

Operatory

Logiczne

Operator	Znaczenie	Przykład
,	koniunkcja (AND)	A, B – oba warunki muszą być spełnione
;	alternatywa (OR)	A; B – wystarczy, że spełniony jest jeden
\+	negacja (NOT)	\+ A – warunek A nie jest prawdziwy

Porównania

Operator	Znaczenie
=	unifikacja – sprawdza, czy dwa termy mogą być dopasowane
==	sprawdza, czy dwa termy są dokładnie takie same
==:	porównanie arytmetyczne (po obliczeniu wartości)
\==	negacja równości (symbolicznej)
=\=	negacja równości arytmetycznej
>	większy
>=	większy lub równy

is	oblicza wartość wyrażenia arytmetycznego po prawej stronie
-----------	---

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

```
X = 2+3.           % tylko dopasowanie, brak obliczenia
```

```
?- X is 2+3.
```

```
X = 5.             % wykonano obliczenie
```

```
?- 2+3 ==: 5.
```

```
true.
```

```
?- 2+3 = 5.
```

```
false.
```

Operator **is** wykonuje obliczenie, natomiast **=** tylko sprawdza zgodność struktury.

W Prologu porównania arytmetyczne nie są tym samym co unifikacja (**=**). Muszą być poprzedzone obliczeniem wartości po obu stronach.

Arytmetyczne

Operator	Znaczenie	Przykład	Wynik
+	dodawanie	?- X is 3 + 4.	X = 7
-	odejmowanie lub negacja	?- X is 5 - 2. ?- X is -3.	X = 3 X = -3
*	mnożenie	?- X is 6 * 7.	X = 42
/	dzielenie rzeczywiste	?- X is 7 / 2.	X = 3.5
//	dzielenie całkowite (ang. integer division)	?- X is 7 // 2.	X = 3
mod	reszta z dzielenia (modulo)	?- X is 10 mod 3.	X = 1
rem	reszta z dzielenia z zachowaniem znaku dzielnej	?- X is -10 rem 3.	X = -1
abs(X)	wartość bezwzględna	?- X is abs(-5).	X = 5
sqrt(X)	pierwiastek kwadratowy	?- X is sqrt(16).	X = 4.0
** lub ^	potęgowanie	?- X is 2 ** 3.	X = 8.0
round(X)	zaokrąglenie do najbliższej liczby całkowitej	?- X is round(3.6).	X = 4
floor(X)	zaokrąglenie w dół	?- X is floor(3.6).	X = 3
ceiling(X)	zaokrąglenie w górę	?- X is ceiling(3.2).	X = 4
truncate(X)	obcięcie części ułamkowej	?- X is truncate(3.9).	X = 3
max(A,B)	zwraca większą z wartości	?- X is max(5,9).	X = 9
min(A,B)	zwraca mniejszą z wartości	?- X is min(5,9).	X = 5

Predykaty IN / OUT

Predykat	Co robi
write(X)	wypisuje wartość X
nl	przechodzi do nowej linii
tab(N)	dodaje N spacji
writeln(X)	wypisuje X i kończy nową linią (skrót dla write(X), nl)
format(Wzor, ListaArgumentow)	sformatowany tekst, np. format('Wynik: ~w~n', [X]).
Wzór	
~w	dowolna wartość (write)
~d	liczba całkowita (decimal)
~f	liczba zmiennoprzecinkowa
~s	ciąg znaków (string)
~n	nowa linia
~~	wypisz dosłowny znak ~

```
format('Student ~w ma ocenę ~w z przedmiotu ~w.~n', ['Jan', 5, 'Prolog']).
Student Jan ma ocenę 5 z przedmiotu Prolog.
```

W Prologu sposób zapisu słowa określa jego znaczenie logiczne:

Zapis	Znaczenie	Typ
jan	konkretny symbol (atom)	stała
Jan	zmienna (coś nieznanego, dopasowywanego)	zmienna
'Jan'	atom zaczynający się wielką literą lub zawierający znaki specjalne	stała
"Jan"	lista znaków (ciąg liter [74,97,110]), nie atom	lista kodów ASCII

Zasada praktyczna:

- Jeśli to dane (np. imię, miasto, kierunek) → pisz 'Jan' lub jan.
- Jeśli to zmienna → pisz Jan, Osoba, X itd.
- Jeśli potrzebujesz tekstu z polskimi znakami lub spacją, też użyj apostrofów.

```
miasto('Nowy Sącz').
uczelnia('WSiIZ w Rzeszowie').
```

Backtracking (nawracanie)

To mechanizm, dzięki któremu Prolog automatycznie przeszukuje wszystkie możliwe ścieżki logiczne, by znaleźć rozwiązanie zapytania. Jeśli jakaś ścieżka prowadzi do sprzeczności, program cofa się do ostatniego miejsca, w którym miał wybór, i próbuje innego dopasowania.

Za każdym razem, gdy interpreter napotyka wybór (więcej niż jedno możliwe dopasowanie), zapamiętuje tzw. **punkt powrotu (choice point)**. Jeśli dalsza ścieżka prowadzi do błędu lub **fail**, Prolog cofa się do ostatniego punktu i próbuje następnego dopasowania.

Wiele faktów – wiele rozwiązań:

```
owoc(jablko).
owoc(gruszka).
owoc(sliwka).
?- owoc(X).
```

Wynik:

```
X = jablko ;
X = gruszka ;
X = sliwka ;
false.
```

Każde ; oznacza nawrócenie — Prolog wraca i szuka kolejnego dopasowania.

Predykat fail

Predykat fail zawsze kończy się niepowodzeniem. Używa się go po to, by zmusić Prolog do cofnięcia się i poszukania kolejnych rozwiązań. Na pierwszy rzut oka wydaje się, że po prostu „kończy program błędem”. Ale w Prologu nie ma błędu w sensie „wyjątku” — jest niepowodzenie logiczne, i właśnie to wykorzystuje fail.

```
owoc(jablko).
owoc(gruszka).
owoc(sliwka).
wypisz_owoce :-
    owoc(X),
    write(X), nl,
    fail.
```

Zapytanie:

```
?- wypisz_owoce.
```

Wynik:

```
jablko
gruszka
sliwka
false.
```

Co się stało?

1. `owoc(X)` → pasuje `jablko`, wypisuje `jablko`.
2. `fail` → mówi: „to nie koniec, wróć do poprzedniego punktu i szukaj dalej.”
3. Interpreter wraca do `owoc(X)` i znajduje `gruszka`, wypisuje, znów `fail`, wraca... itd.
4. Dopiero gdy zabraknie kolejnych `owoc(X)` — cały predykat kończy się `false`.

Dlaczego **false** na końcu?

`fail` kończy się niepowodzeniem, więc cały `wypisz_owoce` również kończy się niepowodzeniem. W Prologu to nie „błąd”, tylko oznaka, że nie ma już żadnych nowych rozwiązań. Jeśli chcesz zakończyć bez `false`, możesz dodać na końcu pustą regułę jako „hamulec awaryjny”:

```
wypisz_owoce. % reguła zapasowa – po wyczerpaniu rozwiązań
```

Inny przykład:

```
kolor(czerwony).
kolor(zielony).
kolor(niebieski).
pokaz_kolory :-
    kolor(X),
    write('Kolor: '), write(X), nl,
    fail.
pokaz_kolory :- write('--- koniec listy ---'), nl.
```

Zapytanie:

```
?- pokaz_kolory.
Kolor: czerwony
Kolor: zielony
Kolor: niebieski
--- koniec listy ---
true.
```

Predykat odcięcia – !

Odcięcie (cut) mówi Prologowi: „Nie wracaj do poprzednich wyborów – to już ostateczna ścieżka” i jest to sposób na kontrolę backtrackingu.

Wersja bez odcięcia

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

Zapytanie:

```
?- max(5, 3, M).
M = 5 ;
false.
```

Prolog najpierw dopasuje pierwszą regułę, ale nadal spróbuje drugiej (nawet jeśli już znalazł dobre rozwiązanie).

Wersja z odcięciem

```
max(X, Y, X) :- X >= Y, !.
max(_, Y, Y).
```

Zapytanie:

```
?- max(5, 3, M).
M = 5.
```

Nie ma już drugiego dopasowania — ! odcina wszystkie alternatywne ścieżki. Interpreter, który natrafi na !, traktuje wszystko, co było wcześniej w tej regule, jako ostateczne — nie wraca, nawet jeśli dalej program mógłby znaleźć inne rozwiązanie.

Uwaga: ! nie sprawia, że wynik jest bardziej prawdziwy. ! tylko mówi Prologowi: nie cofaj się już, zaakceptuj to rozwiązanie nawet jeśli w innej gałęzi byłoby jakieś inne.

Zadania

1	<p>Utwórz nowy plik o nazwie <code>operacje.pl</code>:</p> <p>a) Zdefiniuj predykaty:</p> <pre>suma(A,B,S) roznica(A,B,R) iloczyn(A,B,M) iloraz(A,B,D)</pre> <p>b) Przetestuj je.</p>								
2	<p>Utwórz plik o nazwie <code>figury.pl</code>, gdzie zdefiniujesz predykaty obliczania pól i obwodów:</p> <p>a) Prostokąta b) Trójkąta c) Koła</p> <p>Dodaj walidację danych dla każdej z figur oraz odpowiednie predykaty <code>dane_figury</code>, która łączy te 3 informacje.</p>								
3	<p>Utwórz plik <code>kolory.pl</code>:</p> <p>a) Zdefiniuj fakt <code>kolor(...)</code> dla 4 różnych kolorów, b) Zdefiniuj fakt <code>przedmiot(...)</code> dla 5 różnych przedmiotów, c) Zdefiniuj predykat <code>przypisz(X, Y)</code> łączący przedmiot oraz kolor go określający. d) Przetestuj kod korzystając z narzędzia śledzenia (<code>trace.</code>) i obserwuj, jak Prolog przechodzi kolejne etapy nawracania.</p> <p>TIP: Zwróć uwagę na etapy <code>Call</code>, <code>Exit</code>, <code>Redo</code> i <code>Fail</code>.</p>								
4	<p>Utwórz plik <code>labirynt.pl</code> zawierający symulację fragmentu labiryntu znajdującego się poniżej. Określ fakty <code>droga(X,Y)</code>, oznaczających, że można przejść bezpośrednio z X do Y, a następnie prześledź proces <code>backtrackingu</code>:</p> <p>a) Czy istnieje ścieżka a-f? b) Gdzie można dotrzeć zaczynając w punkcie a?</p> <table border="1"><tr><td>a</td><td>c</td><td>e</td><td>f</td></tr><tr><td>b</td><td>d</td><td></td><td></td></tr></table> <p>Litery oznaczają miejsca, a cienkie krawędzie drzwi, przez które można przejść. Np. z pkt A można przejść bezpośrednio do B i C, ale do innych punktów trzeba już znaleźć drogę poprzez poprzedzające je miejsca.</p>	a	c	e	f	b	d		
a	c	e	f						
b	d								
5	<p>Mając bazę studentów różnych kierunków (<code>student(imie, kierunek)</code>) w pliku <code>studenci.pl</code> zdefiniuj predykat, który wypisze wszystkich informatyków a na koniec wypisze podsumowanie.</p> <p>TIP: Użyj <code>fail</code>, aby wymusić przejście przez całą bazę.</p>								

6	<p>Mając bazę wyników kolokwium z programowania (<code>wynik(imie, liczba_pkt)</code>) w pliku <code>kolokwium.pl</code> napisz predykat, który:</p> <ol style="list-style-type: none"> dla każdego studenta wypisze jego imię, punktację i ocenę opisową, zastosuje <code>!</code> do zatrzymania dalszych dopasowań po przydzieleniu oceny, użyje <code>fail</code>, by przejść przez wszystkich studentów, na końcu wypisze komunikat „Koniec raportu”.
7	<p>System wczytuje dane użytkowników, ale niektóre imiona zapisano różnie:</p> <pre>uzytkownik(jan). uzytkownik('Jan'). uzytkownik("Jan").</pre> <p>Napisz predykat, który dla każdego wpisu wypisze jego formę i typ, używając <code>write</code> i <code>format</code>, a na końcu wyświetli liczbę znalezionych przypadków (z użyciem akumulatora).</p> <p>TIP: <code>atom(X)</code> zwraca, czy dana wartość jest atomem, a <code>string(X)</code> czy tekstem.</p>

Zadania do prezentacji

1	<p>Kalkulator logiczny</p> <p>Należy stworzyć prosty system, który umożliwia wykonywanie podstawowych działań matematycznych: dodawanie, odejmowanie, mnożenie i dzielenie.</p> <p>Program ma przyjmować zapytania w postaci logicznej, np.:</p> <pre>?- dzialanie(+, 2, 3, X). X = 5.</pre> <p>lub:</p> <pre>?- kalkulator(/, 10, 2, W). W = 5.</pre> <p>Zadaniem programu jest rozpoznanie rodzaju działania, obliczenie wyniku oraz zwrócenie wartości w sposób poprawny logicznie.</p> <p>Wynik = <wartość>.</p> <p>TIP: Pamiętaj o zabezpieczeniu przed przypadkiem dzielenia przez zero. TIP: A co, kiedy podany operator jest nieznany?</p> <p>Wymagania rozszerzające (dla chętnych):</p> <ul style="list-style-type: none"> dodanie działania potęgowania (<code>^</code> lub <code>**</code>), obsługa operacji modulo (<code>mod</code>), implementacja „trybu odwrotnego” – użytkownik podaje wynik, a program ma odnaleźć brakujący argument (np. <code>?- dzialanie(+, 2, X, 7). → X = 5</code>).
---	---

Część prezentacyjna (do realizacji na kolejnych zajęciach):

Krótką prezentacją obejmującą:

- Opis działania programu – struktura reguł, sposób dopasowania operatorów.
- Omówienie użytych operatorów Prologa (is, =, =:=).
- Wskazanie różnicy między unifikacją a obliczeniem.
- Pokaz działania programu na wybranych przykładach (w SWI-Prologu).
- Dodatkowy komentarz: jak można by rozbudować ten kalkulator o nowe funkcje (np. pamięć ostatniego wyniku, lista operacji, wczytywanie z pliku).

Czas prezentacji: 3–5 minut

Forma: demonstracja działania kodu + krótka analiza na slajdach lub w edytorze kodu.

Wybierz jeden z predykatów z zadań laboratorium:

- Uruchom go normalnie, żeby pokazać jego zachowanie (w tym wypisywanie przez format lub write).
- Następnie uruchom go ponownie pod trace:
- Zanotuj fragment przebiegu, w którym widać:
 - Call: (Prolog próbuje spełnić cel),
 - Exit: (udało się),
 - Redo: (cofnięcie i próba kolejnej ścieżki),
 - Fail: (niepowodzenie i powrót).
- Wytłumacz, co wywołało Redo i Fail:
 - Czy był to fail. w kodzie?
 - Czy to był koniec faktów, więc Prolog szukał kolejnych dopasowań i się odbił od ściany?
 - Czy ! zatrzymało dalsze cofanie?
- Wytłumacz, jak zachowuje się wypisywanie (format, write, nl) w kontekście backtrackingu:
 - Czy linia wypisała się raz czy wiele razy?
 - Czy Prolog wypisuje to samo kilka razy, bo fail zmusza go do przejścia przez wszystkie rozwiązania?

Minimalny wymagany układ ich prezentacji / opisu:

- Krótki opis predykatu
- Kod predykatu w wersji końcowej
- Wyjście bez trace
- Fragment z trace
- Komentarz interpretacyjny

2