# Paradygmaty programowania

Każdy paradygmat to inny sposób myślenia o tym, jak opisać problem i przekazać komputerowi, co ma zrobić i w jaki sposób to osiągnąć.

## Paradygmat imperatywny

- Najstarszy i najbardziej „mechaniczny".
- Program to ciąg instrukcji, które zmieniają stan maszyny krok po kroku.
- Programista mówi komputerowi dokładnie, jak ma coś zrobić.

Przykładowe języki: C, Pascal, Python (częściowo), JavaScript.

Myślenie: „Zrób to, potem tamto, a na końcu wypisz wynik."

## Paradygmat proceduralny

- Odmiana imperatywnego, ale bardziej uporządkowana.
- Kod dzieli się na procedury / funkcje, które realizują konkretne zadania.
- Każda procedura jest jak klockowy element większej układanki.

Przykładowe języki: C, Fortran, Ada.

Myślenie: „Podziel duży problem na mniejsze, łatwiejsze części."

## Paradygmat obiektowy

- Program składa się z obiektów, które mają dane (stan) i zachowania (metody).
- Obiekty komunikują się między sobą, tworząc system przypominający rzeczywisty świat.

Przykładowe języki: Java, C#, C++, Python.

Myślenie: „Samochód ma silnik i potrafi się uruchomić, gdy kierowca przekręca kluczyk."

## Paradygmat funkcyjny

- Program jest zbiorem funkcji matematycznych, które przekształcają dane w inne dane.
- Nie ma tu stanów ani efektów ubocznych — każda funkcja zawsze zwraca ten sam wynik dla tych samych argumentów.

Przykładowe języki: F#, Haskell, Lisp, Scala.

Myślenie: „Weź dane, przepuść je przez funkcje, odbierz wynik."

## Paradygmat logiczny

- Program składa się z faktów i reguł.
- Zamiast mówić komputerowi, jak coś zrobić, określasz, co jest prawdą — a komputer sam wnioskuje, jakie rozwiązania są poprawne.

Przykładowe języki: Prolog, Mercury.

Myślenie: „Jeśli Jan jest ojcem Piotra, a Piotr jest ojcem Agaty, to Jan jest dziadkiem Agaty."

## Podział języków: imperatywne vs deklaratywne

| Rodzaj | Jak się myśli | Przykład | Kluczowa idea |
|---|---|---|---|
| Imperatywne | „powiedz komputerowi, **jak** coś zrobić, krok po kroku” | C, Python, Java | program = sekwencja instrukcji modyfikujących stan |
| Deklaratywne | „powiedz komputerowi, **co** chcesz osiągnąć” | SQL, Prolog, F# | program = opis zależności między danymi |

Porównanie:

W języku imperatywnym dajesz dokładny przepis: „Weź 3 jajka, ubij, dodaj mąkę, cukier, wymieszaj, wstaw do piekarnika na 180°C.”

W języku deklaratywnym mówisz tylko: „Chcę ciasto czekoladowe.” Komputer sam dobiera kroki i składniki, by osiągnąć ten efekt.

Myślenie imperatywne to kontrola nad każdym ruchem miksera, deklaratywne – zaufanie, że piekarnik wie, co robi.

## Pozostałe paradygmaty

Równoległy / współbieżny: program wykonuje wiele rzeczy naraz (np. obsługa wielu wątków).

Reaktywny: reaguje na zdarzenia (np. GUI, aplikacje mobilne).

Deklaratywny to parasol nad funkcyjnym i logicznym – łączy podejścia, gdzie program opisuje fakty, a nie wydaje polecenia.

# PROLOG

Prolog (od PROgramming in LOGic) to język, który działa inaczej niż większość tego, co znacie. Nie mówisz mu, jak coś zrobić – tylko opisujesz, co jest prawdą, a on sam to rozumuje. To programowanie w duchu „opisz świat, a komputer znajdzie odpowiedź".

Prolog to przykład języka deklaratywnego i logicznego, czyli takiego, w którym:

- program to zbiór faktów (np. Jan jest ojcem Piotra),
- oraz reguł (np. X jest dziadkiem Y, jeśli X jest ojcem Z, a Z jest ojcem Y).

Potem zadajesz pytania (tzw. zapytania), a Prolog odpowiada, co jest prawdą w tym świecie.

Początki Prologa sięgają Francji, lat 70. XX wieku, a dokładniej – Uniwersytetu w Marsylii. Język stworzył Alain Colmerauer wraz z Robertem Kowalskim (tak, to nazwisko nieprzypadkowo brzmi swojsko – Kowalski był Brytyjczykiem polskiego pochodzenia). Pierwsza wersja powstała w 1972 roku, czyli w czasach, gdy nikt jeszcze nie mówił o AI – tylko o sztucznej inteligencji pisanej przez pełne słowa. Ich pomysł był prosty, ale rewolucyjny:

*„Skoro komputery są dobre w logice, to czemu nie pozwolić im myśleć jak logik?"*

W latach 80. Japonia wpadła na pomysł stworzenia tzw. „komputerów piątej generacji" – inteligentnych maszyn, które będą rozumieć język naturalny, prowadzić rozmowy i wyciągać wnioski. Zgadnij, w jakim języku planowano je oprogramować? Tak, właśnie w Prologu. Projekt był ambitny i... kosztowny. Rząd Japonii zainwestował miliardy jenów, ale finalnie technologia nie dogoniła marzeń. Mimo to Prolog stał się ważnym kamieniem milowym w rozwoju AI i logiki komputerowej.

> Colmerauer żartował, że nazwał Prolog „językiem logicznym", ale gdy pokazywał go innym inżynierom, większość odpowiadała:
>
> *„To jest zbyt logiczne, żeby działało."*
>
> Okazało się jednak, że działa — tylko wymaga innego sposobu myślenia.

## Wykorzystanie PROLOG'a

- w sztucznej inteligencji (systemy ekspertowe, rozumowanie logiczne),
- w lingwistyce komputerowej (analiza składni, semantyka języka naturalnego),
- w planowaniu i rozwiązywaniu łamigłówek (np. sudoku, krzyżówki logiczne),
- w edukacji – bo uczy myślenia „na poziomie reguł", a nie instrukcji.

## Dlaczego warto się go nauczyć?

- Bo zmienia sposób myślenia o programowaniu – uczy precyzji i rozumienia zależności, zamiast klepania instrukcji.
- Bo pokazuje, że logika matematyczna naprawdę potrafi działać w praktyce.
- Bo jego idea przetrwała – wiele współczesnych języków (np. Python, SQL, F#) ma elementy deklaratywne właśnie z Prologa.

## Działanie PROLOG'a

Prolog działa na zasadzie wnioskowania logicznego – dokładniej rezolucji i unifikacji. Brzmi groźnie, ale chodzi o to, że komputer:

1. sprawdza, czy dane fakty i reguły mogą prowadzić do spełnienia zapytania,
2. szuka wartości zmiennych, które to umożliwiają.

Czyli:

Ty podajesz warunki, Prolog sprawdza wszystkie logiczne ścieżki. Jeśli coś pasuje – dostajesz true i konkretne wartości. Jeśli nie – false i cisza. Np.

```
% Fakty:
ojciec(jan, piotr).
ojciec(piotr, agata).

% Reguła:
dziadek(X, Y) :- ojciec(X, Z), ojciec(Z, Y).

% Zapytanie:
?- dziadek(jan, agata).
% Wynik:
% true.
```

## Unifikacja

Unifikacja to proces dopasowywania dwóch wyrażeń (tzw. termów) poprzez znalezienie takich wartości zmiennych, które sprawiają, że oba wyrażenia stają się identyczne. Np.

```
ojciec(jan, X) = ojciec(jan, piotr)
```

Prolog widzi, że po lewej jest `ojciec(jan, X),` po prawej `ojciec(jan, piotr)` i stwierdza: „Dopasuję to, jeśli X = piotr."

Unifikacja porównuje term po termie (wyrażeniu):

- symbole(funktory) muszą być takie same (ojciec z ojciec),
- liczba argumentów też (obie strony mają po 2 argumenty),
- zmienne mogą przyjąć wartość, by dopasowanie się udało.

Jeśli wszystko się zgadza, Prolog mówi **true** i zapamiętuje to przypisanie.

## Rezolucja

Rezolucja to metoda wnioskowania, dzięki której Prolog może „dowodzić" prawdziwości zapytań na podstawie reguł i faktów. Polega ona na szukaniu ciągu logicznych powiązań (dowodów), które prowadzą od zapytania do znanych faktów. Innymi słowy — rezolucja to proces udowadniania, a unifikacja to narzędzie, którym to robi. Np.

```
ojciec(jan, piotr).
ojciec(piotr, agata).
dziadek(X, Y) :- ojciec(X, Z), ojciec(Z, Y).

?- dziadek(jan, agata).
```

Prolog widzi, że `dziadek(X, Y)` to reguła. Aby potwierdzić `dziadek(jan, agata)`, musi znaleźć takie Z, że: `ojciec(jan, Z)` i `ojciec(Z, agata)`

Teraz uruchamia się **unifikacja**:

- `ojciec(jan, Z)` dopasowuje się do `ojciec(jan, piotr)` → Z = piotr
- potem `ojciec(Z, agata)` dopasowuje się do `ojciec(piotr, agata)` → Z = piotr nadal pasuje

Wszystkie warunki spełnione → **rezolucja udana**, wynik true.

Prolog stosuje tzw. rezolucję wsteczną (backward chaining). Oznacza to, że wnioskowanie przebiega „od celu do przesłanek" — program zaczyna od zapytania użytkownika i próbuje ustalić, jakie warunki muszą być spełnione, aby to zapytanie było prawdziwe.

Dzięki temu można zadawać pytania otwarte, np.:

```
?- ojciec(jan, Kto).
```

i otrzymywać kolejne rozwiązania (po wciśnięciu ;):

```
Kto = piotr ;
Kto = anna ;
false.
```

## Obserwacja w SWI

Aby zobaczyć, jak Prolog wykonuje proces rezolucji, można użyć wbudowanego narzędzia trace:

```
?- trace.
?- dziadek(jan, agata).
```

Po włączeniu śledzenia program wyświetla kolejne kroki unifikacji i rezolucji, m.in.:

- `Call:` – rozpoczęcie sprawdzania celu,
- `Exit:` – cel został spełniony,
- `Redo:` – próba znalezienia kolejnego rozwiązania,
- `Fail:` – brak dopasowania.

Dzięki temu można dokładnie prześledzić proces logicznego rozumowania programu.

## Zadania

**1**

Utwórz nowy plik o nazwie zwierzeta.pl.

a) Zdefiniuj co najmniej 10 faktów opisujących różne zwierzęta oraz ich właściwości, np. gatunek, środowisko życia, sposób poruszania się.np.

```
ssak(pies).
ptak(wrobel).
mieszka(pies, dom).
mieszka(wrobel, drzewo).
```

b) Wykonaj kilka zapytań testowych:

```
?- ssak(kot).
?- mieszka(X, drzewo).
?- ptak(X).
```

c) Dopisz własne przykłady i przetestuj, jak Prolog odpowiada w przypadku wielu rozwiązań (użyj ;).

**2**

Utwórz plik rodzina.pl.

a) Zdefiniuj fakty dotyczące relacji rodzinnych.
b) Zdefiniuj regułę określającą, kiedy dwie osoby są rodzeństwem.

TIP: Zwróć uwagę, że Prolog generuje wszystkie możliwe kombinacje – w ten sposób realizuje proces rezolucji i backtrackingu.

**3**

W tym samym pliku zdefiniuj regułę, która pozwala ustalić, kto jest przodkiem kogo i przetestuj jej działanie.

a) Uruchom narzędzie śledzenia (trace.) i obserwuj, jak Prolog przechodzi kolejne etapy rezolucji i unifikacji.

TIP: Zwróć uwagę na etapy Call, Exit, Redo i Fail.

**4**

Napisz prostą regułę określającą relację „kuzynostwa” na podstawie danych z poprzedniego zadania (osoby, których rodzice są rodzeństwem).

a) Zbadaj, czy reguła zwraca prawidłowe wyniki dla wprowadzonych danych.

## Zadania do prezentacji

**1**

Baza wiedzy o studentach i zajęciach

a) Utwórz plik studenci.pl.
b) Zdefiniuj fakty:

```
student(vadym, informatyka, 5).
student(marek, informatyka, 5).
student(ola, ekonomia, 3).
prowadzi(kowalski, programowanie).
prowadzi(nowak, bazy_danych).
```

c) Utwórz regułę:

```
uczy_sie_z(X, Y) :- student(X, Kier, Sem), student(Y, Kier, Sem),
X \= Y.
```

d) Przetestuj zapytania:

```
?- uczy_sie_z(vadym, Y).
?- uczy_sie_z(X, marek).
```

e) W prezentacji wyjaśnij, jak działa unifikacja w tej regule oraz jak można rozszerzyć bazę o relację `uczy(X, Przedmiot)` lub `prowadzi_dla(Pracownik, Kierunek)`.

**2**

Prosty system ekspertowy „Pogoda”

a) Utwórz plik pogoda.pl.
b) Wprowadź fakty opisujące warunki atmosferyczne i zalecenia:

```
jest_zimno.
pada_deszcz.

ubierz(kurtka) :- jest_zimno, pada_deszcz.
ubierz(sweter) :- jest_zimno, \+ pada_deszcz.
ubierz(koszulka) :- \+ jest_zimno, \+ pada_deszcz.
```

c) Wykonaj zapytania:

```
?- ubierz(Co).
```

d) Zmodyfikuj fakty i obserwuj, jak zmienia się wynik.
e) Dla prezentacji przygotuj komentarz, jak działa negacja (\+) i jak Prolog znajduje odpowiednie dopasowania.

**3**

Własny mini-projekt

Opracuj krótką bazę wiedzy opisującą wybrany fragment rzeczywistości, np.:

- system doradzający wybór środka transportu,
- klasyfikator gatunków roślin,

| | |
|---|---|
| | • prostą bazę filmów i reżyserów.<br><br>Zadbaj o:<br>• minimum 8 faktów i 2 reguły,<br>• co najmniej jedno zapytanie z wynikiem wielokrotnym,<br>• komentarze opisujące strukturę bazy.<br><br>Przygotuj krótką (3–5 minut) prezentację wyjaśniającą, jak działa program i w jaki sposób Prolog wnioskuje na podstawie danych. |