X_1_2 MyVariable X _
```

ترم مرکب: از قرار گیری ترم اتمی، پرانتز باز، تعدادی ترم که با ویرگول از هم جدا شده‌اند و پرانتز بسته بدست می‌آیند.

```
is_bigger(horse, X) f(g(X,_), 7) 'My Functor'(dog)
```

۲.۲ گزاره‌ها

همچنین اجتماع مجموعه‌ی اتم‌ها و ترم‌های مرکب، مجموعه گزاره [۶]های پرولوگ را تشکیل می‌دهند.

۳.۲ نمادگذاری ترم‌های مرکب در مستندات

بطور قراردادی، در مستندات و کتاب‌های موجود پیرامون زبان پرولوگ، ترم مرکبی مانند `length` با 2 آرگومان را با نماد `length/2` نمایش می‌دهند.

[۶] Predicate

۴.۲ کلازها

مجموعه‌ی قواعد [۷] و فرض‌ها در پرولوگ، مجموعه‌ی کلازها را تشکیل می‌دهند.
فرضها: در مثال بالا بطور غیر رسمی با فرضها آشنا شده ایم. از قرار گرفتن یک «نقطه» بعد از گزاره، یک فرض حاصل می‌شود.

```
bigger(whale, _). life_is_beautiful.
```

قواعد: یک قاعده شامل «سر»، علامت :- و «بدنه»ی قاعده است که بدنه دنباله‌ای از گزاره‌هاست که با علامت ویرگول از هم جدا شده‌اند. نماد «نقطه» نمایانگر پایان تعریف قاعده است.

معنی شهودی قاعده این است که هدف بیان شده توسط سر قاعده درست است اگر بتوان نشان داد که تمامی عبارات بدنه‌ی قاعده برقرار هستند.

```
is_smaller(X, Y) :- is_bigger(Y, X).
aunt(Aunt, Child) :-
  sister(Aunt, Parent),
  parent(Parent, Child).
```

[۷] Rule

۵.۲ برنامه

دنباله‌ای از کلازها، یک برنامه‌ی پرولوگ را تشکیل می‌دهند.

۶.۲ کوئری‌ها

بعد از کامپیال، برنامه پرولوگ با ثبت کوئری‌ها در مفسر اجرا می‌شود. یک کوئری، از نظر ساختار زبانی، همان ساختار بدنه‌ی قاعده را داراست. کوئری‌ها را در مفسر پرولوگ، بعد از علامت ?- می‌نویسند.

```
?- is_bigger(elephant, donkey).
?- small(X), green(X), slimy(X).
```

۷.۲ لیست‌ها

لیست‌ها را می‌توان در حقیقت با بهره‌گیری از ترم‌ها تعریف کرد. نماد [] را برای لیست خالی در نظر می‌گیریم و لیست تک عنصری را معادل ترم مرکب زیر با استفاده از فانکتور ویژه‌ی «.» در نظر می‌گیریم:

```
[a] = .(a, [])
```

بهمین ترتیب:

```
[a, b, c] = .(a, .(b, .(c, [])))
```

به عنصر ابتادیی لیست، «سر»[۸] و به لیستی که سر لیست اولیه به آن متصل است، «دم»[۹] یا «دنباله»ی لیست اولی می‌گوییم. از نمادگذاری [H|T] نیز می‌توان برای ساختن لیست‌ها استفاده کرد.

---

[۸] Head
[۹] Tail

مثال‌های زیر، تعاریف بالا در مورد لیست‌ها را بهتر بیان می‌کنند:

```
?- .(a, .(b, .(c, []))) = [a,b,c].
true.

?- .(a,.(b,X)) = [a,b,c].
X = [c].

?- [H|T] = [a,b,c].
H = a,
T = [b, c].

?- [H|T] = [a].
H = a,
T = [].

?- [a|[b,c]] = [a,b,c].
true.
```

۳ مثال: بستار تعدی رابطه‌ی bigger

حال می‌توانیم رابطه‌ی جدیدی را با استفاده از قواعد زیر تعریف کنیم:

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

۱.۳ چند کوئری از مثال باغ‌وحش

اکنون می‌توانیم سوالات گوناگونی در جهان باغ‌وحش فرضی‌مان مطرح کنیم، مثلا اینکه آیا فیل از میمون بزرگتر است یا خیر؟

```
?- is_bigger(elephant, monkey).
Yes
```

یا اینکه: به ازای چه مقادیری از X، می‌توان گفت X از donkey بزرگتر است؟

```
?- is_bigger(X, donkey).
X = horse ;
X = elephant ;
No
```

```
?- is_bigger(X, donkey).
X = horse ;
X = elephant ;
No
```

پس از دریافت جواب از هر قسمت، حرف «;» را وارد می‌کنیم تا جواب‌های جایگزین را ببینیم. عبارت `No` در آخر به این دلیل است که غیر از جواب‌های بالایی، جواب جایگزین دیگری وجود ندارد.

۲.۳ عملیات تطبیق

دو ترم مطابق [۱۰] هستند اگر یکسان باشند یا با مقداردهی به متغیرهایشان یکسان شوند. لازم به ذکر است که مقدار داده شده به تکرارهای مختلف یک متغیر در سراسر ترم، باید مقدارهای یکسانی داشته باشند. تنها عبارت استثنایی این قانون، متغیر ویژه‌ی «_» است که به متغیر ناشناس معروف است.
از اوپراتور «=» برای عمل تطابق استفاده می‌شود. به مثال‌های زیر توجه کنید:

```
?- is_bigger(X, dog) = is_bigger(elephant, dog).
X = elephant
Yes
?- p(X, 2, 2) = p(1, Y, X).
No
?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes
```

[۱۰] Two terms match

به مثال‌های پیچیده‌تر زیر توجه کنید:

```
?- f(a, g(X, Y)) = f(X, Z), Z = g(W, h(X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes
?- X = my_functor(Y).
X = my_functor(_G177)
Y = _G177
Yes
```

در مثال بالا، متغیر G177_، متغیری است که پرولوگ برای رسیدن به هدف ایجاد کرده. به این معنی که با انجام دو تطابق آخر، می‌توان به هدف (که تطابق X با my_functor(Y) است رسید.

۴ تفاوت با سایر زبان‌های متداول

۱.۴ عملیات پیدا کردن عنصر کمینه‌ی یک لیست را در نظر بگیرید

در پرولوگ، تعاریف و فرضها در فایلی نوشته و ذخیره می‌شوند. سپس بوسیله‌ی مفسر پرولوگ، از داده‌های جهان پرسش انجام می‌شود.

برای مثال، قطعه کد زیر، رابطه‌ی دوتایی بین لیست و عنصر کمینه‌ی آن را بطور استقرایی تعریف می‌کند:

```
my_min([X], X).
my_min([H | T], H) :- my_min(T, U), H =< U.
my_min([H | T], U) :- my_min(T, U), U < H.
```

با تفسیر این فایل توسط مفسر پرولوگ، میتوان به این صورت از رابطه‌ی دوتایی استفاده کرد:

```
$ swipl -f my_min.prolog
?- my_min([-4,3,-4,5,6],X).
X = -4 ;
false.
```

این برخلاف روند معمول اکثر زبان‌های متداول است. مثلا برای انجام همین کار در پایتون می‌نویسیم:

```
def find_min(input_list):
  min_elem = None
  for elem in input_list:
    if min_elem == None or elem < min_elem:
      min_elem = elem
  return min_elem
print find_min([-4,3,-4,5,6])
```

و بعد:

```
$ python find_min.py
-4
```

تفاوت اینجاست که بجای جایگزینی یک تابع با مقدار مورد انتظار، رویه‌ها را بصورت رابطه‌هایی تعریف می‌کنیم و از سیستم، مقادیری از جواب را می‌خواهیم که رابطه برقرار باشد.

۲.۴ انجام محاسبات عددی

با استفاده از رابطهی دوتایی is می‌توان با اعداد در پرولوگ کار کرد. به مثال زیر که تعریف فاکتوریل در این زبان است، توجه کنید:

```
factorial(0, 1).

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

۵ برش

برش [۱۱] ، مکانیزمی برای کنترل حدس‌های پرولوگ در فرایند Backtrack است، بدین ترتیب که در صورت رسیدن به اوپراتور مخصوص کات در پرولوگ « ! »، مقادیر جایگزین احتمالی برای سر قاعده فراموش می‌شوند. مثال زیر مفهوم کات را در پرولوگ بهتر بیان می‌کند:

سارقی قصد دزدیدن جواهرات از جواهرفروشی‌های شهری را دارد. برخی از مغازه‌های شهر فاقد سیستم‌های ایمنی پیشرفته هستند و توانایی کافی برای به دام انداختن سارق را ندارند. باقی مغازه‌ها مجهز به سیستم‌های پیشرفته‌ی ایمنی هستند و درصورت ورود سارق به آن مغازه‌ها، وی را به دام می‌اندازند. مغازه‌ای که به آن دستبرد می‌زند از آن جنس نداشته باشد، به مغازه‌ی دیگری دستبرد می‌زند.

---

[۱۱] Cut

اگر بخواهیم با گزاره‌ای، موفقیت سارق در سرقت جنس مورد نظرش را بررسی کنیم، جهان را بصورت زیر مدل می‌کنیم.

```
۱ jewelry_store(shop_2).
۲ jewelry_store(shop_3).
۳
۴ is_insecure(shop_1).
۵ is_insecure(shop_3).
۶
۷ mug(Shop) :- jewelry_store(Shop), !, is_insecure(Shop).
```

حال، درصورتی که پرسیده شود

```
?- mug(X).
```

روند زیر طی می‌شود: جواهرفروشی‌های تعریف شده در جهان فرضی ما، به ترتیب تعریف، فروشگاه‌های ۲ و ۳ هستند. یعنی به جای X، ترتیب فروشگاه‌های ۲ و ۳ قرار می‌گیرند. پس از جایگزینی فروشگاه ۲ بعنوان Shop در خط ۷، به علامت «!» می‌رسیم. با رسیدن به این عملگر، گزینه‌های جایگزین برای متغیرهای «سر» قاعده فراموش می‌شوند. (تنها گزینه دیگر، جایگزینی فروشگاه سوم بجای متغیر Shop بود. حال چون فروشگاه ۲ فروشگاه مجهز به سیستم‌های پیشرفته‌ی امنیتی‌ست، شرط نا امن بودن فروشگاه برقرار نمی‌شود، پس سارق موفق به سرقت از فروشگاه نمی‌شود.

دقت کنید اگر جای خطوط ۱ و ۲ در کد بالا عوض می‌شد، ابتدا فروشگاه ۳ برای سرقت امتحان می‌شد، و چون فروشگاه ۳ یک فروشگاه ناامن است، سارق موفق به سرقت از ان فروشگاه می‌شد. بنابرین:

«چنین نیست که ترتیب قواعد مشخص شده در یک برنامه پرولوگ بی‌اهمیت باشد»

۶ چند برنامه نمونه

۱.۶ اتصال دو لیست به هم

```
my_concat([], L, L).
my_concat([H|T], L, [H|TL]) :- my_concat(T, L, TL).
```

۲.۶ پیدا کردن عنصر انتهایی لیست

```
my_last1([X], X).
my_last1([H|T], X) :- my_last1(T, X).

my_last2(L, X) :- my_concat(LL, [X], L).
```

۳.۶ برگرداندن لیست

```
my_reverse([], []).
my_reverse([H | T], L) :-
  my_reverse(T, TR), my_concat(TR, [H], L).
```

```
mergesort([], []).
mergesort([A], [A]).

mergesort([A, B | Rest], S) :-
  divide([A, B | Rest], L1, L2),
  mergesort(L1, S1),
  mergesort(L2, S2),
  my_merge(S1, S2, S).

divide([], [], []).
divide([A], [A], []).

divide([A, B | R], [A | Ra], [B | Rb]) :-  divide(R, Ra, Rb).

my_merge(A, [], A).
my_merge([], B, B).

my_merge([A | Ra], [B | Rb], [A | M]) :-
  A =< B,
```

```
  my_merge(Ra, [B | Rb], M).
my_merge([A | Ra], [B | Rb], [B | M]) :-
  A > B,
  my_merge([A | Ra], Rb, M).
```

References

[1] Endriss, Ulle. *An Introduction to Prolog Programming*, Institute for Logic, Language and Computation

[2] Literate Programs wiki contributors. *Merge sort (Prolog)*, Literate Programs wiki (`http://en.literateprograms.org/Merge_sort_(Prolog)`)