Wprowadzenie do Sztucznej Inteligencji
Laboratorium lista 0.2
Elementy języka PROLOG: reguły i rekurencja

Przemysław Kobylański

Część I
Wprowadzenie

1 Reguły

Przypomnijmy z poprzedniej listy zadań fakty z pliku `olimp.pl`:

```
rodzice(uranus, gaia, rhea).
rodzice(uranus, gaia, cronus).
rodzice(cronus, rhea, zeus).
rodzice(cronus, rhea, hera).
rodzice(cronus, rhea, demeter).
rodzice(zeus, letom, artemis).
rodzice(zeus, leto, apollo).
rodzice(zeus, demeter, persephone).
```

Aby zdefiniować predykaty `ojciec/2` i `matka/2` można użyć następujących dwóch reguł:

```
ojciec(X, Y) :- rodzice(X, _, Y).
matka(X, Y)   :- rodzice(_, X, Y).
```

Reguła ma postać

Head :- Body.

Gdzie `Head` jest formułą atomową wyrażającą warunek, który zachodzi **jeśli** spełniony jest warunek `Body`, przy czym warunek `Body` może być takiej postaci jak warunki w zapytaniach:

- może być formułą atomową;
- może być koniunkcją warunków oddzielonych przecinkami;
- może być alternatywą warunków oddzielonych średnikami;
- może być postaci `W1 -> W2; W3` i należy go rozumieć następująco:

  > *Jeśli spełniony jest warunek W1, to sprawdzić czy spełniony jest warunek W2. W przeciwnym przypadku sprawdzić czy spełniony jest warunek W3.*

Poniżej zdefiniowano kilka predykatów wykorzystujących predykaty `ojciec/2` i `matka/2`:

```
rodzic(X, Y) :- ojciec(X, Y).
rodzic(X, Y) :- matka(X, Y).
dziadek(X, Z) :- ojciec(X, Y), rodzic(Y, Z).
babcia(X, Z) :- matka(X, Y), rodzic(Y, Z).
```

2 Poszukiwanie odpowiedzi

Przedstawimy na przykładzie jak system PROLOG poszukuje odpowiedzi na zadane pytanie.

Należy zaznaczyć, że **formalny** opis działania programu w PROLOGU wymaga znajomości SLD-rezolucji będącej odmianą zasady rezolucji. Na potrzeby naszego kursu wystarczy jednak **nieformalny** opis zamieszczony poniżej.

Załóżmy, że chcemy poznać która bogini `X` była babcią Zeusa:

```
?- babcia(X, zeus).
```

Warunek `babcia(X, zeus)` stanowi cel do udowodnienia. System poszukuje w programie czy zna definicję predykatu `babcia/2` i znajduje regułę:

```
babcia(X, Z) :- matka(X, Y), rodzic(Y, Z).
```

W PROLOGU zasięgiem zmiennej jest jeden fakt, jedna reguła lub jedno zapytanie. Z tego powodu zmienna `X` z zapytania i zmienna `X` z powyższej reguły są różnymi zmiennymi. System PROLOG aby rozróżnić zmienne stosuje przemianowanie. My w tak krótki przykładzie nie będziemy przemianowywać zmiennych ale trzeba zdawać sobie sprawę, że jest to konieczne dla poprawnego działania programu.

Warunek `babcia(X, zeus)` zostaje zastąpiony zgodnie z definicją i po podstawieniu `zeus` za zmienną `Z`. Uzyskujemy nowy cel: `matka(X, Y), rodzic(Y, zeus)`.

Warunek `matka/2` jest zdefiniowany jest regułą:

```
matka(X, Y) :- rodzice(_, X, Y).
```

Opierając się na powyższej regule możemy zastąpić cel `matka(X, Y), rodzic(Y, zeus)` nowym celem: `rodzice(_, X, Y), rodzic(Y, zeus)`.

W pierwszym fakcie definiującym warunek `rodzice/3` zapisano, że `rodzice(uranus, gaia, rhea)`, zatem po podstawieniu za `X` wartości `gaia` a za `Y` wartości `rhea` otrzymujemy nowy cel: `rodzic(rhea, zeus)`.

Warunek `rodzic/2` zdefiniowany jest za pomocą dwóch reguł:

```
rodzic(X, Y) :- ojciec(X, Y).
rodzic(X, Y) :- matka(X, Y).
```

i każda z nich musi być rozpatrzona osobno.

System PROLOG zabiera się w pierwszej kolejności za regułę zamieszczoną wcześniej w tekście programu (bliżej jego początku).

Zatem aby sprawdzić czy `rhea` jest rodzicem `zeusa` zostanie sformułowany nowy cel do udowodnienia: `ojciec(rhea, zeus)`.

Reguła definiująca warunek `ojciec/2`:

```
ojciec(X, Y) :- rodzice(X _, Y).
```

wymaga odszukania wśród faktów definiujących warunek `rodzice/3` takiego, w którym na pierwszej pozycji (pierwszym argumentem) jest stała `rhea`.

Nie ma takiego faktu zatem skorzystanie z reguły

```
rodzic(X, Y) :- ojciec(X, Y)
```

skończyło się niepowodzeniem i system PROLOG przystępuje do próby skorzystania z drugiej reguły definiującej warunek `rodzic/2`:

```
rodzic(X, Y) :- matka(X, Y).
```

Tym razem nowym celem jest: `matka(rhea, zeus)`.

Zgodnie z regułą:

```
matka(X, Y) :- rodzice(_, X, Y).
```

konieczne jest odszukanie faktu definiującego warunek `rodzice/3`, który na drugiej pozycji miałby stałą `rhea` a na trzeciej stałą `zeus`.

Jest taki fakt `rodzice(cronus, rhea, zeus)` zatem cel został udowodniony i może pojawić się odpowiedź:

```
X = gaia
```

Jako ciekawostkę można podać, że jeśli poprosimy system PROLOG o znalezienie drugiej odpowiedzi, to po raz drugi otrzymamy tę samą odpowiedź:

```
?- babcia(X, zeus).
X = gaia ;
X = gaia ;
false.
```

Nie jest to błędem, ponieważ system raz wywnioskował, że Gaia jest babcią jako matka matki Zeusa a drugi raz jako matka ojca Zeusa (rodzice Zeusa byli rodzeństwem).

3 Rekurencja

W języku PROLOG nie ma instrukcji pętli jaką znamy z języków imperatywnych (algorytmicznych). Naturalnym sposobem powtórzenia sprawdzania warunku jest jego wywołanie rekurencyjne.

Rozpatrzmy warunek `przodek(X, Y)`, który jest spełniony gdy `X` jest przodkiem dla `Y`.

Pierwszą myślą jak wyrazić ten warunek w postaci programu jest napisanie następujących reguł:

```
przodek(X, Y) :- rodzic(X, Y).
przodek(X, Y) :- rodzic(X, Z1), rodzic(Z1, Y).
przodek(X, Y) :- rodzic(X, Z1), rodzic(Z1, Z2), rodzic(Z2, Y).
...
```

Wadą takiego podejścia jest to, że powyższa definicja wymaga nieskończenie wielu reguł.

Możemy jednak zauważyć, że **przodek** to albo **rodzic** albo **rodzic przodka**.

Zapisać możemy to w postaci następujących dwóch reguł:

```
przodek(X, Y) :- rodzic(X, Y).
przodek(X, Y) :- rodzic(X, Z), przodek(Z, Y).
```

Dla przykładu zapytajmy się czyim przodkiem był Cronus:

```
?- przodek(cronus, X).
X = zeus ;
X = hera ;
X = demeter ;
```

```
X = artemis ;
X = apollo ;
X = persephone ;
X = persephone ;
false.
```

Tym razem dwukrotnie otrzymaliśmy odpowiedź, że był on przodkiem Persephone a to dlatego, że rodzice Persephone byli rodzeństwem.

4 Funkcje rekurencyjne

Przedstawimy teraz kilka wybranych przykładów funkcji rekurencyjnych.

4.1 Silnia

Zwróć uwagę w poniższej definicji predykatu `fact/2` na obliczenie `N-1` przed wywołaniem rekurencyjnym `fact(N1, F1)` i obliczenie `N*F1` po powrocie z rekurencyjnego wywołania.

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

Przykładowe obliczenia:

```
?- fact(10, X).
X = 3628800 .

?- fact(100, X).
X = 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000000 .
```

4.2 Ciąg Fibonacciego

Podobnie jak w przykładzie z obliczaniem silni musimy obliczyć nowe wartości parametrów przed wywołaniem rekurencyjnym, natomiast wynik obliczyć po powrocie z wywołania rekurencyjnego:

```
fib(0, 0).
fib(1, 1).
fib(N, F) :- N > 1, N1 is N-1, N2 is N-2, fib(N1, F1), fib(N2, F2), F is F1+F2.
```

Przykładowe obliczenia:

```
?- fib(15, X).
X = 610 .

?- fib(20, X).
X = 6765 .

?- fib(25, X).
X = 75025 .
```

Zwróć uwagę na długi czas poszukiwania odpowiedzi na trzecie pytanie.

Czas poszukiwania odpowiedzi i liczbę kroków wnioskowania do tego potrzebnych można sprawdzić korzystając z predykatu `time/1`:

```
?- time(fib(15, X)).
% 3,945 inferences, 0.001 CPU in 0.001 seconds (70% CPU, 6013720 Lips)
X = 610 .

?- time(fib(20, X)).
% 43,780 inferences, 0.144 CPU in 0.147 seconds (98% CPU, 305017 Lips)
X = 6765 .

?- time(fib(25, X)).
% 485,568 inferences, 23.148 CPU in 23.219 seconds (100% CPU, 20976 Lips)
X = 75025 .
```

Jak widać obliczenie piętnastego wyrazu ciągu Fibonacciego trwało 0.001 sekundy, dwudziestego wyrazu 0.144 sekundy a dwudziestego piątego wyrazu aż 23.148 sekund gdyż potrzebnych było 485568 kroków wnioskowania.

Czy pamiętasz z wcześniejszych kursów dlaczego tak długo trwa wyliczanie wartości wyrazów ciągu Fibonacciego ze wzoru rekurencyjnego? W punkcie o tablicowaniu wyników pokażemy jak w prosty sposób temu zaradzić.

5 Styl programowania

Podane wcześniej przykłady predykatów nie były napisane zgodnie z zasadami dobrego stylu programowania. Otóż przyjęło się warunki z ciała reguły pisać w kolejnych wierszach stosując wcięcie.

Dla przykładu predykat `przodek/2` powinien zatem być zapisany następująco:

```
przodek(X, Y) :-
    rodzic(X, Y).
przodek(X, Y) :-
    rodzic(X, Z),
    przodek(Z, Y).
```

Natomiast predykat `fib/2` następująco:

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1,
    N1 is N-1,
    N2 is N-2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1+F2.
```

6 Tablicowanie wyników

Powrócimy do przykładu z ciągiem Fibonacciego. Tym razem napiszemy predykat `fib2/2`, który będzie zachowywał się w sposób bardziej "inteligentny". Otóż jeśli ma on policzyć jakiś wyraz ciągu Fibonacciego, to najpierw będzie sprawdzał czy już liczył ten wyraz i jeśli tak, to odda wcześniej policzoną wartość a jeśli nie, to policzy wartość po raz pierwszy i zapamięta ją aby nie musieć liczyć po raz drugi.

Już raz policzone wyniki będziemy zapamiętywać dodając nowe fakty, zatem predykat `fib2/2` musi być zadeklarowany jako dynamiczny:

```
:- dynamic fib2/2.

fib2(0, 0).
fib2(1, 1).
fib2(N, F) :-
    N > 1,
    N1 is N-1,
    N2 is N-2,
    fib2(N1, F1),
    fib2(N2, F2),
    F is F1+F2,
    asserta(fib2(N, F)).
```

Jak widać jedyną różnicą w definicji powyższego predykatu jest dodanie zapamiętania faktu po wyliczeniu wyniku. Proszę zwrócić uwagę na konieczność użycia predykatu `asserta/1` gdyż dodawanie na końcu definicji predykatem `assertz/1` mijałoby się z celem (najpierw PROLOG korzystałby z reguły zanim zauważyłby dodane na końcu fakty).

O tym, że faktycznie obliczenia przyspieszyły można przekonać się analizując poniższy dialog:

```
?- time(fib2(20, X)).
% 96 inferences, 0.000 CPU in 0.000 seconds (75% CPU, 2000000 Lips)
X = 6765 .

?- time(fib2(30, X)).
% 50 inferences, 0.000 CPU in 0.000 seconds (71% CPU, 943396 Lips)
X = 832040 .

?- time(fib2(50, X)).
% 100 inferences, 0.000 CPU in 0.000 seconds (77% CPU, 1219512 Lips)
X = 12586269025 .

?- time(fib2(100, X)).
% 250 inferences, 0.000 CPU in 0.000 seconds (82% CPU, 1111111 Lips)
X = 354224848179261915075 .

?- time(fib2(1000, X)).
% 4,500 inferences, 0.007 CPU in 0.008 seconds (91% CPU, 606714 Lips)
X = 434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288 75 .

?- time(fib2(1000, X)).
% 0 inferences, 0.000 CPU in 0.000 seconds (64% CPU, 0 Lips)
X = 434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288 75 .
```

Zwróć uwagę, że kiedy po raz drugi zapytano się o o tysięczny wyraz ciągu Fibonacciego, to odpowiedź uzyskano natychmiast.

Część II
Zadania i polecenia

Zadanie 1

Załóżmy, że dany jest predykat `przodek/2`. Napisz predykat `krewny/2`, który wyraża relację między dwiema osobami, w której są tylko te pary dwóch różnych[1] osób, które mają wspólnego przodka.

Zadanie 2

Załóżmy, że warunek `is_a(Object, Class)` wyraża relację między obiektem a klasą do której ten obiekt należy, warunek `a_kind_of(SubClass, SuperClass)` wyraża relację między podklasą a nadklasą, oraz warunek `has(Class, Property)` wyraża relację między klasą a własnością jaką posiadają obiekty z tej klasy.

Utwórz plik `objects.pl` zawierający następujące fakty:

```
is_a(fig1, square).
is_a(fig2, circle).
a_kind_of(square, figure).
a_kind_of(circle, figure).
has(square, size).
has(circle, radius).
has(figure, color).
```

Dopisz do pliku `objects.pl` definicję warunku `subclass(SubClass, Class)`, który zachodzi między podklasą a jej nadklasą (bezpośrednio lub pośrednio gdyż podklasa podklasy jest również podklasą).

Dopisz do pliku `objects.pl` definicję warunku `super_has/2`, który zachodzi między obiektem a własnością jaką on posiada (uwzględnij dziedziczenie własności w podklasach).

Na przykład w przypadku faktów z pliku `objects.pl` poprawne są następujące odpowiedzi:

```
?- super_has(fig1, X).
X = size ;
X = color ;
false.

?- super_has(X, radius).
X = fig2 ;
false.

?- super_has(X, color).
X = fig1 ;
X = fig2 ;
false.
```

[1] Warunek różne można sprawdzić predykatem `\=/2`.

Zadanie 3

Napisz w PROLOGU definicję funkcji[2] Ackermana:

$$A(m,n) = \begin{cases} n+1 & \text{gdy } m = 0 \text{ i } n \geq 0, \\ A(m-1,1) & \text{gdy } m > 0 \text{ i } n = 0, \\ A(m-1, A(m,n-1)) & \text{gdy } m > 0 \text{ i } n > 0 \end{cases}$$

Oblicz wartość powyższej funkcji dla $m = 4$ i $n = 1$. Czy udało się policzyć wynik?

Zastosuj tablicowanie wyników i powtórz powyższe obliczenia.

[2]Pamiętaj, że w PROLOGU funkcja to relacja między argumentami a wartością.