Przykłady prostych zdań i ich reprezentacja w Prologu

Zdania wyrażające fakty można zapisać jako **klauzule** Prologu. Zwróćmy na początek uwagę, by wszystkie klauzule były pisane z małej litery. Praktyczne jest również upraszczanie polskiej fleksji:

```
Mam kolege Zdzicha.          kolega_zdzich.
Mam kolege Rycha.            kolega_rychu.

Zdzich ma ksiazke.           zdzich_ma_ksiazka.
Zdzich ma Mercedesa.         zdzich_ma_mercedes.
Zdzich ma Mariole.           zdzich_ma_mariola.

Zdzich lubi Mercedesa.       zdzich_lubi_mercedes.
Rychu lubi wino.             rychu_lubi_wino.
Zona Rycha lubi wino.        zona_rycha_lubi_wino.
```

Prolog może zapamiętać te fakty, a następnie pytany o nie odpowie twierdząco, a pytany o jakiekolwiek inne fakty odpowie przecząco.

```
?- kolega_zdzich.            ?- kolega_roman.
yes                          no
```


Jednak ten sposób wyrażania faktów posiada szereg wad. Na przykład, gdybyśmy zapytali o ostatni fakt na liście w formie: `rycha_zona_lubi_wino` to Prolog nie zauważyłby związku tego faktu z wcześniej wprowadzonym `zona_rycha_lubi_wino`:

```
?- zona_rycha_lubi_wino.
yes
?- rycha_zona_lubi_wino.
no
```

Dlatego do wyrażania faktów lepiej użyć **struktur**:

```
kolega(zdzich).
kolega(rychu).

ma(zdzich, ksiazka).
ma(zdzich, mercedes).
ma(zdzich, mariola).

lubi(zdzich, mercedes).
lubi(rychu, wino).
lubi(zona(rychu), wino).
```

```
?- kolega(zdzich).
yes
?- lubi(X, wino).
X = rychu ? ;
X = zona(rychu)
yes
```


Struktury i predykaty

Prolog pozwala na wprowadzanie faktów strukturalnych, co daje dużą swobodę. W korzystaniu z tej swobody, podobnie jak z innych swobód, warto stosować jednak pewną dyscyplinę, np.:

Czy zamiast pisać tak:

```
kolega(zdzich).
kolega(rychu).

ma(zdzich, ksiazka).
ma(zdzich, mercedes).
ma(zdzich, mariola).

lubi(zdzich, mercedes).
lubi(rychu, wino).
lubi(zona(rychu), wino).
```

nie moglibyśmy równie dobrze tak:

```
kolega(zdzich).
rychu(kolega).

zdzich(ma, ksiazka).
mercedes(ma, zdzich).
ma(mariola, zdzich).

zona(rychu, lubi, wino).
zona(rychu, lubi(wino)).
lubi(zona(rychu(wino))).
lubi(zona, rychu, wino).
```

Odpowiedź: formalnie owszem, praktycznie nie bardzo.

Struktura najwyższego poziomu (zewnętrzna) traktowana jest przez Prolog jako symbol **relacji** wyrażającej jakiś związek między jej argumentami.

Struktury wewnętrzne (mogą być dowolnie zagnieżdżone) traktowane są jako **funkcje** określające obiekt pozostający w zależności od innych obiektów.

Zapis relacji wyraża fakt logiczny, i nazywany jest **predykatem**. Przekształcając fakty wyrażone w normalnym języku na predykaty logiczne najczęściej używamy orzeczenia zdania (czasownika) jako symbolu predykatu. Podmiot staje się argumentem predykatu, podobnie jak dopełnienia.

Zatem predykat powinien mieć stałą liczbę argumentów, które powinny mieć swoje określone role, aczkolwiek mogą istnieć podobne predykaty z różną liczbą argumentów, np.:

```
ma2( kto, co )
ma3( kto, co, kiedy )
ma4( kto, co, kiedy, gdzie )
```

W Prologu takie predykaty mogą mieć jednakową nazwę, ponieważ Prolog potrafi je odróżnić po liczbie argumentów: `ma/2, ma/3, ma/4`

Zmienne w Prologu

Symbol termu rozpoczynający się wielką literą (lub podkreślnikiem _) jest zawsze zmienną w Prologu. Symbol predykatu nie może być zmienną i nie może zaczynać się wielką literą. Zmienna w aksjomacie jest traktowana jako kwantyfikowana uniwersalnie, a zmienna w zapytaniu jako kwantyfikowana egzystencjalnie. Zakresem zmiennych jest cała klauzula w której występują.

Zawartość bazy danych:

```
/* some like wine */
likes(ed, wine).
likes(wife(ed), wine).

/* everyone likes beer */
likes(X, beer).

/* dick likes his merc */
likes(dick, mercedes).
```

Odpowiadanie na pytania:

```
?- likes(X, wine).
X = ed ? ;
X = wife(ed)
yes
?- likes(ed, X).
X = wine ? ;
X = beer
yes
?- likes(X, beer), likes(X, mercedes).
X = dick ? ;
no
```

W pierwszych dwóch zapytaniach zmienna X jest za każdym razem inną zmienną. Natomiast w trzecim zapytaniu (o piwie i mercedesie), są dwa wystąpienia tej samej zmiennej X, więc muszą mieć tę samą wartość.

Operator unifikacji

Operator unifikacji = porównuje operandy. Jeśli obie są termami stałymi, zwraca logiczny wynik porównania (identyczne czy nie). Jeśli jeden lub oba operandy są zmiennymi, wtedy wynik jest zawsze prawdą, z efektem ubocznym przypisania zmiennej stałego operandu. Gdy oba operandy były zmiennymi to pozostają nimi, ale są zunifikowane, czyli muszą mieć równą wartość w przyszłości.

```
?- wife(ed) = meg.
no

?- wife(ed) = X.
X = wife(ed)
yes

?- wife(X) = wife(Y).
Y = X
yes
```

```
/* no artihmetic */
?- 2 + 3 = 5.
no

?- 2 + 3 = X.
X = 2+3
yes

/* dont try */
?- father(son(X)) = X.
```

Jak widać w przykładach po lewej stronie, unifikacja jest elastyczna i porównuje wyrażenia strukturalnie, przypisując zmienne by dopasować wyrażenia. Jednak żadne obliczenia arytmetyczne nie są wykonywane.

Wprowadzanie faktów i odpowiadanie na pytania

Zapisane w postaci klauzul fakty można wprowadzić do Prologu, który przyjmuje je jako **aksjomaty**, umieszcza (po kolei) w swojej bazie danych, i zaczyna w nie wierzyć (bezgranicznie).

Można również wprowadzać fakty (w tym samym formacie, zakończone kropką) jako zapytania, na które Prolog ma odpowiedzieć. W czasie normalnej pracy Prolog jest właśnie w takim trybie odpowiadania na pytania. Aby wprowadzić aksjomaty używamy specjalnego predykatu `consult/1`, który wczytuje fakty z podanego pliku, albo z wejścia: `consult(user)`. Pojedyncze fakty można również wprowadzać predykatami `asserta` i `assertz` (patrz dalej).

Prolog odpowiada na pytania przez przeszukiwanie swojej bazy danych, w kolejności wprowadzonych aksjomatów, dopasowując predykat i kolejne argumenty zapytania do argumentów aksjomatów.

Posługiwanie się gprolog-iem

`gprolog` jest łatwo dostępnym interpreterem Prologu. Można go wywołać w taki sposób, żeby od razu na starcie wczytał fakty zawarte w określonym pliku:

```
> gprolog --init-goal "consult('zdzich2.pro')"
compiling zdzich2.pro for byte code...
zdzich2.pro compiled, 10 lines read - 1004 bytes written, 15 ms
GNU Prolog 1.2.18
By Daniel Diaz
Copyright (C) 1999-2004 Daniel Diaz
| ?- kolega(zdzich).

(1 ms) yes
```


`gprolog` posiada szereg rozszerzeń w odniesieniu do standardu Prologu, jak również szereg zmiennych konfiguracyjnych. Na przykład, w przypadku zapytania o fakt, którego symbol predykatu nie jest znany, `gprolog` domyślnie generuje błąd.

Można ustawić flagę konfiguracyjną, aby takie zapytanie otrzymywało po prostu odpowiedź negatywną:

```
| ?- jest_fajnie.
uncaught exception: error(existence_error(procedure,jest_fajnie/0),top_level/
| ?- set_prolog_flag(unknown, fail).

yes
| ?- jest_fajnie.
no
```

Ponieważ `set_prolog_flag` jest predykatem (jak wszystko w Prologu), więc wywołuje się go w trybie zadawania pytań. Próba wprowadzenia go z pliku w trybie `consult` byłaby równoważna próbie przedefiniowania predykatu wbudowanego, co jest niedopuszczalne.

SWI Prolog

Podobnie można wywołać inny wygodny interpreter Prologu — SWI Prolog:

```
> pl -f zdzich2.pro
% /home/witold/cla/ai/Prolog/zdzich2.pro compiled 0.00 sec, 2,768 bytes
Welcome to SWI-Prolog (Version 5.6.6)
Copyright (c) 1990-2005 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- kolega(X).
X = zdzich ;
X = rychu ;

No
```


Ćwiczenie

Uruchom jakiś system Prologu, wprowadź kilka prostych faktów (z argumentami oraz bez) za pomocą `consult(user)`. Zadając pytania sprawdź poprawność wprowadzonych danych. Za pomocą klawisza `;` (średnika) wymuś wyświetlanie wszystkich możliwych wartości jakiejś zmiennej.

Pod hasłem `"prolog tutorial"` wygoogluj kilka stron z wprowadzeniem do Prologu. Skopiuj znalezione tam proste przykładowe programy, np. `hanoi`, i spróbuj je uruchomić zgodnie z załączonymi wyjaśnieniami.

Na przykład:
`http://www.learnprolognow.org/`
`http://www.cs.bham.ac.uk/~pjh/prolog_course/se207.html`

Reguły

Oprócz faktów prostych, bezwarunkowych, możemy wyrażać w Prologu fakty zawierające spójnik implikacji (wstecznej):

```
lubi(zdzich, X) :- lubi(X, wino).
```

Fakty tego typu nazywa się **regułami**.

Prolog poszukuje odpowiedzi na zadawane pytania dopasowując pytanie do kolejnych faktów w bazie danych. Gdy pytanie dopasowuje się do faktu prostego, odpowiedź jest natychmiastowa. Gdy fakt jest regułą to Prolog próbuje dopasować pytanie do jej lewej strony. Gdyby to się udało, Prolog wywołuje się rekurencyjnie, aby udowodnić fakty po prawej stronie reguły. Sukces wywołania rekurencyjnego oznacza odpowiedź pozytywną na oryginalne pytanie.

```
likes(dick, X) :- likes(X, wine).
likes(dick, mercedes).
likes(dick, wine).
likes(ed, wine).
likes(wife(ed), wine).
```

```
?- likes(dick,X).
X = dick ;
X = ed ;
X = wife(ed) ;
X = mercedes ;
X = wine.
```


Spójniki i konotacje logiczne

Operator `:-` występujący w regułach można traktować jako spójnik, ponieważ pozwala on tworzyć klauzule złożone z prostych. W sensie logicznym odpowiada on implikacji skierowanej wstecz $\Leftarrow$. Implikację można stosować tylko w definicjach reguł (nie w zapytaniach), i to tylko raz.

W Prologu istnieją jeszcze spójniki koniunkcji $\wedge$, zapisywanej przecinkiem, oraz alternatywy logicznej $\vee$, zapisywanej średnikiem:

```
?- ma(zdzich, mercedes), ma(zdzich, alfa_romeo_8c).
No

?- ma(zdzich, mercedes); ma(zdzich, alfa_romeo_8c).
true
```

Koniunkcję i alternatywę można stosować tylko po prawej stronie reguł (w poprzednikach implikacji). Lewa strona reguły musi być termem atomowym. Taki formuły logiczne, bez alternatyw, albo z alternatywami wyłącznie negatywnych literałów, albo z najwyżej jednym literałem pozytywnym, nazywa się **klauzulami Horna**.

Zatem można stwierdzić, że Prolog operuje jedynie klauzulami Horna.

Wyjaśnienie proceduralne

Wyrażanie faktów połączonych koniunkcją jest równoważne zapisywaniu tych faktów oddzielnie (baza danych Prologu jest jedną wielką koniunkcją logiczną), a więc nie jest de facto potrzebne. Jednak wyrażanie faktów połączonych alternatywą daje możliwości, których nie da się bez alternatywy uzyskać:

```
ma(zdzich, pieniadze); ma(zdzich, alfa_romeo_8c).
```

Ten fakt wyraża ważną własność pieniędzy: albo się je trzyma, albo się je wydaje. Ale jak Prolog miałby korzystać z takiego faktu, w wyszukiwaniu odpowiedzi na zapytanie? Nie może ani skorzystać z części pierwszej, i odpowiedzieć, że Zdzich ma gotówkę, bo nie wie tego na pewno, ani z części drugiej, z tego samego powodu.

Takiego problemu nie ma, gdy alternatywa występuje po prawej stronie reguły:

```
jest_cool(X) :- ma(X, pieniadze); ma(X, alfa_romeo_8c).
```

Aby udowodnić, że ktoś jest „cool" (ktokolwiek, niekoniecznie nasz bohater Zdzich), wystarczy sprawdzić, że ma kasę, lub alfę.

Czyli można zapisywać te klauzule, z którymi radzi sobie algorytm wyszukiwania.

Negacja, a może raczej jej brak

W Prologu nie ma spójnika negacji. Jednak istnieje wbudowany predykat `not`, którego znaczenie można określić jako: „nie da się udowodnić, że ...".
W niektórych przypadkach można go używać w charakterze negacji, ale niekiedy daje on nieoczekiwane wyniki.

Zawartość bazy danych:

```
man(dick).
dog(spot).
```

```
?- man(X).
X = dick ?
yes
?- not(man(spot)).
yes
?- not(man(X)).
no
```

Można byłoby oczekiwać, że Prolog znajdzie indywiduum, które nie jest człowiekiem.

Można byłoby oczekiwać, że skoro dało się udowodnić `not(man(spot))` to tym bardziej powinno dać się udowodnić `not(man(X))`.

Jedno i drugie oczekiwanie zawodzi. Można jedynie wyciągnąć wniosek, że `not` jest dziwną, nieintuicyjną formą przeczenia.

Predykatu negacji `not` można używać tylko w zapytaniach, a nie w stwierdzeniach zapamiętywanych w bazie danych.

Wyjaśnienie dlaczego tylko taka forma przeczenia jest dostępna w Prologu pojawi się później, a na razie musimy przyjąć, że obowiązuje myślenie pozytywne, i staramy się nic nie negować.

Jednak zwróćmy uwagę, że sam Prolog jest naładowany myśleniem negatywnym, bo zaprzecza wszystkiemu, co nie jest dla niego oczywiste po sprawdzeniu swojej bazy danych. Tę własność, negowania wszystkiego co nie jest jawnie znane, nazywa się **założeniem świata zamkniętego** (*Closed-World Assumption, CWA*).

W wielu interpreterach Prologu predykat `not` nie występuje jako taki. Zamiast niego jest dostępny operator `\+` o takim samym działaniu.

Obliczenia na strukturach

To co nazywamy w Prologu strukturą, czyli zapis predykatu z argumentami, można traktować jako strukturę danych, i budować z ich użyciem obliczenia.

Rozważamy następującą arytmetykę, gdzie wprowadzamy liczby za pomocą symbolu `zero` i struktury `s(X)`, która oznacza następnik (następną liczbę po) `X`. Jest to tzw. arytmetyka Peano. Na przykład, liczbę 5 zapisujemy w postaci: `s(s(s(s(s(zero)))))`. Chcemy zdefiniować dodawanie za pomocą predykatu `dodaj(Skl1, Skl2, Suma)` prawdziwego zawsze, gdy `Suma` jest sumą podanych dwóch składników, i fałszywego w pozostałych przypadkach:

```
suma(zero, Skl2, Skl2).
suma(s(X), Skl2, s(Suma)) :- suma(X, Skl2, Suma).
```

Aby prowadzić obliczenia w tej arytmetyce musimy posługiwać się notacją następników, np. żeby obliczyć 3+4:

```
?- suma(s(s(s(zero))), s(s(s(zero)))), X).
X = s(s(s(s(s(s(zero))))))
```

Zdefiniowanie mnożenia wymaga trochę więcej zachodu, spróbuj!

Obliczenia „wstecz”

Zauważmy, że w zdefiniowanym predykacie `suma` dwa pierwsze argumenty stanowią dane, a trzeci argument stanowi wynik obliczeń. Jest tak, podobnie jak w innych językach programowania, w których funkcje mogą mieć argumenty typu „out” i zwracać w nich wyniki. Podobnie, jak w tych innych językach „funkcja” nie jest dokładnie funkcją w sensie matematycznym, tak w Prologu predykat nie jest dokładnie predykatem w sensie logicznym.

Jednak Prolog nie ma mechanizmu deklarowania, który argument jest typu „out”, zatem co by się stało, gdybyśmy zamiast zadawać proste pytania typu 3+4=? zaczęli zadawać równania do rozwiązania, typu 3+?=4:

```
?- suma(s(s(s(zero))), X, s(s(s(zero)))).
X = s(zero) ;
No
```

Dobrze, jedynym rozwiązaniem powyższego równania jest `s(zero)`, i nic innego. Ta zdolność do prowadzenia obliczeń „wstecz” jest efektem ubocznym prologowego algorytmu przeszukiwania bazy danych i dopasowywania wzorców.

Oczywiście możemy posunąć się dalej, i zadać pytanie, na które odpowiedź nie jest jednoznaczna: ?+?=4. Uzyskamy wszystkie rozkłady liczby 4 na składniki:

```
?- suma(X, Y, s(s(s(zero)))).

X = zero
Y = s(s(s(zero))) ;

X = s(zero)
Y = s(s(zero)) ;

X = s(s(zero))
Y = s(zero) ;

X = s(s(s(zero)))
Y = zero ;

No
```

Mając zdefiniowane mnożenie moglibyśmy dokonywać faktoryzacji liczb, a nawet wyciągać pierwiastki! Spróbuj.

Prawdziwe liczby

Prolog potrafi posługiwać się prawdziwymi liczbami, porównywać je, i obliczać wyrażenia liczbowe, choć to ostatnie robi niechętnie. Sprawdźmy to, wykorzystując operator porównania `=`.

```
?- 0 = 0.
Yes

?- 0 = 1.
No

?- 2+2 = 4.
No

?- 2+2 = X.
X = 2+2
Yes
```


Prolog uważa, że jego podstawowym zadaniem jest przeszukiwanie bazy danych i dopasowywanie termów, i nie będzie zawracał sobie głowy obliczaniem wartości, gdy któryś z termów jest wyrażeniem liczbowym. Zaklęciem, które zmusza Prolog do wykonania obliczeń jest operator `is`, który oblicza wyrażenie po prawej stronie i podstawia (lub porównuje) pod zmienną po lewej stronie:

```
rowna_sie2(X, Y) :- X1 is X, X1 = Y.
```

Mamy teraz wyniki dobre:

```
?- rowna_sie(2+2, 4).
Yes

?- rowna_sie(2+2, X).
X = 4
```

ale również nadal nieakceptowalne, albo błędy:

```
?- rowna_sie(2+2,2+2).
No

?- rowna_sie(X, 2+2).
ERROR: (user://4:136):
 is/2: Arguments are not sufficiently instantiated
```

Porażka w pierwszym przykładzie po prawej wynika z nieobliczenia drugiego argumentu. Jednak jak pokazuje ostatni przykład, obliczenia prowadzone przez `is` nie tolerują niepodstawionego argumentu po prawej stronie, zatem `is` musi być stosowane ostrożnie, po zbadaniu postaci posiadanych argumentów: czy są wartościami, czy wyrażeniami, czy podstawione, czy nie. Prolog posiada szereg mechanizmów do sprawdzania tej postaci (patrz dalej).

Operatory infiksowe

Zasadniczo Prolog stosuje zapis formuł (zwanych strukturami) w notacji funkcyjnej, czyli symbol operacji i lista argumentów w nawiasach okrągłych, oddzielonych przecinkami. Jest jednak dopuszczalne użycie składni operatorowej, czyli argumenty rozdzielone, poprzedzone, lub poprzedzające symbol operatora, bez nawiasów ani przecinków. Prolog dopuszcza zapis:

`a + b` jako równoważną alternatywę zapisu: `+(a,b)`

Wyrażenia te są całkowicie równoważne, a wręcz identyczne, ponieważ to pierwsze traktowane jest jako pewna dodatkowa forma zapisu, i konwertowane do postaci po prawej w czasie parsowania przez Prolog.

Operatory tego typu jak `+` można również definiować w programach, co pozwala na posługiwanie się dowolnymi symbolami operatorów. Na przykład możemy wprowadzić symbol prefiksowego unarnego minusa (lub przeczenia) oraz symbol infiksowego operatora potęgowania:

```
?-op( 9, fx, ~ ). /* operator minus (unarny) */
?-op(10, yfx, ^ ). /* operator potegowania */
```

Wtedy każde wyrażenie postaci `a ^ b` będzie przez Prolog konwertowane do postaci `^(a,b)` i obliczane zgodnie z istniejącymi definicjami predykatu `^`

Operatory porównania w Prologu

Równość albo równoważność posiada wiele oblicz w Prologu. Poza operatorem unifikacji `=`, który wykonuje porównanie strukturalne z unifikacją zmiennych, istnieją porównania numeryczne, które pozwalają obliczać wartości wyrażeń arytmetycznych. Wymagają one by obliczane numerycznie termy były w pełni podstawione, i miały wartość liczbową:

`X is Y` — operand prawostronny Y może być wyrażeniem arytmetycznym, którego wartość liczbowa jest dopasowana do operandu lewostronnego X, który może być zmienną

`X =:= Y` — wartości arytmetyczne wyrażeń X i Y są równe

`X =\= Y` — wartości arytmetyczne wyrażeń X i Y są różne

Ponadto, istnieją porównania strukturalne, które nie wyliczają wartości liczbowej, a wymagają pełnej, literalnej identyczności:

`X == Y` — termy X i Y są identyczne, mają identyczną strukturę i identyczne argumenty z dokładnością do nazwy, np. `X==Y` jest zawsze nieprawdą

`X \== Y` — termy X i Y nie są identyczne


```
?- 3+4 = 4+3.
no % structures differ
?- 3+4 = 3+4.
yes
?- X = 3+4.
X = 3+4
yes
?- 3+X = 3+4.
X = 4
yes

?- 3+4 == 4+3.
no
?- 3+X == 3+4.
no
?- +(3,X) == 3+X.
yes

?- 3+4 \== 4+3.
yes
```

```
?- X is 3+4.
X = 7
yes
?- X = 7, X is 3+4.
X = 7
yes
?- X is 3+4, X = 7.
X = 7
yes
?- 3+4 is 4+3.
no % left arg.must be unassigned var.
   % or evaluate to a number

?- 3+4 =:= 4+3.
yes % calculates both values
?- X =:= 3+4.
error % both args must have values
?- a =:= 3+4.
error % and they must be arithm.values

?- 3+4 =\= 4+3.
no
```


Listy

Prolog ma jedną prawdziwą strukturę danych jaką jest **lista**. Lista jest sekwencją elementów, które mogą być atomami, bądź listami. Listy zapisujemy w nawiasach kwadratowych oddzielając poszczególne elementy przecinkami, np.:

```
[a]
[X,Y]
[1,2,3,4]
[a,[1,X],[],[],a,[a]]   /* ta lista ma 6 elementow */
```

Listy można również zapisywać podając „głowę" (element początkowy) i „resztę" listy, co ma duże znaczenie gdy ta reszta zapisana jest za pomocą zmiennej, np.:

```
[a|R]             /* ta lista ma co najmniej 1 element,R moze byc []*/
[1|[2|[3|[4|[]]]]] /* dokladnie rowna liscie [1,2,3,4] */
[1|[2|[3|[4]]]]    /* inny sposob zapisu listy [1,2,3,4] */
[1,2|[3,4]]        /* inny dopuszczalny zapis listy [1,2,3,4] */
```

Listę w notacji [Głowa|Reszta] można również zapisać jako strukturę .(Głowa,Reszta) (nazwą termu jest kropka).

Pomimo iż większość współczesnych interpreterów Prologu posiada wiele operacji na listach (przykładowe predykaty zdefiniowane poniżej nazywają się odpowiednio: `member` i `append`), jest pouczające przestudiowanie rekurencyjnych implementacji podstawowych takich operacji.

Predykat `element` sprawdza, czy coś jest elementem listy:

```
element(X,[X|_]).
element(X,[_|Y]) :- element(X,Y).
```

Ten predykat złącza dwie listy i unifikuje z trzecim argumentem.

```
merge([],X,X).
merge([X|Y],Z,[X|Q]) :-
                    merge(Y,Z,Q).
```

Wypróbuj poniższe zapytania:

```
?- merge([a,b,c],[w,x,y,z],L).
?- merge([a,b],Y,[a,b,c,d]).
?- merge(Y,[c,d],[a,b,c,d]).
?- merge([b,c],Y,[a,b,c,d]).
?- merge(X,Y,[a,b,c,d]).
?- merge(X,Y,Z).
```

Spróbuj: napisz definicję predykatu określającego ostatni element listy.

Debugowanie programów

Prolog zawiera kilka predykatów wspomagających analizę programów i umożliwiających śledzenie ich wykonania:

`spy/1` — ustawia śledzenie wykonania danego predykatu, który można podać w formie: `pred/n` wyróżniając wersję o danej liczbie argumentów,

`trace/0` — włącza śledzenie wszystkiego,

`nospy/1, notrace/0` — kasuje śledzenie,

`nodebug/0` — kasuje wszystkie `spy`,

`debugging/0` — listuje wszystkie `spy`,

`listing/1` — wyświetla komplet definicji dla jednego konkretnego predykatu,

`listing/0` — wyświetla wszystkie definicje posiadanych predykatów (poza predykatami wbudowanymi, które nie mają definicji źródłowej).

Przykład

Można zdefiniować „zdanie" języka polskiego jako listę słów w szyku pasującym do gramatyki naszego pięknego języka, w uproszczonej wersji:

```
rzeczownik(adam).
rzeczownik(stolarz).
rzeczownik(murarz).
czasownik(muruje).
czasownik(hebluje).
podmiot(X) :- rzeczownik(X).
orzeczenie(X) :- czasownik(X).
zdanie([X,Y]) :- podmiot(X), orzeczenie(Y).
```

Ten schemat pozwala sprawdzać przykłady różnych konstrukcji, czy są zdaniami (wyłącznie z punktu widzenia gramatyki, nie wnikając w ich sens):

```
?- zdanie([stolarz, muruje]).

Yes
?- zdanie([hebluje, stolarz]).

No
```


Można oczywiście prosić o uzupełnienie częściowo podstawionego zdania, lub generować całe zdania:

```
?- zdanie(X).
X = [adam, muruje] ;
X = [adam, hebluje] ;
X = [stolarz, muruje] ;
X = [stolarz, hebluje] ;
X = [murarz, muruje] ;
X = [murarz, hebluje] ;
```

Jako ćwiczenie spróbuj rozwinąć powyższy schemat zdania tak, aby dopuszczał również zdania bardziej skomplikowane, z różnymi okolicznikami (miejsca, czasu), i/lub zdania złożone.

Uzupełnij zasób słów i wypróbuj swój programik na różnych mniej lub bardziej rzeczywistych konstrukcjach zdaniowych.

Ćwiczenie — permutacje

W ćwiczeniu z gramatyką, listy potrzebne były jedynie do tego, by zdania mogły być różnej długości. W wielu programach konieczne jest jednak analizowanie zawartości list i skuteczne nimi manipulowanie. Dobrym ćwiczeniem jest napisanie predykatu `permutacja(X,Y)`, który sprawdza, czy jego argumenty są listami, z których jedna jest permutacją drugiej, czyli listą składającą się z tych samych elementów (w tych samych ilościach), tylko być może w innej kolejności:

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

Yes
?- permutacja([a,a,c],[c,c,a]).

No
```

Spróbuj napisać taki predykat. Następnie sprawdź możliwość generacji wszystkich permutacji jakiejś listy przez uruchamianie predykatu z jednym argumentem niepodstawionym (zmienną).

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

Schemat: Generate and Test

Wiele programów w Prologu można sensownie napisać według pewnego przydatnego schematu. Rozważmy na początek prosty generator, generujący kolejne liczby naturalne, jeśli tylko jakiś predykat będzie o te liczby w kółko prosił:

```
liczba(0).
liczba(N) :- liczba(M), N is M + 1.
```

Aby zademonstrować jego działanie, czyli spowodować wielokrotne wznawianie obliczeń tego predykatu, możemy wykorzystać zeroargumentowy predykat `fail`, który po prostu zwraca fałsz:

```
liczba(N), write(N), nl, fail.
```

Możemy teraz budować programy składające się z generatora potencjalnych rozwiązań jakiegoś abstrakcyjnego zadania, i predykatu testującego, który jedynie sprawdza czy proponowany obiekt jest akceptowalnym rozwiązaniem:

```
generate(X), test(X), gotowe(X).
```


Na przykład, w celu generacji liczb pierwszych wystarczy generować po kolei liczby całkowite, i użyć predykatu sprawdzającego podzielność, a raczej jej brak:

```
% The sieve of Eratosthenes, from Clocksin & Mellish 2ed p.170
%       finding the prime numbers up to 98.

main :- primes(98, X), write(X), nl.

primes(Limit, Ps) :- integers(2, Limit, Is), sift(Is, Ps).

/* integers(F,T,L) puts the integers from F to T into list L */
integers(Low, High, [Low | Rest]) :-
        Low =< High, !, M is Low+1, integers(M, High, Rest).
integers(_,_,[]).

/* sift(L1,L2) sifts non-prime numbers from L1, puts rest into L2 */
sift([],[]).
sift([I | Is], [I | Ps]) :- remove(I,Is,New), sift(New, Ps).

/* remove(N,L1,L2) removes from L1 multiples of number N into L2 */
remove(P,[],[]).
remove(P,[I | Is], Nis) :- 0 is I mod P, !, remove(P,Is,Nis).
remove(P,[I | Is], [I | Nis]) :- not(0 is I mod P),!,remove(P,Is,Nis).
```


Zawieszanie i wznawianie obliczeń

http://www.inf.ed.ac.uk/teaching/courses/aipp/
http://www.inf.ed.ac.uk/teaching/courses/aipp/lecture_slides/07_Cut.pdf

Odcięcie

Odcięcie, zapisywane znakiem wykrzyknika (!), jest operatorem mającym wartość logiczną prawdy, ale jednocześnie blokującym mechanizm nawracania Prologu do dalszych punktów wyboru. Przyjrzyjmy się na przykładach, co to dokładnie oznacza:

Dla następujących definicji:

```
fakt(a).
fakt(b) :- !.
fakt(c).
```

Każdy z faktów a,b,c, jest indywidualnie spełniony, jednak gdy Prolog próbuje wszystkich definicji po kolei, i po drodze napotka operator odcięcia, to nie może już kontynuować obliczeń i zwraca odpowiedź negatywną.

```
?- fakt(a).

Yes
?- fakt(b).

Yes
?- fakt(c).

Yes
?- fakt(X).

X = a ;

X = b ;

No
```


Jak widać poniżej, obecność odcięcia w definicji fakt powoduje zakłócenie jego wywoływania przez inne fakty.

```
fakt2(X,Y) :- fakt(X), X = Y.

/**************************/

?- fakt2(X,a).
X = a
Yes

?- fakt2(X,b).
X = b
Yes

?- fakt2(X,c).
No
```

Zmiana kolejności sprawdzanych warunków opóźnia wykonanie odcięcia, co trochę pomaga, ale nadal odcina ono pewne rozwiązania.

```
fakt3(X,Y) :- X = Y, fakt(X).

/**************************/

?- fakt3(X,c).
X = c
Yes

?- fakt3(X,Y).
X = a
Y = a ;

X = b
Y = b ;

No
```

To po co jest nam właściwie potrzebny operator odcięcia?

Odcięcie — przypadek 1: utwierdza wybór reguły

Wyobraźmy sobie predykat sum_to służący do obliczania sumy liczb od 1 do jakiejś wartości. Drugi argument przeznaczony jest na wynik obliczeń.

```
sum_to( 1, 1 ).
sum_to( N, R ) :-
    N1 is N - 1,
    sum_to( N1, R1 ),
    R is R1 + N.
```

To rozwiązanie działa poprawnie, z wyjątkiem kilku przypadków specjalnych, na przykład, kiedy użytkownik wywoła program ze złymi danymi, albo kiedy naciśnie „;" zmuszając program do wznawiania obliczeń:

```
?- sum_to(5,X).

X = 15 ;
ERROR: (user://1:22):
        Out of local stack
?- sum_to(5,14).
ERROR: (user://1:27):
        Out of local stack
```


To jest przykład programu, który nie powinien w ogóle po obliczeniu jednego wyniku wznawiać obliczeń, ponieważ nie ma innej sumy liczb niż ta pierwotnie wyliczona. Można to uwzględnić za pomocą odcięcia (wersja po lewej):

```
sum_to( 1, 1 ) :- !.
sum_to( N, R ) :-
    N1 is N - 1,
    sum_to( N1, R1 ),
    R is R1 + N.
```

```
sum_to( N, 1 ) :- N < 1, !, fail.
sum_to( 1, 1 ).
sum_to( N, R ) :-
    N1 is N - 1,
    sum_to( N1, R1 ),
    R is R1 + N.
```

Pozostaje jeszcze przypadek, z którym ta wersja sobie nie radzi, gdy pierwszy argument jest od razu ujemny. Radzi sobie z tym wersja powyżej po prawej.

Okazuje się jednak, że istnieje proste rozwiązanie niewykorzystujące odcięcia, które rozwiązuje wszystkie powyższe problemy:

```
sum_to( 1, 1 ).
sum_to( N, R ) :-
    N > 1,
    N1 is N - 1,
    sum_to( N1, R1 ),
    R is R1 + N.
```


Odcięcie — przypadek 2: stwierdza fałszywość celu

Odcięcie — przypadek 3: odcina niepotrzebne możliwości

Odcięcie — problemy

Prolog — operacje na bazie danych

Normalnie w trakcie pracy Prolog jest w trybie odpowiadania na pytania. Fakty są dodawane do bazy danych Prologu przez użycie predykatu `consult`. Jednak w trakcie pracy można zarówno dodawać nowe fakty (atomowe i/lub struktury) do bazy wiedzy, jak również usuwać istniejące. Powoduje to jakby samo-modyfikację programu.

`asserta(term), assertz(term)` — dodaje fakt term do bazy danych, odpowiednio na początek i na koniec

`retract(term)` — kasuje fakt term z bazy danych, o ile w niej był

Uwaga: przy nawracaniu Prologu efekty działania tych operacji nie są kasowane, to znaczy poprzednie stany bazy danych nie są odtwarzane!.

Prolog — operacje na termach

```
var(term) /* prawdziwy jesli term jest niepodstawiona zmienna */

nonvar(term) /* odwrotnie niz var */

atom(term) /* czy term jest podstawionym literalem atomowym (nie stringiem) *

integer(term)

atomic(term) /* atom lub integer */

clause         /* C&M(4)p.115 */

functor         /* C&M(2)p.120(4)p.117 */

arg /* C&M(2)p.122(4)p.119 */

=.. /* C&M(2)p.173,123(4)p.120 */
```


Prolog — operacje wejścia/wyjścia

Wczytywanie i wypisywanie termów:

```
?- read(X).     /* wczytuje z terminala jeden term zakonczony
                   kropka '.' i podstawia go pod zmienna X;
                   na koncu pliku read zwraca end_of_file  */
?- write(X).    /* wypisuje na terminalu wartosc termu
                   aktualnie podstawiona pod zmienna X     */
?- nl.          /* wypisuje znak nowej linii na terminalu  */
```

Wczytywanie i wypisywanie znaków:

```
?- get(X)       /* czyta znak (w postaci numerycznego kodu znaku */
?- put(X)       /* pisze jeden znak, np. put(104) wypisuje znak 'h' */
```

Operacje na plikach:

```
?- tell('nowy')./* otwiera nowy plik o nazwie 'nowy' i przelacza
                   standardowe wyjscie na ten plik;
                   nastepne operacje wyjscia dzialaja na tym pliku */
?- told.        /* zamyka plik aktualnie otwarty i
                   przelacza standardowe wyjscie na terminal  */
```



```
?- see('stary').     /* otwiera istniejacy plik do zapisu     */
?- seen.             /* konczy czytanie z pliku i zamyka plik */
```

Czytanie całych plików w trybie definiowania aksjomatów:

```
?- consult(plik1).   /* w skrocie mozna: [plik1]  */
?- reconsult(plik2). /* w skrocie mozna: [-plik2] */
```

Przykład: zapisanie na pliku wszystkich aksjomatów definiujących predykaty 'ma' i 'lubi':

```
?- tell('program'), listing(ma), listing(lubi), told.
     /* predykat listing wypisuje na terminalu wszystkie posiadane
        klauzule; np.: listing, listing(ma), listing(ma/2)      */
```