Prolog. Programowanie

Autorzy: W. F. Clocksin, C. S. Mellish
Tłumaczenie: Tomasz Żmijewski
ISBN: 83-7197-918-5
Tytuł oryginału: Programming in Prolog
Format: B5, stron: 274

Programowanie w Prologu różni się zasadniczo od programowania w językach strukturalnych, takich jak Pascal czy C i językach obiektowych jak Java. Dla wielu osób zaczynających przygodę z Prologiem zaskoczeniem jest fakt, że pisanie programu w tym języku nie polega na kodowaniu algorytmu. Programista opisuje obiekty i związki między nimi, a także podaje warunki, jakie powinno spełniać szukane rozwiązanie. System sam przeprowadza obliczenia w oparciu o podane zależności logiczne, zaś programista jedynie częściowo może wpływać na sposób działania programu.

Książka „Prolog. Programowanie" to podręcznik tego niezwykłego języka programowania stosowanego przy rozwiązywaniu problemów z różnych dziedzin: od logiki matematycznej i symbolicznego rozwiązywania równań przez analizę języka naturalnego, aż do zagadnień związanych ze sztuczną inteligencją. Zawiera ona:

- Wprowadzenie do Prologu
- Podstawowe struktury danych
- Nawracanie, sterowanie nawracaniem za pomocą symbolu odcięcia
- Operacje wejścia/wyjścia
- Predykaty
- Składnię reguł gramatycznych i analizę języka naturalnego
- Wiele przykładowych programów

Wszystkim rozdziałom towarzyszą ćwiczenia. Uzupełnieniem tekstu książki są dodatki omawiające m.in. rozwiązania ćwiczeń i różnice między najważniejszymi wersjami Prologu.

„Prolog. Programowanie" to książka dla studentów matematyki i informatyki, a także dla wszystkich zainteresowanych programowaniem opartym na regułach logicznych. Jeśli chcesz podjąć wyzwanie i nauczyć się Prologu, jest książka dla Ciebie.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 1.

Wprowadzenie

Prolog

Prolog to komputerowy język programowania. Jego początki sięgają roku 1970, od tego czasu używano go w aplikacjach związanych z przetwarzaniem symbolicznym, w takich dziedzinach, jak:

- relacyjne bazy danych,
- logika matematyczna,
- rozwiązywanie problemów abstrakcyjnych,
- przetwarzanie języka naturalnego,
- automatyzacja projektowania,
- symboliczne rozwiązywanie równań,
- analiza struktur biochemicznych,
- różne zagadnienia z dziedziny sztucznej inteligencji.

Osoby dopiero zaczynające swoją przygodę z Prologiem są zaskoczone tym, że pisanie programu w Prologu nie polega na opisywaniu algorytmu, jak to ma miejsce w tradycyjnych językach programowania. Zamiast tego programiści Prologu zajmują się raczej formalnymi relacjami i obiektami związanymi z danym problemem, badając, które relacje są „prawdziwe" dla szukanego rozwiązania. Tak więc Prolog może być uważany za język *opisowy* i *deklaratywny*. Programowanie w Prologu polega przede wszystkim na opisaniu znanych faktów i relacji dotyczących problemu, w mniejszym stopniu na podawaniu kolejnych kroków algorytmu. Kiedy programujemy w Prologu, sposób pracy komputera częściowo wynika z deklaratywnej semantyki Prologu, częściowo z tego, że Prolog na podstawie danego zbioru faktów może wnioskować nowe fakty, a jedynie częściowo na podstawie jawnie podanych przez programistę instrukcji sterujących.

Obiekty i relacje

Prolog to komputerowy język programowania używany do rozwiązywania problemów dotyczących *obiektów* i *relacji* między nimi.

Kiedy na przykład mówimy, że „Jan ma książkę", informujemy, że istnieje relacja własności między obiektem „Jan" a obiektem „książka". Co więcej, jest to relacja uporządkowana: Jan ma książkę, ale książka nie ma Jana. Jeśli zadajemy pytanie „Czy Jan ma książkę?", staramy się poznać relację. Wiele problemów możemy opisać, wskazując obiekty i ich relacje. Rozwiązanie problemu polega na zażądaniu od komputera informacji o obiektach i relacjach, które można z naszego programu wywnioskować.

Nie wszystkie relacje jawnie określają wszystkie obiekty, które ich dotyczą. Kiedy na przykład mówimy „Biżuteria jest cenna", oznacza to, że istnieje relacja „bycia cennym" dotycząca biżuterii. Nie wiadomo, dla kogo biżuteria jest cenna ani dlaczego. Wszystko zależy od tego, co zamierzamy wyrazić. Jeśli tego typu relacje będziemy modelować w Prologu, to liczba podawanych szczegółów zależeć będzie od tego, jakiej odpowiedzi oczekujemy od komputera.

Mówimy tutaj o obiektach, ale nie należy mylić tego z popularną metodologią programowania — programowaniem obiektowym. W programowaniu obiektowym obiekt to struktura danych, która może dziedziczyć pola i metody z hierarchii klas, do której sama należy. Wprawdzie początki programowania obiektowego można datować na środek lat 60., lecz popularność zyskało w latach 80. i 90., kiedy to pojawiły się takie języki jak Smalltalk-80, C++ czy Java.

Z kolei Prolog ewoluował niezależnie od początku lat 70., zaś jego zaczątkiem były postępy programowania w logice. Prologu nie należy porównywać z językami obiektowymi, takimi jak C++ i Java, gdyż Prolog służy do całkiem innych zadań i całkiem inne znaczenie ma w nim słowo „obiekt". Dzięki elastyczności Prologu możliwe jest napisanie w nim programu, który będzie interpretował podobny do Prologu język obiektowy, ale to już całkiem inne zagadnienie. Reasumując, kiedy mówimy o obiektach w Prologu, nie chodzi nam o struktury danych dziedziczące pola i metody z klasy, ale o byty, które można opisać termami.

Prolog stanowi praktyczną i wydajną implementację szeregu aspektów „inteligentnego" wykonywania programu: braku determinizmu, równoległości i wywoływania procedur według wzorca. Prolog zawiera ujednoliconą strukturę danych, *term*, na bazie której tworzone są wszystkie dane oraz same programy Prologu. Program prologowy składa się ze zbioru klauzul, a każda klauzula to albo fakt opisujący pewną informację, albo reguła mówiąca, jak rozwiązanie można powiązać z danymi faktami. Tak więc Prolog można uważać za pierwszy krok na drodze ku ostatecznemu celowi programowania w logice. W książce tej nie będziemy zanadto zastanawiać się nad dalszymi implikacjami programowania w logice, nie będzie nas też specjalnie interesowało, dlaczego Prolog nie jest gotowym językiem programowania w logice. Zamiast tego skoncentrujemy się na pisaniu przydatnych programów za pomocą istniejących obecnie systemów standardowego Prologu.

Zanim zaczniemy programować, trzeba jeszcze wspomnieć o jeszcze jednej rzeczy. Wszyscy do zapisu relacji używamy reguł. Na przykład reguła „Dwoje ludzi jest siostrami, jeśli oboje są kobietami i mają tych samych rodziców" objaśnia znaczenie bycia siostrą. Mówi nam ona też, jak znaleźć dwoje ludzi będących siostrami: wystarczy po prostu sprawdzić, czy oboje to kobiety i czy mają tych samych rodziców. Ważną rzeczą jest to, że reguły zwykle są bardzo uproszczone, ale są wystarczająco precyzyjne, aby być *definicjami*. Nie można przecież oczekiwać, że definicja powie nam o jakimś obiekcie wszystko.

Zapewne większość ludzi zgodzi się co do tego, że tak naprawdę „bycie siostrami" znaczy o wiele więcej niż wynika to z powyższej reguły. Kiedy jednak rozwiązujemy pewien konkretny problem, koncentrujemy się jedynie na regułach z tym problemem związanych. Powinniśmy zatem przyjąć specjalnie przygotowaną, uproszczoną definicję, która w danym wypadku będzie wystarczająca.

Programowanie

W tym rozdziale pokażemy najważniejsze elementy języka w prawdziwych programach, ale nie będziemy zanadto zagłębiać się w szczegóły, zasady formalne czy wyjątki. Na razie nie będziemy silili się na kompletność wykładu czy jego formalną poprawność. Chcemy, by czytelnik szybko posiadł umiejętność pisania przydatnych praktycznie programów, więc musimy skoncentrować się na podstawach: faktach, zapytaniach, zmiennych, połączeniach i regułach. Inne elementy Prologu, np. listy czy rekurencja, będą omawiane w dalszych rozdziałach.

Programowanie w Prologu składa się z:

- Deklarowania *faktów* dotyczących obiektów i związków między nimi.
- Definiowania *reguł* dotyczących obiektów i związków między nimi.
- Zadawania *zapytań* o obiekty i związki między nimi.

Załóżmy na przykład, że wprowadziliśmy do systemu Prologu przedstawioną wyżej regułę dotyczącą sióstr. Moglibyśmy zadać zapytanie, czy Maria i Janina są siostrami. Prolog przeszuka swoje informacje o Marii i Janinie, następnie odpowie *yes* lub *no*, zależnie od swojego stanu wiedzy. Tak więc możemy uważać Prolog za zbiór faktów i reguł, które są używane do udzielania odpowiedzi na zapytania. Programowanie w Prologu polega na podawaniu faktów i reguł, zaś system potrafi wnioskować nowe fakty na podstawie już istniejących.

Prolog to język konwersacyjny, co oznacza, że użytkownik prowadzi pewnego rodzaju konwersację z komputerem. Załóżmy, że siedzimy sobie przy terminalu i mamy użyć Prologu. Terminal komputera ma *klawiaturę* i *wyświetlacz*. Za pomocą klawiatury wprowadza się do komputera dane, zaś komputer na wyświetlaczu (może to być monitor czy drukarka) pokazuje swoje odpowiedzi. Prolog oczekuje, że użytkownik wprowadzi wszystkie fakty i reguły dotyczące rozwiązywanego problemu. Następnie Prolog będzie odpowiadał na zadawane pytania.

Teraz zajmiemy się kolejno podstawowymi elementami składowymi Prologu. Nie będziemy na razie szczegółowo omawiać wszystkich aspektów tego języka, na to czas przyjdzie później, w dalszych rozdziałach.

Fakty

Najpierw omówimy *fakty* opisujące obiekty. Załóżmy, że chcemy w Prologu zanotować fakt, że „Jan lubi Marię". Fakt ten dotyczy dwóch obiektów, Jana i Marii, oraz zawiera relację „lubienia". W prologu fakty zapisuje się w postaci:

```
lubi(jan,maria).
```

Trzeba pamiętać o kilku rzeczach:

- Nazwy wszystkich relacji i obiektów muszą się zaczynać małymi literami, jak powyżej: `lubi`, `jan`, `maria`.
- Najpierw zapisuje się relację, a potem jej obiekty rozdzielone przecinkami i ujęte w nawias okrągły.
- Fakt musi się kończyć kropką (`.`).

Podczas definiowania związków między obiektami w formie faktów należy zwrócić uwagę na kolejność obiektów umieszczanych w nawiasie. Kolejność ta jest wprawdzie dowolna, ale trzeba się na jakąś zdecydować i potem się jej trzymać, aby zapewnić jednolitą interpretację tych faktów. Na przykład, w powyższym fakcie osobę lubiącą podajemy jako pierwszą, zaś lubianą jako drugą. Wobec tego fakt `lubi(jan,maria)` nie jest równoważny faktowi `lubi(maria,jan)`. Pierwszy z nich, zgodnie z przyjętą przez nas kolejnością, mówi, że Jan lubi Marię. Jeśli chcemy powiedzieć, że Maria lubi Jana, musimy to jawnie zapisać:

```
lubi(maria,jan).
```

Oto zestaw faktów wraz z ich interpretacją w języku naturalnym[1]:

`cenny(zloto).`	Złoto jest cenne.
`kobieta(janina).`	Janina jest kobietą.
`posiada(jan,zloto).`	Jan posiada złoto.
`ojciec(jan,maria).`	Jan jest ojcem Marii.
`daje(jan,gazeta,maria).`	Jan daje Marii gazetę.

Za każdym razem, kiedy używamy jakiejś nazwy, dotyczy ona konkretnego obiektu. Czytelnik zna język polski, więc oczywiste jest, że `jan` i `maria` muszą dotyczyć osób,

[1] W faktach i obiektach nie należy używać polskich liter. Zgodnie ze standardem Prologu w nazwach relacji i obiektów można korzystać jedynie z alfabetu łacińskiego i pewnych znaków dodatkowych, które zostaną podane w dalszych rozdziałach — *przyp. tłum.*

ale znaczenie takich nazw jak `zloto` nie od razu będzie jasne. Tego typu słowa nazywane są słowami niepoliczalnymi. Kiedy używamy jakiejś nazwy, musimy przypisać jej *interpretację*.

Przykładowo, nazwa `zloto` może odnosić się do obiektu — wtedy zwykle chodzi nam o jakiś kawałek złota. W takiej sytuacji fakt `cenny(zloto)` oznacza, że dany kawałek złota, któremu przypisaliśmy nazwę `zloto`, jest cenny. Z drugiej strony, możemy uznać, że nazwa `zloto` dotyczy pierwiastka o liczbie atomowej 79, wobec czego pisząc `cenny(zloto)` mówimy, że pierwiastek chemiczny złoto jest cenny. Tak więc nazwę można różnie interpretować, a o wyborze konkretnej interpretacji decyduje programista. Nie stanowi to problemu, pod warunkiem, że będziemy konsekwentnie trzymać się jednej interpretacji. Ważne jest ustalenie tej interpretacji dostatecznie wcześnie, kiedy jeszcze dokładnie wiemy, co dana nazwa miała oznaczać.

Teraz trochę o stosowanej terminologii. Nazwy obiektów występujące w nawiasach nazywamy *argumentami*. Pamiętajmy, że w informatyce słowo „argument" nie ma nic wspólnego z potocznym znaczeniem tego słowa, czyli „racją w dyskusji". Nazwę relacji znajdującą się przed nawiasem nazywamy *predykatem*. Wobec tego predykat `cenny` ma jeden argument, zaś predykat `lubi` ma dwa argumenty.

Nazwy obiektów i relacji są całkowicie dowolne. Zamiast zapisu `lubi(jan,maria)` równie dobrze moglibyśmy zastosować `a(b,c)`, wiedząc, że `a` oznacza *lubienie*, `b` oznacza *Jana*, a `c` oznacza *Marię*. Jednak zwykle nazwy dobiera się tak, aby ułatwić zapamiętywanie ich znaczenia. Wobec tego z góry trzeba ustalić znaczenie poszczególnych nazw i kolejność argumentów, a potem trzeba się ściśle trzymać przyjętej konwencji.

Relacje mogą mieć dowolną liczbę argumentów. Jeśli chcemy zdefiniować predykat `gra`, w którym podamy dwóch graczy i nazwę gry, będziemy potrzebowali trzech argumentów. Oto dwa przykłady takich faktów:

```
gra(jan,maria,futbol).
gra(janina,staszek,badminton).
```

Jak się przekonamy dalej, użycie wielu argumentów jest konieczne do zapamiętania złożonych powiązań między relacjami.

Można deklarować różne fakty, także fakty, które nie są prawdziwe w świecie zewnętrznym. Można na przykład napisać `krol(jan,francja)`, aby poinformować, że *Jan jest królem Francji*. Jest to oczywiście nieprawda, biorąc pod uwagę, że rządy monarsze we Francji zostały obalone około roku 1792. Prolog jednak nic o tym nie wie i nie chce wiedzieć; fakty Prologu pozwalają po prostu zapisywać dowolne relacje między obiektami.

W Prologu zbiór faktów nazywamy *bazą danych*. Tego słowa używa się zawsze, kiedy mamy do czynienia ze zbiorem faktów (a, jak dowiemy się potem, także reguł) używanych do rozwiązania pewnego problemu.

Zapytania

Kiedy mamy już jakieś fakty, możemy zadawać dotyczące ich zapytania. W Prologu zapytanie wygląda tak samo jak fakt, ale umieszcza się przed nim specjalny symbol — pytajnik i minus (`?-`). Rozważmy zapytanie:

```
?- posiada(maria,gazeta).
```

Jeśli `maria` to *osoba o imieniu Maria*, a `gazeta` to pewna konkretna gazeta, to powyższe zapytanie odpowiada pytaniu *Czy Maria ma gazetę?* lub *Czy istnieje fakt mówiący, że Maria ma gazetę?* Nie jest to natomiast zapytanie o wszystkie gazety, ani o gazety w ogólności.

Kiedy zadajemy Prologowi zapytanie, przeszukuje on stworzoną wcześniej bazę danych i szuka faktów *pasujących* do faktu podanego w zapytaniu. Dwa fakty *pasują* do siebie, jeśli mają takie same predykaty (tak samo pisane) i takie same są odpowiadające sobie ich argumenty. Jeśli Prolog znajdzie fakt pasujący do zapytania, odpowie `yes` (tak). Jeśli fakt taki nie zostanie znaleziony, odpowiedzią będzie `no` (nie). Odpowiedzi pokazywane są na monitorze bezpośrednio pod pytaniem. Załóżmy, że mamy następującą bazę danych:

```
lubi(jarek,ryby).
lubi(jarek,maria).
lubi(maria,ksiazka).
lubi(jan,ksiazka).
lubi(jan,francja).
```

Jeśli wprowadzimy takie właśnie fakty, możemy zadawać poniższe zapytania i otrzymamy pokazane niżej odpowiedzi (odpowiedzi komputera są pogrubione):

```
?- lubi(jarek,pieniadze).
no
?- lubi(maria,jarek).
no
?- lubi(maria,ksiazka).
yes
```

Odpowiedzi na pierwsze trzy pytania nie powinny budzić żadnych wątpliwości. W Prologu odpowiedź `no` należy interpretować jako *nie znaleziono niczego pasującego do pytania*. Trzeba pamiętać, że `no` nie jest równoważne z odpowiedzią *nie* — wyobraźmy sobie bazę danych o słynnych Grekach:

```
osoba(sokrates).
osoba(arystoteles).

atenczyk(sokrates).
```

Możemy zadawać następujące zapytania:

```
?- atenczyk(sokrates).
yes
?- atenczyk(arystoteles).
no
```

Wprawdzie być może Arystoteles żył niegdyś w Atenach, ale nie możemy tego *udowodnić* na podstawie pokazanej bazy danych.

A co się stanie, gdy zadamy zapytanie o relację, której nie ma w bazie danych? Załóżmy, że używamy powyższej bazy danych z relacją `lubi` i zadajemy jak najbardziej rozsądne zapytanie:

```
?- krol(jan,francja).
```

Nasza baza danych niczego nie mówi o królach, choć w bazie występują zarówno `jan`, jak i `francja`. W większości systemów Prologu udzielona zostanie odpowiedź **no**, gdyż na podstawie bazy danych nie można udowodnić żadnych faktów dotyczących królów. Niemniej standard języka Prolog pozwala albo udzielić odpowiedzi **no**, albo przed udzieleniem takiej odpowiedzi wygenerować ostrzeżenie, albo może zostać zaprezentowany komunikat o błędzie. Przykładowo, gdybyśmy korzystali z naszej bazy danych o Grekach i zadalibyśmy zapytanie

```
?- grek(sokrates).
```

to choć w naszej bazie danych jest informacja, że Sokrates to Ateńczyk, nie wystarcza to do *udowodnienia*, że jest on Grekiem: baza danych nie mówi niczego o Grekach, więc z punktu widzenia standardu języka dopuszczalna jest odpowiedź:

```
Existence error: procedure grek
no
```

To, jak konkretny system się w takiej sytuacji zachowa, zależy od sposobu zaimplementowania w nim standardu Prologu, więc nie będziemy się tego typu szczegółami zajmować.

Omówione dotąd fakty i zapytania nie są zbyt interesujące — jedyne, co potrafimy, to uzyskać te same informacje, które wprowadziliśmy wcześniej do bazy. Znacznie ciekawsze byłoby uzyskanie odpowiedzi na pytania *Co lubi Maria?* czy *Kto żył w Atenach?* Do tego właśnie służą *zmienne*.

Zmienne

Jeśli chcielibyśmy dowiedzieć się, co lubi Jan, bardzo nieporęcznie byłoby pytać *Czy Jan lubi książki?*, *Czy Jan lubi Marię?* i tak dalej, a Prolog każdorazowo udzielałby odpowiedzi `yes` lub `no`. Znacznie rozsądniej byłoby zapytać Prolog, co lubi Jan. Odpowiednie pytanie miałoby postać *Czy Jan lubi X?* W chwili zadawania takiego zapytania nie wiemy, co *oznacza X*; chcielibyśmy, aby to Prolog podał nam wszystkie możliwości. Otóż w Prologu możemy nie tylko nazywać konkretne obiekty, ale też używać takich nazw jak *X*, oznaczających obiekty, które będą wskazywane przez Prolog. Tego typu nazwy nazywamy *zmiennymi*.

Kiedy Prolog używa zmiennej, może być ona *ukonkretniona* lub *nieukonkretniona*. Zmienna jest ukonkretniona, jeśli odpowiada jakiemuś konkretnemu obiektowi. Zmienna nie jest ukonkretniona, kiedy nie wiemy, jakiemu obiektowi może odpowiadać.

Prolog odróżnia zmienne od nazw konkretnych obiektów w ten sposób, że *wszystkie* nazwy zaczynające się wielkimi literami są traktowane jako zmienne.

Kiedy Prolog otrzymuje zapytanie zawierające zmienną, przeszukuje wszystkie fakty, aby znaleźć obiekt, który może zastąpić zmienną. Kiedy zatem pytamy *Czy Jan lubi X?*, Prolog przeszukuje wszystkie fakty, by znaleźć rzeczy, które lubi Jan.

Zmienna taka jak *X* nie nazywa sama konkretnego obiektu, ale może zastępować obiekt, którego nazwy nie potrafimy podać. Na przykład, nie możemy nadać nazwy *czemuś, co lubi Jan*, więc w Prologu używa się tu innego zapisu; zamiast zapytania

```
?- lubi(jan, coś, co lubi Jan).
```

używamy zmiennych:

```
?-lubi(jan,X).
```

Zmienne w razie potrzeby mogą mieć dłuższe nazwy, na przykład poniższe zapytanie jest poprawne:

```
?- lubi(jan,Coscolubijan).
```

Dlaczego to działa? Po prostu zmienna może być dowolną nazwą zaczynającą się wielką literą. Przyjmijmy, że mamy do czynienia z poniższą bazą faktów i zapytaniem:

```
lubi(jan,kwiaty).
lubi(jan,maria).
lubi(pawel,maria).

?- lubi(jan,X).
```

Zapytanie interpretujemy jako *Czy mamy coś, co lubi Jan?* Prolog odpowie na nie:

```
X=kwiaty
```

i będzie czekał na dalsze instrukcje, które wkrótce też omówimy. Jak Prolog uzyskał powyższy wynik? Kiedy zadajemy mu powyższe zapytanie, zmienna `X` nie jest początkowo ukonkretniona. Prolog przegląda bazę danych szukając faktów *pasujących* do zapytania. Póki argument jest zmienną nieukonkretnioną, może być zastępowany *dowolnym* innym argumentem występującym w tym samym miejscu w faktach. Prolog wyszukuje dowolne fakty z predykatem `lubi` i pierwszym argumentem `jan`. drugi argument może być dowolny, gdyż zapytanie w drugim argumencie zawiera zmienną nieukonkretnioną. Kiedy odpowiedni fakt zostanie znaleziony, od tej chwili `X` oznacza drugi argument znalezionego faktu. Prolog przeszukuje bazę danych w takiej kolejności, w jakiej ją wpisano (czyli od góry do dołu), wobec czego fakt `lubi(jan,kwiaty)` znajdowany jest jako pierwszy. Zmienna `X` od tej chwili oznacza obiekt `kwiaty`, czyli jest *ukonkretniona* jako `kwiaty`. Prolog *oznacza miejsce* bazy danych, w którym znaleziono pasujący fakt; użycie tego znacznika omówimy wkrótce.

Kiedy Prolog znajduje fakt pasujący do zapytania, pokazuje obiekty podstawione pod zmienne. W tym wypadku była tylko jedna zmienna, `X`, i pasowała do obiektu `kwiaty`. Prolog czeka na dalsze polecenia. Wciśnięcie klawisza *Enter* oznacza, że wystarcza nam znalezione rozwiązanie i dalsze już nas nie interesują. Jeśli wciśniemy klawisz *;* (i *Enter*), Prolog podejmie dalsze przeszukiwanie bazy danych *zaczynając od miejsca wstawienia znacznika*.

Załóżmy, że po uzyskaniu od Prologu pierwszej odpowiedzi (X=kwiaty) zażądaliśmy wznowienia poszukiwań przez wciśnięcie ;. Oznacza to, że chcemy znaleźć inną odpowiedź na to samo zapytanie, czyli chcemy znaleźć inny obiekt, który można podstawić pod X. Wobec tego Prolog musi „zapomnieć", że X oznaczało kwiaty i podjąć poszukiwania z nieukonkretnioną zmienną X. Wyszukujemy inne możliwe rozwiązanie, więc zaczynamy od znacznika. Następny znaleziony pasujący fakt to lubi(jan, maria). Zmienna X jest *ukonkretniana* wartością maria, Prolog wstawia znacznik na fakcie lubi(jan,maria). Prolog wyświetla X=maria i czeka na dalsze polecenia. Jeśli wpiszemy następny średnik, Prolog będzie szukał kolejnego rozwiązania. W naszym przykładzie nie ma już innych obiektów, które lubiłby Jan, więc Prolog kończy wyszukiwanie i pozwala zadawać kolejne zapytania lub deklarować dalsze fakty.

Co się stanie, jeśli mając takie same fakty, jak powyżej, zadamy zapytanie:

```
?- lubi(X,maria).
```

Oznacza to pytanie *Czy jakiś obiekt lubi Marię?* Obiektami takimi będą jan i pawel. Znowu w celu uzyskania wszystkich możliwych odpowiedzi korzystamy ze średnika i RETURN:

`?- lubi(X,maria).`	nasze zapytanie
`X=jan;`	pierwsza odpowiedź; wcisnęliśmy średnik (;) i RETURN
`X=pawel;`	druga odpowiedź; znów wcisnęliśmy średnik (;) i RETURN
`no`	nie ma więcej odpowiedzi

Koniunkcje

Załóżmy, że chcielibyśmy odpowiadać na zapytania dotyczące bardziej złożonych relacji, na przykład takich: *Czy Jan i Maria lubią się nawzajem?* Jednym ze sposobów realizacji takiego zapytania byłoby zapytanie, czy Jan lubi Marię, a jeśli Prolog odpowiedziałby yes, zapytanie, czy Maria lub Jana. Tak więc problem składa się z dwóch odrębnych *celów*, które Prolog musi wykazać. Kombinacje tego typu są wśród programistów Prologu wyjątkowo rozpowszechnione, więc wymyślono dla nich specjalną notację. Załóżmy, że mamy bazę danych:

```
lubi(maria,czekolada).
lubi(maria,wino).
lubi(jan,wino).
lubi(jan,maria).
```

Chcemy wiedzieć czy Jan i Maria lubią się nawzajem. W tym celu pytamy *Czy Jan lubi Marię?* i *Czy Maria lubi Jana?*. Spójnik *i* oznacza, że interesuje nas koniunkcja celów: chcemy spełnić je obydwa. W Prologu spójnik ten zapisujemy jako przecinek (,):

```
?- lubi(jan,maria), lubi(maria,jan).
```

Przecinek czytamy jako *i*, można za jego pomocą rozdzielać dowolną liczbę celów, które mają być spełnione w celu udzielenia odpowiedzi na pytanie. Kiedy Prolog otrzymuje ciąg celów (rozdzielonych przecinkami), stara się spełnić wszystkie cele, znajdując pasujące do nich cele z bazy danych. Aby spełniona była cała sekwencja, spełnione muszą być wszystkie cele składowe. Jeśli do pokazanej powyżej bazy zadamy powyższe zapytanie, odpowiedzią będzie `no`. Pierwszy cel jest prawdziwy, bo mamy w bazie informację, że Jan lubi Marię, ale nie występuje nigdzie fakt `lubi(maria, jan)`. Skoro chcieliśmy wiedzieć, czy lubią się nawzajem, odpowiedź na całe zapytanie brzmi `no`.

Korzystając ze zmiennych i koniunkcji, możemy tworzyć naprawdę złożone zapytania. Wiemy już, że nie można udowodnić, iż Jan i Maria lubią się nawzajem, więc chcielibyśmy wiedzieć, czy istnieje coś, co oboje lubią: *Czy istnieje coś, co lubi zarówno Jan, jak i Maria?* Zapytanie takie składa się z dwóch celów:

- Najpierw sprawdzamy czy istnieje jakiś obiekt `X` lubiany przez Marię.
- Następnie sprawdzamy czy Jan także lubi `X`, cokolwiek by to było.

Powyższe dwa cele w Prologu można zapisać jako koniunkcję:

```
?- lubi(maria,X), lubi(jan,X).
```

Prolog najpierw próbuje spełnić pierwszy cel zapytania. Jeśli zostanie on znaleziony w bazie danych, miejsce w bazie danych zostanie zaznaczone i Prolog spróbuje spełnić drugi cel. Jeśli on też zostanie spełniony, Prolog oznaczy miejsce *tego celu* w bazie danych i mamy już rozwiązanie. Najważniejsze, o czym trzeba pamiętać to to, że każdy cel wstawia do bazy osobną zakładkę.

Jeśli drugi cel nie zostanie jednak spełniony, Prolog będzie się starał spełnić inaczej cel poprzedni (w tym wypadku pierwszy). Pamiętajmy, że Prolog dla każdego celu przeszukuje całą bazę danych. Jeśli fakt z bazy danych zostanie dopasowany, Prolog zaznaczy to miejsce, na wypadek gdyby potrzebne było ponowne dopasowywanie tego celu. Kiedy jednak trzeba inaczej dopasować cel, Prolog zaczyna przeszukiwanie od znacznika celu, nie od początku bazy danych. Nasze powyższe zapytanie *Czy istnieje coś, co lubi Maria i jednocześnie lubi to Jan?*, stanowi przykład użycia mechanizmu *nawracania*:

1. W bazie danych odszukiwany jest pierwszy cel. Kiedy drugi argument `X` jest nieukonkretniony, może przybrać dowolną wartość. Pierwszym znalezionym dopasowaniem jest `lubi(maria,czekolada)`. Wobec tego od tej chwili `X` przybiera wartość `jedzenie` *wszędzie* tam, gdzie pojawia się `X`. Prolog oznacza w bazie danych miejsce znalezienia faktu na wypadek, gdyby konieczne było wznowienie wyszukiwania.
2. W bazie danych szuka się faktu `lubi(jan,czekolada)`. Wynika to stąd, że następnym celem jest `lubi(jan,X)`, zaś `X` oznacza teraz `czekolada`. Faktu takiego w bazie nie ma, więc cel zawodzi i Prolog stara się znaleźć inne dopasowanie `lubi(maria,X)`, tym razem jednak przeszukiwanie bazy zaczyna się od zaznaczonego miejsca. Najpierw konieczne jest cofnięcie ukonkretnienia zmiennej `X`, aby ponownie mogła przybrać dowolną wartość.

3. Oznaczone miejsce to `lubi(maria,czekolada)`, więc od tego faktu zaczyna się dalsze wyszukiwanie. Nie doszliśmy jeszcze do końca bazy, więc można dalej sprawdzać co lubi Maria; następny pasujący fakt to `lubi(maria,wino)`. Zmienna `X` otrzymuje teraz wartość `wino`, Prolog ponownie oznacza miejsce wystąpienia tego faktu.
4. Tak jak poprzednio, Prolog stara się uzgodnić drugi cel, tym razem szukając faktu `lubi(jan,wino)`. Tego celu Prolog nie próbuje uzgadniać z innymi wartościami; cel jest nowy, więc przeszukiwana jest cała baza danych. Fakt zostaje szybko znaleziony i Prolog generuje odpowiedni komunikat. Cel został uzgodniony, więc Prolog oznacza odpowiednie miejsce w bazie danych, aby w razie potrzeby zacząć dalsze przeszukiwanie od tego miejsca. W bazie danych znajdują się znaczniki wszystkich celów, które Prolog uzgadniał.
5. W tej chwili spełnione są już oba cele, zmiennej `X` odpowiada nazwa `wino`. Pierwszy cel spowodował zaznaczenie w bazie danych faktu `lubi(maria,wino)`, a drugi — faktu `lubi(jan,wino)`.

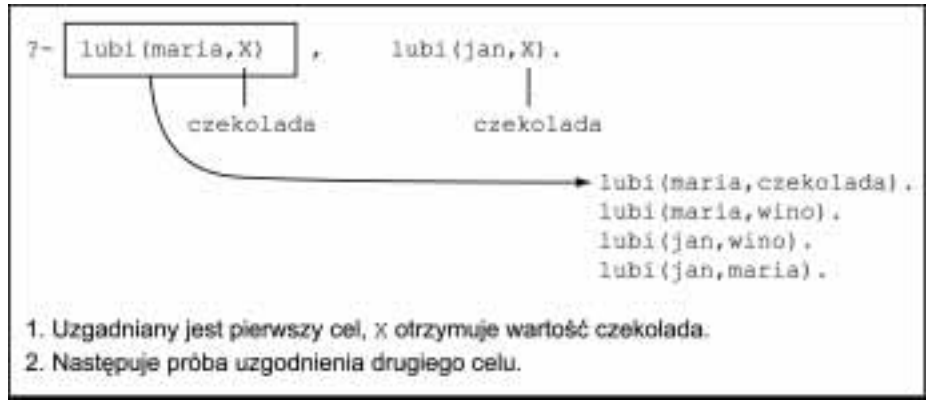
Tak jak w przypadku wszystkich zapytań, Prolog odnajduje pierwszą odpowiedź, zatrzymuje się i czeka na dalsze instrukcje. Jeśli wpiszemy `;`, poszukiwanie rozwiązań będzie kontynuowane. Wiemy, że odbywa się to przez próby innego spełnienia obydwóch zadanych celów od zostawionych wcześniej znaczników.

Reasumując, koniunkcję celów interpretujemy od strony lewej do prawej, poszczególne cele rozdzielamy przecinkami. Każdy cel może mieć lewego i prawego sąsiada. Oczywiście pierwszy cel nie ma sąsiada lewego, a ostatni — prawego. Kiedy Prolog ma do czynienia z koniunkcją celów, stara się spełnić je kolejno, od lewej do prawej. Jeśli cel zostaje spełniony, Prolog umieszcza w bazie danych znacznik w miejscu z tym celem powiązanym — można na to patrzeć jak na narysowanie strzałki od celu do spełniającego go miejsca w bazie danych, przy tym ukonkretniane są wszystkie nieukonkretnione dotąd zmienne. Wszystko to ma miejsce w pierwszym kroku powyższego wyliczenia. Jeśli zmienna jest ukonkretniana, ukonkretniane są od razu wszystkie jej wystąpienia. Następnie Prolog stara się uzgodnić prawego sąsiada danego celu zaczynając poszukiwanie rozwiązań od początku bazy danych.

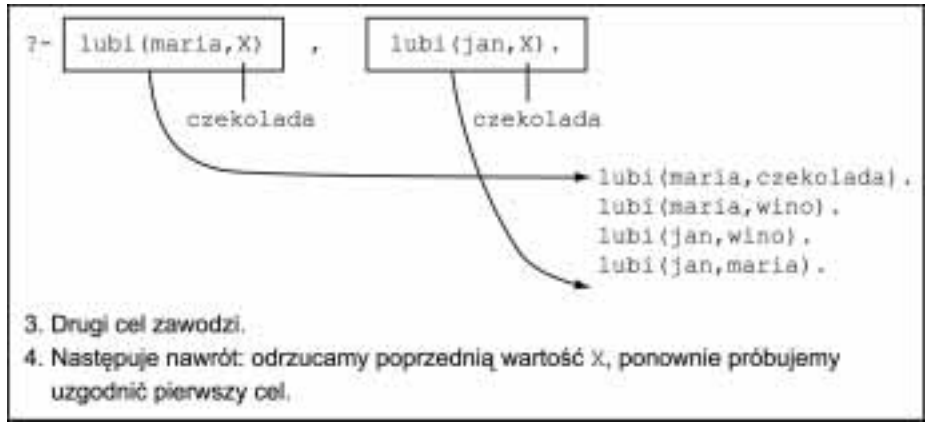
Spełniane są kolejno wszystkie cele, przy czym w bazie danych zostawiane są znaczniki (czyli w naszym przykładzie rysujemy strzałki od celów do odpowiednich faktów). Kiedy któryś cel zawodzi (nie można znaleźć pasującego do niego faktu), Prolog się cofa i stara się inaczej uzgodnić lewego sąsiada, zaczynając poszukiwanie od jego znacznika. W takiej chwili wycofywane jest ukonkretnienie wszystkich zmiennych, które miało miejsce w czasie realizacji sąsiedniego celu. Kiedy nie udaje się uzgodnić kolejnych celów, do których dochodzimy teraz od prawej strony, Prolog wycofuje się w lewo, aż kiedy braknie lewego sąsiada, cała koniunkcja zawodzi. Takie próby wycofywania się i uzgadniania kolejnych celów koniunkcji nazywamy *nawracaniem*. Nawracanie omówimy jeszcze w następnym rozdziale, a dokładnie zrobimy to w rozdziale 4.

Badając przykłady, dla wygody możemy zapisywać pod każdą zmienną z celu wartość, na jaką zmienna ta została ukonkretniona. Warto też rysować strzałkę od celu do

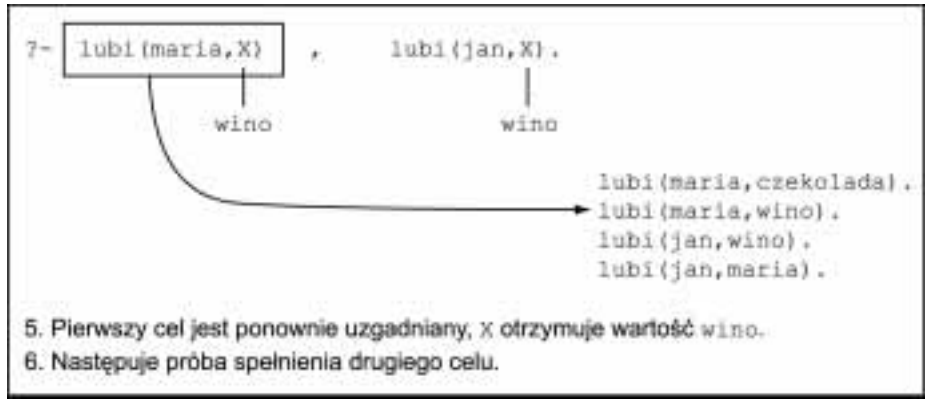
jego znacznika w bazie danych. Poniżej pokazano cztery rysunki, które pomogą zrozumieć przedstawiony przykład. Na każdym rysunku pokazano całą bazę danych i zapytanie wraz z ponumerowanymi komentarzami. Cele już uzgodnione dodatkowo obramowano.



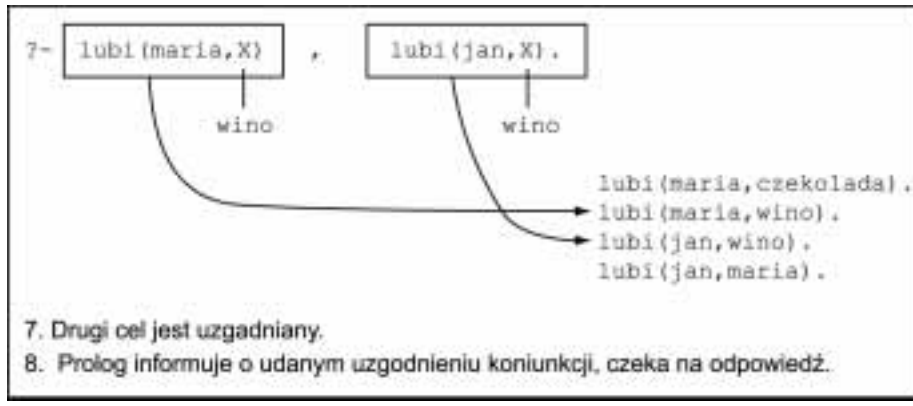
1. Uzgadniany jest pierwszy cel, X otrzymuje wartość czekolada.
2. Następuje próba uzgodnienia drugiego celu.



3. Drugi cel zawodzi.
4. Następuje nawrót: odrzucamy poprzednią wartość X, ponownie próbujemy uzgodnić pierwszy cel.



5. Pierwszy cel jest ponownie uzgadniany, X otrzymuje wartość wino.
6. Następuje próba spełnienia drugiego celu.




```
X=wino
```

W całej tej książce postaramy się pokazać, gdzie w przykładach zachodzi nawracanie i jak wpływa ono na sposób rozwiązywania problemów. Nawracanie jest tak ważne, że poświęcimy mu cały rozdział 4.

Ćwiczenie 1.1. Poprowadź dalej powyższą analizę obrazkową zakładając, że odpowiedzieliśmy systemowi średnikiem nakazującym znaleźć inne rozwiązania zapytania.

Reguły

Załóżmy, że chcemy zapisać fakt, że Jan lubi wszystkich ludzi. Jednym ze sposobów byłoby zapisanie szeregu kolejnych faktów:

```
lubi(jan,alfred).
lubi(jan,bertrand).
lubi(jan,cyryl).
lubi(jan,dawid).
```

Jednak jest to rozwiązanie niewygodne, szczególnie jeśli w bazie danych mamy setki ludzi. Innym sposobem byłoby stwierdzenie, że *Jan lubi wszystkie obiekty będące osobami*. Fakt taki ma postać *reguły* mówiącej, kogo lubi Jan. Taki zapis jest bardziej zwarty od wypisywania kolejno wszystkich osób lubianych przez Jana.

W Prologu reguł używa się do zapisania, że jakiś fakt *zależy* od grupy innych faktów. W języku polskim do stworzenia reguły używamy słówka „jeśli", na przykład:

Używam parasola, jeśli pada.
Jan kupuje wino, jeśli jest ono tańsze od piwa.

Reguł używa się też do zapisywania definicji, na przykład:

X jest ptakiem jeśli:
X jest zwierzęciem i
X ma pióra.

lub

X jest siostrą Y, jeśli:
X jest kobietą i
X i Y mają takich samych rodziców.

W powyższych definicjach zapisanych w języku naturalnym użyto zmiennych *X* i *Y*. Trzeba pamiętać, że zmienne oznaczają te same obiekty we wszystkich wystąpieniach w regule; gdyby było inaczej, byłoby to niezgodne z duchem definicji w ogóle. Na przykład w powyższej regule opisującej ptaka nie można wykazać, że Fred jest ptakiem tylko dlatego, że Fido jest zwierzęciem, a Maria ma pióra. Ta sama zasada jednolitej interpretacji zmiennych obowiązuje także w regułach Prologu. Jeśli `X` w jednym miejscu oznacza Freda, to wszystkie `X` w tej samej regule oznaczają też Freda.

Reguła to *ogólne stwierdzenie dotyczące obiektów i ich powiązań*. Możemy na przykład powiedzieć, że Fred jest ptakiem, jeśli Fred jest zwierzęciem i Fred ma pióra, oraz że Bertrand jest ptakiem, jeśli Bertrand jest zwierzęciem i Bertrand ma pióra. Możemy zatem pozwolić, by zmienna miała różne wartości w przypadkach *użycia* reguły, zaś w samej regule musi być interpretowana jednolicie.

Przyjrzyjmy się teraz kilku przykładom, począwszy od reguły z jedną zmienną i z koniunkcją.

Jan lubi każdego, kto lubi wino,

czyli

Jan lubi wszystko, jeśli to lubi wino,

a gdy zapiszemy to samo za pomocą zmiennych

Jan lubi X, jeśli X lubi wino.

W Prologu reguła składa się z *głowy* i *treści*, które połączone są symbolem składającym się z dwukropka i myślnika (`:-`). Powyższą regułę można zatem zapisać następująco:

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

Należy zauważyć, że reguły także kończą się `kropką`. Głowa powyższej reguły to `lubi(jan,X)`. Treść, tym razem `lubi(X,wino)`, zawiera koniunkcję celów, które muszą być spełnione, aby głowa była prawdziwa. Możemy na przykład uczynić Jana osobą staranniej szukającą obiektów sympatii, dodając do treści reguły więcej celów rozdzielonych `przecinkami`:

```
lubi(jan,X) :- lubi(X,wino), lubi(X,jedzenie).
```

Oznacza to, że Jan lubi wszystkich, którzy lubią wino i jedzenie. Załóżmy teraz, że Jan lubi panie, które lubią wino:

```
lubi(jan,X) :- kobieta(X), lubi(X,wino).
```

Zawsze kiedy analizujemy regułę Prologu, musimy zwrócić uwagę na to, gdzie znajdują się w niej zmienne. W powyższej regule trzykrotnie użyto zmiennej `X`. Kiedy

zmienna ta jest ukonkretniana jakimś obiektem, jest ukonkretniana w *całym swoim zakresie obowiązywania*. W konkretnych przypadkach użycia reguły zakres obowiązywania to cała reguła, od głowy do kropki na końcu. Wobec tego, jeśli w powyższej regule zmienna `X` zostanie ukonkretniona, aby wskazywała Marię, Prolog będzie starał się uzgodnić cele `kobieta(maria)` i `lubi(maria,wino)`.

Następnie przyjrzyjmy się regule, w której używana jest więcej niż jedna zmienna. Niech baza danych zawiera fakty dotyczące rodziny królowej Wiktorii. Będziemy korzystać z predykatu `rodzice` mającego trzy argumenty: `rodzice(X,Y,Z)` oznacza, że *rodzice X to Y i Z*. Drugi argument to matka, a trzeci to ojciec. Użyjemy też predykatów `kobieta` i `mezczyzna`, które nie wymagają objaśnienia. Oto przykład części zawartości bazy:

```
mezczyzna(albert).
mezczyzna(edward).

kobieta(alicja).
kobieta(wiktoria).

rodzice(edward,wiktoria,albert).
rodzice(alicja,wiktoria,albert).
```

Teraz użyjemy opisanej wcześniej reguły dotyczącej *siostry*. W regule tej definiujemy dwuargumentowy predykat `siostra(X,Y)` mówiący, że `X` jest siostrą `Y`. `X` jest siostrą `Y`, jeżeli:

- `X` jest kobietą,
- matką `X` jest `M`, a ojcem `O`,
- `Y` ma takich samych matkę i ojca, jak `X`.

Możemy zapisać zatem następującą regułę:

```
siostra(X,Y) :-
   kobieta(X),
   rodzice(X,M,O),
   rodzice(Y,M,O).
```

Albo tę samą regułę można zapisać w jednym wierszu:

```
siostra(X,Y) :- kobieta(X), rodzice(X,M,O), rodzice(Y,M,O).
```

Matkę i ojca oznaczają zmienne `M` i `O`, choć moglibyśmy użyć też zmiennych `Matka` i `Ojciec`. Zauważmy, że te zmienne nie występują w głowie reguły; niemniej są traktowane tak samo, jak wszelki inne zmienne. Kiedy Prolog używa reguły, zmienne `M` i `O` początkowo są nieukonkretnione, więc podczas pierwszego dopasowywania celu `rodzice(X,M,O)` są im przypisywane wartości. Kiedy jednak już raz zostaną ukonkretnione, ukonkretnienie to dotyczy *wszystkich* wystąpień `M` i `O` w całej regule. Poniższy przykład pomoże zrozumieć sposób użycia tych zmiennych. Zadajmy zapytanie:

```
?- siostra(alicja,edward).
```

Prolog będzie przetwarzał takie zapytanie następująco:

1. Najpierw zapytanie dopasowywane jest do głowy reguły `sister`, więc `X` ukonkretniana jest jako `alicja`, a `Y` jako `edward`. Znacznik odpowiadający zapytaniu umieszczony jest na znalezionej regule. Następnie Prolog próbuje spełnić kolejno trzy cele z treści reguły.
2. Pierwszym celem jest `kobieta(alicja)`, gdyż wcześniej `X` ukonkretniono wartością `alicja`. Cel jest prawdziwy, gdyż istnieje odpowiedni fakt w bazie, więc ten cel nie zawodzi. W tej sytuacji Prolog oznacza odpowiednie miejsce w bazie danych (trzeci zapis). Nie dochodzi do ukonkretnienia żadnych dodatkowych zmiennych i Prolog przechodzi do uzgadniania następnego celu.
3. Prolog szuka faktu `rodzice(alicja,M,O)`, gdzie `M` i `O` dopiero mają być ukonkretnione. Znaleziony zostaje fakt `rodzice(alicja,wiktoria,albert)`, więc cel jest uzgodniony. Prolog oznacza w bazie danych szóstą pozycję, ukonkretnia `M` wartością `wiktoria` i `O` wartością `albert` (można ponownie użyć zapisu wartości pod celem). Prolog przechodzi do uzgadniania następnego celu.
4. Obecnie Prolog szuka faktu `rodzice(edward,wiktoria,albert)`, gdyż już w zapytaniu ukonkretniono `Y` jako `edward`, zaś w poprzednim kroku ukonkretniono `M` i `O`. Cel udaje się uzgodnić, oznaczany jest odpowiedni fakt (piąty w bazie). Jako że był to ostatni cel koniunkcji, cała koniunkcja jest prawdziwa i fakt `siostra(alicja,edward)` uznawany jest za prawdziwy, wobec czego Prolog odpowiada **yes**.

Załóżmy teraz, że interesuje nas czy Alicja jest czyjąś siostrą; odpowiednie zapytanie ma postać

```
?- siostra(alicja,X).
```

Prolog postąpi w następujący sposób:

1. Zapytanie pasuje do głowy reguły `siostra`; zmienna `X` reguły jest ukonkretniana wartością `alicja`. Zmienna `X` zapytania pozostaje nieukonkretniona, więc nieukonkretniona jest też zmienna `Y` reguły. Zmienne te jednak zostają ze sobą *powiązane*: kiedy jedna z nich zostanie ukonkretniona, druga zostanie ukonkretniona tak samo. Na razie jednak obie są jeszcze wolne.
2. Pierwszy cel, `kobieta(alicja)`, udaje się uzgodnić jak poprzednio.
3. Drugi cel to `rodzice(alicja,M,O)`, jest on dopasowywany do faktu `rodzice(alicja,wiktoria,edward)`. Znamy już zatem wartości zmiennych `M` i `O`.
4. Wartość `Y` nie jest jeszcze znana, więc trzecim celem jest `rodzice(Y,wiktoria,albert)`. Cel ten dopasowywany jest do faktu `rodzice(edward,wiktoria,albert)`, więc zmienna `Y` ukonkretniona jest wartością `edward`.
5. Cel jest uzgodniony, więc uzgodniona jest cała reguła z wartościami `X` jako `alicja` (znana z zapytania) i `Y` jako `edward`. Jako że zmienna `Y` z reguły jest powiązana ze zmienną `X` z zapytania, `X` także ma wartość `edward`. Prolog wyświetla wynik, **`X=edward`**.

Jak zwykle, Prolog czeka na decyzję, czy ma szukać kolejnych rozwiązań zapytania. To akurat zapytanie, które zadaliśmy, więcej rozwiązań już nie ma, zaś sposób dojścia Prologu do tego wniosku jest przedmiotem ćwiczenia z końca niniejszego rozdziału.

Jak widzieliśmy, Prolog może pobierać informacje o predykacie `lubi` w dwóch postaciach: możemy podawać fakty i reguły. Ogólnie rzecz biorąc, predykat definiuje się jako zbiór faktów i reguł; jedne i drugie nazywamy *klauzulami* predykatu. Słowa *klauzula* używać będziemy zawsze mając na myśli fakt lub regułę.

Zastanówmy się teraz nad następującym zagadnieniem: *dana osoba może coś ukraść, jeśli jest złodziejem i coś lubi, zaś to coś jest wartościowe*. W Prologu zapisujemy to:

```
moze_ukrasc(P,T) :- zlodziej(P), lubi(P,T).
```

Używamy dwuargumentowego predykatu `moze_ukrasc`, gdzie `P` oznacza potencjalnego złodzieja, a `T` kradzioną rzecz. Reguła ta opiera się na klauzulach `zlodziej` i `lubi`; obie one mogą być zapisane jako mieszanina faktów i reguł. Przyjmijmy, że baza danych jest taka sama jak poprzednio, ale między symbole `/*...*/` wstawiliśmy numery klauzul. Pokazane symbole służą do zapisu *komentarzy*. Komentarze są przez Prolog pomijane, ale można je dodawać do programu, aby zwiększyć jego czytelność. Dalej będziemy odwoływać się do umieszczonych w komentarzach numerów klauzul.

```
/*1*/  zlodziej(jan).

/*2*/  lubi(maria,czekolada).
/*3*/  lubi(maria,wino).
/*4*/  lubi(jan,X) :- lubi(X,wino).

/*5*/  moze_ukrasc(X,Y) :- zlodziej(X), lubi(X,Y).
```

Zauważmy, że definicja `lubi` składa się z trzech klauzul: dwóch faktów i jednej reguły. Teraz przyjrzyjmy się, co się będzie działo, kiedy zadamy zapytanie *Co może ukraść Jan?* Najpierw zapiszmy to zapytanie w formie:

```
?- moze_ukrasc(jan,X).
```

Prolog będzie działał następująco:

1. Najpierw odszukana zostanie klauzula pasująca do `moze_ukrasc`, czyli klauzula 5. Prolog oznaczy to miejsce w bazie danych. Jest to reguła, więc aby prawdziwa była głowa, spełnione muszą być cele z jej treści. Wobec tego zmienna `X` z reguły jest ukonkretniana wartością `jan` z zapytania. Widzimy, że mamy do ukonkretnienia dwie zmienne: `X` z zapytania i `Y` z reguły, obie zmienne zostają powiązane. Aby reguła została uzgodniona, uzgodnione muszą zostać jej cele; szukanie zaczyna się od pierwszego celu, `zlodziej(jan)`.
2. Cel jest uzgadniany, gdyż istnieje po prostu fakt `zlodziej(jan)` (klauzula 1.). Prolog oznacza to miejsce w bazie danych, nie są ukonkretniane żadne inne zmienne. Prolog następnie stara się spełnić drugi cel klauzuli 5. `X` jest równoważne wartości `jan`, więc Prolog szuka dopasowań do `lubi(jan,Y)`. Zmienna `Y` nadal nie jest ukonkretniona.

3. Cel `lubi(jan,Y)` pasuje do głowy reguły z klauzuli 4. Zmienna `Y` występująca w treści reguły jest powiązana z `X` z głowy, obie zmienne są nieukonkretnione. Aby regułę spełnić, szukamy celu `lubi(X,wino)`.
4. Cel jest spełniany, gdyż istnieje fakt `lubi(maria,wino)` (klauzula 3.). Wobec tego `X` oznacza teraz nazwę `maria`. Cel klauzuli 4. został spełniony, więc spełniona jest cała reguła i wynika z niej fakt `lubi(jan,maria)`. Wobec faktu powiązania `Y` z klauzuli 5. z `X` (z klauzuli 4.), `X` też otrzymuje wartość `maria`.
5. Klauzula 5. jest spełniona, zmienna `Y` jest ukonkretniona wartością `maria`. `Y` była powiązana z drugim argumentem zapytania, więc szukana wartość to `maria`.

Wybraliśmy ten przykład, aby pokazać, jak łatwo jest uzyskać niespodziewane rezultaty, właśnie takie jak „Jan może ukraść Marię". Do tego wniosku Prolog doszedł następująco:

> *Aby ukraść cokolwiek, Jan musi być przede wszystkim złodziejem. Z klauzuli 1. wynika, że tak właśnie jest. Następnie Jan musi lubić to, co ma ukraść. Z klauzuli 4. wynika, że Jan lubi wszystko, co lubi wino. Z klauzuli 3. wynika, że Maria lubi wino, wobec czego Jan lubi Marię. Wobec tego, że spełnione są obydwa warunki ukradzenia czegoś, Jan może ukraść Marię.*

Zauważmy, że fakt, iż Maria lubi czekoladę (klauzula 2.) jest w całym rozumowaniu w ogóle nie używany.

W powyższym przykładzie wielokrotnie w kolejnych klauzulach używaliśmy zmiennych `X` i `Y`. Na przykład w regule `moze_ukrasc` zmienna `X` oznacza obiekt, który może coś ukraść, zaś w regule `lubi` oznacza obiekt lubiany. Aby nasz program działał prawidłowo, Prolog musi uwzględnić, że ta sama zmienna w dwóch różnych wywołaniach klauzul może mieć różne znaczenie. Znajomość zakresu obowiązywania zmiennych pozwala uniknąć pomyłek w tym zakresie. Można byłoby użyć łatwiejszych do zinterpretowania nazw zmiennych, ale użyliśmy krótkich, niewiele mówiących nazw, aby pokazać, jak ustalany jest zakres obowiązywania zmiennych.

Podsumowanie i ćwiczenia

Omówiliśmy już najważniejsze zasady Prologu. W szczególności zapoznaliśmy się z:

- Dodawaniem faktów dotyczących obiektów.
- Zadawaniem zapytań o fakty.
- Użyciem zmiennych i ich zakresami.
- Koniunkcją jako sposobem łączenia zdań spójnikiem „*i*".
- Zapisywaniem relacji jako reguł.
- Podstawami nawracania.

Tak mały zbiór technik wystarcza do tworzenia przydatnych programów obsługujących niewielkie bazy danych. Warto w celu utrwalenia materiału wykonać podane poniżej ćwiczenia.

Przed rozpoczęciem pisania programów w wybranym systemie Prologu należy zajrzeć do podręcznika, aby obejrzeć przykład sesji. Nieco praktycznych porad znajduje się także w rozdziale 8.

Teraz, kiedy znamy już podstawy Prologu, możemy przejść do następnego rozdziału, w którym wyjaśnione zostaną niektóre zagadnienia w tym rozdziale tylko wspomniane. Pokażemy też, jak w Prologu traktowane są liczby. W następnych kilku rozdziałach będziemy mogli docenić możliwości Prologu i wygodę jego użycia.

Ćwiczenie 1.2. Kiedy reguła `siostra` zostanie zastosowana do pokazanej wcześniej bazy danych opisującej rodzinę królowej Wiktorii, otrzymamy więcej niż jedną odpowiedź. Wyjaśnij, skąd te odpowiedzi się biorą i jakie one są.

Ćwiczenie 1.3. Inspiracją do tego ćwiczenia było ćwiczenie z książki „Logic for Problem Solving" (*Logika a rozwiązywanie problemów*) Roberta Kowalskiego wydanej przez North Holland w 1979 roku. Załóżmy, że zapisano w formie klauzul Prologu następujące relacje:

```
ojciec(X,Y)         /* X jest ojcem Y */
matka(X,Y)          /* X jest matką Y */
mezczyzna(X)        /* X jest mężczyzną */
kobieta(X)          /* X jest kobietą */
rodzic(X,Y)         /* X jest rodzicem Y */
diff(X,Y)           /* X i Y są różne */
```

Należy zapisać klauzule definiujące relacje:

```
jest_matka(X)       /* X jest matką */
jest_ojcem(X)       /* X jest ojcem */
jest_synem(X)       /* X jest synem */
siostra(X,Y)        /* X jest siostrą Y */
dziadek(X,Y)        /* X jest dziadkiem Y */
rodzenstwo(X,Y)     /* X i Y są rodzeństwem */
```

Przykładowo, można byłoby napisać regułę `ciotka` korzystającą z danych wcześniej reguł `kobieta`, `rodzenstwo` i `rodzic`:

```
ciotka(X,Y) :- kobieta(X), rodzenstwo(X,Z), rodzic(Z,Y).
```

Można też tę regułę zapisać inaczej:

```
ciotka(X,Y) :- siostra(X,Z), rodzic(Z,Y).
```

gdy ma się do dyspozycji regułę `siostra`.

Ćwiczenie 1.4. Korzystając z przedstawionej w tym rozdziale reguły `siostra` wyjaśnij, dlaczego obiekt może być swoją własną siostrą. Jak należy zmienić tę regułę, aby tak nie było? Wskazówka: należy założyć, że dany jest predykat `diff` z ćwiczenia 1.3.