Baza programisty

Pragmatic Bookshelf

Siedem języków w siedem tygodni

Praktyczny przewodnik nauki **języków programowania**

Siedmiotygodniowa podróż po czterech odmiennych **paradygmatach programowania**, siedmiu różnych **stylach składni** i czterech dekadach **rozwoju języków**!

- Poznaj najważniejsze modele programowania i techniki obsługi współbieżności
- Opanuj tajniki systemu prototypów i dynamicznych typów
- Zostań wszechstronnym programistą, gotowym zmierzyć się z każdym projektem!

Helion

Bruce A. Tate

Siedem języków w siedem tygodni. Praktyczny przewodnik nauki języków programowania

Autor: Bruce A. Tate
Tłumaczenie: Radosław Meryk
ISBN: 978-83-246-3379-1
Tytuł oryginału: Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages
Format: 168×237, stron: 368

Siedmiotygodniowa podróż po czterech odmiennych paradygmatach programowania, siedmiu różnych stylach składni i czterech dekadach rozwoju języków!

- Poznaj najważniejsze modele programowania i techniki obsługi współbieżności
- Opanuj tajniki systemu prototypów i dynamicznych typów
- Zostań wszechstronnym programistą, gotowym zmierzyć się z każdym projektem!

Jeśli myślisz, że to kolejna książka z serii „Jak schudnąć 50 kilogramów w trzy dni" albo „Jak zostać obrzydliwie bogatym w dwa tygodnie", na szczęście się mylisz! Oto podręcznik, który w siedem tygodni przedstawi Ci najważniejsze modele programowania na przykładzie siedmiu przydatnych języków. Zaproponowana tu innowacyjna forma nauki pozwoli Ci poznawać je dzień po dniu. Zaczniesz od krótkiego omówienia składni i możliwości każdego języka, by na końcu wypróbować go w akcji. I choć po lekturze tej książki nie staniesz się ekspertem, opanujesz to, co w każdym z przedstawionych tu języków jest kluczowe. Będziesz mógł tworzyć czytelniejszy, lepszy kod z mniejszą ilością powtórzeń. Zdobędziesz także niezwykle cenną umiejętność – zaczniesz sprawnie wykorzystywać pojęcia z jednego języka w celu znalezienia kreatywnych rozwiązań w innym!

W książce tej opisano jeden język programowania logicznego, dwa z pełną obsługą pojęć obiektowych, cztery o charakterze funkcyjnym i jeden prototypowy – wszystko po to, by zapewnić Ci możliwie najbardziej wszechstronne przygotowanie programistyczne. Lepiej przyswoisz sobie także techniki obsługi współbieżności, będące kręgosłupem następnej generacji aplikacji internetowych, oraz poznasz sposoby wykorzystywania filozofii „Let it crash" Erlanga do budowy systemów odpornych na awarie.

Jakie praktyczne języki poznasz dzięki tej książce?

- Ruby – język obiektowy, a przy tym łatwy w użytkowaniu i czytelny
- Io – prototypowy język, wyposażony w unikatowy mechanizm dystrybucji komunikatów
- Prolog – język oferujący łatwe rozwiązania, które w Javie lub C byłyby bardzo kłopotliwe
- Scala – jeden z języków nowej generacji, przeznaczony na maszynę wirtualną Javy
- Erlang – język funkcyjny, z mechanizmami obsługi współbieżności, na którym działa już kilka słynnych baz danych w stylu cloud
- Clojure – język, w którym wykorzystano strategię wersjonowania baz danych w celu zarządzania współbieżnością
- Haskell – język o charakterze czysto funkcyjnym

Jeden z tych języków może już wkrótce stać się Twoim ulubionym narzędziem!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.
Prolog

Sally Dibbs, Dibbs Sally. 461-0192.
Raymond

Ach, ten Prolog. Czasami spektakularnie błyskotliwy, innym razem tak samo frustrujący. Zdumiewające odpowiedzi uzyskujemy tylko wtedy, kiedy wiemy, jak należy zadawać pytania. Porównałbym go do postaci z *Rain Mana*[1]. Pamiętam, jak jeden z głównych bohaterów, Raymond, niewiele myśląc, wyrecytował numer telefonu Sally Dibbs po przeczytaniu dzień wcześniej książki telefonicznej. Zarówno w przypadku Raymonda, jak i Prologu często zadaję sobie dwa pytania: „Skąd on to wiedział" i „Jak on mógł tego nie wiedzieć?". Jest kopalnią wiedzy, jeśli tylko uda nam się właściwie sformułować pytanie.

Prolog znacząco różni się od innych języków, z którymi dotychczas mieliśmy do czynienia. Zarówno Io, jak i Ruby zaliczają się do **języków imperatywnych**. W językach imperatywnych formułujemy „przepisy". Dokładniej mówiąc, instruujemy komputer, w jaki sposób należy wykonać zadanie. Języki imperatywne wyższego poziomu dają programistom nieco więcej swobody. Pozwalają na połączenie wielu dłuższych kroków w jeden. Ogólnie rzecz biorąc, programowanie sprowadza się jednak do zdefiniowania listy składników i opisania krok po kroku procesu wypiekania ciasta.

[1] *Rain Man*. DVD. Reżyseria Barry Levinson. 1988; Los Angeles, CA: MGM, 2000.

Zanim podjąłem próbę napisania tego rozdziału, poświęciłem kilka tygodni na eksperymentowanie z Prologiem. W tym czasie skorzystałem z kilku samouczków. Studiowałem między innymi przykłady z samouczka J.R. Fishera[2]. W poznaniu terminologii i struktury programu pomógł mi też samouczek A. Aaby'ego[3]. Wykonałem również wiele samodzielnych ćwiczeń.

Prolog jest językiem deklaratywnym. Programista podaje fakty i reguły wnioskowania, a Prolog znajduje rozwiązanie. To tak, jakbyśmy poszli do dobrego cukiernika. Opisujemy mu ciastka, które nam smakują, a on sam, na podstawie przekazanych reguł, dobiera składniki i piecze ciasto. Programując w Prologu, nie trzeba znać odpowiedzi na pytanie **jak**. Za wyciąganie wniosków odpowiedzialny jest komputer.

Wystarczy trochę poszperać w internecie, aby znaleźć przykłady rozwiązania sudoku za pomocą programu składającego się z mniej niż dwudziestu linijek kodu. Są też programy do układania kostki Rubika, czy też rozwiązywania popularnych łamigłówek, takich jak Wieża Hanoi (zaledwie kilkanaście linijek). Prolog był jednym z pierwszych języków programowania logicznego, które odniosły sukces. Programista formułuje logiczne twierdzenia, a Prolog ocenia, czy są one prawdziwe. W podawanych twierdzeniach mogą być luki. Prolog stara się wypełnić luki w taki sposób, by niekompletne fakty tworzyły prawdziwe stwierdzenia.

4.1. O Prologu

Prolog jest językiem programowania logicznego opracowanym w 1972 roku przez Alaina Colmerauera i Phillippe'a Roussela. Język ten zyskał popularność w przetwarzaniu języka naturalnego. Obecnie ten szacowny język dostarcza podstaw programowania dla szerokiej gamy problemów — począwszy od planowania zadań, a skończywszy na systemach ekspertowych. Ten bazujący na regułach język można wykorzystać do wyrażania logiki i zadawania pytań. Tak jak SQL, Prolog przetwarza bazy danych, choć w przypadku Prologu dane składają się z reguł i związków logicznych. Tak jak SQL, Prolog można podzielić na dwie części: jednej do opisywania danych i drugiej do zadawania pytań o dane. W Prologu dane mają postać logicznych reguł. Oto podstawowe bloki budulcowe Prologu:

---

[2] *http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html*

[3] *http://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tutorial.html*

- **Fakty**. Fakt jest podstawowym twierdzeniem dotyczącym rzeczywistości (Piggy to świnia, świnie lubią błoto).
- **Reguły**. Reguła definiuje wniosek dotyczący faktów w określonej rzeczywistości (zwierzę lubi błoto, jeśli jest świnią).
- **Zapytanie**. Zapytanie jest pytaniem dotyczącym wybranej rzeczywistości (czy Piggy lubi błoto?).

Fakty i reguły są zapisywane w **bazie wiedzy**. Kompilator Prologu kompiluje bazę wiedzy, przekształcając ją na postać pozwalającą na wydajne formułowanie zapytań. Studiując przykłady zamieszczone w tym rozdziale, będziemy wykorzystywali Prolog do przedstawienia bazy wiedzy. Następnie spróbujemy bezpośrednio wydobyć dane i skorzystać z Prologu do powiązania ze sobą reguł w taki sposób, aby zdobyć informacje, których nie znamy.

Dość wprowadzenia. Zabierzmy się do pracy.

4.2. Dzień 1. Świetny kierowca

W *Rain Manie* Raymond powiedział swojemu bratu, że jest świetnym kierowcą, ponieważ potrafi sprawnie prowadzić samochód po parkingu z prędkością 10 kilometrów na godzinę. Raymond używał wszystkich głównych podzespołów samochodu — kierownicy, hamulców, pedału gazu — ale używał ich w ograniczony sposób. Taki jest nasz dzisiejszy cel. Użyjemy Prologu do sformułowania pewnych faktów, zdefiniowania reguł oraz wykonania pewnych podstawowych zapytań. Prolog, tak jak Io, jest językiem o niezwykle prostej składni. Bardzo szybko można się nauczyć podstawowych reguł. Prawdziwa zabawa zaczyna się w chwili, kiedy pojęcia ułożą się w warstwy, tworząc interesującą kompozycję. Jeśli to jest dla czytelnika pierwsze spotkanie z Prologiem, to gwarantuję, że albo zmieni sposób swojego myślenia, albo będzie skazany na niepowodzenie. Szersze rozwinięcie tematu pozostawimy na dalszy dzień.

Najpierw podstawa. Trzeba przygotować działającą instalację. Dla potrzeb niniejszej książki używam GNU Prolog w wersji 1.3.1. Należy zachować ostrożność. Dialekty Prologu różnią się pomiędzy sobą. Będę się starał, aby stąpać po „wspólnym gruncie", ale czytelnicy, którzy wybiorą inną wersję Prologu, będą musieli odrobić zadanie domowe. Dzięki temu zrozumieją różnice występujące w wybranej

przez siebie odmianie języka. Poniżej zamieściłem opis sposobu korzystania z języka — niezależnie od wybranej wersji.

Proste fakty

W niektórych językach używanie wielkich i małych liter jest wyłącznie w gestii programisty, ale w Prologu wielkość liter ma znaczenie. Jeśli słowo rozpoczyna się małą literą, to jest to **atom** — ustalona wartość, tak jak symbol w Ruby. Jeśli natomiast rozpoczyna się wielką literą lub symbolem podkreślenia, to jest to **zmienna**. Zmienne mogą się zmieniać, atomy nie. Spróbujmy stworzyć prostą bazę wiedzy zawierającą kilka faktów. Wpisz poniższy kod w edytorze:

Pobierz: prolog/przyjaciele.pl

```
lubi(wallace, ser).
lubi(gromit, ser).
lubi(wendolene, owce).

przyjaciel(X, Y) :- \+(X = Y), lubi(X, Z), lubi(Y, Z).
```

Powyższy plik to baza wiedzy zawierająca fakty i reguły. Pierwsze trzy instrukcje to fakty, natomiast ostatnia instrukcja jest regułą. Fakty są bezpośrednimi obserwacjami rzeczywistości. Reguły są logicznymi wnioskami dotyczącymi rzeczywistości. Na razie zwróćmy uwagę na pierwsze trzy wiersze. Wszystkie one opisują fakty. `wallace`, `gromit` i `wendolene`[4] są atomami. Fakty można zinterpretować następująco: `wallace` lubi `ser`, `gromit` lubi `ser`, a `wendolene` lubi `owce`. Spróbujmy wykorzystać fakty w praktyce.

Uruchamiamy interpreter Prologu. Czytelnicy używający wersji GNU Prolog powinni w tym celu wpisać polecenie `gprolog`. Następnie w celu załadowania pliku należy wpisać następujące polecenie:

```
| ?- ['przyjaciele.pl'].
compiling C:/Seven Languages/przyklady/przyjaciele.pl for byte code...
C:/Seven Languages/przyklady/przyjaciele.pl compiled, 5 lines read - 976 bytes written, 15 ms

(16 ms) yes
| ?-
```

Jeśli Prolog nie oczekuje na pośrednie wyniki, to udzieli odpowiedzi `yes` (tak) lub `no` (nie). W naszym przypadku załadowanie pliku zakończyło się pomyślnie, dlate-

[4] Wallace, Gromit i Wendolene to postacie z brytyjskiego filmu animowanego *Wallace i Gromit: Golenie owiec*, 1995 — *przyp. tłum.*

go Prolog odpowiedział yes. Możemy zacząć zadawać pytania. Najprostsze są pytania o potwierdzenie bądź zaprzeczenie określonych faktów. Spróbujmy zadać kilka tego rodzaju pytań:

```
| ?- lubi(wallace, owce).

no
| ?- lubi(gromit, ser).

yes
```

Powyższe pytania są dość intuicyjne. Czy wallace lubi owce? Nie. Czy gromit lubi ser? Tak. Nie jest to zbyt ciekawe. Prolog jedynie powtarza jak papuga fakty z bazy wiedzy. Nieco ciekawiej robi się w przypadku, gdy spróbujemy zbudować jakąś logikę. Przyjrzyjmy się mechanizmowi wnioskowania.

Proste wnioski i zmienne

Wypróbujmy regułę przyjaciel:

```
| ?- przyjaciel(wallace, wallace).

no
```

Jak widać, Prolog analizuje wpisane reguły i odpowiada na pytania twierdząco bądź przecząco. Wnioskowanie to coś więcej niż widać na pierwszy rzut oka. Ponownie przyjrzyjmy się regule przyjaciel:

Aby X mógł być przyjacielem Y, nie może być taki sam jak Y. Przyjrzyjmy się pierwszej części występującej z prawej strony symbolu :-. Jest to tzw. **cel częściowy** (ang. *subgoal*). \+ jest operatorem negacji logicznej. Zatem instrukcja \+(X = Y) oznacza X nie równa się Y.

Wypróbujmy kilka innych zapytań:

```
| ?- przyjaciel(gromit, wallace).

yes
| ?- przyjaciel(wallace, gromit).

yes
```

X jest przyjacielem Y, jeśli można udowodnić, że X lubi jakieś Z, a Y lubi to samo Z. Zarówno wallace, jak i gromit lubią ser, dlatego odpowiedź na to pytanie jest twierdząca.

Spróbujmy zagłębić się w kod. W zapytaniach X nie jest równe Y, co udowadnia pierwszy cel częściowy. W zapytaniu będą użyte pierwszy i trzeci cel częściowy:

`lubi(X, Z)` i `lubi(Y, Z)`. `gromit` i `wallace` lubią `ser`, zatem udowodniliśmy drugi i trzeci cel częściowy. Wypróbujmy inne zapytanie:

```
| ?- przyjaciel(wendolene, gromit).

no
```

W tym przypadku interpreter Prologu musiał wypróbować kilka możliwych wartości `X`, `Y` i `Z`:

- `wendolene`, `gromit` i `ser`;
- `wendolene`, `gromit` i `owce`.

Żadna kombinacja nie spełniała obu celów — tzn. aby `wendolene` lubiła `Z` i jednocześnie by `gromit` także lubił `Z`. Nie istnieje takie `Z`, dlatego maszyna wnioskowania udzieliła odpowiedzi „nie" — to nie są przyjaciele.

Spróbujmy sformalizować terminologię. Poniższa instrukcja:

```
przyjaciel(X, Y) :- \+(X = Y), lubi(X, Z), lubi(Y, Z).
```

jest regułą Prologu z trzema zmiennymi: `X`, `Y` i `Z`. Regułę tę opisujemy `przyjaciel/2` — jest to skrócona forma reguły `przyjaciel` z dwoma parametrami. Reguła ta zawiera trzy cele częściowe oddzielone od siebie przecinkami. Aby reguła była prawdziwa, wszystkie cele częściowe muszą być prawdziwe. A zatem nasza reguła oznacza, że `X` jest przyjacielem `Y`, jeśli `X` i `Y` nie są sobie równe, a jednocześnie zarówno `X`, jak i `Y` lubią to samo `Z`.

Wypełnianie luk

Wykorzystaliśmy Prolog do uzyskania odpowiedzi „tak" lub „nie" na postawione pytania. Zastosowania Prologu nie ograniczają się jednak wyłącznie do tego. W tym punkcie wykorzystamy maszynę wnioskowania do znalezienia wszystkich możliwych odpowiedzi na pytanie. Aby to zrobić, określimy w zapytaniu **zmienną**.

Przeanalizujmy następującą bazę wiedzy:

Pobierz: prolog/zywnosc.pl

```
zywnosc_typ(velveeta, ser).
zywnosc_typ(ritz, krakers).
zywnosc_typ(konserwa, mięso).
zywnosc_typ(kiełbasa, mięso).
zywnosc_typ(jolt, napój).
zywnosc_typ(twinkie, deser).
```

```
smak(słodki, deser).
smak(pikantny, mięso).
smak(pikantny, ser).
smak(słodki, napój).

zywnosc_smak(X, Y) :- zywnosc_typ(X, Z), smak(Y, Z).
```

W bazie wiedzy mamy kilka faktów. Niektóre z nich, na przykład `zywnosc_typ(velveeta, ser)`, oznaczają, że żywność jest określonego typu. Inne, na przykład `smak(słodki, deser)`, oznaczają, że żywność określonego typu ma charakterystyczny smak. Na koniec mamy regułę o nazwie `zywnosc_smak`, która umożliwia wywnioskowanie smaku żywności określonego typu. Żywność `X` ma `zywnosc_smak Y`, jeśli żywność posiada `zywnosc_typ Z` oraz to `Z` ma charaterystyczny smak. Spróbujmy skompilować ten skrypt:

```
| ?- ['code/prolog/zywnosc.pl'].
compiling C:/Seven Languages/przyklady/zywnosc.pl for byte code...
C:/Seven Languages/przyklady/zywnosc.pl compiled, 13 lines read - 1567 bytes written, 15 ms

yes
```

Możemy teraz zadać kilka pytań.

```
| ?- zywnosc_typ(Co, mięso).

Co = konserwa ? ;

Co = kiełbasa ? ;

no
```

Zwróćmy uwagę na interesujący aspekt. Daliśmy Prologowi zadanie: „Znajdź pewną wartość zmiennej `Co`, która spełnia zapytanie `zywnosc_typ(Co, mięso)`". Prolog znalazł jedną taką wartość: `konserwa`. Kiedy wpisaliśmy polecenie ; poprosiliśmy, aby Prolog znalazł inną wartość zmiennej. W odpowiedzi uzyskaliśmy wartość `kiełbasa`. Wartości te łatwo było znaleźć, ponieważ zapytania bazowały na prostych faktach. Następnie zadaliśmy pytanie o kolejną wartość, a Prolog odpowiedział `no`. Takie zachowanie jest trochę niespójne. Gdyby Prolog potrafił wykryć, że nie ma więcej alternatywnych wartości, zobaczylibyśmy odpowiedź `yes`. Jeśli Prolog nie może natychmiast stwierdzić, czy istnieje więcej alternatywnych rozwiązań, bez dodatkowych obliczeń, zwraca `no` i oczekuje na kolejne polecenie. Własność ta istnieje wyłącznie dla wygody użytkownika. Jeśli Prolog potrafi udzielić odpowiedzi wcześniej, to jej udzieli. Spróbujmy zadać kilka dodatkowych pytań:

```
| ?- zywnosc_smak(kiełbasa, słodki).
no

| ?- smak(słodki, Co).
```

```
Co = deser ? ;
Co = napój

yes
```

Nie, kiełbasa nie jest słodka. Jakiego typu żywność jest słodka? Deser i napój. Wszystko to są fakty. Pozwólmy jednak, aby Prolog sam wyciągnął wnioski:

```
| ?- zywnosc_smak(Co, pikantny).

Co = velveeta ? ;

Co = konserwa ? ;

Co = kiełbasa ? ;

no
```

Zapamiętajmy, `zywnosc_smak(X, Y)` to reguła, a nie fakt. Poprosiliśmy interpreter Prologu o znalezienie wszystkich wartości spełniających zapytanie „Jakie typy żywności mają pikantny smak?". Aby znaleźć rozwiązanie, Prolog musiał powiązać ze sobą proste fakty dotyczące żywności — jej typów i smaków. Silnik wnioskowania musiał przeanalizować wszystkie możliwe kombinacje wartości, dla których wszystkie cele częściowe zostały osiągnięte.

Kolorowanie map

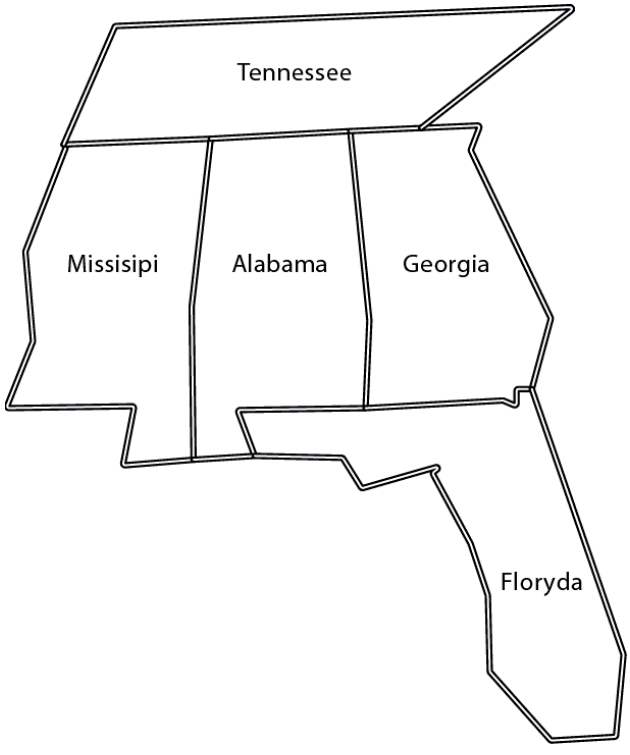
Spróbujmy wykorzystać tę samą koncepcję do kolorowania map. Przeanalizujmy poniższy przykład, aby uzyskać bardziej spektakularny pogląd na Prolog. Zamierzamy pokolorować mapę południowowschodniego rejonu Stanów Zjednoczonych. Na rysunku 4.1 zamieszczono mapę stanów, które będziemy kolorować. Zakładamy, że dwa stany pokolorowane na taki sam kolor nie mogą się ze sobą stykać.

Zakodujemy kilka prostych faktów.

Pobierz: prolog/mapa.pl

```
różne(czerwony, zielony). różne(czerwony, niebieski).
różne(zielony, czerwony). różne(zielony, niebieski).
różne(niebieski, czerwony). różne(niebieski, zielony).

pokoloruj(Alabama, Missisipi, Georgia, Tennessee, Floryda) :-
    różne(Missisipi, Tennessee),
    różne(Missisipi, Alabama),
    różne(Alabama, Tennessee),
    różne(Alabama, Missisipi),
    różne(Alabama, Georgia),
    różne(Alabama, Floryda),
    różne(Georgia, Floryda),
    różne(Georgia, Tennessee).
```



Rysunek 4.1. Mapa wybranych południowowschodnich stanów USA

Mamy trzy kolory. Przekazujemy Prologowi informację o zbiorach różnych kolorów do wykorzystania w kolorowaniu mapy. Jest również reguła. W regule `pokoloruj` informujemy Prolog o tym, które stany ze sobą sąsiadują. To wszystko. Wypróbujmy poniższy kod:

```
| ?- pokoloruj(Alabama, Missisipi, Georgia, Tennessee, Floryda).

Alabama = niebieski
Floryda = zielony
Georgia = czerwony
Missisipi = czerwony
Tennessee = zielony ?
```

Z pewnością istnieje sposób pokolorowania tych pięciu stanów za pomocą trzech kolorów. Aby uzyskać inne możliwe kombinacje, wystarczy wpisać `a`. Wykonaliśmy zadanie zaledwie za pomocą kilkunastu linijek kodu. Logika jest śmiesznie prosta — nawet dziecko zorientowałoby się, o co w niej chodzi. W pewnym momencie trzeba jednak zadać sobie pytanie…

Gdzie jest program?

W powyższym kodzie nigdzie nie ma implementacji algorytmu! Spróbujcie rozwiązać taki sam problem w dowolnym języku proceduralnym. Czy to rozwiązanie byłoby zrozumiałe? Pomyślcie, co trzeba by było zrobić, aby rozwiązać podobnie złożony problem logiczny w takich językach jak Ruby lub Io. Oto jeden z możliwych sposobów postępowania:

1. sformułowanie logiki rozwiązania problemu,
2. wyrażenie logiki w programie,
3. znalezienie wszystkich możliwych danych wejściowych,
4. sprawdzenie działania programów dla tych danych.

Pisanie takiego programu mogłoby zająć sporo czasu. W Prologu logikę wyraża się w postaci faktów i wniosków. Następnie można zadawać pytania. Programista piszący programy w tym języku nie jest odpowiedzialny za tworzenie receptur krok po kroku. Prolog nie służy do zapisywania algorytmów w celu rozwiązywania problemów logicznych. Prolog służy do opisywania świata i prezentowania problemów logicznych, które komputer próbuje rozwiązywać.

Pozwólmy komputerom wykonywać swoją pracę.

Unifikacja. Część 1.

W tym momencie nadszedł czas, aby trochę się cofnąć i zaprezentować więcej teorii. Spróbujmy rzucić nieco więcej światła na unifikację. W niektórych językach stosujemy podstawienia zmiennych. Na przykład w Javie lub w Ruby `x = 10` oznacza: podstaw `10` do zmiennej `x`. Unifikacja dwóch struktur to dążenie do tego, aby obie stały się identyczne. Przeanalizujmy następującą bazę wiedzy:

Pobierz: prolog/zwierzeta.pl

```
kot(lew).
kot(tygrys).

dorota(X, Y, Z) :- X = lew, Y = tygrys, Z = niedźwiedź.
dwa_koty(X, Y) :- kot(X), kot(Y).
```

W tym przykładzie symbol `=` oznacza „zunifikuj” — tzn. spowoduj, aby obie strony były identyczne. W bazie wiedzy są zapisane dwa fakty: lwy i tygrysy są kotami. Mamy także dwie proste reguły: W regule `dorota/3` argumenty `X`, `Y` i `Z` to odpowiednio

lew, tygrys i niedźwiedź. W regule dwa_koty/2 argument X to kot i Y to kot. Powyższą bazę wiedzy możemy wykorzystać do dokładniejszego wyjaśnienia unifikacji.

Najpierw spróbujmy zastosować pierwszą regułę. Skompilujemy, a następnie wykonamy proste zapytanie bez parametrów.

```
| ?- dorota(lew, tygrys, niedźwiedź).
yes
```

Pamiętajmy: unifikacja oznacza „znajdź wartości, dla których dwie strony są sobie równe". Po prawej stronie Prolog wiąże argumenty X, Y i Z z wartościami lew, tygrys i niedźwiedź. Wartości te pasują do odpowiadających im wartości po lewej stronie. Oznacza to, że unifikacja przebiegła pomyślnie. Prolog odpowiada yes. Powyższy przypadek jest dosyć prosty, można go jednak trochę skomplikować. Unifikacja może działać po obu stronach implikacji. Spróbujmy wykonać następujący kod:

```
| ?- dorota(Jeden, Dwa, Trzy).

Dwa = tygrys
Jeden = lew
Trzy = niedźwiedź

yes
```

W tym przykładzie występuje dodatkowa warstwa pośrednia. W funkcji celu Prolog unifikuje argumenty X, Y i Z do wartości lew, tygrys i niedźwiedź. Po lewej stronie Prolog wiąże argumenty X, Y i Z ze zmiennymi Jeden, Dwa i Trzy, a następnie wyświetla wyniki.

Przejdźmy teraz do ostatniej reguły: dwa_koty/2. Reguła ta mówi, że dwa_koty(X, Y) jest prawdą, jeśli można udowodnić, że zarówno X, jak i Y to koty. Wypróbujmy poniższy kod:

```
| ?- dwa_koty(Jeden, Dwa).

Jeden = lew
Dwa = lew ?
```

Prolog wyświetlił pierwsze rozwiązanie: lew i lew to dwa koty. Przenalizujmy sposób dojścia do tej konkluzji:

1. Sformułowaliśmy zapytanie: dwa_koty(Jeden, Dwa). Prolog powiązał zmienną Jeden z X oraz Dwa z Y. W celu rozwiązania problemu Prolog musi obliczyć funkcje celu.
2. Pierwszy cel to kot(X).

3. Funkcję tę spełniają dwa fakty: `kot(lew)` i `kot(tygrys)`. Prolog podejmuje próbę sprawdzenia pierwszego faktu. Podstawia do argumentu `X` wartość `lew` i przechodzi do następnej funkcji celu.
4. Teraz Prolog wiąże zmienną `Y` z `kot(Y)`. Rozwiązanie tej funkcji celu Prolog znajduje w taki sam sposób jak w pierwszym przypadku — wybiera wartość `lew`.
5. Obie funkcje celu są spełnione, zatem reguła jest prawdziwa. Prolog wyświetlił wartości `Jeden` i `Dwa`, dla których reguła jest spełniona, i odpowiedział `yes`.

Mamy zatem pierwsze rozwiązanie, dla którego reguły są prawdziwe. Czasami jedno rozwiązanie nam wystarcza. Czasem potrzebujemy więcej niż jednego. Możemy teraz przeglądać po kolei inne rozwiązania, wpisując symbol ;. Możemy też zażądać wyświetlenia wszystkich pozostałych rozwiązań. W tym celu wystarczy wcisnąć `a`.

```
Dwa = lew ? a

Jeden = lew
Dwa = tygrys

Jeden = tygrys
Dwa = lew

Jeden = tygrys
Dwa = tygrys

(1 ms) yes
```

Zwróćmy uwagę, że Prolog przeanalizował listę wszystkich kombinacji argumentów `X` i `Y`, uwzględniając dostępne informacje w funkcjach celu oraz odpowiednich faktach. Jak przekonamy się później, unifikacja pozwala również na przeprowadzanie złożonego dopasowywania na podstawie struktury danych. Tyle wystarczy na pierwszy dzień. Nieco bardziej złożonymi przykładami zajmiemy się drugiego dnia.

Prolog w praktyce

Zetknięcie się z „programem" zaprezentowanym w ten sposób może być dość osobliwym przeżyciem. W Prologu często nie tworzymy precyzyjnych receptur krok po kroku, a jedynie przepis na placek, który trzeba wyjąć z pieca, kiedy już będzie gotowy. Kiedy uczyłem się Prologu, bardzo pomógł mi wywiad z osobą, która korzystała z tego języka w praktyce. Rozmawiałem z Brianem Tarboksem — naukow-

cem, który skorzystał z Prologu do opracowania harmonogramów pracy personelu laboratorium z delfinami w projekcie badawczym.

Wywiad z Brianem Tarboksem — naukowcem badającym delfiny

Bruce: *Czy może nam pan opowiedzieć o swoich doświadczeniach z nauki Prologu?*

Brian: *Prologu zacząłem się uczyć pod koniec lat osiemdziesiątych podczas studiów na Uniwersytecie Hawajskim w Manoa. Pracowałem w Kewalo Basin Marine Mammal Laboratory. Prowadziłem badania możliwości poznawczych delfinów butelkonosych. Większość dyskusji w laboratorium dotyczyła różnych teorii na temat sposobów myślenia delfinów. Pracowaliśmy głównie z delfinem o imieniu Akeakamai — zdrobniale nazywaliśmy go Ake. Wiele rozmów zaczynało się mniej więcej tak: „Wydaje mi się, że Ake postrzega tę sytuację w taki to a taki sposób…".*

Postanowiłem, że w mojej pracy skupię się na stworzeniu modelu pasującego do naszego postrzegania sposobu widzenia świata przez Ake'a. Miałem zamiar zaprezentować co najmniej podzbiór tego, nad czym prowadziliśmy badania. Gdyby ten model potrafił przewidzieć rzeczywiste zachowania Ake'a, zyskalibyśmy potwierdzenie naszych teorii dotyczących sposobu jego rozumowania.

Prolog jest cudownym językiem, ale dopóki nie przyzwyczaimy się do niego, otrzymywane wyniki mogą wydawać się nam dość dziwne. Pamiętam jedno z moich pierwszych doświadczeń z Prologiem. Napisałem coś w stylu $x = x + 1$. *Prolog odpowiedział „no". Języki zazwyczaj nie mówią „nie". Czasami można uzyskać błędne wyniki, innym razem program nie chce się skompilować, ale nigdy nie spotkałem się z językiem, który by do mnie mówił „nie". Zwróciłem się więc do pomocy technicznej i powiedziałem, że uzyskałem odmowę wykonania polecenia, gdy chciałem zmienić wartość zmiennej. Zapytali mnie: „A czemu miałaby służyć zmiana wartości zmiennej?". Pomyślałem, co u licha? Jaki język nie pozwala ci zmieniać wartości zmiennych? Po pewnym czasie poznawania Prologu staje się jasne, że zmienne albo mają jakąś konkretną wartość, albo są niezwiązane, ale wtedy jeszcze tego nie wiedziałem.*

Bruce: *W jaki sposób wykorzystał pan Prolog?*

Brian: *Stworzyłem dwa główne systemy: symulator delfina i program do tworzenia harmonogramów pracy laboratorium. Laboratorium prowadziło cztery doświadczenia dziennie z każdym z czterech delfinów. Musicie wiedzieć, że delfiny doświadczalne to*

niezwykle ograniczony zasób. Każdy delfin pracował na potrzeby innego doświadczenia, a każde doświadczenie wymagało innego personelu. Pewne role, na przykład trenera delfinów, mogło odgrywać zaledwie kilka osób. Inne role, na przykład rejestratorów danych, mogły być odgrywane przez szersze grono osób, ale pomimo to musiały to być osoby odpowiednio przeszkolone. Większość doświadczeń wymagała personelu złożonego z sześciu do dwunastu osób. Mieliśmy do dyspozycji licealistów, studentów i ochotników ze społeczności Earthwatch. Każda osoba miała własny plan pracy oraz indywidualny zakres umiejętności. Opracowanie takiego harmonogramu pracy laboratorium, który zapewniałby realizację wszystkich zadań, było pełnoetatowym zadaniem dla jednej osoby.

Z tego powodu postanowiłem napisać w Prologu program do tworzenia harmonogramu. Okazało się, że ten problem idealnie pasował do możliwości Prologu. Najpierw zdefiniowałem zbiór faktów opisujących umiejętności wszystkich członków personelu, plan pracy każdego z nich oraz wymagania każdego doświadczenia. Po wykonaniu tych zadań pozostało jedynie powiedzieć Prologowi: „zrób to". Dla każdego zadania wymienionego w doświadczeniu Prolog znalazł dostępną osobę z odpowiednimi umiejętnościami i przypisał ją do zadania. Następnie kontynuował pracę tak długo, aż spełnił wszystkie potrzeby doświadczenia lub doszedł do wniosku, że ich spełnienie jest niemożliwe. Jeśli nie mógł znaleźć prawidłowego rozwiązania, zaczynał cofać poprzednie powiązania i podejmował kolejną próbę z inną kombinacją. Na koniec albo rozwiązanie zostało znalezione, albo można było wyciągnąć wniosek, że ograniczenia dla danego doświadczenia są zbyt ostre.

***Bruce**: Czy może pan przytoczyć jakieś interesujące przykłady faktów, reguł lub asercji powiązanych z delfinami, które miałyby sens dla naszych czytelników?*

***Brian**: Pamiętam, że była jedna bardzo spektakularna sytuacja, kiedy symulowany delfin pomógł nam zrozumieć rzeczywiste zachowanie Ake'a. Ake odpowiadał na język gestów zawierający takie „zdania" jak „skok przez", czy „prawa piłka ogon dotknij". Dawaliśmy mu instrukcje, a on reagował.*

Jednym z celów moich badań była próba nauczenia Ake'a nowych słów, na przykład „nie". W tym kontekście zdanie „dotknij nie piłka" oznaczało polecenie dotknięcia czegokolwiek poza piłką. Dla Ake'a był to trudny problem do rozwiązania. Przez pewien czas trening przynosił dobre rezultaty. Jednak w pewnym momencie Ake zaczął chować się pod wodą zawsze, kiedy usłyszał pewną instrukcję. Nie byliśmy w stanie

tego zrozumieć, było to bardzo frustrujące. Nie możesz przecież spytać delfina, dlaczego coś zrobił. Z tego względu zdefiniowaliśmy zadanie symulatorowi delfina i uzyskaliśmy interesujące wyniki. Chociaż delfiny są bardzo inteligentne, zazwyczaj starają się znaleźć najprostsze rozwiązanie problemu. Tę samą heurystykę zastosowaliśmy w odniesieniu do symulatora. Okazało się, że język gestów Ake'a zawierał „słowo" opisujące jedno z okien w zbiorniku. Większość trenerów zapomniała o tym słowie, ponieważ korzystano z niego rzadko. Symulator delfina odkrył regułę, zgodnie z którą „okno" było poprawną odpowiedzią na polecenie „nie piłka". Było również poprawną odpowiedzią na polecenia „nie skok", „nie rura" i „nie ringo". Zabezpieczyliśmy się przed stosowaniem tego schematu dla innych obiektów, zmieniając zbiór obiektów w zbiorniku przy każdej próbie, ale — co oczywiste — nie mogliśmy usunąć okna. Okazało się, że kiedy Ake płynął na dno zbiornika, ustawiał się obok okna, choć ja tego nie mogłem zobaczyć.

***Bruce**: Co w Prologu podoba się panu najbardziej?*

***Brian**: Model programowania deklaratywnego jest bardzo interesujący. Ogólnie rzecz biorąc, jeśli potrafisz opisać problem, to potrafisz go również rozwiązać. W przypadku większości języków zdarzało mi się w pewnych sytuacjach dyskutować z komputerem. Mówiłem: „Przecież wiesz, co mam na myśli. Po prostu to zrób!". Symbolem tego zachowania mogą być błędy zgłaszane przez kompilatory języków C lub C++ w rodzaju „oczekiwano średnika". Jeśli „oczekiwano średnika", to dlaczego go nie wstawiono, by sprawdzić, czy to rozwiązuje problem?*

W Prologu moja rola w rozwiązaniu problemu planowania sprowadzała się do stwierdzenia „Chciałbym, aby dzień wyglądał w taki oto sposób, zatem zrób to tak". W odpowiedzi komputer znajdował rozwiązanie.

***Bruce:** Co sprawiło panu największy kłopot?*

***Brian**: Prolog jest rozwiązaniem typu „wszystko albo nic" dla większości problemów — przynajmniej tych, z którymi miałem do czynienia. W przypadku problemu planowania pracy laboratorium zdarzało się, że program „myślał" przez 30 minut, po czym albo wyświetlał doskonałe rozwiązanie planu dnia, albo po prostu wyświetlał odpowiedź „no". W tym przypadku oznaczało to zbyt ostre ograniczenia dla danego dnia, takie, które nie pozwalały na znalezienie pełnego rozwiązania. Prolog nie oferuje niestety rozwiązań częściowych ani nie informuje o tym, w którym miejscu ograniczenia są zbyt ostre.*

To, co zostało zaprezentowane powyżej, to niezwykle interesująca koncepcja. Nie trzeba opisywać sposobu rozwiązania problemu. Trzeba jedynie opisać problem. Językiem opisu problemu jest logika — wyłącznie logika. Należy określić fakty i reguły wnioskowania, a Prolog zajmie się resztą. Programy w Prologu są na wyższym poziomie abstrakcji w porównaniu do programów w innych językach. Harmonogramy i schematy zachowań to świetne przykłady problemów nadających się do rozwiązania za pomocą Prologu.

Czego nauczyliśmy się pierwszego dnia?

Dziś nauczyliśmy się podstawowych bloków budulcowych języka Prolog. Zamiast kodowania kroków, które prowadziłyby Prolog do znalezienia rozwiązania, kodowaliśmy wiedzę, wykorzystując czystą logikę. Prolog wykonał ciężką pracę zinterpretowania tej wiedzy w celu znalezienia rozwiązania problemów. Logikę należy umieścić w bazie wiedzy. Następnie wystarczy formułować zapytania do tej bazy.

Po stworzeniu kilku baz wiedzy skompilowaliśmy je, a następnie zadawaliśmy pytania. Zapytania mają dwie formy. Po pierwsze, zapytanie pozwala na określenie faktów. Prolog poinformuje nas o tym, czy te fakty są prawdziwe, czy fałszywe. Następnie należy utworzyć zapytanie zawierające jedną lub więcej zmiennych. Prolog obliczy wszystkie możliwości wartości zmiennych powodujące prawdziwość podanych faktów.

Dowiedzieliśmy się, że Prolog przetwarza reguły poprzez analizowanie po kolei klauzul wybranej reguły. Dla każdej klauzuli Prolog stara się spełnić każdy z celów, próbując dobrać możliwe kombinacje zmiennych. W ten sposób działają wszystkie programy w Prologu.

W kolejnych punktach przeprowadzimy bardziej złożone wnioskowanie. Dowiemy się również, w jaki sposób wykonywać działania arytmetyczne oraz wykorzystywać bardziej złożone struktury danych, na przykład listy. Zapoznamy się również ze strategiami iterowania po listach.

Dzień 1. Praca do samodzielnego wykonania

Poszukaj:

- darmowych samouczków Prologu,
- forum wsparcia technicznego (dostępnych jest kilka),
- podręcznika online dla używanej wersji Prologu.

Wykonaj następujące ćwiczenia:

- Stwórz prostą bazę wiedzy. Powinna ona reprezentować ulubione książki i autorów.
- Znajdź w bazie wiedzy wszystkie książki napisane przez jednego autora.
- Stwórz bazę wiedzy reprezentującą muzyków i instrumenty.
- Zaprezentuj muzyków oraz gatunek tworzonej przez nich muzyki.
- Znajdź wszystkich muzyków, którzy grają na gitarze.

4.3. Dzień 2. Piętnaście minut do Wapnera

Zrzędliwy sędzia Wapner z programu *The People's Court* jest obsesją głównej postaci z *Rain Mana*. Tak jak większość osób autystycznych Raymond ma obsesję na punkcie znanych postaci. Studiujemy ten enigmatyczny język i powoli wszystko zaczyna układać się w całość. Być może jesteś jednym ze szczęśliwców, którzy rozumieją wszystko od początku, ale jeśli tak nie jest, poproszę o cierpliwość. Dzisiaj jest rzeczywiście „piętnaście minut do Wapnera". Spokojnie! Potrzeba nam kilku dodatkowych narzędzi w przyborniku. Nauczymy się używać rekurencji, operacji arytmetycznych i list. Kontynuujmy naukę!

Rekurencja

Ruby i Io to imperatywne języki programowania. Wymagają zdefiniowania każdego kroku algorytmu. Prolog jest pierwszym językiem deklaratywnym, którym się zajmujemy. Podczas przetwarzania kolekcji elementów, na przykład list lub drzew, często korzystamy z rekurencji zamiast iteracji. Zajmiemy się rekurencją i wykorzystamy ją do rozwiązania pewnych problemów wymagających prostego wnioskowania. Następnie zastosujemy tę samą technikę w odniesieniu do list oraz do wykonywania działań arytmetycznych.

Przyjrzyjmy się bazie danych zamieszczonej poniżej. Prezentuje ona rozbudowane drzewo rodziny Waltonów — postaci z serialu *Waltonowie* z roku 1963 oraz kolejnych serii. W bazie zdefiniowano relację `ojciec`, która służy do wnioskowania związków pomiędzy potomkami i przodkami. Ponieważ przodek może być ojcem, dziadkiem lub pradziadkiem, powstaje konieczność zagnieżdżania reguł bądź iteracji. Ponieważ mamy do czynienia z językiem deklaracyjnym, trzeba zastosować zagnieżdżanie. Jedna z klauzul w klauzuli `przodek` będzie wykorzystywała klauzulę

`przodek`. W tym przypadku `przodek(Z, Y)` to rekurencyjny cel częściowy. Treść bazy wiedzy zamieszczono poniżej:

Pobierz: prolog/rodzina.pl

```
ojciec(zeb, john_boy_sr).
ojciec(john_boy_sr, john_boy_jr).

przodek(X, Y) :-
    ojciec(X, Y).
przodek(X, Y) :-
    ojciec(X, Z), przodek(Z, Y).
```

`ojciec` to zasadniczy zbiór faktów pozwalających na obliczenie celu częściowego w sposób rekurencyjny. Reguła `przodek/2` zawiera dwie klauzule. Kiedy reguła składa się z kilku klauzul, to tylko jedna klauzula musi być prawdziwa, aby cała reguła była prawdziwa. Potraktujmy przecinki pomiędzy celami częściowymi jako warunki `and`, natomiast kropki pomiędzy klauzulami jako warunki `or`. Pierwsza klauzula mówi „`X` jest przodkiem `Y`, jeśli `X` jest ojcem `Y`". Ta relacja jest oczywista. Regułę tę możemy wypróbować w następujący sposób:

```
| ?- przodek(john_boy_sr, john_boy_jr).

true ?
no
```

Prolog odpowiedział `true` (prawda): John Boy senior jest przodkiem Johna Boya juniora. Pierwsza klauzula bazuje na prostym fakcie.

Druga klauzula jest bardziej złożona: `przodek(X, Y) :- ojciec(X, Z), ojciec(Z, Y).` Ta klauzula mówi, że `X` jest przodkiem `Y`, jeśli można udowodnić, że `X` jest ojcem `Z` i jednocześnie ten sam `Z` jest przodkiem `Y`.

Doskonale! Spróbujmy skorzystać z drugiej klauzuli:

```
| ?- przodek(zeb, john_boy_jr).

true ?
```

Tak, `zeb` jest przodkiem Johna Boya juniora. Można oczywiście spróbować wykorzystać zmienne w zapytaniach. Robi się to w następujący sposób:

```
| ?- przodek(zeb, Kto).

Kto = john_boy_sr ? a

Kto = john_boy_jr

no
```

Widzimy również, że `zeb` jest przodkiem Johna Boya juniora i Johna Boya seniora. Predykat `przodek` działa także w przeciwną stronę:

```
| ?- przodek(Kto, john_boy_jr).

Kto = john_boy_sr ? a

Kto = zeb

(1 ms) no
```

Doskonale! Możemy skorzystać z tej reguły w naszej bazie wiedzy w dwóch celach: by znaleźć przodków oraz by znaleźć potomków.

Krótkie ostrzeżenie. W przypadku używania rekurencyjnych celów częściowych trzeba zachować ostrożność. Każdy rekurencyjny cel częściowy wykorzystuje miejsce na stosie, które w końcu się wyczerpie. Języki deklaratywne często rozwiązują ten problem za pomocą techniki optymalizacji znanej jako **rekurencja ogonowa** (ang. *tail recursion*). Jeśli da się umieścić rekurencyjny cel częściowy na końcu reguły rekurencyjnej, to Prolog zoptymalizuje wywołanie — wyeliminuje odwołanie do stosu i wykorzysta stałą z pamięci. Nasze wywołanie jest rekurencją ogonową, ponieważ rekurencyjny cel częściowy `przodek(Z, Y)` jest ostatnim celem w regule rekurencyjnej. Kiedy w programie w Prologu nastąpi błąd krytyczny spowodowany wyczerpaniem się miejsca na stosie, będzie to znak, że należy poszukać sposobu optymalizacji z wykorzystaniem rekurencji ogonowej.

Po omówieniu tego ostatniego „narzędzia w przyborniku" możemy przyjrzeć się listom i krotkom.

Listy i krotki

Listy i krotki są bardzo ważną częścią Prologu. Listę można określić jako `[1, 2, 3]`, natomiast krotkę jako `(1, 2, 3)`. Listy są kontenerami o zmiennym rozmiarze, natomiast krotki są kontenerami o stałym rozmiarze. Możliwości zarówno list, jak i krotek stają się bardziej wyraźne, jeśli pomyślimy o nich w kategoriach unifikacji.

Unifikacja. Część 2.

Jak pamiętamy, kiedy Prolog unifikuje zmienne, próbuje przyrównać do siebie lewą i prawą stronę porównania. Dwie krotki są ze sobą zgodne, jeśli mają tę samą liczbę elementów oraz wszystkie one są zunifikowane. Przeanalizujmy kilka przykładów:

```
| ?- (1, 2, 3) = (1, 2, 3).

yes
| ?- (1, 2, 3) = (1, 2, 3, 4).

no
| ?- (1, 2, 3) = (3, 2, 1).

no
```

Dwie krotki są zunifikowane, jeśli wszystkie ich elementy są zunifikowane. Krotki w pierwszym porównaniu pasowały do siebie dokładnie. W drugim nie miały tej samej liczby elementów, a w trzecim nie miały tych samych elementów w tej samej kolejności. Spróbujmy wprowadzić kilka zmiennych:

```
| ?- (A, B, C) = (1, 2, 3).

A = 1
B = 2
C = 3

yes
| ?- (1, 2, 3) = (A, B, C).

A = 1
B = 2
C = 3

yes
| ?- (A, 2, C) = (1, B, 3).

A = 1
B = 2
C = 3

yes
```

Właściwie nie ma znaczenia, po której stronie są zmienne. Są zunifikowane, jeśli Prolog może je do siebie dopasować. Teraz przyjrzyjmy się listom. Jest z nimi podobnie jak z krotkami.

```
| ?- [1, 2, 3] = [1, 2, 3].

yes
| ?- [1, 2, 3] = [X, Y, Z].

X = 1
Y = 2
Z = 3

yes
| ?- [2, 2, 3] = [X, X, Z].
```

```
X = 2
Z = 3

yes
| ?- [1, 2, 3] = [X, X, Z].

no
| ?- [] = [].
```

Interesujące są dwa ostatnie przykłady. `[X, X, Z]` i `[2, 2, 3]` są zunifikowane, ponieważ Prolog może je dopasować, jeśli podstawi `X = 2`. `[1, 2, 3] = [X, X, Z]` nie są zunifikowane, ponieważ wykorzystano `X` zarówno na pierwszej, jak i na drugiej pozycji, a 1 nie równa się 2. Listy mają własność, której krotki nie mają. Listy można zapisać w postaci `[Głowa|Ogon]`. W przypadku unifikacji listy zapisanej za pomocą takiej konstrukcji `Głowa` zostanie powiązana z pierwszym elementem listy, a `Ogon` z całą resztą, w następujący sposób:

```
| ?- [a, b, c] = [Głowa|Ogon].

Głowa = a
Ogon = [b,c]

yes
```

Wyrażenie `[Głowa|Ogon]` nie zunifikuje się z pustą listą. Jednoelementowa lista daje się jednak prawidłowo dopasować:

```
| ?- [] = [Głowa|Ogon].

no
| ?- [a] = [Głowa|Ogon].

Głowa = a
Ogon = []

yes
```

Oto kilka bardziej złożonych kombinacji:

```
| ?- [a, b, c] = [a|Ogon].

Ogon = [b,c]

(1 ms) yes
```

Prolog dopasował wartość `a` i zunifikował resztę listy z listą `Ogon`. Uzyskany w ten sposób ogon także można rozdzielić na głowę i ogon:

```
| ?- [a, b, c] = [a|[Głowa|Ogon]].

Głowa = b
```

```
Ogon = [c]

yes
```

Spróbujmy pobrać trzeci element:

```
| ?- [a, b, c, d, e] = [_, _|[Głowa|_]].

Głowa = c

yes
```

Znak podkreślenia (_) jest symbolem wieloznacznym, który unifikuje się z dowolną wartością. Ogólnie rzecz biorąc, oznacza on „nie interesuje mnie, co znajdzie się na tej pozycji". Poinstruowaliśmy Prolog, aby pominął pierwsze dwa elementy, a resztę podzielił na głowę i ogon. Z członem `Głowa` będzie powiązany trzeci element, a końcowy symbol _ oznacza ogon, zatem pozostała część listy zostanie zignorowana.

To powinno wystarczyć na początek. Unifikacja jest potężnym narzędziem, a używanie jej w połączeniu z listami i krotkami jeszcze zwiększa te możliwości.

W tym momencie czytelnicy powinni znać zasadnicze struktury danych w Prologu, a także sposoby działania unifikacji. Jesteśmy teraz gotowi, aby połączyć te elementy z regułami i asercjami w celu wykonywania działań matematycznych na wyrażeniach logicznych.

Arytmetyka list

W ramach naszego kolejnego przykładu postanowiłem zaprezentować sposób wykorzystywania rekurencji i arytmetyki w celu wykonywania działań na listach. Poniższe przykłady służą do zliczania elementów oraz obliczania podsumowań i średnich. Całą ciężką pracę wykonuje pięć reguł.

Pobierz: prolog/arytmetyka_list.pl

```
policz(0, []).
policz(LiczbaElementów, [Głowa|Ogon]) :- policz(LiczbaElementówOgona, Ogon), :-
    LiczbaElementów is LiczbaElementówOgona + 1.

suma(0, []).
suma(SumaŁącznie, [Głowa|Ogon]) :- suma(Suma, Ogon), SumaŁącznie is Głowa + Suma.

średnia(Średnia, Lista) :- suma(Suma, Lista), policz(LiczbaElementów, Lista), Średnia is :-
    Suma/LiczbaElementów.
```

Najprostszym przykładem jest predykat `policz`. Można go wykorzystać w następujący sposób:

```
| ?- policz(Co, [1]).

Co = 1 ? ;

no
```

Reguły są trywialnie proste. Liczba elementów pustej listy wynosi `0`. Liczba elementów niepustej listy jest równa liczbie elementów ogona plus jeden. Przeanalizujmy krok po kroku, jak to działa.

- Wprowadziliśmy zapytanie `policz(Co, [1])`, które nie może zostać zunifikowane z pierwszą regułą, ponieważ lista nie jest pusta. W związku z tym przechodzimy do sprawdzenia celów drugiej reguły: `policz(LiczbaElementów, [Głowa|Ogon])`. Następuje unifikacja, powiązanie zmiennej `Co` ze zmienną `LiczbaElementów`, zmiennej `Głowa` z wartością `1` i zmiennej `Ogon` z wartością `[]`.
- Po unifikacji pierwszym celem jest `policz(LiczbaElementówOgona, [])`. Staramy się udowodnić cel częściowy. Tym razem unifikujemy z pierwszą regułą. Wiążemy zmienną `LiczbaElementówOgona` z wartością `0`. Pierwsza reguła jest teraz spełniona. Możemy zatem przejść do kolejnego celu.
- Teraz wyznaczamy wartość wyrażenia `LiczbaElementów is LiczbaElementówOgona + 1`. Możemy zunifikować zmienne. Zmienną `LiczbaElementówOgona` wiążemy z wartością 0, a zatem zmienna `LiczbaElementów` ma wartość `0 + 1`, czyli `1`.

To wszystko. Nie definiowaliśmy procesu rekurencyjnego. Definiowaliśmy tylko reguły logiczne. Następny przykład dotyczy sumowania elementów listy. Oto kod tych reguł:

```
suma(0, []).
suma(SumaŁącznie, [Głowa|Ogon]) :- suma(Suma, Ogon), SumaŁącznie is Głowa + Suma.
```

Powyższy kod działa identycznie jak reguła liczenia elementów. Także zawiera dwie klauzule — przypadek bazowy i przypadek rekurencyjny. Sposób użycia tej reguły jest podobny:

```
| ?- suma(Co, [1, 2, 3]).

Co = 6 ? ;

no
```

Jeśli spojrzymy na to z punktu widzenia języka imperatywnego, dojdziemy do wniosku, że reguła `suma` działa dokładnie tak, jak należałoby oczekiwać w języku rekurencyjnym.

Wartość reguły `suma` pustej listy wynosi zero. W pozostałych przypadkach suma jest równa wartości `Głowa` plus `suma` z `Ogon`.

Można to jednak zinterpretować inaczej. Nie powiedzieliśmy Prologowi, w jaki sposób oblicza się sumy. Opisaliśmy jedynie sumy w postaci reguł i celów. Spełnienie pewnych celów wymaga od maszyny wnioskującej spełnienia pewnych celów częściowych. Interpretacja deklaratywna jest następująca: „Suma pustej listy wynosi zero, natomiast suma wszystkich elementów na liście ma wartość `SumaŁącznie`, jeśli można udowodnić, że suma ogona i głowy wynosi `SumaŁącznie`". Zastąpiliśmy rekurencję pojęciem dowodzenia celów i celów częściowych. Działanie reguły `policz` można wyjaśnić podobnie: liczba elementów pustej listy wynosi zero. Liczba elementów niepustej listy wynosi jeden (liczba elementów głowy) plus liczba elementów ogona.

Tak jak zwykle w przypadku logiki, reguły mogą bazować na innych regułach. Na przykład możemy użyć reguły `suma` razem z regułą `policz` w celu obliczenia średniej:

```
średnia(Średnia, Lista) :- suma(Suma, Lista), policz(Policz, Lista), Średnia is Suma/Policz.
```

A zatem `średnia` argumentu `Lista` wynosi `Średnia`, jeśli można udowodnić, że:

- `suma` tego argumentu `Lista` wynosi `Suma`.
- Wartość `policz` tego argumentu `Lista` wynosi `Policz`.
- `Średnia` wynosi `Suma/Policz`.

Powyższy kod działa zgodnie z oczekiwaniami:

```
| ?- średnia(Co, [1, 2, 3]).

Co = 2.0 ? ;

no
```

Wykorzystywanie reguł w dwóch kierunkach

W tym momencie czytelnik powinien mieć dość wyraźny obraz działania rekurencji. W tym punkcie trochę „pozmieniam biegi" i omówię regułę `append`. Reguła `append (Lista1, Lista2, Lista3)` jest prawdziwa, jeśli lista `Lista3` jest sumą list `Lista1 + Lista2`. To interesująca reguła, którą można wykorzystywać na różne sposoby.

Ta niepozorna reguła daje wiele możliwości. Można ją wykorzystywać na wiele różnych sposobów. Jako wykrywacz kłamstwa:

```
| ?- append([olej], [woda], [olej, woda]).

yes
```

```
| ?- append([olej], [woda], [olej, smar]).

no
```

Jako narzędzie do tworzenia list.

```
| ?- append([piwo], [szampan], Co).

Co = [piwo,szampan]

yes
```

Do odejmowania list:

```
| ?- append([przybranie_deseru], Kto, [przybranie_deseru, pasta_do_podłogi]).

Kto = [pasta_do_podłogi]

yes
```

Do obliczania możliwych permutacji:

```
| ?- append(Jeden, Dwa, [jabłka, pomarańcze, banany]).

Dwa = [jabłka, pomarańcze, banany] ? a
Jeden = [] ? a

Dwa = [pomarańcze, banany]
Jeden = [jabłka]

Dwa = [banany]
Jeden = [jabłka, pomarańcze]

Dwa = []
Jeden = [jabłka, pomarańcze, banany]

(15 ms) no
```

Jak widać, jedna reguła daje nam cztery. Na pierwszy rzut oka wydaje się, że utworzenie takiej reguły wymaga dużej ilości kodu. Spróbujmy się dowiedzieć ile. Postaramy się napisać samodzielnie regułę Prologu `append`, ale nazwiemy ją po swojemu — `dołącz`. Zadanie wykonamy w kilku krokach:

1. Napiszemy regułę o nazwie `dołącz(Lista1, Lista2, Lista3)` umożliwiającą dołączenie pustej listy do listy `Lista1`.
2. Dodamy regułę, która dołącza jeden element z listy `Lista1` do listy `Lista2`.
3. Dodamy regułę, która dołącza drugi i trzeci element z listy `Lista1` do listy `Lista2`.
4. Spróbujemy wprowadzić uogólnienia.

A więc do dzieła. Pierwsza czynność polega na dołączeniu pustej listy do listy `Lista2`. Napisanie reguły, która jest potrzebna do osiągnięcia tego celu, nie jest trudne.

Pobierz: prolog/dolacz_krok1.pl

```
dołącz([], Lista, Lista).
```

Żadnych problemów. Reguła `dołącz` jest prawdziwa, jeśli pierwszy parametr jest listą, a następne dwa parametry są takie same.

To działa.

```
| ?- dołącz([], [heniek], Co).

Co = [heniek]

yes
```

Przejdźmy do następnego kroku. Dodamy regułę, która dołącza pierwszy element z listy `Lista1` na początek listy `Lista2`.

Pobierz: prolog/dolacz_krok2.pl

```
dołącz([], Lista, Lista).
dołącz([Głowa|[]], Lista, [Głowa|Lista]).
```

Aby napisać regułę `dołącz(Lista1, Lista2, Lista3)`, rozbijemy listę `Lista1` na głowę i ogon, przy czym ogon będzie pustą listą. Trzeci element rozbijemy na głowę i ogon, wykorzystując głowę listy `Lista1` oraz listę `Lista2` jako ogon. Pamiętajmy o skompilowaniu bazy wiedzy. Działa bez zarzutu:

```
| ?- dołącz([malfoy], [potter], Co).

Co = [malfoy,potter]

yes
```

Możemy teraz zdefiniować kolejnych kilka reguł opisujących łączenie list o długościach 2 i 3 elementów. Działają one w taki sam sposób:

Pobierz: prolog/dolacz_krok3.pl

```
dołącz([], Lista, Lista).
dołącz([Głowa|[]], Lista, [Głowa|Lista]).
dołącz([Głowa1|[Głowa2]|[]]], Lista, [Głowa1| Głowa2|Lista]).
dołącz([Głowa1|[Głowa2]|[Głowa3|[]]]], Lista, [Głowa1, Głowa2, Głowa3|Lista]).

| ?- dołącz([malfoy, granger], [potter], Co).

Cot = [malfoy,granger,potter]

yes
```

Mamy tu przypadek bazowy oraz strategię, zgodnie z którą każdy cel częściowy zmniejsza liczbę elementów na pierwszej liście i rozszerza trzecią listę. Druga lista pozostaje stała. Mamy teraz wystarczająco dużo informacji, by spróbować uogólnić wyniki. Poniżej reguła `dołącz` zdefiniowana z wykorzystaniem reguł zagnieżdżonych:

Pobierz: prolog/dolacz.pl

```
dołącz([], Lista, Lista).
dołącz([Głowa|Ogon1], Lista, [Głowa|Ogon2]) :-
    dołącz(Ogon1, Lista, Ogon2).
```

Objaśnienie tego niewielkiego fragmentu kodu jest niezwykle proste. Pierwsza klauzula mówi, że w wyniku dołączenia pustej listy do argumentu `Lista` otrzymujemy tę samą wartość argumentu `Lista`. Druga klauzula mówi, że dołączenie listy `Lista1` do listy `Lista2` daje w wyniku listę `Lista3` w przypadku, gdy głowy list `Lista1` i `Lista3` są takie same oraz można udowodnić, że w wyniku scalenia listy `Lista1` z listą `Lista2` otrzymamy ogon listy `Lista3`. Prostota i elegancja tego rozwiązania są testamentem możliwości Prologu.

Zobaczmy, co się dzieje w zapytaniu `dołącz([1,2],[3], Co)`. Dla każdego kroku przeanalizujemy unifikacje. Pamiętajmy o tym, że zagnieździliśmy reguły, zatem przy każdej próbie udowodnienia celu częściowego mamy inną kopię zmiennych. Najważniejsze miejsca oznaczyłem literami. Dzięki temu można z łatwością śledzić przykład. W każdym przebiegu pokażę, co się będzie działo w czasie, kiedy Prolog podejmie próbę udowodnienia kolejnego celu częściowego.

- Rozpoczynamy od następującej reguły:
  `dołącz([1,2], [3], Co)`
- Pierwsza reguła nie jest spełniona, ponieważ `[1, 2]` nie jest pustą listą. Unifikujemy ją w następujący sposób:
  `dołącz([1|[2]], [3], [1|Ogon2-A]) :- dołącz([2], [3], [Ogon2-A])`

  Unifikacja wszystkich członów poza drugim przebiegła pomyślnie. Przejdźmy teraz do celów. Unifikujemy prawą stronę.
- Spróbujemy zunifikować regułę `dołącz([2], [3], [Ogon2-A])`. W efekcie otrzymujemy:
  `dołącz([2|[ ]], [3], [2|Ogon2-B]) :- dołącz([ ], [3], Ogon2-B)`

  Zwróćmy uwagę, że `Ogon2-B` jest ogonem listy `Ogon2-A`. Nie jest to ta sama lista co początkowa lista `Ogon2`. Teraz jednak musimy ponownie zunifikować prawą stronę.
  `dołącz([ ], [3], Ogon2-C) :- dołącz([ ], [3], [3]) .`

- Wiemy zatem, że `Tail2-C` ma wartość `[3]`. Możemy teraz pójść w górę łańcucha. Przyjrzyjmy się trzeciemu parametrowi, który na każdym kroku dołącza listę `Ogon2`. `Ogon2-C` ma wartość `[3]`, co oznacza, że `[2|Ogon2-2]` to lista `[2, 3]` i na koniec `[1|Ogon2]` to lista `[1, 2, 3]`. Zmienna `Co` ma wartość `[1, 2, 3]`.

Prolog wykonał tu mnóstwo pracy. Radzę analizować tę listę tak długo, aż wszystko stanie się zrozumiałe. Unifikowanie zagnieżdżonych celów częściowych to podstawowa umiejętność — niezbędna do rozwiązywania zaawansowanych problemów w niniejszej książce.

Przestudiowaliśmy właśnie jedną z najważniejszych funkcji Prologu. Poświęćmy trochę czasu na analizę rozwiązań w taki sposób, aby stały się zrozumiałe.

Czego nauczyliśmy się drugiego dnia?

W tym punkcie zajęliśmy się podstawowymi blokami budulcowymi wykorzystywanymi w Prologu do organizowania danych: listami i krotkami. Studiowaliśmy także zagnieżdżanie reguł pozwalające na przedstawienie problemów, które w innych językach rozwiązuje się za pomocą instrukcji iteracyjnych. Przyjrzeliśmy się dokładniej unifikacji oraz sposobowi działania Prologu podczas porównywania obu stron operatorów `:-` lub `=`. Przekonaliśmy się, że pisząc reguły, opisujemy zasady logiczne, a nie algorytmy. To Prolog wypracowuje rozwiązanie.

Pokazaliśmy również, w jaki sposób wykorzystuje się działania arytmetyczne. Dowiedzieliśmy się, jak wykorzystuje się podstawowe działania arytmetyczne i zagnieżdżone cele częściowe do obliczania podsumowań i średnich.

Na koniec pokazaliśmy, w jaki sposób wykorzystuje się listy. Zaprezentowaliśmy, jak dopasować jedną bądź kilka zmiennych z listą, ale co ważniejsze, pokazaliśmy, w jaki sposób można dopasować ze zmiennymi głowę listy wraz z jej pozostałymi elementami, używając wzorca `[Głowa|Ogon]`. Wykorzystaliśmy tę technikę do rekurencyjnego iterowania po listach. Poznane bloki budulcowe będą nam służyć jako baza do rozwiązywania złożonych problemów, z którymi spotkamy się trzeciego dnia.

Dzień 2. Praca do samodzielnego wykonania

Poszukaj:

- Implementacji algorytmów do wyznaczania ciągu Fibonacciego i obliczania silni. W jaki sposób one działają?

- Społeczności użytkowników Prologu. Do rozwiązywania jakich problemów członkowie tej społeczności używają Prologu?

Czytelnicy poszukujący bardziej zaawansowanych zadań mogą przeanalizować następujące problemy:

- Implementację Wieży Hanoi. W jaki sposób działa algorytm rozwiązania tego zadania?
- Jakie problemy mogą występować podczas posługiwania się wyrażeniami zanegowanymi? Dlaczego należy zachować ostrożność, posługując się takimi wyrażeniami w Prologu?

Wykonaj następujące zadania:

- Odwracanie elementów listy.
- Wyszukiwanie najmniejszego elementu na liście.
- Sortowanie elementów listy.

4.4. Dzień 3. Podbić Vegas

Czytelnicy powinni teraz lepiej rozumieć powody, dla których porównałem Prolog do *Rain Mana*, autystycznego geniusza. Choć czasami trudno to zrozumieć, wspaniale jest myśleć o programowaniu w taki sposób. Jednym z moich ulubionych momentów w *Rain Manie* jest sytuacja, kiedy brat Raya zdaje sobie sprawę z tego, że potrafi on liczyć karty. Raymond razem z bratem jadą do Vegas i prawie rozbijają bank. W tym punkcie zobaczycie Prolog z takiej strony, która pozostawi uśmiech na Waszych twarzach. Kodowanie przykładów w niniejszym rozdziale było w równym stopniu szalone co radosne. Spróbujemy rozwiązać dwie popularne łamigłówki, które idealnie nadają się dla Prologu — są to systemy z ograniczeniami.

Jeśli ktoś chce, może samodzielnie spróbować napisać program do rozwiązywania tych łamigłówek. Jeśli tak, to próbujcie opisywać znane Wam reguły dotyczące każdej z łamigłówek zamiast prezentowania Prologowi rozwiązania krok po kroku. Rozpoczniemy od niewielkiego sudoku, a następnie w ramach samodzielnej pracy tego dnia pozwolimy czytelnikom na opracowanie rozwiązania dla większego sudoku. Potem przejdziemy do rozwiązywania klasycznego problemu ośmiu hetmanów.

Rozwiązywanie sudoku

Zakodowanie programu do rozwiązywania sudoku było dla mnie prawie magiczne. Sudoku to tabelka składająca się z wierszy, kolumn i bloków. Typowa łamigłówka jest diagramem 9×9, w którym niektóre pola są wypełnione, a inne puste. Każda komórka w diagramie ma numer od 1 do 9 odpowiadający numerowi kwadratu 3×3. Zadaniem rozwiązującego jest takie wypełnienie diagramu, aby w każdym wierszu, w każdej kolumnie i w każdym z dziewięciu kwadratów znalazło się po jednej cyfrze od 1 do 9.

Rozpoczniemy od sudoku o rozmiarach 4×4. Zasady są identyczne, choć rozwiązanie będzie prostsze. Rozpocznijmy od opisania rzeczywistości — tego, co wiemy o obowiązujących zasadach. Mamy diagram złożony z czterech wierszy, czterech kolumn i czterech kwadratów. Kwadraty o numerach 1 – 4 zamieszczono w poniższej tabeli.

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Pierwszym naszym zadaniem będzie utworzenie zapytania. Jest to dość proste. Mamy łamigłówkę i jej rozwiązanie w postaci `sudoku(Łamigłówka, Rozwiązanie)` Użytkownik może wprowadzić łamigłówkę w postaci listy, wpisując znaki podkreślenia w miejscu nieznanych liczb. Może to wyglądać następująco:

```
sudoku([_, _, 2, 3,
        _, _, _, _,
        _, _, _, _,
        3, 4, _, _],
        Rozwiązanie).
```

Jeśli istnieje rozwiązanie, to Prolog je wyświetli. Kiedy rozwiązywałem tę łamigłówkę w Ruby, musiałem zdefiniować algorytm jej rozwiązywania. W Prologu nie trzeba się o to martwić. Trzeba jedynie wprowadzić zasady gry. Oto one:

- Liczby w łamigłówce i rozwiązaniu muszą być takie same.
- Plansza sudoku jest diagramem złożonym z szesnastu komórek o wartościach 1 – 4.
- Plansza składa się z czterech wierszy, czterech kolumn i czterech kwadratów.
- Łamigłówka jest rozwiązana, jeśli w żadnym wierszu, kolumnie i kwadracie nie ma powtarzających się elementów.

Zacznijmy od początku. Liczby w rozwiązaniu i łamigłówce powinny być ze sobą zgodne:

Pobierz: prolog/sudoku4_krok1.pl

```
sudoku(Łamigłówka, Rozwiązanie) :-
        Rozwiązanie = Łamigłówka.
```

Mamy pewien postęp. Nasz „program do rozwiązywania sudoku" działa dla przypadku, w którym plansza nie ma pustych miejsc.

```
| ?- sudoku([4, 1, 2, 3,
             2, 3, 4, 1,
             1, 2, 3, 4,
             3, 4, 1, 2], Rozwiązanie).

Rozwiązanie = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

Format nie jest piękny, ale intencje są czytelne. Otrzymaliśmy szesnaście liczb — wiersz po wierszu. Jednak chyba trochę zbyt wiele żądamy od Prologu:

```
| ?- sudoku([1, 2, 3], Rozwiązanie).

Rozwiązanie = [1,2,3]

yes
```

Teraz diagram nie jest prawidłowy, a nasz program wyświetlił informację, że istnieje poprawne rozwiązanie. Jest oczywiste, że trzeba wprowadzić ograniczenie diagramu do szesnastu elementów. Mamy jeszcze jeden problem. Wartości w komórkach mogą być dowolne:

```
| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Rozwiązanie).

Rozwiązanie = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6]

yes
```

Aby rozwiązanie mogło być prawidłowe, wszystkie liczby, które się w nim znajdują, muszą mieć wartości od 1 do 4. Problem ten ma wpływ na dwie sprawy. Po pierwsze, program może generować nieprawidłowe rozwiązania. Po drugie, Prolog nie posiada wystarczającej ilości informacji do testowania dozwolonych wartości w każdej komórce. Inaczej mówiąc, zbiór wyników nie został **ograniczony**. Oznacza to, że nie podaliśmy reguł, które definiują dozwolone wartości w każdej komórce. Z tego powodu Prolog nie będzie w stanie odgadnąć, jakie powinny być te wartości.

Spróbujmy rozwiązać ten problem poprzez zdefiniowanie następnej reguły łamigłówki. Mówi ona, że diagram składa się z szesnastu komórek, z których każda zawiera wartości z zakresu 1 – 4. Program GNU Prolog zawiera wbudowany predykat służący do wyrażania możliwych wartości. Mowa o predykacie `fd_domain(Lista, DolnaGranica, GórnaGranica)`. Predykat ten zwraca wartość `true`, jeśli wszystkie wartości należące do argumentu `Lista` mieszczą się w zakresie pomiędzy wartościami `DolnaGranica` a `GórnaGranica` włącznie. Musimy zapewnić, aby wszystkie wartości na liście `Łamigłówka` mieściły się w zakresie 1 – 4.

Pobierz: prolog/sudoku4_krok2.pl

```
sudoku(Łamigłówka, Rozwiązanie) :-
        Rozwiązanie = Łamigłówka,
        Łamigłówka = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],
fd_domain(Łamigłówka, 1, 4).
```

Zunifikowaliśmy argument `Łamigłówka` z listą złożoną z szesnastu zmiennych i wprowadziliśmy ograniczenie na elementy do wartości z zakresu 1 – 4. Teraz jeśli użytkownik wprowadzi nieprawidłowe sudoku, to Prolog odpowie mu `no`:

```
| ?- sudoku([1, 2, 3], Rozwiązanie).

no

| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Rozwiązanie).

no
```

Przejdźmy teraz do zasadniczej części rozwiązania. Reguła numer 3 mówi, że plansza składa się z wierszy, kolumn i kwadratów. Musimy podzielić łamigłówkę na wiersze, kolumny i kwadraty. Teraz widać, po co nadaliśmy komórkom takie nazwy. Dzięki nim z łatwością opiszemy wiersze.

```
Wiersz1 = [S11, S12, S13, S14],
Wiersz2 = [S21, S22, S23, S24],
Wiersz3 = [S31, S32, S33, S34],
Wiersz4 = [S41, S42, S43, S44],
```

W podobny sposób można opisać kolumny:

```
Kol1 = [S11, S21, S31, S41],
Kol2 = [S12, S22, S32, S42],
Kol3 = [S13, S23, S33, S43],
Kol4 = [S14, S24, S34, S44],
```

i kwadraty:

```
Kwadrat1 = [S11, S12, S21, S22],
Kwadrat2 = [S13, S14, S23, S24],
Kwadrat3 = [S31, S32, S41, S42],
Kwadrat4 = [S33, S34, S43, S44].
```

Po podzieleniu planszy na części możemy przejść do następnej reguły. Rozwiązanie jest prawidłowe, jeśli żaden wiersz, kolumna i kwadrat nie zawierają powtarzających się elementów. Do sprawdzania obecności powtarzających się elementów wykorzystamy predykat programu GNU Prolog — `fd_all_different(Lista)`. Predykat ten zwraca wartość „prawda", jeśli wszystkie elementy należące do argumentu `Lista` są różne. Musimy zdefiniować regułę, która sprawdza, czy wszystkie wiersze, kolumny i kwadraty są prawidłowe. Do tego celu wykorzystamy prostą regułę:

```
prawidłowa([]).
prawidłowa([Głowa|Ogon]) :-
    fd_all_different(Głowa),
    prawidłowa(Ogon).
```

Ten predykat jest prawdziwy, jeśli wszystkie listy zawarte w argumencie są różne. Pierwsza klauzula mówi, że pusta lista jest prawidłowa. Druga klauzula mówi, że lista jest prawidłowa, jeśli elementy pierwszej listy są różne oraz pozostała część listy jest prawidłowa.

Wystarczy tylko wywołać naszą regułę `prawidłowa(Lista)`:

```
prawidłowa([Wiersz1, Wiersz2, Wiersz3, Wiersz4,
       Kol1, Kol2, Kol3, Kol4,
       Kwadrat1, Kwadrat2, Kwadrat3, Kwadrat4]).
```

Uwierzcie mi lub nie, ale właśnie skończyliśmy. Nasz program potrafi rozwiązać sudoku o rozmiarach 4×4:

```
| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Rozwiązanie).

Rozwiązanie = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

Po przekształceniu do bardziej czytelnej postaci otrzymujemy:

4 1 2 3
2 3 4 1
1 2 3 4
3 4 1 2

Kompletny program zamieszczono poniżej:

Pobierz: prolog/sudoku4.pl

```
prawidłowa([]).
prawidłowa([Głowa|Ogon]) :-
    fd_all_different(Głowa),
    prawidłowa(Ogon).


sudoku(Łamigłówka, Rozwiązanie) :-
        Rozwiązanie = Łamigłówka,

        Łamigłówka = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],

        fd_domain(Rozwiązanie, 1, 4),

        Wiersz1 = [S11, S12, S13, S14],
        Wiersz2 = [S21, S22, S23, S24],
        Wiersz3 = [S31, S32, S33, S34],
        Wiersz4 = [S41, S42, S43, S44],

        Kol1 = [S11, S21, S31, S41],
        Kol2 = [S12, S22, S32, S42],
        Kol3 = [S13, S23, S33, S43],
        Kol4 = [S14, S24, S34, S44],

        Kwadrat1 = [S11, S12, S21, S22],
        Kwadrat2 = [S13, S14, S23, S24],
        Kwadrat3 = [S31, S32, S41, S42],
        Kwadrat4 = [S33, S34, S43, S44],

prawidłowa([Wiersz1, Wiersz2, Wiersz3, Wiersz4,
       Kol1, Kol2, Kol3, Kol4,
       Kwadrat1, Kwadrat2, Kwadrat3, Kwadrat4]).
```

Kogoś, komu do tej pory Prolog się jeszcze nie spodobał, ten przykład powinien poprowadzić we właściwym kierunku. Gdzie jest program? Nie pisaliśmy żadnego programu. Opisaliśmy jedynie reguły obowiązujące w grze: diagram składa się z szesnastu komórek, z których każda zawiera wartości z zakresu 1 – 4 i w żadnym wierszu, kolumnie lub kwadracie nie może być powtarzających się wartości. Rozwiązanie łamigłówki zajęło kilkanaście wierszy kodu i nie wymagało żadnej wiedzy na temat strategii rozwiązywania sudoku. W pracy do samodzielnego wykonania czytelnik otrzyma szansę rozwiązania sudoku składającego się z dziewięciu wierszy. Nie będzie to zbyt trudne zadanie.

Pokazana łamigłówka to wspaniały przykład typu problemów, do których rozwiązywania Prolog świetnie się nadaje. Mamy zbiór ograniczeń, które łatwo wyrazić, a które trudno spełnić. Przyjrzyjmy się innej łamigłówce, w której występują ostre ograniczenia zasobów: problemowi ośmiu hetmanów

Ośmiu hetmanów

Problem dotyczy rozstawienia na szachownicy ośmiu hetmanów. Żadne dwa hetmany nie mogą dzielić tego samego wiersza, tej samej kolumny lub tej samej przekątnej. Z pozoru problem ten może wydawać się trywialny. Po prostu dziecinna gra. Patrząc jednak z drugiej strony, wiersze, kolumny i przekątne można uznać za ograniczone zasoby. Nasza branża pełna jest problemów z ograniczeniami. Przyjrzyjmy się, w jaki sposób można rozwiązać je w Prologu.

Najpierw przyjrzymy się, w jaki sposób powinno wyglądać zapytanie. Każdego hetmana można wyrazić w postaci krotki `(Wiersz, Kolumna)`. `Szachownica` to lista krotek. Predykat `ośmiu_hetmanów(Szachownica)` ma wartość „prawda", jeśli argument prezentuje poprawną szachownicę. Zapytanie przyjmie następującą postać:

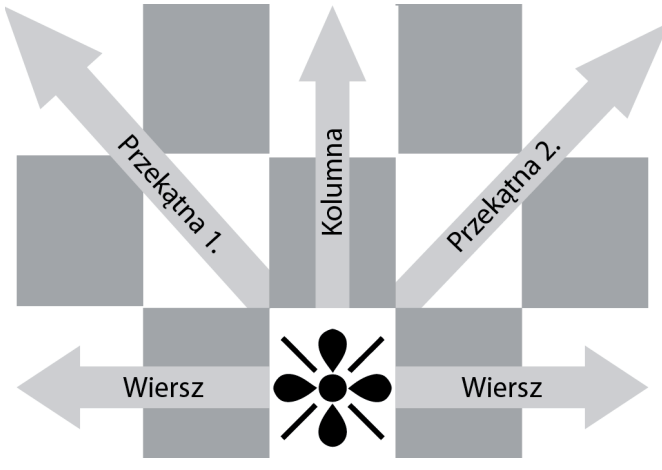
```
ośmiu_hetmanów([(1, 1), (3, 2), ...]).
```

Przyjrzyjmy się celom, jakie muszą być spełnione, aby można było rozwiązać łamigłówkę. Jeśli ktoś chciałby spróbować rozwiązać problem tej łamigłówki bez zaglądania do rozwiązania, proponuję przyjrzeć się tym celom. Kompletne rozwiązanie zaprezentuję w dalszej części tego rozdziału.

- Na szachownicy jest ośmiu hetmanów.
- Każdy hetman znajduje się w wierszu o numerze 1 – 8 i kolumnie o numerze 1 – 8.
- Nie może być dwóch hetmanów w żadnym wierszu.
- Nie może być dwóch hetmanów w żadnej kolumnie.
- Nie może być dwóch hetmanów na tej samej przekątnej (z południowego zachodu na północny wschód).
- Nie może być dwóch hetmanów na tej samej przekątnej (z północnego zachodu na południowy wschód).

Wiersze i kolumny muszą być unikatowe, ale w przypadku przekątnych trzeba zachować większą ostrożność. Każdy hetman znajduje się na dwóch przekątnych — jednej prowadzącej z lewego dolnego narożnika (północnego zachodu) do górnego

prawego narożnika (południowego wschodu) oraz drugiej prowadzącej z górnego lewego narożnika do dolnego prawego — tak jak na rysunku 4.2 na następnej stronie. Reguły te powinny być jednak stosunkowo łatwe do zakodowania.



Rysunek 4.2. Reguły problemu ośmiu hetmanów

Tak jak poprzednio rozpoczynamy od początku listy. Na szachownicy znajduje się ośmiu hetmanów. To oznacza, że rozmiar naszej listy musi wynosić osiem. Zaprogramowanie takiej reguły nie jest trudne. Można skorzystać z predykatu `policz`, z którym spotkaliśmy się wcześniej w tej książce, albo z wbudowanego w Prolog predykatu `length`. Predykat `length(Lista, N)` jest prawdziwy, jeśli `Lista` zawiera `N` elementów. Tym razem zamiast pokazywania każdego celu z osobna przeanalizujemy cele, które muszą być osiągnięte, aby można było rozwiązać cały problem. Oto pierwszy cel:

```
ośmiu_hetmanów(Lista) :- length(Lista, 8).
```

Następnie musimy sprawdzić, czy wszystkie hetmany na liście mają prawidłowe pozycje. Utworzymy regułę sprawdzającą, czy hetman ma prawidłową pozycję:

```
prawidłowy_hetman((Wiersz, Kolumna)) :-
    Zakres = [1,2,3,4,5,6,7,8],
    należy(Wiersz, Zakres), należy(Kolumna, Zakres).
```

Predykat `należy` działa tak, jak można oczekiwać: sprawdza przynależność. Pozycja hetmana jest prawidłowa, jeśli zarówno wiersz, jak i kolumna są liczbami całkowitymi z zakresu 1 – 8. Następnie utworzymy regułę sprawdzającą, czy cała szachownica zawiera prawidłowe pozycje hetmanów:

```
prawidłowa_szachownica([]).
prawidłowa_szachownica([Głowa|Ogon]) :- prawidłowy_hetman(Głowa), :-
    prawidłowa_szachownica(Ogon).
```

Pusta szachownica jest prawidłowa. Niepusta szachownica jest prawidłowa, jeśli pierwszy jej element zawiera prawidłową pozycję hetmana oraz reszta szachownicy jest prawidłowa.

Idąc dalej, kolejna reguła mówi, że w tym samym wierszu nie mogą się znaleźć dwa hetmany. Do rozwiązania kilku następnych ograniczeń potrzebna jest niewielka pomoc. Podzielimy program na fragmenty, które będą mogły nam pomóc w opisaniu problemu: czym są wiersze, kolumny i przekątne? Najpierw zajmiemy się wierszami. Utworzymy funkcję o nazwie `wiersze(Hetmany, Wiersze)`. Powyższa funkcja zwraca prawdę, jeśli argument `Wiersze` jest listą elementów `Wiersz` wszystkich hetmanów.

```
wiersze([], []).
wiersze([(Wiersz, _)|HetmanyOgon], [Wiersz|WierszeOgon]) :-
    wiersze(HetmanyOgon, WierszeOgon).
```

Powyższa funkcja wymaga nieco wyobraźni, choć niezbyt wiele. Funkcja `wiersze` wywołana dla pustej listy zwraca pustą listę, natomiast funkcja `wiersze(Hetmany, Wiersze)` zwraca listę `Wiersze`, jeśli `Wiersz` odpowiadający pierwszemu hetmanowi na liście pasuje do pierwszego elementu listy `Wiersze` oraz jeżeli `wiersze` ogona listy `Hetmany` są ogonem listy `Wiersze`. Dla osób, które tego nie rozumieją, proponuję analizę z wykorzystaniem kilku list testowych. Na szczęście dla kolumn obowiązują te same reguły. Zamiast wierszy użyjemy jednak kolumn:

```
kolumny([], []).
kolumny([(_, Kolumna)|HetmanyOgon], [Kolumny|KolumnyOgon]) :-
    kolumny(HetmanyOgon, KolumnyOgon).
```

Logika tej funkcji jest identyczna jak funkcji `wiersze`, choć dopasowujemy drugi element krotki opisującej hetmana zamiast pierwszego.

Następnie ponumerujemy przekątne. Najprostszym sposobem ich ponumerowania jest wykonanie prostego odejmowania i dodawania. Jeśli wartości `północ` i `zachód` wynoszą 1, to przekątnym prowadzącym z północnego zachodu na południowy wschód przypiszemy wartość `Kolumna – Wiersz`. Oto predykat potrzebny do opisania tych przekątnych:

```
przekątne1([], []).
przekątne1([(Wiersz, Kolumna)|HetmanyOgon], [Przekątna|PrzekątneOgon]) :-
    Przekątna is Kolumna - Wiersz,
    przekątne1(HetmanyOgon, PrzekątneOgon).
```

Powyższa reguła działa identycznie jak reguły dla wierszy i kolumn, z jednym dodatkowym ograniczeniem: `Przekątna is Kolumna - Wiersz`. Warto zwrócić uwagę, że to nie jest unifikacja. Jest to predykat `is`, który służy do sprawdzenia, czy rozwiązanie jest w pełni poprawne. Na koniec opiszemy przekątną z południowego wschodu na północny zachód. Robimy to w następujący sposób:

```
przekątne2([], []).
przekątne2([(Wiersz, Kolumna)|HetmanyOgon], [Przekątna|PrzekątneOgon]) :-
    Przekątna is Kolumna + Wiersz,
    przekątne2(HetmanyOgon, PrzekątneOgon).
```

Formuła jest nieco zawiła. Proponuję jednak wypróbować kilka wartości. Łatwo zauważyć, że hetmany o takiej samej sumie numeru wiersza i kolumny leżą na jednej przekątnej. Teraz, kiedy mamy reguły opisujące wiersze, kolumny i przekątne, pozostaje spełnienie warunku, aby wiersze, kolumny i przekątne były różne.

Aby można było zobaczyć wszystkie reguły w tym samym kontekście, poniżej zamieszczono całe rozwiązanie. Testy dla wierszy i kolumn to ostatnie osiem klauzul.

Pobierz: prolog/hetmany.pl

```
prawidłowy_hetman((Wiersz, Kolumna)) :-
    Zakres = [1,2,3,4,5,6,7,8],
    należy(Wiersz, Zakres), należy(Kolumna, Zakres).

prawidłowa_szachownica([]).
prawidłowa_szachownica([Głowa|Ogon]) :- prawidłowy_hetman(Głowa), :-
    prawidłowa_szachownica(Ogon).

wiersze([], []).
wiersze([(Wiersz, _)|HetmanyOgon], [Wiersz|WierszeOgon]) :-
    wiersze(HetmanyOgon, WierszeOgon).

kolumny([], []).
kolumny([(_, Kolumna)|HetmanyOgon], [Kolumny|KolumnyOgon]) :-
    kolumny(HetmanyOgon, KolumnyOgon).

przekątne1([], []).
przekątne1([(Wiersz, Kolumna)|HetmanyOgon], [Przekątna|PrzekątneOgon]) :-
    Przekątna is Kolumna - Wiersz,
    przekątne1(HetmanyOgon, PrzekątneOgon).

przekątne2([], []).
przekątne2([(Wiersz, Kolumna)|HetmanyOgon], [Przekątna|PrzekątneOgon]) :-
    Przekątna is Kolumna + Wiersz,
    przekątne2(HetmanyOgon, PrzekątneOgon).

ośmiu_hetmanów(Szachownica) :-
    length(Szachownica, 8),
    prawidłowa_szachownica(Szachownica).
```

```
    wiersze(Szachownica, Wiersze),
    kolumny(Szachownica, Kolumny),
    przekątne1(Szachownica, Przekątne1).
    przekątne2(Szachownica, Przekątne2).

    fd_all_different(Wiersze),
    fd_all_different(Kolumny),
    fd_all_different(Przekątne1),
    fd_all_different(Przekątne2),
```

Gdybyśmy w tym momencie uruchomili program, to zacząłby on działać… działać… i działać. Po prostu istnieje zbyt wiele kombinacji, by można je było skutecznie posortować. Jeśli się nad tym zastanowimy, łatwo dojdziemy do przekonania, że w każdym wierszu może być dokładnie jeden hetman. Aby przyspieszyć znalezienie rozwiązania, możemy wprowadzić szachownicę w następującej postaci:

```
| ?- ośmiu_hetmanów([(1, A), (2, B), (3, C), (4, D), (5, E), (6, F), (7, G), (8, H)]).

A = 1
B = 5
C = 8
D = 6
E = 3
F = 7
G = 2
H = 4 ?
```

To rozwiązanie działa zadowalająco, ale program w dalszym ciągu wykonuje zbyt wiele obliczeń. Dość łatwo można wyeliminować niektóre wiersze i jednocześnie uprościć API. Poniżej zamieszczono trochę bardziej zoptymalizowaną wersję.

Pobierz: prolog/hetmany_zoptymalizowane.pl

```
prawidłowy_hetman((Wiersz, Kolumna)) :- należy(Kolumna, [1,2,3,4,5,6,7,8]).

prawidłowa_szachownica([]).
prawidłowa_szachownica([Głowa|Ogon]) :- prawidłowy_hetman(Głowa), :-
    prawidłowa_szachownica(Ogon).

kolumny([], []).
kolumny([(_, Kolumna)|HetmanyOgon], [Kolumny|KolumnyOgon]) :-
    kolumny(HetmanyOgon, KolumnyOgon).

przekątne1([], []).
przekątne1([(Wiersz, Kolumna)|HetmanyOgon], [Przekątna|PrzekątneOgon]) :-
    Przekątna is Kolumna - Wiersz,
    przekątne1(HetmanyOgon, PrzekątneOgon).

przekątne2([], []).
przekątne2([(Wiersz, Kolumna)|HetmanyOgon], [Przekątna|PrzekątneOgon]) :-
    Przekątna is Kolumna + Wiersz,
    przekątne2(HetmanyOgon, PrzekątneOgon).
```

```
ośmiu_hetmanów(Szachownica) :-
    Szachownica = [(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)],
    prawidłowa_szachownica(Szachownica).

    kolumny(Szachownica, Kolumny),
    przekątne1(Szachownica, Przekątne1).
    przekątne2(Szachownica, Przekątne2).

    fd_all_different(Kolumny),
    fd_all_different(Przekątne1),
    fd_all_different(Przekątne2),
```

Ogólnie rzecz biorąc, wprowadziliśmy jedną istotną zmianę. Dopasowaliśmy listę `Szachownica` z wartościami `(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)` po to, aby znacznie zmniejszyć całkowitą liczbę permutacji. Usunęliśmy również reguły związane z wierszami i wyświetlaniem wyników. Na moim starym MacBooku znalezienie wszystkich rozwiązań zajęło około trzech minut.

Końcowe wyniki są zadowalające. Stworzyliśmy niewielką bazę wiedzy na temat rozwiązania. Opisaliśmy reguły łamigłówki oraz zastosowaliśmy kilka reguł logicznych w celu przyspieszenia procesu wyszukiwania rozwiązania. W przypadku pewnej klasy problemów Prolog okazuje się niezastąpiony.

Czego nauczyliśmy się trzeciego dnia?

W dzisiejszym dniu zebraliśmy kilka koncepcji, które wykorzystaliśmy w Prologu do rozwiązania kilku klasycznych łamigłówek. Problemy z ograniczeniami mają wiele wspólnych cech z klasycznymi aplikacjami. Wystarczy zdefiniować ograniczenia i znaleźć rozwiązanie. Nigdy nie pomyślelibyśmy o tym, aby w imperatywny sposób zdefiniować złączenie dziewięciu tabel SQL, ale nawet nie mrugnęliśmy okiem, kiedy przyszło nam w taki sposób rozwiązać problem logiczny.

Rozpoczęliśmy od rozwiązania sudoku. Rozwiązanie go w Prologu okazało się niezwykle proste. Odwzorowaliśmy szesnaście zmiennych na wiersze, kolumny i kwadraty. Następnie opisaliśmy reguły łamigłówki, które zmuszają do tego, aby każdy wiersz, kolumna i kwadrat były unikatowe. To wystarczyło, aby Prolog zaczął metodycznie analizować możliwości, co doprowadzało do szybkiego znalezienia rozwiązania. Do stworzenia intuicyjnego API używaliśmy symboli wieloznacznych i zmiennych, ale w żaden sposób nie opisywaliśmy technik rozwiązania.

Następnie skorzystaliśmy z Prologu do rozwiązania problemu ośmiu hetmanów. Tak jak wcześniej zakodowaliśmy reguły gry i poleciliśmy Prologowi znaleźć rozwiąza-

nie. Ten klasyczny problem wymaga intensywnych obliczeń — istnieją 92 rozwiązania, ale nawet przy naszym uproszczonym podejściu znalezienie rozwiązania zajęło kilka minut.

W dalszym ciągu nie znam wszystkich sztuczek i technik wymaganych do rozwiązywania złożonych łamigłówek sudoku, jednak znając Prolog, nie muszę ich znać. Aby się bawić, muszę znać jedynie reguły gry.

Dzień 3. Praca do samodzielnego wykonania

Poszukaj:

- Prolog zawiera również mechanizmy wejścia-wyjścia. Poszukaj predykatów służących do wyświetlania wartości zmiennych.
- Znajdź sposób wykorzystania predykatów wyświetlających dane w celu wyświetlenia tylko pozytywnych rozwiązań. W jaki sposób to działa?

Wykonaj następujące zadania:

- Zmodyfikuj program do rozwiązywania sudoku w taki sposób, by pozwalał na rozwiązywanie łamigłówek 6×6 (bloki 3×2) oraz 9×9 (bloki 3×3).
- Wprowadź mechanizm wyświetlania ciekawszych rozwiązań.

Czytelników, którzy są entuzjastami łamigłówek, Prolog może bardzo wciągnąć. Osobom, które chcą dokładniej przyjrzeć się zaprezentowanym tu łamigłówkom, proponuję rozpoczęcie od problemu ośmiu hetmanów.

- Rozwiąż problem ośmiu hetmanów, pobierając listę hetmanów. Zamiast reprezentowania hetmana za pomocą krotki zaprezentuj go za pomocą liczby całkowitej z zakresu 1 – 8. Wiersz hetmana wyznacz na podstawie jego pozycji na liście, a kolumnę na podstawie wartości na liście.

4.5. Prolog. Podsumowanie

Prolog to jeden z najstarszych języków spośród omawianych w niniejszej książce, ale zasady jego działania są w dalszym ciągu interesujące i ważne także dziś. Prolog to programowanie logiczne. Z Prologu korzystamy w celu przetwarzania reguł złożonych z klauzul, a te z kolei są złożone z ciągu celów.

Programowanie w Prologu składa się z dwóch zasadniczych kroków. Najpierw budujemy bazę wiedzy składającą się z faktów logicznych oraz wniosków związanych

z dziedziną problemu. Następnie kompilujemy bazę wiedzy i zadajemy pytania dotyczące dziedziny. Niektóre z pytań są asercjami. Prolog odpowiada na nie `yes` (tak) lub `no` (nie). W innych zapytaniach występują zmienne. Prolog wypełnia te luki w taki sposób, by zapytania zwracały prawdę.

W Prologu zamiast prostego przypisywania wartości stosowany jest proces zwany **unifikacją**. W wyniku przeprowadzenia tego procesu dopasowywane są wartości zmiennych po obu stronach reguły. Czasami w celu wyciągnięcia wniosku Prolog musi wypróbować wiele różnych kombinacji zmiennych.

Zalety

Prolog może służyć do rozwiązywania różnorodnych problemów, począwszy od systemów planowania lotów dla linii lotniczych, a skończywszy na systemach finansowych. Nauka Prologu wymaga sporo wysiłku, ale złożone problemy, jakie można rozwiązywać za pomocą Prologu oraz podobnych mu języków, rekompensują włożony wysiłek.

Przypomnijmy sobie pracę Briana Tarboksa z delfinami. Udało mu się opisać proste fakty na temat rzeczywistych zachowań delfinów i na ich podstawie wyciągnąć przełomowe wnioski. Udało mu się również rozdzielić niezwykle ograniczone zasoby i za pomocą Prologu opracować stosowny harmonogram. Poniżej wyszczególniono kilka obszarów, w których dziś wykorzystuje się Prolog.

Przetwarzanie języka naturalnego

Prolog był jednym z pierwszych języków wykorzystywanych do rozpoznawania języków. Modele napisane w Prologu potrafią dla wybranego języka naturalnego zastosować bazę wiedzy złożoną z faktów i reguł i na ich podstawie zaprezentować złożony i nieprecyzyjny język w postaci konkretnych reguł możliwych do przetwarzania przez komputery.

Gry

Gry stają się coraz bardziej złożone. W szczególności coraz bardziej skomplikowane staje się modelowanie zachowania współzawodniczących graczy lub przeciwników. Za pomocą modeli zapisanych w Prologu można z łatwością zaprezentować zachowania różnych postaci występujących w grach. Prolog można także wykorzy-

stać do tworzenia zachowań i kreowania nowych rodzajów przeciwników. To zbliża grę do rzeczywistości i poprawia jej atrakcyjność.

Semantyczna Sieć

Semantyczna Sieć (ang. *Semantic Web*) to próba nadania znaczenia serwisom i informacjom zgromadzonym w internecie tak, by mogły być łatwiej przetwarzane przez komputery. Zasoby są opisywane za pomocą języka opisu zasobów (*Resource Description Language* — RDF). Następnie opis ten podlega kompilacji i w ten sposób powstaje baza wiedzy. Baza wiedzy w połączeniu z możliwościami przetwarzania w Prologu języka naturalnego dają użytkownikom olbrzymie możliwości. Istnieje wiele programów napisanych w Prologu oferujących ten rodzaj funkcji dla serwerów WWW.

Sztuczna inteligencja

Sztuczna inteligencja (*Artificial Intelligence* — AI) koncentruje się wokół wyposażania maszyn w inteligencję. Wspomniana inteligencja może przyjmować różne formy, choć w każdym przypadku jakiś „agent" modyfikuje działania na podstawie złożonych reguł. Prolog bryluje w tym obszarze, zwłaszcza wtedy, kiedy reguły są konkretne i bazują na logice formalnej. Z tego względu Prolog jest czasami nazywany **językiem programowania logicznego**.

Szeregowanie

Prolog świetnie nadaje się do rozdzielania ograniczonych zasobów. Znanych jest wiele przypadków, kiedy Prolog był używany do tworzenia mechanizmów szeregujących systemów operacyjnych oraz innych zaawansowanych systemów szeregowania.

Słabe punkty

Prolog to język, który przez lata ulegał zmianom. Pomimo tego jest on pod wieloma względami przestarzały i posiada istotne ograniczenia.

Zastosowania

Chociaż Prolog jest świetny w swojej dziedzinie, jest to jednak specyficzna dziedzina niszowa — programowanie logiczne. Nie jest to język ogólnego przeznaczenia. Ma również kilka ograniczeń związanych z projektem języka.

Bardzo duże zbiory danych

W Prologu drzewo decyzyjne jest przeszukiwane „najpierw w głąb" — z ustalonym zbiorem reguł dopasowywane są wszystkie możliwe kombinacje. W wielu językach i kompilatorach proces ten jest dość dobrze zoptymalizowany. Pomimo tego strategia ta jest wewnętrznie kosztowna obliczeniowo, zwłaszcza w przypadku dużych zbiorów danych. Ponadto zastosowanie tej metody zmusza użytkowników Prologu do rozumienia sposobu działania języka. Tylko w ten sposób mogą oni utrzymać rozmiar zbiorów danych na akceptowalnym poziomie.

Mieszanie modelu imperatywnego z deklaratywnym

Podobnie jak w przypadku wielu języków z rodziny języków funkcyjnych, zwłaszcza tych, w których znaczącą rolę odgrywa rekurencja, użytkownik musi rozumieć sposób, w jaki Prolog interpretuje rekurencyjne reguły. Często do rozwiązywania nawet przeciętnie złożonych problemów trzeba wykorzystywać reguły rekurencji ogonowej. Stosunkowo łatwo można stworzyć aplikacje w Prologu, które nie dają się skalować i mogą służyć do obsługi jedynie trywialnych zbiorów danych. Bardzo często opracowanie efektywnych reguł — takich, które można skalować do akceptowalnego poziomu — wymaga od programisty dobrej znajomości sposobu działania Prologu.

Wnioski końcowe

Podczas pracy nad kilkoma językami na potrzeby tej książki zdarzało mi się doznawać olśnienia, kiedy zdawałem sobie sprawę z tego, jak złych narzędzi używałem do rozwiązywania wielu problemów. Prolog jest jednym z tych języków, który uświadamiał mi to szczególnie często. Polecam wszystkim, aby używali Prologu do rozwiązywania zadań, które szczególnie pasują do tego języka. Tego bazującego na regułach logicznych języka można używać w połączeniu z innymi językami ogólnego przeznaczenia — na podobnej zasadzie, na jakiej używa się języka SQL wewnątrz języków Ruby lub Java. Uważne połączenie możliwości kilku języków pozwala na sprawniejsze programowanie.

Skorowidz

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

Ś

T

U

W

X

Z

Siedem języków w siedem tygodni

Praktyczny przewodnik nauki **języków programowania**

Jeśli myślisz, że to kolejna książka z serii „Jak schudnąć 50 kilogramów w trzy dni" albo „Jak zostać obrzydliwie bogatym w dwa tygodnie", na szczęście się mylisz! Oto podręcznik, który w siedem tygodni przedstawi Ci najważniejsze modele programowania na przykładzie siedmiu przydatnych języków. Zaproponowana tu innowacyjna forma nauki pozwoli Ci poznawać je dzień po dniu. Zaczniesz od krótkiego omówienia składni i możliwości każdego języka, by na końcu wypróbować go w akcji. I choć po lekturze tej książki nie staniesz się ekspertem, opanujesz to, co w każdym z przedstawionych tu języków jest kluczowe. Będziesz mógł tworzyć czytelniejszy, lepszy kod z mniejszą liczbą powtórzeń. Zdobędziesz także niezwykle cenną umiejętność – zaczniesz sprawnie wykorzystywać pojęcia z jednego języka w celu znalezienia kreatywnych rozwiązań w innym!

W książce tej opisano jeden język programowania logicznego, dwa z pełną obsługą pojęć obiektowych, cztery o charakterze funkcyjnym i jeden prototypowy – wszystko po to, by zapewnić Ci możliwie najbardziej wszechstronne przygotowanie programistyczne. Lepiej przyswoisz sobie także techniki obsługi współbieżności, będące kręgosłupem następnej generacji aplikacji internetowych, oraz poznasz sposoby wykorzystywania filozofii „Let it crash" Erlanga do budowy systemów odpornych na awarie.

Bruce A. Tate prowadzi w Austin, w stanie Teksas, firmę RapidRed, która zajmuje się tworzeniem aplikacji w Ruby. Jest autorem ponad dziesięciu podręczników informatycznych, wydawanych na całym świecie. Należą do nich: „From Java to Ruby", „Deploying Rails Applications", „Beyond Java" oraz zdobywczyni prestiżowej nagrody Jolt – książka „Better, Faster, Lighter Java".

Pragmatic Bookshelf

Jakie praktyczne języki poznasz dzięki tej książce?

1. **Ruby** – język obiektowy, a przy tym łatwy w użytkowaniu i czytelny
2. **Io** – prototypowy język, wyposażony w unikatowy mechanizm dystrybucji komunikatów
3. **Prolog** – język oferujący łatwe rozwiązania, które w Javie lub C byłyby bardzo kłopotliwe
4. **Scala** – jeden z języków nowej generacji, przeznaczony na maszynę wirtualną Javy
5. **Erlang** – język funkcyjny, z mechanizmami obsługi współbieżności, na którym działa już kilka słynnych baz danych w stylu cloud
6. **Clojure** – język, w którym wykorzystano strategię wersjonowania baz danych w celu zarządzania współbieżnością
7. **Haskell** – język o charakterze czysto funkcyjnym

Jeden z tych języków może już wkrótce stać się Twoim ulubionym narzędziem!

ISBN 978-83-246-3379-1

Cena 59,00 zł