Аритметика

Средствата на ПРОЛОГ, които разгледахме до момента са напълно достатъчни да реализират аритметика с числа (вж. предикатите `nat, sum, leq,...`). Така обаче всички сметки ще минават през интерпретатора на ПРОЛОГ и ще бъдат изключително бавни. Затова в ПРОЛОГ е предвидена възможност за смятане с аритметични изрази както във всеки друг език за програмиране.

Оператор наричаме едно- или двуаргументен предикатен или функционален символ, който се записва инфиксно, т.е. предикатният или функционалният символ е между аргументите си и не се използват скоби. Досега сме се запознали с оператора за унификация (=) и неговото отрицание (\=). ПРОЛОГ възприема записа `X = Y` като `'='(X,Y)`.

Аритметичните операции в ПРОЛОГ са оператори за улеснение на записа:

- `+` (събиране)
- `-` (изваждане)
- `*` (умножение)
- `**` (степенуване)
- `/` (деление на реални числа)
- `//` (целочислено деление)
- `mod` (остатък при целочислено деление).

Аритметичен израз е терм, в който са използвани аритметичните операции, които имат същия приоритет, както в математиката (и други езици за програмиране). Например:

```
X + 2 * (3 - 4 / 5.6 + ( 3 ** (Y mod 2) // 5)))
```

Аритметичният израз не се пресмята от ПРОЛОГ автоматично, а се възприема като терм. За да накараме ПРОЛОГ да изчисли стойността на аритметичен израз, използваме оператора `is` следния начин:

```
<променлива> is <аритметичен-израз>
```

Ако `<променлива>` няма стойност по време на оценката на горния атом, `<аритметичен-израз>` се пресмята и стойността му се присвоява на `<променлива>`. Ако в участват променливи, то те трябва да имат конкретни стойности в момента на оценяване на аритметичния израз, в противен случай се издава съобщение за грешка. Ако `<променлива>` има стойност или е число, то is извършва проверка дали стойността на `<аритметичен-израз>` съвпада със стойността на `<променлива>` и връща съответно Yes или No. Израз със is не се преудовлетворява.

```
?-X is 2+3*5.
X = 18
```

```
?-Y is X*5.
Error: X not instantiated
?-X is 2+3*5,Y is X*5.
X = 18
Y = 90;
No
?-X is 2+3*5, X is 18.
X = 18
Yes
?-1 is 2.
No
```

Друг начин за да накараме ПРОЛОГ да сметне аритметичен израз е да използваме някои от операторите за сравнение:

- `=:=` (равенство)
- `=\=` (неравенство)
- `<`
- `>`
- `=<`
- `>=`

Операторите за по-малко или равно и по-голямо или равно се пишат така, че да не приличат на "стрелкичка" - тя има значение на импликация в ПРОЛОГ. Операторите за сравнение се използват по следния начин:

```
<аритметичен-израз> <оператор> <аритметичен-израз>
```

И двата аритметични изрази се пресмятат, след което се сравняват и резултатът е Yes или No. Операторите за сравнение не се преудовлетворяват.

Аритметични генератори

Генератор наричаме предикат, който последователно при преудовлетворяване изброява със или без повторение всички елементи на дадено множество.

Примери:

```
int(0).
int(X) :- int(X1), X is X1+1.
```

`int` е пример за безкраен генератор, понеже би-могъл да се удволетворява безкрайно. При въпрос `?-int(X)` се генерират последователно естествените числа.

```
between(X,Y,X) :- X=<Y.
between(X,Y,Z) :- X<Y,X1 is X+1,between(X1,Y,Z).
```

`?-between(a,b,X).` генерира последователно всички естествени числа между конкретните стойност `a` и `b`. `between` е краен генератор.

Отрицание

В ПРОЛОГ е възможно да се записва отрицание на атом. В сила са обаче някои ограничения:

- отрицание може да се записва само в цел или в тялото на правило.
- отрицание на атома `A` се записва `not(A)` или още `\+ A`
- отрицанието not(A) се удовлетворява само ако атомът A поставен като цел не се удовлетворява за *никои* стойности на променливите си
- отрицанието not(A) не се удовлетворява само ако атомът A поставен като цел се удовлетворява за някои стойности на променливите си. При отрицание нямаме начин да разберем кои са тези стойности, затова казваме, че отрицанието не е намиращо.

Пример за използване на отрицание е предикатът diff, който проверява дали списък се състои от различни (в смисъл на унификация) елементи

```
diff([]).
diff([X|T]) :- not(member(X,T)),diff(T).
```

Отсичане

ПРОЛОГ предлага възможност да контролиране процеса на извод като "отрязвате" ненужните преудовлетворявания. Специалният символ cut (!) записан в цел или тялото на правило забранява връщането назад (бектрекинга) наляво от него. Можем да си мислим за това в термините на дърво на извод - cut отрязва клона на дървото на извод от мястото където е поставен.

За пример да разгледаме предиката `member`:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Нека например искаме да си отговорим на въпроса има ли в един списък число от 3 до 10, което е четно:

```
?-
between(3,10,X),member(X,[1,3,9,17,38,59,110,11,3,12]),X mod
2 =:= 0.
```

Очевидно отговорът на този въпрос трябва да е No. ПРОЛОГ ще работи по следния начин: за всяко число X от 3 до 10 (генерирано от between), ще претърси целия списък и *всеки път* когато намери X в списъка ще прави проверка за четност. Ако тази проверка е неуспешна (както е в нашия случай), ПРОЛОГ ще се върне и ще претърси списъка до края. Бихме могли

да си спестим претърсването на списъка до края в случай, че сме намерили числото X в списъка веднъж, а проверката за четност се е провалила. За целта е достатъчно да променим member по следния начин:

```
member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).
```

Така когато ПРОЛОГ намери дадено число в списъка, предикатът member няма да се преудовлетвори заради cut. По друг начин казано - cut отрязва пътя наляво и надолу от себе си. Така когато се установи, че например намереното число 3 не удовлетворява проверката за четност, member няма да продължи да претърсва списъка до края, а ще пропадне заради cut и ще се извърши връщане назад към between.

Cut в повечето случаи може да се спести, като се използва някаква проверка, например:

```
member(X,[X|_]).
member(X,[Y|T]) :- Y\=X,member(X,T).
```

Добрият стил на програмиране на ПРОЛОГ изисква да използвате ! само за оптимизация, да не разчитате на него за построяване на логиката на програмите си и да знаете винаги как да замените дадено използване на ! с негово еквивалентна проверка.

Отсичането и отрицанието са взаимно заменяеми, например, за да напишете еквивалент на оператора if P then Q else R бихте могли да използвате отрицание:

```
S :- P,Q.
S :- not(P),R.
```

или отсичане:

```
S :- P,!,Q.
S :- R.
```

т.е. ако P е вярно - Q и не се връщай повече назад и надолу; ако P не е вярно, ще попаднем във втората клауза; там можем да си спестим проверката not(P) - не бихме могли да попаднем долу, ако P беше вярно заради !.

Отрицанието може да се реализира чрез !:

```
not(A) :- A,!,fail.
not(A).
```

където `fail` е атом, който никога не се удовлетворява(той може от своя страна да се дефинира например така:

```
fail :- 1 = 2.
```