STEROWANIE PROCESEM WNIOSKOWANIA.

Predykat	Objaśnienie
true/0	zawsze spełniony, deterministyczny
fail/0	zawsze zawodzi, deterministyczny
cut/0 lub !	odcięcie; zawsze spełniony
not(W) lub \+W	spełniony, gdy warunek W zawodzi
repeat/0	zawsze spełniony, niedeterministyczny

I. Predykat `fail/0`

Predykat **`fail`** jest zawsze niespełniony. Ponieważ próba realizacji celu **`fail`** zawsze kończy się porażką, użycie tego predykatu powoduje wymuszenie nawrotu.

Predykat **`fail`** może być wykorzystany do wymuszania poszukiwania następnych (wszystkich) rozwiązań dla danego celu. Zademonstrujemy to na następującym przykładzie.

Dana jest baza:

```
kobieta(katarzyna).
kobieta(anna).
kobieta(maria).
kobieta(marianna).
kobieta(marta).
```

Zdefiniujemy predykat **`kobiety/0`** następująco:

```
kobiety :- kobieta (X), write(X), nl.
```

Dla celu

```
? - kobiety.
```

na ekranie zostanie wypisane tylko

```
katarzyna
Yes
? -
```

Dodając na końcu definicji **`kobiety`** predykat **fail** (zawsze niespełniony)

```
kobiety :- kobieta (X), write(X), nl,
                fail.
```

wymusimy nawroty, a tym samym na ekranie zostaną wypisane wszystkie kobiety z bazy

```
? - kobiety
   katarzyna
   anna
   maria
   marta
false
? -
```

Ponieważ predykat `fail` jest zawsze niespełniony realizacja całej procedury **`kobiety`** kończy się niepowodzeniem. Aby tego uniknąć dodajemy do tej definicji jeszcze jedną klauzulę-fakt: `kobiety.`

```
kobiety :- kobieta (X), write(X), nl,
              fail.
kobiety.
```

Wtedy otrzymamy:

```
? - kobiety
        katarzyna
        anna
        maria
        marta

     true
     ? -
```

II. Predykat odcięcia `cut ( ! )`

Predykat odcięcia **`cut`** jest zawsze spełniony i służy do ograniczania nawrotów. Zamiast **`cut`** używamy wygodniejszej i bardziej zwracającej uwagę postaci "**!**".

Wystąpienie predykatu odcięcia zmienia sposób wykonywania nawrotów w obliczeniach:

Jeżeli w trakcie procesu wnioskowania nastąpiło uzgodnienie odcięcia, to podczas nawrotu zostaną pominięte:

a) wszystkie alternatywne drogi poszukiwania rozwiązania podcelów znajdujących się przed znakiem odcięcia,

b) wszystkie następne klauzule procedury, w której odcięcie nastąpiło.

Pod względem znaczenia, można wyróżnić trzy główne obszary zastosowań predykatu odcięcia:

1. Z góry wiadomo, że pewne rozwiązania częściowe nie doprowadzą do zrealizowania celu głównego. Ten sposób użycia odcięcia wpływa jedynie na zwiększenie efektywności działania programu, a nie na jego wynik.
Takie zastosowanie odcięcia nazywamy zielonym.

Przyklad . Zademonstrujemy użycie tego sposobu odcięcia w procedurze wyznaczającej znak liczby.

Z definicji znaku liczby mamy:

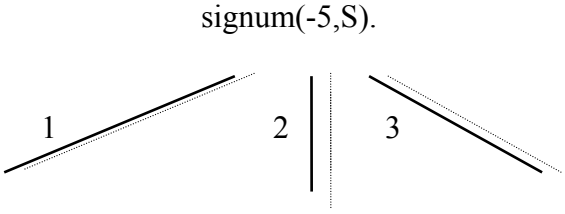
$$signum(x) = \begin{cases} -1 \text{ dla } x < 0, \\ 0 \text{ dla } x = 0, \\ 1 \text{ dla } x > 0. \end{cases}$$

Wiadomo, że zawsze zachodzi tylko jeden przypadek.

Odpowiednia procedura w Prologu jest np. postaci:

```
/*1*/  signum(X,-1) :- X<0.
/*2*/  signum(X,0) :- X=0.
/*3*/  signum(X,1) :- X>0.
```

Drzewo poszukiwania rozwiązania dla celu: ? - signum(-5,S). jest następujące:



X = -5, S = -1, -5 < 0 sukces

X = -5, S=0, -5 = 0 porażka

X = -5, S=1, -5 > 0, porażka

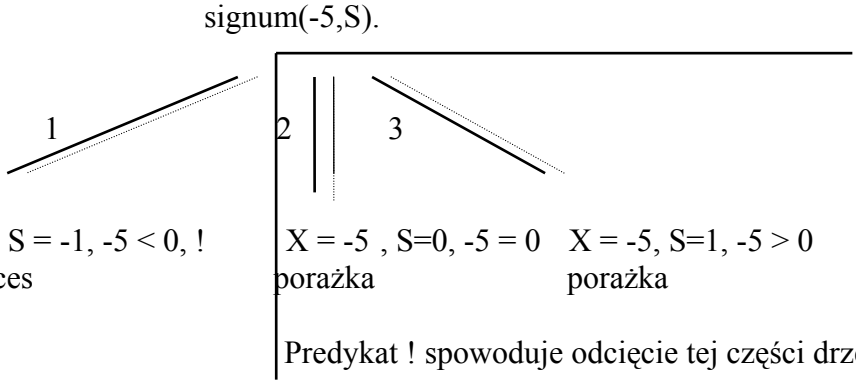
Widzimy, że jeżeli pierwsza droga doprowadziła do sukcesu, sprawdzanie pozostałych dróg nie jest już potrzebne, ponieważ w naszej sytuacji możliwy jest tylko jeden sukces.

Stosując predykat odcięcia następująco:

```
/*1*/  signum (X,-1) :- X < 0, !.
/*2*/  signum (X,0) :- X = 0, !.
/*3*/  signum (X,1) :- X > 0, !.
```

mamy gwarancję, że w przypadku znalezienia rozwiązania, inne rozwiązania nie są już poszukiwane.

Drzewo poszukiwań dla celu: ? - signum(-5,S). jest teraz następujące:



X = -5, S = -1, -5 < 0, ! sukces

X = -5 , S=0, -5 = 0 porażka

X = -5, S=1, -5 > 0 porażka

Predykat ! spowoduje odcięcie tej części drzewa

2. Logika problemu wymaga, by zapobiec w pewnych przypadkach możliwości poszukiwania rozwiązań alternatywnych. W takich sytuacjach użycie odcięcia jest

równoważne stwierdzeniu, że aparat wnioskowania wybrał właściwe rozwiązanie w danym konkretnym przypadku.
Odcięcie, które ogranicza liczbę rozwiązań nazywamy czerwonym. Odcięcie czerwone należy stosować z wielką ostrożnością!

Przykład. Następujące fakty określają dostępne w hurtowni towary:

```
/*  wyrob(nazwa, cena, ilosc)  */

     wyrob (towar1,10,10).
     wyrob (towar2,50,220).
     wyrob (towar3,90,500).
     wyrob (towar4,110,80).
```

Zakładając, że użytkownika interesuje zakup jakiegokolwiek towaru po cenie niższej niż 100, w ilości większej niż 200 sztuk określamy regułę:

```
     zakup_towaru (Nazwa) :- wyrob(Nazwa,Cena, Ilosc),
                             Cena < 100,
                             Ilosc > 200.
```

Dla celu

```
     ?- zakup_towaru(Nazwa).
```

otrzymujemy dwa rozwiązania:

```
     towar2,
     towar3.
```

Wprowadzając w regule zakup_towaru odcięcie:

```
     zakup_towaru (Nazwa) :- wyrob(Nazwa,Cena, Ilosc),
                             Cena < 100,
                             Ilosc > 200, !.
```

rozwiązania dla celu

```
     ?- zakup_towaru(Nazwa).
```

zostaną ograniczone do pierwszego napotkanego:

```
     towar2.
```

3. Użycie odcięcia w kombinacji z predykatem `fail`, który zawsze zawodzi, zazwyczaj jako dwa ostatnie podcele w klauzuli `(..., !, fail.`), co oznacza niepowodzenie wykonania całej procedury, a nie tylko klauzuli.
W tym przypadku jest to równoważne następującemu stwierdzeniu: jeśli podcele poprzedzające w ciele reguły predykat odcięcia zostały zrealizowane, oznacza to konieczność odrzucenie rozwiązania.

III. Negacja przez niepowodzenie – not

Może być stosowana wobec klauzul ukonkretnionych (bez zmiennych).

Jeżeli warunek W jest spełniony, to not(W) nie jest spełniony.
Jeżeli warunek W nie jest spełniony, to not(W) jest spełniony.

Zapis w Prologu:

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

Jeżeli prawdziwości faktu W nie można wykazać za pomocą dostępnych danych i reguł, to negacja faktu W jest prawdziwa. Zatem w programach prologowych zakłada się, że informacje nieobecne w bazie są fałszywe. Oznacza to, że negacja przez niepowodzenie nie jest dokładnym odzwierciedleniem negacji logicznej

Jeśli do bazy:

kobieta(katarzyna).
kobieta(anna).
kobieta(maria).
kobieta(marianna).
kobieta(marta).

dodamy regułę

mezczyzna(X):- not(kobieta(X)).

to dla celu

? – mezczyzna(X).

otrzymamy odpowiedź

false,

a dla celu

? – mezczyzna(ewa).

otrzymamy odpowiedź

true

Faktu kobieta(ewa) nie można wykazać na podstawie danych i reguł programu, więc cel `not( kobieta(ewa))` jest spełniony, czyli również cel `mężczyzna(ewa)` jest spełniony. Zatem zgodnie z regułami wnioskowania `ewa` jest mężczyzną.

Jeżeli do definicji kobieta/1, dołączymy definicję osoba/1

```
osoba(bernard).
osoba(adam).
osoba(piotr).
osoba(jan).
osoba(katarzyna).
osoba(anna).
osoba(maria).
osoba(marianna).
osoba(marta).

kobieta(katarzyna).
kobieta(anna).
kobieta(maria).
kobieta(marianna).
kobieta(marta).
```

wtedy definicja

```
mezczyzna(X):- osoba(X), not kobieta(X).
```

będzie poprawna.

Teraz otrzymamy odpowiedzi:

```
1 ?- mężczyzna(X).

     X = bernard ;
     X = adam ;
     X = piotr ;
     X = jan ;
     false

2 ?- mężczyzna(ewa).

     false
```

III. ORGANIZOWANIE PĘTLI.

Do organizowania pętli stosuje się predykat systemowy **repeat** o nieskończonym działaniu wyrażonym w następującej definicji:

```
repeat.
repeat :- repeat.
```

Jeżeli zastosujemy predykat **repeat** jako jeden z podcelów w jakiejś regule realizacja pierwszej klauzuli predykatu **repeat**, będącej faktem, zawsze kończy się sukcesem. Jeżeli któryś z następujących po **repeat** podcelów rozważanej reguły skończy się porażką w wyniku procesu nawracania następuje próba realizacji drugiej klauzuli predykatu **repeat**, będącej regułą - czyli wygenerowanie następnego celu do realizacji, tzn. następnego **repeat**. Proces ten będzie powtarzał się tak długo, aż wszystkie podcele reguły zostaną zrealizowane. Podcel, który kończy się porażką przy kolejnych próbach realizacji reguły zwykle jest ostatnią klauzulą reguły i zawiera warunek, którego spełnienie powoduje zakończenie pętli. Proces ten przypomina użycie instrukcji *repeat .... until ....* w Pascalu.

Zastosowanie predykatu **repeat** pokażemy na przykładzie programu **kalkulator1**, który dla podanego wyrażenia arytmetycznego wyznacza jego wartość. Program pyta, czy kontynujemy obliczenia. W przypadku odpowiedzi "nie" kończy działanie, w przeciwnym przypadku pobiera następne wyrażenie.

```
kalkulator1:-
     repeat,
     write('Podaj wyrazenie arytmetyczne'),
     read(E),
     V is E,
     write(V),nl,nl,
     write('Czy kontynuowac? (t/n):'),
     get(Odp),nl,
     (Odp=78; Odp=110).
```

Objaśnienia:

- Predykat **read** służy do czytania termów (w naszym przypadku będzie to wyrażenie arytmetyczne np. 2*sin(5)+7). Wprowadzany term **musi** być zakończony kropką.
- Klauzula "V is E" powoduje ukonkretnienie zmiennej V wartością wyrażenia E.
- Predykat **get** czyta pojedynczy znak i zapamiętuje go w postaci kodu Ascii. Po pojedynczym znaku nie stawiamy kropki.
- Ostatnia klauzula jest alternatywą celów (użycie średnika " ; " między nimi). Liczby 78 i 110 są kodami Ascii odpowiednio liter N i n.

Klauzula ta będzie spełniona, jeżeli nasza odpowiedź będzie "nie", tzn wtedy, gdy naciśniemy klawisz N lub n. Jest to więc warunek, którego spełnienie kończy działanie pętli.

Może się tak zdarzyć, że przedstawiona konstrukcja pętli nie jest możliwa do wykonania. Na przykład wtedy, gdy warunek musimy sprawdzić przed wykonaniem następnych operacji, które powinny znaleźć się w pętli. W takiej sytuacji możemy wymusić nawroty korzystając z predykatu **fail**. Zakończenie procesu iteracji, po rozpoznaniu informacji szczególnej osiągamy za pomocą predykatu odcięcia **!** .

Aby zilustrować tę sytuację zmienimy trochę nasz program kalkulator1. Program kalkulator2 będzie tak długo odczytywał wyrażenia aż napotka słowo “stop”.

```
kalkulator2:-
     repeat,
     write('Podaj wyrazenie arytmetyczne :'),
     read(E),
     sprawdz(E).

sprawdz(stop):-!.
sprawdz(E):-
     V is E,
     write(V),nl,nl,
     fail.
```

Polecenie :

Sprawdź działanie tego programu dla celu

? - kalkulator2.

Zakładamy, że wprowadzamy ciąg wyrażeń postaci, np:

2+3, 6/3, stop.

Trochę inna wersja kalkulator2:

```
kalkulator3:-
     repeat,
     write('Podaj wyrazenie arytmetyczne :'),
     read(E),
     (E=stop; (V is E, write(V),nl,nl,fail)).
```