

Prolog

Programowanie

W. F. Clocksin, C. S. Mellish

*Programowanie
w logice*



- Naucz się Prologu – języka programowania opartego na regułach logicznych
- Rozwiązuj w prosty sposób problemy związane z logiką matematyczną i analizą języka naturalnego
- Poznaj praktyczne sposoby pisania programów w Prologu
- Sprawdź swoją wiedzę korzystając z licznych ćwiczeń



wydawnictwo

Spis treści

Wstęp	7
Rozdział 1. Wprowadzenie	11
Prolog.....	11
Obiekty i relacje	12
Programowanie.....	13
Fakty	14
Zapytania.....	16
Zmienne.....	17
Koniunkcje.....	19
Reguły	23
Podsumowanie i ćwiczenia	28
Rozdział 2. Prolog z bliska	31
Składnia.....	31
Stałe	32
Zmienne.....	32
Struktury.....	33
Znaki.....	34
Operatory.....	35
Równość i unifikacja.....	37
Arytmetyka	38
Spełnianie celów — podsumowanie.....	42
Udane spełnienie koniunkcji celów.....	42
Cele i nawracanie.....	45
Unifikacja	47
Rozdział 3. Korzystanie ze struktur danych.....	49
Struktury a drzewa.....	49
Listy	51
Przeszukiwanie rekurencyjne	54
Odwzorowania	57
Porównywanie rekurencyjne.....	60
Łączenie struktur	62
Akumulatory	66
Struktury różnicowe.....	68
Rozdział 4. Nawracanie i odcięcie.....	71
Generowanie wielu rozwiązań.....	72
Odcięcie.....	75

Typowe zastosowania odcięcia.....	80
Potwierdzanie wyboru reguły	80
Użycie odcięcia z predykatem fail	84
Kończenie generowania możliwych rozwiązań i ich sprawdzanie	86
Niebezpieczeństwa wynikające ze stosowania odcięcia	89
Rozdział 5. Wejście i wyjście	91
Czytanie i pisanie termów.....	92
Czytanie termów.....	92
Pisanie termów	93
Czytanie i pisanie znaków	96
Czytanie znaków.....	96
Pisanie znaków.....	97
Wczytywanie zdań.....	98
Czytanie z plików i pisanie do plików.....	101
Otwieranie i zamykanie strumieni.....	102
Zmiana bieżącego strumienia wejściowego i wyjściowego	103
Konsultowanie.....	104
Deklarowanie operatorów.....	105
Rozdział 6. Predykaty wbudowane	107
Wprowadzanie nowych klauzul.....	107
Sukces i porażka	109
Klasyfikacja termów	110
Przetwarzanie klauzul jako termów	111
Tworzenie składników struktur i sięganie do nich	114
Wpływ na nawracanie	118
Tworzenie celów złożonych.....	119
Równość.....	122
Wejście i wyjście	122
Obsługa plików.....	124
Wyliczanie wyrażeń arytmetycznych.....	124
Porównywanie termów.....	126
Badanie działania Prologu	127
Rozdział 7. Przykładowe programy	129
Sortowany słownik w formie drzewa	129
Przeszukiwanie labiryntu.....	132
Wieże Hanoi.....	135
Program magazynowy.....	136
Przetwarzanie list.....	137
Zapis i przetwarzanie zbiorów.....	140
Sortowanie	142
Użycie bazy danych.....	145
random	145
gensym	146
findall	147
Przeszukiwanie grafów.....	149
Odsiej Dwójki i odsiej Trójki.....	153
Różniczkowanie symboliczne	155
Odwzorowywanie struktur i przekształcanie drzew	157
Przetwarzanie programów	160
Literatura	163

Rozdział 8. Usuwanie błędów w programach prologowych.....	165
Układ programów	166
Typowe błędy.....	168
Śledzenie programu.....	171
Śledzenie i punkty kontrolne	177
Sprawdzanie celu	179
Sprawdzanie przodków.....	180
Zmiana poziomu śledzenia.....	181
Zmiana sposobu spełnienia celu.....	182
Inne opcje	183
Podsumowanie	184
Poprawianie błędów	184
Rozdział 9. Użycie reguł gramatycznych w Prologu	187
Parsowanie.....	187
Problem parsowania w Prologu	190
Notacja reguł gramatyki	194
Dodatkowe argumenty	196
Dodatkowe warunki	199
Podsumowanie	201
Przekształcanie języka na logikę.....	202
Ogólniejsze zastosowanie reguł gramatyki	204
Rozdział 10. Prolog a logika.....	207
Krótkie wprowadzenie do rachunku predykatów.....	207
Postać klauzulowa.....	210
Zapis klauzul	215
Rezolucja i dowodzenie twierdzeń.....	216
Klauzule Horna	220
Prolog.....	220
Prolog i programowanie w logice	222
Rozdział 11. Projekty w Prologu.....	225
Łatwiejsze projekty.....	225
Projekty zaawansowane	227
Dodatek A Odpowiedzi do niektórych ćwiczeń.....	231
Dodatek B Klauzulowa postać programów	235
Dodatek C Przenośne programy w standardowym Prologu	241
Przenośność standardu Prologu	241
Różne implementacje Prologu.....	242
Czego się wystrzegać	243
Definicje wybranych predykatów standardowych	244
Przetwarzanie znaków.....	245
Dyrektywy	247
Wejście i wyjście strumieniowe.....	247
Różne	249
Dodatek D Różne wersje Prologu.....	251
Dodatek E Dialekt edynburski.....	255
Dodatek F micro-Prolog	263
Skorowidz.....	267

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 opisanie 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 „Bizuteria 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 zależć 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 początkiem 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 zaniechać 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 zaniechać się w szczegóły, zasady formalne czy wyjątki. Na razie nie będziemy siłili się na kompletność wykładu czy jego formalną poprawność. Chcemy, by czytelnik szybko posiadał 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.

Żałómy na przykład, że wprowadziliśmy do systemu Prologu przedstawioną wyżej regułę dotyczącą siostr. 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. Żałómy, 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 naturalnym¹:

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

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

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

Przykładowo, nazwa `zloto` może odnosić się do obiektu — wtedy zwykle chodzi 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 nawiasie nazywamy *argumentami*. Pamiętajmy, że w informatyce słowo „argument” nie ma wspólnego z potocznym znaczeniem tego słowa, czyli „racją w dyskusji”. Nazwę relacji znajdującą się przed nawiasem nazywamy *predykatem*. Wobec tego predykat `lubi` 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, 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.

Mozna deklarować różne fakty, także fakty, które nie są prawdziwe w świecie rzeczywistym. 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 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.

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 — *przypr. tłum.*

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. Założmy, 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? Założmy, 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 wystarczy 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 nieporęczne 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:

```
lub(jan, kwiaty).
lub(jan, maria).
lub(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 *lub* i pierwszym argumentem *jan*, drugi argument może być dowolny, gdyż zapytanie w drugim argumente zawiera zmienną nieukonkretnioną. Kiedy odpowiedni fakt zostanie znaleziony, od tej chwili *X* oznacza drugi argument znalezionej fakt. Prolog przeszukuje bazę danych w takiej kolejności, w jakiej ją wpisał (czyli od góry do dołu), wobec czego fakt *lub(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 wystarczy 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 *lub(jan, maria)*. Zmienna *X* jest *ukonkretniana* wartością *maria*, Prolog wstawia znacznik na fakt *lub(jan, maria)*. Prolog wyświetla *X=maria* i czeka na dalsze polecenia. Jeśli wpisujemy 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; wciśnięliśmy średnik (;) i RETURN
X=pawel;	druga odpowiedź; znów wciśnięliśmy 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. Założmy, że mamy bazę danych:

```
lub(maria, czekolada).
lub(maria, wino).
lub(jan, wino).
lub(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)*.

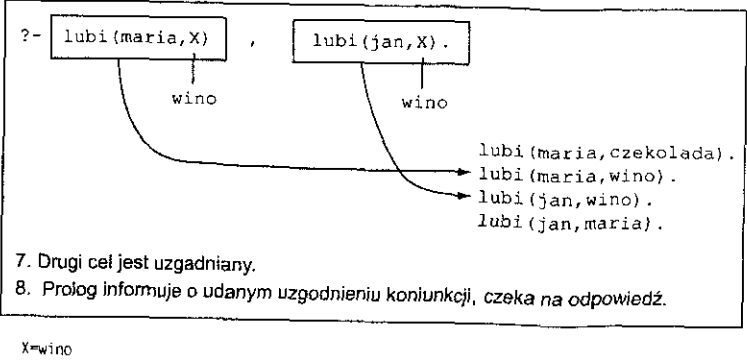
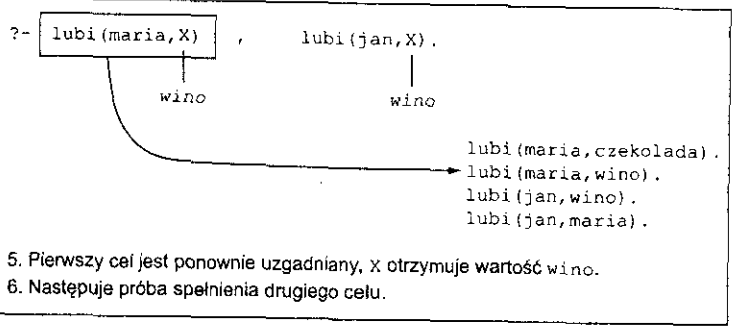
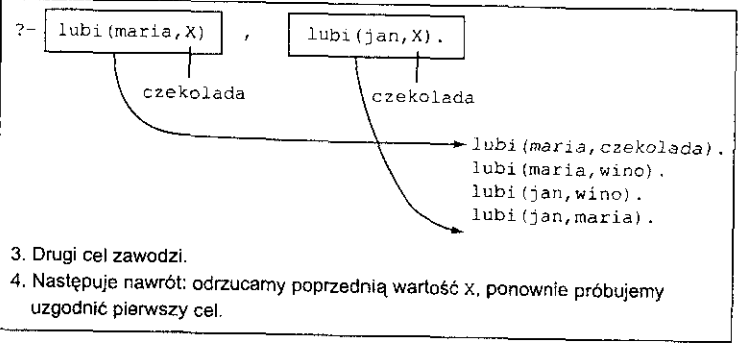
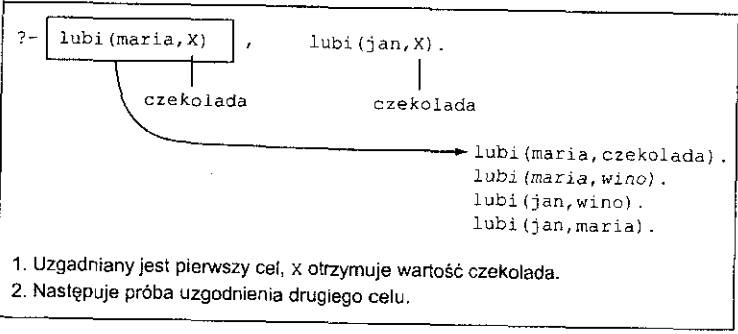
Tak jak w przypadku wszystkich zapytań, Prolog odnajduje pierwszą odpowiedź, zatrzymuje się i czeka na dalsze instrukcje. Jeśli wpisujemy:, poszukiwanie rozwiązań będzie kontynuowane. Wiemy, że odbywa się to przez próby innego spełnienia obydwóch zadanych celów od zostawionych wcześniej znaczników.

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 powiązany — 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.



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:

- `lubim(jan, alfred) .`
- `lubim(jan, bertrand) .`
- `lubim(jan, cyryl) .`
- `lubim(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 reguły 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.

Reguły 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, poczynawszy 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:

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

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

`lub(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:

`lub(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 `lub(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 wszelkie 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).`

Wyższa Szkoła
Zarządzania i Inżynierii
ul. Armii Krajowej 4, 30-150 NIEKŁOSZÓW
tel. (012) 638-65-77, fax (012) 637-63-47
Wpis do KRS 0000154311, NIP 14-12607-27600-00
Konto BBS Orlakow 15431115-12607-27600-00
NIP 677-17-58-169 REGON 350814846

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,0)`, gdzie `M i 0` dopiero mają być ukonkretnione. Znalezione zostaje fakt `rodzice(alicja,wiktoria,albert)`, więc cel jest uzgodniony. Prolog oznacza w bazie danych szóstą pozycję, ukonkretnia `M` wartością `wiktoria` i `0` 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 0`. 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`.

Zalóżmy 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,0)`, jest on dopasowywany do faktu `rodzice(alicja,wiktoria,edward)`. Znamy już zatem wartości zmiennych `M i 0`.
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 predykatie `lubi` w dwóch postaciach: możemy podawać fakty i reguły. Ogólnie rzecz biorąc, predykat definiuje się jako zbiór faktów i reguł; jedno 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 ukrąść, 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 ukrąść 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 oznacza 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` korzystając 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.

Rozdział 2.

Prolog z bliska

W tym rozdziale dokładnie omówimy poszczególne części programu prologowego, które przedstawiliśmy w rozdziale poprzednim. Prolog zawiera mechanizmy pozwalające na strukturyzację danych oraz strukturyzację prób uzgadniania celów. Strukturyzacja danych związana jest ze znajomością składni używanej do zapisu danych. Strukturyzacja sposobu uzgadniania celów związana jest z mechanizmem nawracania.

Składnia

Składnia języka opisuje dopuszczalne sposoby zestawiania słów. W języku polskim zdanie „Widzę zebra” jest poprawne, ale zdanie „Zebra widzę ja” już nie jest poprawne. W pierwszym rozdziale nie omawialiśmy dokładnie składni Prologu, po prostu pokazaliśmy ją na przykładach. Dalej znajduje się podsumowanie składni omawianych dotąd elementów.

Programy Prologu składają się z *termów*. Term to *stała*, *zmienna* lub *struktura*. Wszystkie rodzaje termów widzieliśmy w poprzednim rozdziale, tylko nie znaliśmy tych nazw. Każdy term zapisywany jest jako ciąg *znaków*; znaki podzielono na cztery następujące kategorie:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

+ - * / \ ~ ^ < > . : ? @ # \$ %

Pierwszy wiersz to wielkie litery, drugi to małe litery, trzeci — cyfry, zaś czwarty to znaki specjalne (symbole). W czwartym wierszu brakuje jeszcze części znaków, ale mają one specjalne znaczenie i omówimy je dalej. Każdy rodzaj termu: stała, zmienna czy struktura, rządzi się innymi zasadami tworzenia jej nazwy ze znaków. Teraz zasady te omówimy.

Stałe

Stałe *nazywają* konkretne obiekty lub konkretne relacje. Istnieją dwa rodzaje stałych: atomy i liczby całkowite. Przykładami atomów mogą być nawy używane w poprzednim rozdziale:

```
lubi maria jan ksiazka wino posiada klejnoty moze_ukrasc
```

Specjalne symbole używane do oznaczania pytań `?` i reguł `:` to także atomy. Istnieją dwa rodzaje atomów: składające się z liter i cyfr oraz składające się z symboli. Pierwszy rodzaj atomów zaczyna się małą literą. Atomy składające się z symboli zawierają tylko symbole. Czasami atom musi jednak zaczynać się wielką literą lub cyfrą. Jeśli atom ujęty jest w pojedynczy cudzysłów, `('')`, może zawierać dowolne znaki. Aby poprawić czytelność nazw, można do środka wstawiać podkreślenie `(_)`. Oto kolejne przykłady atomów:

```
a void = 'jan-kowalski' --> jan_kowalski 1eh2304
```

Poniższe przykłady *nie* są już poprawnymi atomami:

```
23041eh jan-kowalski Void _alfa
```

Innym rodzajem stałych. Nie omawialiśmy jeszcze obliczeń w Prologu, ale zajmijmy się tym tematem w dalszej części niniejszego rozdziału. Oto przykłady liczb:

```
-17 -2.67e2 0 1 99.9 512 8192 14765 67344 6.02e-23
```

Większość powyższych przykładów liczb powinna być czytelnikowi znajoma. Symbol „e” oznacza potęgę 10, więc na przykład $-2.67e2$ to -2.67×10^2 czyli po prostu -276 , a $6.02e-23$ to 6.02×10^{-23} .

Poza tym programiści Prologu mają do swojej dyspozycji biblioteki pozwalające między innymi robić obliczenia na liczbach wymiernych oraz liczbach o dowolnej precyzji, ale w tej książce nie będziemy ich potrzebować.

Zmienne

Drugi rodzaj termów używanych w Prologu to zmienne. Zmienne mają postać atomów, ale ich nazwy zaczynają się wielkimi literami lub podkreśleniem. Zmienną należy traktować jak zastępstwo obiektu, którego nie możemy w danej chwili nazwać. Zmienne Prologu można porównać do zaimków języka polskiego. W przedstawianych dotąd przykładach zmienne nazywaliśmy literami `X`, `Y` i `Z`, ale nazwy mogą być dowolnie długie:

```
Odpowiedz Dane PlacaBrutto _3_niewidome_mysz  
Bardzo_dluga_nazwa_jakiej_zmiennej
```

Czasami trzeba jednak użyć zmiennej, której nazwa nie ma znaczenia. Z taką sytuacją mamy do czynienia na przykład wówczas, gdy interesuje nas, czy ktoś lubi Jana, ale nie chcemy nawet wiedzieć, kto. W takich wypadkach używa się *zmiennej anonimowej*.

Zmienną taką oznacza się pojedynczym podkreśleniem; powyższy przykład należałoby zapisać tak:

```
?- lubi(_jan).
```

Wiele zmiennych anonimowych występujących w tej samej klauzuli nie musi mieć jednolitej interpretacji; jest to cecha szczególna tych zmiennych. Zmiennych anonimowych używa się po to, aby nie było konieczne wymyślanie mnóstwa różnych nazw dla zmiennych, które i tak nigdzie indziej nie zostaną użyte.

Struktury

Trzeci rodzaj termów używanych w programach Prologu to *struktury*. Struktury w standardzie języka Prolog nazywane są „termami złożonymi”, ale w niniejszej książce pozostaniemy przy prostszym słowie „struktura”. Struktura to pojedynczy obiekt składający się z zestawu innych obiektów, nazywanych *składnikami* struktury. Składniki są pogrupowane w strukturę, aby ułatwić ich przetwarzanie.

Przykładem struktur znanym każdemu z codziennych zastosowań są karty katalogowe w bibliotece. Taka karta może zawierać szereg informacji (składników): nazwisko autora, tytuł książki, datę wydania, położenie w bibliotece i tak dalej. Niektóre składniki można rozbić na składniki drobniejsze, na przykład nazwisko autora tak naprawdę zawiera imię, ewentualnie inicjały pozostałych imion i nazwisko.

Struktury pomagają zorganizować dane programu, gdyż pozwalają traktować grupę powiązanych informacji jako całość. To, jak potem te dane są rozbijane na elementy, zależy od rozwiązywanego problemu i zajmijmy się tym zagadnieniem później.

Struktury są też przydatne, gdy mamy do czynienia z typowym rodzajem obiektów, a może istnieje wiele obiektów tego rodzaju. Przykładem mogą być książki. W rozdziale 1. używaliśmy faktu

```
posiada(jan,ksiazka)
```

aby zapisać, że Jan ma pewną książkę. Jeśli potem napiszemy

```
posiada(maria,ksiazka)
```

oznaczać to będzie, że Maria ma tę samą książkę, gdyż użyto takiej samej nazwy obiektu. Obiektów nie można rozróżnić inaczej jak przez ich nazwy. Aby dokładnie opisać książki posiadane przez Jana i Marię, można powiedzieć

```
posiada(jan,wichrowe_wzgorza).  
posiada(maria,moby_dick).
```

Jednak w przypadku dużego programu byłoby to rozwiązanie niewygodne, gdyż istniałoby wiele różnych stałych bez kontekstu objaśniającego ich znaczenie. Osoba czytająca taki program mogłaby nie wiedzieć, że `wichrowe_wzgorza` to tytuł książki Emily Brontë, autorki z Yorkshire w Anglii, żyjącej w XIX wieku; przecież Jan mógłby na przykład nazwać swojego królika imieniem „wichrowe_wzgórza”. Struktury pomagają w określeniu kontekstu danych.

Struktury w Prologu zapisujemy podając *funktor* oraz jego *składniki*. Nazwa funktora określa rodzaj struktury, odpowiada ona typom w tradycyjnych językach programowania. Składniki ujęte są w nawias okrągły i oddzielone od siebie przecinkami. Funktor umieszcza się przed nawiasem otwierającym. Oto fakt mówiący, że Jan ma książkę „Wichrowe wzgórza” Emily Brontë:

```
posiada(jan,ksiazka(wichrowe_wzgorza,bronte)).
```

Wewnątrz faktu posiada mamy strukturę o nazwie *ksiazka* z dwoma składnikami, tytułem i autorem. Struktura *ksiazka* pojawia się *wewnątrz* faktu, jako jeden z jego argumentów, zachowuje się jak obiekt należący do relacji. Moglibyśmy utworzyć też strukturę zawierającą dane o autorze, gdyż Brontë to nazwisko trzech pisarek:

```
posiada(jan,ksiazka(wichrowe_wzgorza,autor(emily,bronte))).
```

Struktury mogą być używane w zapytaniach i mogą zawierać zmienne. Możemy przykładowo zapytać, czy Jan ma jakieś książki którejkolwiek z sióstr Brontë:

```
?- posiada(jan,ksiazka(X,autor(Y,bronte))).
```

Jeśli odpowiedź będzie twierdząca, zmienna *X* zostanie ukonkretniona znalezionym tytułem, zaś *Y* imieniem autora. Jeśli informacje te nie są nam potrzebne, możemy użyć zmiennych anonimowych:

```
?- posiada(jan,ksiazka(_ ,autor(_ ,bronte))).
```

Pamiętajmy, że kolejne wystąpienia zmiennej anonimowej nie mają ze sobą nic wspólnego.

Mozna byłoby jeszcze poprawić strukturę opisującą książki przez dodanie argumentu mówiącego, o *który egzemplarz* książki chodzi. Jeśli trzeci argument będzie przykładowo liczbą całkowitą, możemy jednoznacznie identyfikować książkę:

```
posiada(jan,ksiazka(ulisses,autor(james,joyce),3129)).
```

Powyższy zapis interpretujemy następująco: *Jan posiada 3 129. egzemplarz „Ulisses” Jamesa Joyce’a*.

Jeśli wydaje ci się, czytelniku, że składnia struktur i faktów jest taka sama, to masz rację. Predykat (używany w faktach i regułach) jest tak naprawdę funktorem struktury. Argumenty faktu lub reguły są *de facto* składnikami struktury. Zapisywanie programów Prologu jako struktur ma wiele zalet. Obecnie nie ma znaczenia, dlaczego tak jest, ale trzeba pamiętać, że wszystko w Prologu, nawet same programy, składają się ze stałych, zmiennych i struktur.

Znaki

Nazwy stałych i zmiennych stanowią łańcuch znaków. Wprawdzie wszystkie rodzaje nazw (atomy, liczby całkowite, zmienne) rządzą się własnymi zasadami doboru znaków, lecz dobrze wiedzieć, jakie znaki Prolog musi zawsze rozpoznawać. Znak także

może być traktowany jako dana. Znamy już liczby całkowite, więc warto opisać sposób kodowania znaków jako liczb, gdyż wiedza ta jest przydatna do operacji „wejścia” i „wyjścia”; dokładniej zagadnienie to omówimy w rozdziale 5.

Prolog rozróżnia dwie grupy znaków: drukowalne (czarne) i niedrukowalne (białe). Znaki drukowalne pojawiają się na monitorze, zaś niedrukowalne się nie pojawiają, za to powodują wykonanie czegoś: zostawienie odstępu (spacji), przejście do nowego wiersza i tym podobne. Oto wszystkie znaki drukowalne, których można używać¹:

```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # $ % & ' ( ) = - - ^ | \ { } [ ] _ ` @ + ; * : < > , . ? /
```

Nietrudno zauważyć, że jest to uzupełniony zestaw z początku tego rozdziału. Niektóre znaki mają specjalne znaczenie, na przykład nawiasy okrągłe obejmują składniki struktury. W kolejnych rozdziałach przekonamy się jednak, że znaków tych można używać jako danych Prologu. Znaki mogą być wyświetlane, odczytywane z klawiatury, porównywane i poddawane operacjom arytmetycznym.

Operatory

Czasami wygodnie jest zapisać niektóre funktory jako *operatory*, gdyż użycie takiej składni ułatwia czytanie programu. Na przykład operacje arytmetyczne zapisuje się zwykle za pomocą operatorów. Kiedy chcemy zapisać wyrażenie „ $x + y * z$ ”, wywołujemy operatory dodawania, mnożenia i znaku. Jeśli chcielibyśmy zapisać podane wyrażenie tradycyjnie, jako strukturę, zapis ten wyglądałby następująco: $+(x, *(y, z))$ i byłby poprawnym termem Prologu. Operatory są czasem łatwiejsze w użyciu, choć wynika to z tego, że tak uczy się zapisywać operacje w szkole. Poza tym przy zapisie struktur konieczne jest użycie nawiasów, które też mogą przeszkadzać w czytaniu.

Trzeba zaznaczyć jednak, że operatory nie powodują wykonywania jakichkolwiek obliczeń. Dla Prologu $3+4$ nie jest równoważne z 7 , jest to inny zapis termu $+(3,4)$, czyli struktury danych. Potem zajmiemy się sposobem interpretowania struktur jako wyrażań arytmetycznych i ich wyliczaniem zgodnie z regułami arytmetyki.

Najpierw musimy wiedzieć, jak czytać wyrażenia arytmetyczne zawierające operatory. O każdym operatorze musimy wiedzieć trzy rzeczy: znać jego położenie, priorytet i łączność. W tym podrozdziale powiemy, jak to wygląda, ale nie będziemy się zanadto

¹ Większość implementacji Prologu pozwala używać przynajmniej pełnego zestawu znaków jednobajtowych, więc na przykład z polskimi literami. Niektóre implementacje pozwalają korzystać ze znaków Unicode, dzięki czemu liczba dopuszczalnych znaków jest ogromna i obejmuje praktycznie wszystkie znaki narodowe. Niemniej jednak w nazwach można używać jedynie znaków podawanych tu przez autorów, zaś pozostałe znaki mogą być jedynie danymi — *przyp. tłum.*

zagłębiać w szczegóły. Wprawdzie można będzie tworzyć wiele różnych operatorów, ale zajmiemy się jedynie dobrze znanymi atomami $+$, $-$, $*$ i $/$.

Składnia termu z operatorami zależy między innymi od położenia operatora. Operatory takie jak plus ($+$), minus ($-$), gwiazdka ($*$) i ukośnik ($/$) są zapisywane między swoimi argumentami, nazywamy je zatem operatorami *infiksowymi*. Operatory mogą też występować przed swoimi argumentami, jak minus w wyrażeniu „ $-x + y$ ” oznaczający liczbę przeciwną co do znaku do x . Operatory pojawiające się przed swoimi argumentami to operatory *prefiksowe*. W końcu operatory znajdujące się za swoim argumentem to operatory *postfiksowe*. Przykładem jest oznaczenie silni, mające postać wykrzyknika znajdującego się za liczbą, której silnia jest wyliczana, na przykład silnia x to „ $x!$ ”. Tak więc położenie operatora informuje o tym, jak odnosi się on do argumentów. Wszystkie operatory, którymi będziemy się teraz zajmować, są operatorami *infiksowymi*.

Teraz zajmijmy się priorytetem. Kiedy widzimy term „ $x + y * z$ ”, który ma być zinterpretowany jako wyrażenie algebraiczne, wiemy, że najpierw musimy wymnożyć y przez z , a dopiero potem do wyniku dodać x . Wynika to stąd, że wiadomo, że mnożenie i dzielenie wykonuje się przed dodawaniem i odejmowaniem, chyba że użyto w wyrażeniu nawiasów. Z drugiej strony z zapisu $+(x, *(y, z))$ wynika od razu, w jakiej kolejności należy działania wykonywać. Bierze się to stąd, że struktura z funktorem $*$ jest argumentem struktury z funktorem $+$, więc aby móc w ogóle określić argumenty funktora $+$, najpierw trzeba wyznaczyć strukturę funktora $*$. Tak więc jeśli chcemy używać operatorów, musimy ustalić reguły, w jakiej kolejności należy wykonywać operacje. Do tego właśnie używa się *priorytetów*.

Priorytet operatora mówi, które operacje mają być przeprowadzane najpierw. Każdy operator używany w Prologu ma *klasę priorytetu*. Klasa ta jest liczbą całkowitą związaną z operatorem, zaś wartość tej liczby zależy od używanej wersji Prologu. Zawsze jednak operator z wyższym priorytetem ma klasę priorytetu bliższą jedynki. Jeśli klasy priorytetów mieszczą się w zakresie od 1 do 255, operatory pierwszej klasy priorytetu są wykonywane jako pierwsze, przed operatorami z klasy 129. W Prologu operatory mnożenia i dzielenia mają priorytet wyższy od dodawania i odejmowania, więc term $a-b/c$ jest równoważny z termem $-(a/(b.c))$. Nie jest w tej chwili istotne dokładne określanie klas priorytetów, wystarczy pamiętać, w jakiej kolejności operatory są interpretowane.

Na koniec zastanówmy się nad łącznością operatorów. Łączność operatorów dochodzi do głosu, kiedy mamy kilka operatorów o takim samym priorytecie. Jak należy na przykład zinterpretować wyrażenie „ $8/2/2$ ”: jako $(8/2)/2$ czy jako $8/(2/2)$? W pierwszym wypadku otrzymamy 2, w drugim 8. Aby móc rozróżnić, o który przypadek chodzi, musimy wiedzieć, które operatory są *łączne lewostronnie*, a które *prawostronnie*. Operatory łączne lewostronnie muszą mieć po lewej operację o priorytecie takim samym lub niższym, a po prawej — o priorytecie niższym. Przykładowo, wszystkie operatory arytmetyczne (dodawanie, odejmowanie, mnożenie i dzielenie) są łączne lewostronnie, wobec czego wyrażenie „ $8/4/4$ ” jest interpretowane jako $(8/4)/4$. Poza tym „ $5+8/2/2$ ” jest interpretowane jako $5+((8/2)/2)$.

W praktyce, w przypadku wyrażeń sprawiających trudności w interpretacji, stosuje się nawiasy, które pozwalają uniknąć niejasności. W tej książce też będziemy używać bardzo wielu nawiasów okrągłych, ale i tak trzeba znać reguły składniowe obowiązujące operatory.

Pamiętajmy, że struktury zawierające operatory arytmetyczne są strukturami takimi jak wszelkie inne struktury i żadne obliczenia nie są wykonywane, póki nie wymusi tego użycie predykatu `is` omówionego dalej, w podrozdziale „Arytmetyka”.

Równość i unifikacja

Wartym omówienia jest predykat równości, infiksowy operator zapisywany jako `=`. Kiedy staramy się spełnić cel:

?- $X = Y$.

Prolog stara się *dopasować* X i Y , a jeśli się to uda, cel jest uzgodniony. O unifikacji można myśleć jako o *próbie uczynienia* X i Y *równymi*. Predykat równości jest predykatem *wbudowanym*, co oznacza, że jest z góry zdefiniowany w Prologu. Predykat ten działa tak, jakby był zdefiniowany za pomocą faktu

$X = X$.

W ramach ustalonej klauzuli X zawsze jest równy X , z właściwości tej korzystamy definiując równość w pokazany sposób.

Jeśli mamy cel w postaci $X=Y$, gdzie X i Y są dowolnymi termami mogącymi zawierać zmienne nieukonkretnione, czy X i Y są równe, sprawdza się następująco:

- ♦ Jeśli X jest zmienną nieukonkretnioną, a Y nie jest powiązana z żadnym termem, to X i Y są równe. Poza tym X ukonkretniana jest wartością Y . Na przykład poniższe zapytanie nie zawiedzie, poza tym zmienna X zostanie ukonkretniona strukturą `jedzie(student, rower)`:

?- `jedzie(student, rower) = X`.

- ♦ Liczby całkowite i atomy zawsze są sobie równe.

Oto kilka przykładów:

<code>policjant = policjant</code>	uzgodniony
<code>papier = kredka</code>	zawodzi
<code>1066 = 1066</code>	uzgodniony
<code>1206 = 1583</code>	zawodzi

- ♦ Struktury są sobie równe, jeśli mają taki sam funktor oraz taką samą liczbę składników, zaś odpowiadające sobie składniki są sobie równe. Na przykład poniższe zapytanie nie zawiedzie, poza tym zmienna X zostanie ukonkretniona wartością `rower`:

```
jedzie(student, rower) = jedzie(student, X).
```

Zwróćmy uwagę, że powyższe zapytanie jest zapytaniem o predykat `=`, a nie `jedzie`.

Struktury mogą być dowolnie w sobie zagnieżdżane; jeśli takie zagnieżdżone struktury są porównywane, porównywanie może trwać dłużej, gdyż więcej struktur trzeba sprawdzać. Następujący cel:

```
a(b, C, d(e, f, g(h, i, j))) = a(B, c, d, (E, f, g(H, i, j))).
```

zostanie uzgodniony, B zostanie ukonkretnione jako b, C jako c, E jako e, F jako f, H jako h, a J jako j. Co się stanie, jeśli będziemy chcieli porównać dwie nieukonkretnione zmienne? Jest to przypadek szczególny pierwszej z powyższych zasad. Cel zostanie uzgodniony, zaś obie zmienne zostaną *powiązane*, czyli kiedy jedna zostanie ukonkretniona jakimś termem, druga automatycznie zostanie ukonkretniona tym samym termem. Zmienne takie odnoszą się do tego samego, więc w następującej regule drugi argument zostanie ukonkretniony wartością pierwszego argumentu:

```
rowne(X, Y) :- X=Y.
```

Cel `X = Y` zostanie zawsze uzgodniony, jeśli którykolwiek z argumentów zostanie ukonkretniony. Prościej sposobem zapisania takiej reguły jest skorzystanie z faktu, że zmienna jest równa samej sobie i napisanie po prostu:

```
rowne(X, X).
```

Ćwiczenie 2.1. Określ, czy poniższe cele zostaną spełnione i, ewentualnie, które zmienne zostaną jak ukonkretnione:

```
pilotuje(A, londyn) = pilotuje(londyn, paryz)
punkt(X, Y, Z) = punkt(X1, Y1, Z1)
litera(C) = slowo(litera)
rzeczownik(alfa) = alfa
'student' = student
f(X, X) = f(a, b)
f(X, a(b, c)) = f(Z, a(Z, c))
```

Arytmetyka

Komputerów nader często używa się do wykonywania obliczeń. Operacje arytmetyczne przydają się do porównywania liczb i obliczania wyników wyrażeń. W tym podrzdziale pokażemy przykłady jednego i drugiego.

Najpierw przyjrzyjmy się porównywaniu liczb. Jeśli mamy dwie liczby, możemy stwierdzić, czy są sobie równe, czy któraś z nich jest większa bądź mniejsza od drugiej. Prolog zawiera predykaty wbudowane pozwalające porównywać liczby. Predykaty te wyliczają wartości termów, traktując je jako wyrażenia algebraiczne. Argumentami tych predykatów mogą być zmienne ukonkretnione liczbami całkowitymi lub liczby zapisane jako stałe, mogą być też ogólniejszymi wyrażeniami algebraicznymi. Tutaj

używać będziemy predykatów do porównywania liczb, ale potem będziemy też korzystać z ogólniejszych wyrażeń algebraicznych. Zauważmy, że omawiane operatory mogą być zapisywane jako infiksowe:

<code>X = Y</code>	<code>X i Y</code> są tą samą liczbą
<code>X \= Y</code>	<code>X i Y</code> są różnymi liczbami
<code>X < Y</code>	<code>X</code> jest mniejsze od <code>Y</code>
<code>X > Y</code>	<code>X</code> jest większe od <code>Y</code>
<code>X <= Y</code>	<code>X</code> jest mniejsze lub równe <code>Y</code>
<code>X >= Y</code>	<code>X</code> jest większe lub równe <code>Y</code>

Zwróć uwagę, że symbolu *mniejsze lub równe* nie zapisuje się jako `<=`, jak to ma miejsce w wielu językach programowania. Wynika to stąd, że w Prologu można używać atomu `<=`, przypominającego strzałkę do innych, własnych celów.

Operatory porównania są predykatami, więc można się spodziewać, że fakt Prologu można zapisać następująco:

```
2 > 3.
```

aby dopisać, że 2 jest większe od 3. Taki fakt jest całkiem poprawny formalnie, ale Prolog nie pozwala dodawać faktów wykorzystujących predykaty wbudowane. Dzięki temu unika się możliwości niespodziewanej zmiany predykatów wbudowanych. W rozdziale 6. opiszemy wszystkie predykaty wbudowane, łącznie z tymi już wspomnianymi.

Załóżmy teraz, że mamy bazę danych książek Walii z IX i X wieku. Predykat `wlada` jest skonstruowany tak, że cel `wlada(X, Y, Z)` jest uzgadniany, jeśli książę o imieniu `X` rządził od roku `Y` do `Z`. Lista faktów w takiej bazie danych może mieć postać:

```
wlada(rhodri, 844, 878).
wlada(anarawd, 878, 916).
wlada(hywel_dda, 916, 950).
wlada(lago_ap_idwal, 950, 979).
wlada(hywel_ap_teuaf, 979, 985).
wlada(cadwallon, 985, 986).
wlada(maredudd, 986, 999).
```

Załóżmy teraz, że chcemy wiedzieć, kto władał Walią w danym roku. Możemy zdefiniować regułę, której argumentami będą imię władcy i rok, i która przeszuka bazę danych porównując dany rok z datami panowania. Zdefiniujemy predykat `ksiazka(X, Y)`, który nie zawiedzie, jeśli książę `X` zasiadał na tronie w roku `Y`:

```
X byl księciem w roku Y, jeśli:
X panował między rokiem A i B oraz
Y mieści się między A a B z tymi latami włącznie.
```

Pierwszy cel będzie uzgadniany na podstawie faktów `wlada` z powyższej bazy danych. Drugi będzie spełniony, jeśli `Y` będzie równe `A`, `B` lub będzie leżało między `A` a `B`. Warunki takie możemy sprawdzić stosując wyrażenia `A <= Y <= B` oraz `Y = A` lub `Y = B`. W formie Prologu reguła ta uzyska postać:

tel. (012) 637-66-77, tel./fax (012) 637-33-47
 Wójsko Republiki MLN ul. 55 z dnia 11.05.1995r.
 Kancel. GUS Główny 1540/1115-12067-27005-00
 NIP 677-17-58-169 REGON 350814846

```
ksiaze(X,Y) :-
    wlada(X,A,B),
    Y >= A,
    Y <= B.
```

Oto przykładowe zapytania wraz z odpowiedziami Prologu:

```
?- ksiaze(cadwallon,986).
yes
?- ksiaze(rhodri,1979).
no
?- ksiaze(X,900).
X=anarawd
yes
?- ksiaze(X,979).
X=lago_ap_idwal ;
X=hywel_ap_ieuaf
yes
```

Warto zwrócić uwagę na użycie zmiennych w ostatnich przykładach. Upewnij się, że wiesz, jak mechanizm przeszukiwania Prologu pozwala na takie pytania odpowiadać.

Arytmetyki można użyć także do wykonywania obliczeń. Jeśli na przykład znamy liczbę ludności i obszar kraju, możemy wyliczyć jego gęstość zaludnienia. Użyjemy małej bazy danych zawierającej informacje o populacji i obszarze różnych krajów w roku 1976. Predykat `lud` będzie zawierał populację poszczególnych państw. Liczby ludności są dość duże, więc będziemy je zapisywać w milionach: `lud(X,Y)` oznacza, że kraj `X` ma `Y` milionów mieszkańców. Predykat `obszar` określa obszar kraju w milionach mil kwadratowych. Podane liczby nie są zbyt ścisłe, ale wystarczą nam do zaprezentowania arytmetyki Prologu:

```
lud(usa,203).
lud(indie,548).
lud(chiny,800).
lud(brazylia,108).
```

```
obszar(usa,3).
obszar(indie,1).
obszar(chiny,4).
obszar(brazylia,3).
```

Aby teraz określić gęstość zaludnienia jakiegoś kraju, musimy użyć reguły, która podzieli ludność danego kraju przez jego obszar. Użyjemy predykatu `gestosc`, takiego, że cel `gestosc(X,Y)` nie zawiedzie, jeśli kraj `X` będzie miał gęstość zaludnienia wynoszącą `Y`. Odpowiednia reguła Prologu wygląda tak:

```
gestosc(X,Y) :-
    lud(X,P),
    obszar(X,A),
    Y is P/A.
```

Regułę tę interpretujemy następująco:

gęstość zaludnienia kraju `X` wynosi `Y`, jeśli:
kraj `X` ma `P` ludności oraz
obszar `X` to `A` oraz
wyliczamy `Y` jako iloraz `P` i `A`.

Operator infiksowy `is` jest nowością. Jego prawy argument jest termem, który ma być zinterpretowany jako wyrażenie arytmetyczne. Aby uzgodnić predykat `is`, Prolog najpierw wylicza wyrażenie arytmetyczne, zaś wynik jest dopasowywany do lewego argumentu. W powyższym przykładzie po lewej znajduje się nieukonkretniona zmienna `Y`, więc zmienna ta otrzymuje wartość wyliczonego wyrażenia. Wynika stąd, że wszystkie wartości zmiennych po prawej stronie `is` muszą być znane.

Predykatu `is` używa się wówczas, gdy wyliczone ma być wyrażenie arytmetyczne. Pamiętajmy, że `P/A` to zwykła struktura Prologu, taka sama jak na przykład `autor(emily,bronte)`. Jeśli strukturę tę interpretujemy jako wyrażenie arytmetyczne, jest do niej przykładana specjalna operacja wykonująca potrzebne obliczenia. Proces ten nazywamy *ewaluacją* wyrażenia arytmetycznego. Nie wszystkie struktury mogą być ewaluowane jako wyrażenia arytmetyczne — oczywiście jest na przykład, że nie można tak potraktować struktury z funktorem `autor`, gdyż `autor` nie jest operatorem arytmetycznym.

Wróćmy teraz do naszego przykładu z gęstością zaludnienia. Możemy zadać następujące zapytania:

```
?- gestosc(chiny,X).
X=200
yes
?- gestosc(turcja,X).
no
```

W przypadku pierwszego zapytania Prolog odpowiedział `X=200`, co oznacza, że w Chinach na jedną milę kwadratową przypada średnio 200 osób. Drugie zapytanie zawiodło, gdyż w bazie danych nie ma informacji o Turcji.

To, jakich operatorów arytmetycznych można użyć po prawej stronie operatora `is`, zależy od używanego systemu. Wszystkie implementacje Prologu obsługują:

<code>X + Y</code>	suma <code>X</code> i <code>Y</code>
<code>X - Y</code>	różnica <code>X</code> i <code>Y</code>
<code>X * Y</code>	iloczyn <code>X</code> i <code>Y</code>
<code>X / Y</code>	iloraz <code>X</code> i <code>Y</code>
<code>X // Y</code>	całkowity iloraz <code>X</code> przez <code>Y</code>
<code>X mod Y</code>	reszta z dzielenia <code>X</code> przez <code>Y</code>

Ta lista wraz z podaną wcześniej listą operatorów porównania powinna wyjaśnić niemalże wszystko, co dotyczy prostych problemów arytmetycznych.

Spełnianie celów — podsumowanie

Prolog realizuje zadania w odpowiedzi na *zapytania* programisty. Zapytanie to *koniunkcja* celów, które mają być *spełnione*. W Prologu do spełniania celów używa się znanych *klauzul*. Fakt może spowodować, że cel będzie spełniony od razu, zaś reguła pozwala zadanie spełniać stopniowo, przez spełnianie koniunkcji jej *podcelów*. Klauzuli można użyć jedynie wtedy, gdy *pasuje* do spełnianego celu. Jeśli celu nie można spełnić, uruchamiane jest *nawracanie*. Nawracanie polega na przeglądaniu dotąd zrealizowanego programu i próbie *ponownego spełnienia* celów przez znalezienie ich alternatywnych rozwiązań. Jeśli nie wystarcza nam odpowiedź udzielona przez Prolog, możemy sami wymusić nawracanie i kiedy Prolog poda odpowiedź, wpisać średnik. W poniższym punkcie pokażemy na diagramach, jak i kiedy Prolog próbuje spełniać cele koniunkcji.

Udane spełnienie koniunkcji celów

Prolog stara się spełnić cele koniunkcji niezależnie od tego czy występują one w treści reguły, czy w zapytaniu. Odbyna się to w kolejności ich występowania, od lewej do prawej. Oznacza to, że Prolog nie stara się spełnić celu, póki nie zostanie spełniony jego lewy sąsiad. Kiedy cel zostanie spełniony, Prolog stara się spełnić jego prawego sąsiada. Przyjrzyjmy się następującemu prostemu programowi, który opisuje związki rodzinne:

```
kobieta(maria).

rodzic(C,M,F) :- matka(C,M), ojciec(C,F).

matka(jan,anna).
matka(maria,anna).

ojciec(maria,ferdynand).
ojciec(jan,ferdynand).
```

Przyjrzyjmy się kolejnym zdarzeniom pozwalającym odpowiedzieć na zapytanie:

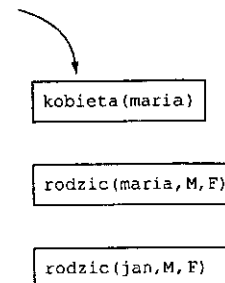
```
?- kobieta(maria), rodzic(maria,M,F), rodzic(jan,M,F).
```

Zadaniem jest upewnienie się, czy maria jest siostrą jana. W tym celu Prolog stara się spełnić listę celów z rysunku 2.1.

Cele zapisujemy jako prostokąty ułożone od góry do dołu. Strzałka zaczynająca się od góry strony wskazuje, które cele zostały już spełnione. Prostokąty znajdujące się pod strzałką to cele, które jeszcze nie były uzgadniane. W miarę wykonywania programu strzałka przesuwa się w górę i w dół, tak jak Prolog stara się spełniać kolejne cele — proces ten nazywamy *przeptywem spełniania*. W naszym przykładzie strzałka najpierw pojawia się na górze strony, jak pokazano na rysunku. Potem jest przesuwana w dół, przez kolejne cele, w miarę ich spełniania. Na rysunku 2.2. pokazano sytuację, która będzie miała miejsce pod koniec wykonywania programu.

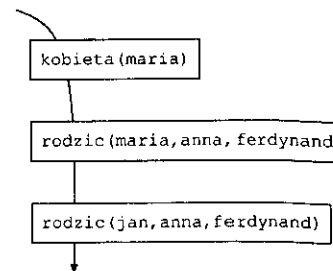
Rysunek 2.1.

Sekwencje jeszcze nie spełnionych celów



Rysunek 2.2.

Sekwencje spełnionych podcelów, zmienne zostały już ukonkretnione



Zwróćmy uwagę, że w miejsce zmiennych M i F używane są już ich wartości. Diagram pokazuje z grubszą co się dzieje, ale nie pokazuje *jak* trzy cele są spełniane. Pokazanie tego wymaga umieszczenia w prostokątach dodatkowych informacji. Zajmijmy się szczegółowo tym, jak spełniany jest drugi cel. Spełnienie tego celu obejmuje przeszukiwanie bazy danych w celu znalezienia *unifikowanej* klauzuli, następnie oznaczenie tego miejsca w bazie danych i spełnienie ewentualnych podcelów.

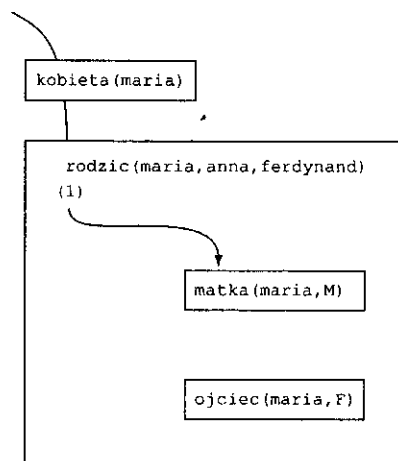
To, jak drugi cel jest spełniany, możemy pokazać rysując odpowiadający mu prostokąt wraz z podcelami. Wybrana klauzula jest zaznaczana za pomocą ujętej w nawias liczby, w tym wypadku (1). Liczba ta mówi, która klauzula ze *zbioru klauzul odpowiedniego predykatu* została wybrana, więc numer 1 oznacza pierwszą klauzulę wybranego predykatu. Informacje te wystarczają do umieszczenia znacznika w bazie danych. Podcele zaznaczamy jako małe prostokąty wewnątrz prostokąta celu. Kiedy wybrana jest klauzula rodzic, sytuacja wygląda tak, jak na rysunku 2.3.

Strzałka weszła już do prostokąta rodzic i przeszła przez nawias oznaczający wybraną klauzulę. Klauzula spowodowała dodanie dwóch podcelów, matka i ojciec. W tej chwili strzałka musi przejść przez mniejsze prostokąty i wyjść z prostokąta rodzic.

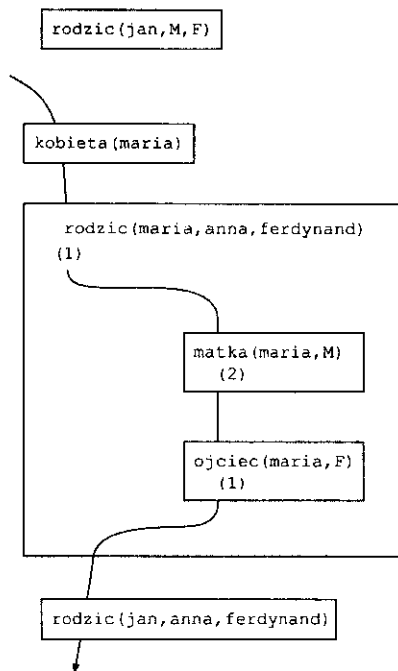
Kiedy strzałka przechodzi przez mniejsze prostokąty, wykonywane są takie same czynności związane z wybieraniem klauzuli i spełnianiem jej podcelów. W naszym przykładzie obydwa podcele są spełniane dzięki znalezieniu faktów w bazie danych, co powoduje ukonkretnienie zmiennych M i F. Na rysunku 2.4. pokazano bardziej szczegółowo sytuację po zrealizowaniu zapytania.

Rysunek 2.3.

Liczba (1) wskazuje, że wybrana została pierwsza klauzula predykatu. Podcele pokazano jako mniejsze prostokąty umieszczone wewnątrz prostokąta celu głównego

**Rysunek 2.4.**

Zapytanie, które się powiodło



Aby wykład był kompletny, powinniśmy pokazać szczegóły spełniania celów `kobieta(maria)` i `rodzic(jan,anna,ferdynand)`, ale nie ma na to tutaj miejsca. Przykład ten pokazuje, jak Prolog stara się spełniać cele w przypadku uzgodnienia koniunkcji celów. Strzałka przechodzi w dół strony przez kolejne prostokąty. Przy wchodzeniu do jakiegokolwiek prostokąta, wybierana jest klauzula i oznaczane jej miejsce w bazie danych. Jeśli klauzula pasuje do celu i jest faktem, strzałka może wyjść z prostokąta (jak w przypadku celów `matka` i `ojciec`). Jeśli klauzula pasuje do celu i jest regułą, tworzone są nowe prostokąty podcelów, przez które strzałka musi przejść przed opuszczeniem prostokąta zewnętrznego.

Cele i nawracanie

Kiedy któryś cel zawodzi (z powodu sprawdzania alternatywnej klauzuli już raz uzgadnianego celu lub użycia przez użytkownika średnika), „przepływ spełniania” jest przekazywany z powrotem drogą, którą do danego miejsca doszedł. Następuje powrót przez prostokąty, które już były raz opuszczone, w celu znalezienia alternatywnych rozwiązań. Kiedy strzałka wraca do miejsca wyboru klauzuli (oznaczonego liczbą w nawiasie), Prolog stara się do danego celu znaleźć klauzulę alternatywną. Najpierw wycofywane jest powiązanie wszystkich zmiennych ukonkretnionych w ramach spełniania danego celu, następnie w bazie odszukiwany jest znacznik. Jeśli znalezione zostanie inne możliwe dopasowanie, oznaczane jest nowe miejsce i przetwarzanie jest kontynuowane tak, jak zostało to opisane wcześniej, w punkcie „Udane spełnienie koniunkcji celów”.

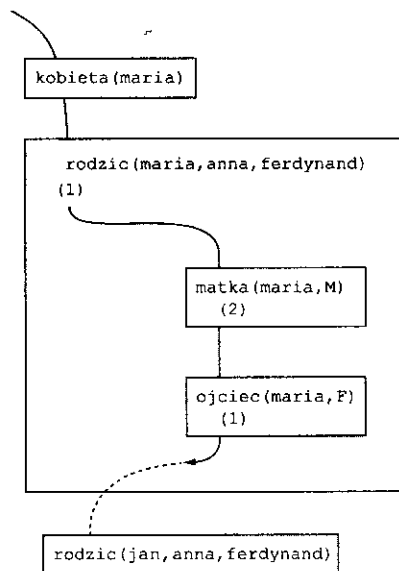
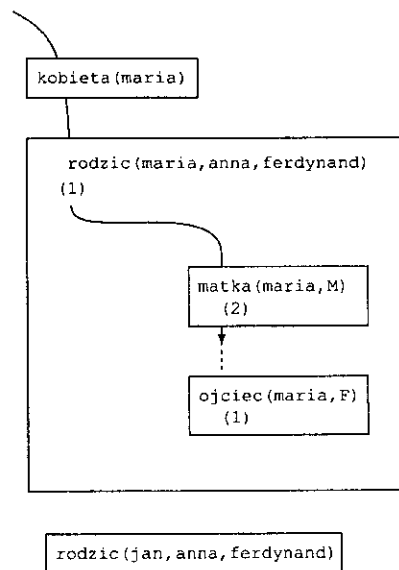
Należy zwrócić uwagę na to, że wszystkie cele „pod” danym celem (nawet jeśli już były badane), będą wykonywane zawsze od początku. Prolog będzie się starał je spełnić, nie będzie to jednak próba spełnienia ponownego, alternatywną drogą. Jeśli nie zostanie znaleziona żadna inna możliwość uzgodnienia, strzałka wycofuje się do poprzedniego znacznika w bazie danych.

Gdyby w naszym przykładzie zawiódł cel `rodzic(jan,anna,ferdynand)`, strzałka wróciłaby w górę z prostokąta `rodzic(jan,anna,ferdynand)` i ponownie weszłaby od dołu do `rodzic(maria,anna,ferdynand)`, co oznaczałoby próbę ponownego spełnienia tego celu. Następnie nastąpiłoby dalsze cofnięcie, ponowne wejście do prostokąta `ojciec(maria,ferdynand)` i miałyby miejsce próba ponownego spełnienia tego celu, co pokazano na rysunku 2.5.

Teraz konieczne jest cofnięcie się jeszcze dalej. Strzałka musi dojść do miejsca, gdzie wybierana była klauzula celu `ojciec`. Przede wszystkim wszystkie zmienne, które były ukonkretnione, są z powrotem zwalniane; dotyczy to zmiennej `f`. Prolog przeszukuje bazę danych zaczynając od pierwszej klauzuli `ojciec` (ona jest zaznaczona) i stara się dla tego celu znaleźć alternatywne dopasowanie. Jeśli Maria ma tylko jednego ojca, nie uda się to, więc strzałka cofnie się dalej w górę, poza prostokąt `ojciec(maria,f)` (ten cel zawiódł) i cofnie się do prostokąta `matka(maria,anna)`, co oznacza próbę jego ponownego dopasowania. Sytuację tę pokazano na rysunku 2.6.

Rysunek 2.5.

Kiedy cel zawodzi

**Rysunek 2.6.**Próba ponownego
uzgodnienia celu

Z podanych przykładów łatwo wywnioskować, jak cele są ponownie spełniane podczas nawracania. Kiedy cel zawodzi, strzałka cofa się w górę za prostokąt zawodzącego celu i wraca do prostokąta celu nadrzędnego. Cofanie się trwa tak długo, aż zostanie znaleziony znacznik w bazie danych. Wszystkie zmienne, które zwolniono w trakcie cofania się, znów są nieukonkretnione. Prolog szuka w bazie danych klauzul z za znacznika. Jeśli taka klauzula zostanie znaleziona, znacznik jest umieszczany w nowym miejscu, ponownie tworzone są prostokąty podcelów i strzałka ponownie podąża w dół. Jeśli żadna inna klauzula nie zostanie znaleziona, strzałka cofa się w górę, do następnego znacznika.

Unifikacja

Poniżej opisano reguły decydujące czy cel można zunifikować z głową klauzuli. W przypadku użycia klauzuli pierwotnie wszystkie zmienne są nieukonkretnione.

- ♦ Nieukonkretniona zmienna unifikuje się z dowolnym obiektem. W wyniku tego zmienna będzie zastąpiona przez ten obiekt.
- ♦ Jeśli zmienna nie jest nieukonkretniona, liczba całkowita lub atom unifikują się jedynie z samymi sobą.
- ♦ Jeśli nie zachodzi żaden z powyższych przypadków, struktura unifikuje się z inną strukturą o takim samym funktorze i takiej samej liczbie argumentów, zaś odpowiadające sobie argumenty też muszą się unifikować.

Przypadkiem wartym osobnego wspomnienia jest unifikacja dwóch nieukonkretnionych zmiennych. Wówczas zmienne stają się ze sobą *powiązane*; będą miały taką samą wartość w momencie ukonkretnienia którejkolwiek z nich. Skojarzenie unifikacji z porównywaniem nie jest przypadkiem, predykat = stara się, aby jego argumenty były równe, unifikując je ze sobą. Teraz możemy zebrać razem wszystko, co mówiliśmy o operatorach, arytmetyce i unifikacji. Załóżmy, że w bazie danych mamy następujące zapisy:

```
suma(5).
suma(3).
suma(X+Y).
```

Niech dane będzie zapytanie:

```
?- suma(2+3).
```

Który z faktów podanych wyżej da się zunifikować z takim zapytaniem? Jeśli zakładasz, że pierwszy, powinieneś wrócić i przeczytać o strukturach i operatorach. W zapytaniu argumentem struktury `suma` jest komponent mający jako funktor znak plus i jako argumenty 2 i 3. Pokazany cel unifikuje się z faktem trzecim, zmienna `X` zostanie ukonkretniona wartością 2, zaś zmienna `Y` — wartością 3. Gdybyśmy faktycznie chcieli wyliczyć sumę, musielibyśmy użyć operatora `is` pisząc:

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

Moglibyśmy też w ramach ćwiczenia stworzyć predykat `add` (dodaj) wiążący dwie liczby całkowite z ich sumą:

```
dodaj(X,Y,Z) :- Z is X+Y.
```

Z samej definicji, X i Y muszą być ukonkretnione.

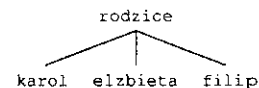
Rozdział 3.

Korzystanie ze struktur danych

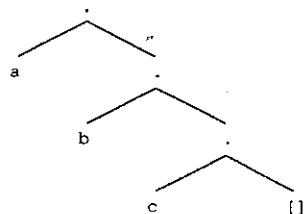
Słowo „rekurencja” dla wielu osób jest słowem nieznanym, być może kojarzącym się niejasno z matematyką. Teraz jednak słowo to nabrało całkiem nowej treści — oznacza potężną technikę programowania nienumerycznego. Rekurencji używa się w dwojaki sposób. Można stosować ją do opisywania struktur, których składnikami są inne struktury lub do zapisywania programów, które muszą spełnić kopie własnych celów, zanim skończą się wykonywać. Osoby początkujące często traktują rekurencję podejrzliwie, zastanawiając się, jak jakaś relacja może definiować samą siebie. W Prologu rekurencja jest naturalną metodą patrzenia na struktury danych i programy. Mamy nadzieję, że po przeczytaniu tego rozdziału dla naszych czytelników rekurencja stanie się oczywistą i wygodną metodą rozwiązywania problemów.

Struktury a drzewa

Zwykle łatwiej jest zrozumieć budowę złożonej struktury, gdy zapisze się ją w formie *drzewa*, którego funktor jest węzłem, zaś składniki struktury gałęziami. Każda gałąź może wskazywać inną strukturę, wobec czego w strukturze możemy zagnieżdżać inne struktury. Diagram drzewa zwykle rysuje się z węzłem głównym (korzeniem) na górze i gałęziami skierowanymi w dół. Oto przykład drzewa odpowiadającego strukturze `rodzice(karol,elzbieta,filip)`:



Strukturę `a+b*c` (czyli `+(a,*(b,c))`) można zapisać jako:



Niektórzy zapisują drzewiasty diagram listy od lewej do prawej, gałęzie danych skierowane są w dół. Powyższa lista zapisana w takiej formie „winnej latorośli” będzie miała postać:

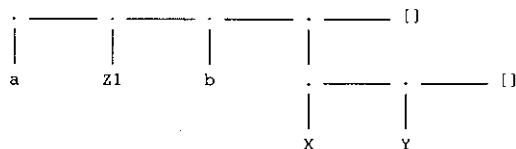


W takim diagramie głowa funktora „kropka” skierowana jest w dół, ogon przedłużany jest w prawo. Koniec listy oznaczony jest symbolem pustej listy. Podstawową zaletą takiego „winnego” diagramu list jest możliwość zapisania całej kartki. Diagram ten bywa przydatny, kiedy trzeba listę zapisać na papierze w celu obejrzenia jej struktury, ale diagramy takie nie są przydatne do zapisywania list w Prologu. Zapis z kropkami jest bardzo niewygodny w przypadku złożonych list, więc stosuje się inną postać zapisu. Polega to na umieszczaniu elementów listy w nawiasach kwadratowych i rozdzielaniu poszczególnych elementów tej listy przecinkami. Na przykład pokazane wcześniej listy zapisuje się jako `[a]` i `[a,b,c]`.

Wygodne jest użycie jako elementów list innych list i zmiennych. Oto przykłady poprawnych list prologowych:

```
[]
[większość,mezczyzn,[lubi,wedkować]]
[a,v1,b,[X,Y]]
```

Zmienne w liście traktowane są tak samo, jak wszystkie inne zmienne w dowolnej strukturze. Można je ukonkretniać w dowolnej chwili, więc staranne dobranie ich położenia pozwoli wstawić do listy „dziury”, które będą mogły być później wypełnione danymi. Aby pokazać sposób użycia list jako elementów listy, przedstawimy „winny diagram” ostatniej z wymienionych wyżej list:



Łatwo zauważyć na tym diagramie, że każdy „poziom” odpowiada liście mającej pewną liczbę elementów: na pierwszym poziomie są cztery elementy, jeden z nich jest listą. Drugi poziom ma dwa elementy i cały ten poziom jest czwartym elementem poziomu wyższego.

Listy przetwarza się poprzez ich podział na głowę i ogon. Głowa listy to pierwszy argument funktora kropka (.). Warto zwrócić uwagę na zbieżność nazwy „głowa” reguły i „głowa” listy. Są to dwie różne rzeczy i choć zdarza się ogólnie powiedzieć o nich „głowy”, to na podstawie kontekstu nietrudno odgadnąć, o którą głowę chodzi. Ogon listy to drugi argument funktora kropka (.). Kiedy lista zapisywana jest w nawiasach kwadratowych, jej głowa jest pierwszym elementem, zaś ogon to wszystko, co pozostaje. Oto zestaw list z zaznaczoną głową i ogonem:

Tabela 3.1. Przykładowe listy, ich głowy i ogony

Lista	Głowa	Ogon
<code>[a,b,c]</code>	<code>a</code>	<code>[b,c]</code>
<code>[]</code>	(brak)	(brak)
<code>[[bury,kot],mruczy]</code>	<code>[bury,kot]</code>	<code>[mruczy]</code>
<code>[bury,[kot,mruczy]]</code>	<code>bury</code>	<code>[[kot,mruczy]]</code>
<code>[bury,[kot,mruczy],cicho]</code>	<code>bury</code>	<code>[[kot,mruczy],cicho]</code>
<code>[X+Y,x+y]</code>	<code>X+Y</code>	<code>[x+y]</code>

Zauważmy, że lista pusta nie ma ani głowy, ani ogona. W ostatnim przykładzie operatora `+` użyto jako funktora struktur `+(X,Y)` i `+(x,y)`.

Z uwagi na to, że typową operacją związaną z listami jest dzielenie ich na głowę i ogon, istnieje specjalny zapis „listy z głową `X` i ogonem `Y`”: `[X|Y]`, gdzie `X` i `Y` rozdzielone są pionową kreską (`|`). Taki wzorec ukonkretnia `X` głową listy, zaś `Y` ogonem listy, na przykład:

```
p([1,2,3]).
p([bury,kot,mruczy,[sobie,pod,nosem]]).
?- p([X|Y]).
X=1 Y=[2,3] ;
X=bury Y=[kot,mruczy,[sobie,pod,nosem]]
?- p([_,_,_,_|X]).
X=[pod,nosem]
```

Więcej przykładów składni list wraz z różnymi przypadkami unifikacji pokazano poniżej; staraliśmy się zunifikować ze sobą listy parami i otrzymaliśmy wyniki, jak w tabeli 3.2.

Z ostatniego przykładu wynika, że korzystając ze składni list, możemy tworzyć struktury do list podobne, ale nie mające na końcu symbolu pustej listy. Struktura `[gniady|rumak]` oznacza strukturę mającą głowę `gniady` i ogon `rumak`. Stała `rumak` nie jest listą ani pustą listą i jak potem się przekonamy, korzystając z takich struktur w ogonie listy, musimy zachować szczególną ostrożność.

Tabela 3.2. Pary list i wynik ich dopasowania; jeśli listy uda się dopasować, pokazywane są wartości zmiennych; w jednym wypadku dopasowanie jest niemożliwe

Lista 1	Lista 2	Wartości zmiennych
[X,Y,Z]	[jan,lubi,ryby]	X = jan Y = lubi Z = ryby
[kot]	[X Y]	X = kot Y = []
[X,Y Z]	[maria,lubi,wino]	X = maria Y = lubi Z = [wino]
[[tu,Y] Z]	[[X,jest,[czarny,pies]]]	X = tu Y = jest Z = [[czarny,pies]]
[srebrne T]	[srebrne,wrota]	T = [wrota]
[dzielny,rumak]	[rumak,X]	(brak)
[czarny,Q]	[P,rumak]	P = czarny Q = rumak

Przeszukiwanie rekurencyjne

Często konieczne jest przeszukiwanie struktur Prologu w celu znalezienia pewnych informacji. Kiedy struktury mogą mieć inne struktury jako argumenty, konieczne jest *szukanie rekurencyjne*.

Załóżmy na przykład, że mamy listę imion koni pochodzących od Coriandera, zwycięzcy wyścigów w Wielkiej Brytanii w roku 1927:

```
[curragh_tip, music_star, park_mill, portland]
```

Teraz przyjmijmy, że chcemy się dowiedzieć czy dany koń znajduje się na liście. W Prologu najpierw sprawdzamy czy interesujący koń jest zapisany w głowie listy. Jeśli tak, mamy już odpowiedź. Jeśli nie, sprawdzamy czy koń występuje w ogonie listy. Oznacza to, że za każdym razem sprawdzamy głowę *ogona* listy, aż dojdziemy do listy pustej — wtedy nasze poszukiwanie kończy się negatywnie.

Aby powyższy algorytm zapisać w Prologu, musimy najpierw stwierdzić, jaki jest związek obiektu i listy. Związkiem tym jest przynależność dobrze znana z życia codziennego: ludzie mogą przynależeć do klubu i tak dalej. Napiszemy predykat `member1`, taki, aby cel `member(X,Y)` był spełniony wtedy, gdy `X` jest termem należącym do listy `Y`. Musimy sprawdzić dwa warunki. Po pierwsze, `X` jest elementem `Y`, jeśli `X` jest

taki sam, jak głowa `Y`. Wprawdzie można byłoby podać jako podcel `X=Y`, ale prościej będzie skorzystać po prostu z unifikacji zmiennych:

```
member(X,[X,_]).
```

Zapis ten interpretujemy: „`X` należy do listy, jeśli jest jej głową”. Ogon listy oznaczamy zmienną anonimową, gdyż nie będziemy go już potrzebować. Regułę tę można byłoby też zapisać jako:

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

Nie powinno być problemów ze zrozumieniem, dlaczego możemy skorzystać z zapisu skrótego używając dwukrotnie tej samej zmiennej, `X`.

Druga i ostatnia reguła mówi, że `X` jest elementem listy, jeśli należy do ogona tej listy — oznaczmy ten ogon przez `Y`. A jak lepiej można sprawdzić przynależność elementu listy do ogona niż używając właśnie predykatu `member`?! Na tym właśnie polega istota rekurencji. Zatem zapis

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

oznacza, że „`X` należy do listy, jeśli należy do jej ogona”. Tym razem zmienną anonimową oznaczyliśmy głowę listy, gdyż głowa ta nie jest nam do niczego potrzebna. Obie podane reguły tworzą cały predykat, dzięki któremu Prolog „wie”, jak ma szukać elementu na liście.

Najważniejsze, o czym trzeba pamiętać w przypadku predykatu rekurencyjnego, to sprawdzenie *warunków granicznych i przypadku rekurencyjnego*. W przypadku predykatu `member` mamy dwa warunki graniczne: kiedy obiekt jest na liście lub kiedy go tam nie ma. Jeśli obiekt jest na liście, odpowiednią obsługę zapewnia pierwsza klauzula, która powoduje zatrzymanie wykonywania predykatu, gdy pierwszy argument pasuje do głowy argumentu drugiego. Drugi warunek dochodzi do głosu, jeśli drugi argument predykatu `member` jest listą pustą.

Jak możemy zapewnić, że warunki graniczne w ogóle kiedykolwiek zajdą? Musimy przyrzec się przypadkowi rekurencyjnemu, tutaj drugiej klauzuli. Zauważmy, że za każdym razem, kiedy sprawdzamy czy zachodzi `member`, wywołujemy cel otrzymując *krótszą* listę. Ogon listy zawsze jest krótszy od listy pierwotnej, więc może zajść jedno z dwojga: albo uruchomiona zostanie pierwsza klauzula `member`, albo jako drugi argument `member` przekazana zostanie lista o długości zero, czyli lista pusta. Kiedy zajdzie którakolwiek z tych dwóch sytuacji, wywołwanie celów `member` kończy się. Pierwszy warunek końcowy uruchamia fakt nie powodujący prób uzgadniania jakichkolwiek dalszych podcelów. Drugi warunek nie jest w ogóle rozpoznawany, więc jeśli zajdzie, `member` zawiedzie:

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
?- member(d,[a,b,c,d,e,f,g]).
yes
?- member(2,[3,a,4,f]).
no
```

¹ Korzystamy z angielskiego terminu, gdyż predykat sprawdzający przynależność elementu do listy tak właśnie zwyczajowo jest nazywany. Podobnie będziemy postępować dalej: jeśli jakieś nazwy są ogólnie przyjęte w brzmieniu angielskim, takimi będziemy je pozostawiać — *przyp. tłum.*

Załóżmy, że zadaliśmy zapytanie:

```
?- member(clygate.[curragh_tip,music_star,park_mill,portland]).
```

Wykonana zostanie druga klauzula member, gdyż clygate nie pasuje do curragh_tip. Zmienna Y jest ukonkretniana wartością [music_star, park_mill, portland], zaś następny cel to sprawdzenie, czy clygate jest elementem podanej listy. Znow uruchamiana jest druga klauzula, znow pobierany jest ogon i wywoływany jest cel member(clygate,[park_mill,portland]). Proces jest kontynuowany, w następnym wywołaniu mamy X o wartości clygate i Y o wartości [portland]. Jeszcze raz wykonywana jest druga klauzula, wartością Y staje się ogon listy [portland], czyli lista pusta, więc cel przybiera postać member(clygate, []). Do takiego celu nie pasuje żadna reguła z bazy danych, więc cel zawodzi i całe zapytanie też.

Bardzo ważną rzeczą jest fakt, że za każdym razem, kiedy member korzysta ze swojej drugiej klauzuli, Prolog traktuje kolejny podcel jako kolejną kopię member. Dzięki temu wartości zmiennych używane w poszczególnych wywołaniach nie mylą się ze sobą.

Predykat member jest bardzo przydatny i będziemy go wielokrotnie używać w dalszej części książki. Jest ważny jako najmniejszy przydatny praktycznie predykat rekurencyjny: definicja member zawiera cele, które spełnić może jedynie sam member. Definicje rekurencyjne w programach prologowych występują nader często i od podanej tutaj definicji zaledwie się nie różnią. Trzeba jednak uważać, aby nie stworzyć definicji cyklicznej, jak poniższa:

```
rodzic(X,Y) :- dziecko(Y,X).
dziecko(A,B) :- rodzic(B,A).
```

W tym wypadku, aby spełnić predykat rodzic, odwołujemy się do predykatu dziecko jako do celu. Jednak w tym ostatnim jako podcel używany jest predykat rodzic. Nie trudno zauważyć, że jeśli zadamy zapytanie odwołujące się do predykatu rodzic lub dziecko, powstanie pętla, w której Prolog nie będzie mógł wywnioskować niczego nowego i pętla ta nigdy się nie skończy.

Kolejna istotna sprawa związana z rekurencją to to, czy rekurencja jest *lewostronna*. Z taką rekurencją mamy do czynienia, kiedy reguła powoduje wywołanie takiego samego celu, jak cel, który spowodował wywołanie tej reguły. Jeśli zatem zdefiniujemy:

```
osoba(X) :- osoba(Y), matka(X,Y).
osoba(adam).
```

i zadamy zapytanie

```
?- osoba(X).
```

Prolog najpierw użyje pierwszej reguły i wygeneruje podcel osoba(Y). Próbując cel ten spełnić, Prolog ponownie wygeneruje taki sam cel, i tak dalej, aż skończy się pamiętać. Gdyby istniała możliwość nawracania, zostałby znaleziony fakt opisujący Adama i generowane byłyby rozwiązania. Problem polega na tym, że aby doszło do nawrotu, któryś cel musi zawieść. W tym wypadku zadanie okazuje się być nieskończenie długie, więc nie ma możliwości aby cel został spełniony, ani aby zawiódł. Morał jest taki:

Nigdy nie zakładaj, że Prolog znajdzie wszystkie fakty i reguły tylko dlatego, że je podałeś. Pisząc programy w Prologu trzeba pamiętać o tym, jak Prolog przeszukuje bazę danych i jak są ukonkretniane zmienne przy używaniu reguł.

W tym przykładzie najprostszym rozwiązaniem byłoby umieszczenie faktu przed regułą, a nie za nią.

```
osoba(adam).
osoba(X) :- osoba(Y), matka(X,Y).
```

Ogólnie należy przyjąć, że fakty powinny być umieszczane przed regułami, kiedy tylko jest to możliwe. Czasami ułożenie reguł w określonej kolejności wystarczy, aby spełnione zostały pewne cele, ale jednocześnie może to uniemożliwić spełnienie innych celów. Przyjrzyjmy się poniższej definicji predykatu jest_lista. Jej cel jest_lista(X) jest uzgadniany, jeśli X jest listą, której ogon ostatniego elementu jest listą pustą:

```
jest_lista([A|B]) :- jest_lista(B).
jest_lista([]).
```

Możemy użyć tej definicji, aby odpowiedzieć na zapytanie

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

albo

```
?- jest_lista([]).
```

albo

```
?- jest_lista(f(1,2,3)).
```

Jeśli jednak zadamy zapytanie

```
?- jest_lista(X).
```

Program wpadnie w pętlę. Możemy stworzyć podobny do jest_lista predykat, który nie będzie już podatny na wpadanie w pętlę:

```
slaba_jest_lista([]).
slaba_jest_lista(_).
```

Tak wersja sprawdza początek listy, a nie sprawdza, czy ostatni ogon to lista pusta, []. Test ten nie jest tak silny, jak jest_lista, ale nie wpadnie w pętlę, jeśli argument będzie zmienną.

Odwzorowania

Jeśli mamy daną strukturę Prologu, często chcielibyśmy stworzyć nową strukturę podobną do starej, ale w jakiś sposób zmienioną. Przechodzimy przez kolejne składniki starej struktury tworząc jednocześnie składniki struktury nowej — taki proces nazywamy *odwzorowywaniem*.

Wyższa Szkoła
Zarządzania i Bankowości
ul. Armii Krajowej 4, 30-150 Kraków
tel. (012) 654-65-77, telefax: (012) 637-33-47
Wpis do Rejestru MER nr 55 z dnia 11.05.1996r.
Konto BUE O/Kraków 15401115-12087-27095-00

Przyjrzyjmy się programowi, który otrzymuje zdania i odpowiada na nie zdaniami podobnymi, ale zmodyfikowanymi. Dialog z tym programem może wyglądać następująco:

```
ty jestes komputerem
ja nie jestem komputerem
czy mowisz po francusku
nie mowie po niemiecku
```

Wprawdzie ten dialog jest wymuszony, ale ma sens; aby go wygenerować, wystarczy zrobić tylko kilka rzeczy:

1. Wczytać zdanie podane przez użytkownika.
2. Zamienić wszystkie formy drugiej osoby na osobę pierwszą.
3. Zmienić słowo czy na nie.
4. Zmienić francusku na niemiecku.
5. Zmienić ty na ja nie.

Jeśli będziemy dobierać odpowiednie zdania, otrzymamy sensowne, zmodyfikowane wypowiedzi. Jednak nie wszystkie zdania dadzą się tak przekształcić, na przykład:

```
czy wiesz, ze cie lubie
nie wiem, ze cie lubisz
```

Kiedy jednak napiszemy podstawową wersję programu, można będzie go potem rozszerzać, aby radził sobie także z innymi rodzajami zdań.

Program prologowy zmieniający jedno zdanie na inne można pisać w ten sposób: najpierw musimy określić związek zdania wyjściowego ze zdaniem pochodnym. Musimy wobec tego zdefiniować w Prologu predykat `zmien(X,Y)`, który umożliwi zmianę zdania `X` na zdanie `Y`. Wygodnym rozwiązaniem jest zapisywanie `X` i `Y` jako list, w których atomami będą słowa, więc zdanie przybierze postać:

```
[to, jest, jakies, zdanie]
```

Kiedy zdefiniujemy już `zmien`, będziemy mogli korzystać z wywołań:

```
?- zmien([czy,mowisz,po,francusku],X).
```

a Prolog odpowie:

```
X=[nie, mowie, po niemiecku].
```

Nie należy przejmować się nieuporządkowaniem zdań wejściowych i wyjściowych oraz ich wyglądem. W następnych rozdziałach powiemy, jak można poprawiać czytelność struktur, na razie interesuje nas jedynie przekształcenie jednej listy w inną.

Predykat `zmien` korzysta z list, więc przede wszystkim musi potrafić poradzić sobie z listą pustą. W takim wypadku listę pustą wymienimy na takąż pustą listę:

```
zmien([],[]).
```

Oznacza to, że „zmiana pustej listy daje listę pustą”. Jeśli taki sposób traktowania listy pustej nie jest jeszcze oczywisty, potem powinno się to wyjaśnić. Następnie musimy zająć się podstawowym zadaniem predykatu `zmien`:

1. Zmiana głowy listy na inne słowo i ustanowienie tego nowego słowa głową nowej listy.
2. Zmiana predykatem `zmien` ogona listy wejściowej i ustanowienie tego nowego ogona ogonem nowej listy.
3. Jeśli przetwarzanie listy się zakończyło, nie trzeba już nic robić, więc kończymy listę wynikową listą pustą, `[]`.

Zapiszmy to samo w sposób nieco bliższy notacji Prologu:

```
zmiana listy z głową G i ogonem O daje listę z głową X i ogonem Y, jeśli:
zamiana słowa G daje słowo X oraz
zamiana listy O daje listę Y.
```

Teraz musimy powiedzieć, co rozumiemy przez zamianę jednego słowa na inne. Wystarczy nam do tego baza danych faktów w postaci `podmien(X,Y)`, oznaczających zamianę słowa `X` na słowo `Y`. Na koniec potrzebujemy jeszcze ogólnej klauzuli, która obejmie wszystkie przypadki słów, dla których nie zdefiniowaliśmy odpowiedników. Jeśli nie jest jeszcze jasne, po co nam ta ostatnia klauzula, wyjaśni się to, kiedy zobaczymy działanie całego programu. Ostatnia klauzula będzie miała postać `podmien(X,X)`, co oznacza polecenie zamiany `X` na `X`. Oto baza danych:

```
podmien(czy,nie).
podmien(ty,[ja,nie]).
podmien(francusku,niemiecku).
podmien(jestes,jestem).
podmien(mowisz,mowie).
podmien(X,X). /* przypadek ogólny */
```

Warto zauważyć, że wyrażenie „ja nie” traktujemy jako listę, dzięki czemu można ją zapisać w pojedynczym argumencie.

Teraz nasz pseudokod powyżej możemy zamienić na kod Prologu, pamiętając, że `[A|B]` oznacza listę z głową `A` i ogonem `B`. Otrzymujemy zatem:

```
zmien([],[]).
zmien([G|O],[X|Y]) :- podmien(G,X), zmien(O,Y).
```

Pierwsza klauzula obsługuje listę pustą i jednocześnie obsługuje koniec listy. Dlaczego? Przyjrzyjmy się wywołaniu:

```
?- zmien([ty,jestes,komputerem],Z).
```

Takie zapytanie pasuje do głównej reguły `zmien`, więc zmienna `G` ma wartość `ty`, zaś `O` wartość `[jestes,komputerem]`. Następnie wywołany jest cel `podmien(ty,X)`; cel ten jest spełniony, zaś zmiennej `X` przypisywana jest wartość `[ja,nie]`. Kiedy `X` staje się głową listy wynikowej (cel `zmien`), pierwszymi słowami nowego zdania są `ja nie`. Następny cel to `zmien([jestes,komputerem],Y)`; używana jest ta sama reguła; słowo `jestes` zamieniane jest na `jestem` i kolejny cel to `zmien([komputerem],Y)`. Jako że nie

udaje się znaleźć faktu `podmien(komputerem,X)`, wykorzystywana jest ostatnia klauzula powodująca zamianę `komputerem` na `komputerem`. W tym wywołaniu ogon był już listą pustą, więc następne wywołanie, `zmien([],Y)`, pasuje do pierwszej klauzuli zmian. Wynikiem jest pusta lista, co kończy przekształcanie zdania (pamiętajmy, że lista kończy się listą pustą). Ostatecznie Prolog odpowiada:

```
Z = [[ja,nie],jestem,komputerem]
```

Wyrażenie `[ja,nie]` jest zwracane tak samo, jak zostało na listę wprowadzone.

Teraz powinno być jasne, dlaczego dodaliśmy fakt `zmien([],[])` i klauzulę ogólną `podmien(X,X)`. Tego typu fakty często występują w programach, w których konieczne jest sprawdzanie warunków końcowych. Powyższe wyjaśnienia wskazują, że z warunkiem końcowym mamy do czynienia wtedy, kiedy lista wejściowa jest pusta i kiedy przeszukane zostaną wszystkie fakty `podmien`. W obu wypadkach mają być wykonane pewne czynności: kiedy lista wejściowa staje się pusta, chcemy zakończyć listę wynikową, a kiedy przeszukano już wszystkie fakty `podmien`, chcemy zostawić słowo bez zmian.

Porównywanie rekurencyjne

Jak pokazywaliśmy w rozdziale 2., Prolog zawiera predykaty do porównywania liczb całkowitych, ale porównywanie struktur jest już znacznie bardziej skomplikowane, bo porównywać trzeba poszczególne składniki tych struktur. Wobec tego, że owe składniki też mogą być strukturami, porównywanie musi być wykonywane rekurencyjnie. Z sytuacją taką mamy do czynienia na przykład wtedy, gdy chcemy porównywać listy dowolnej długości.

Wyobraźmy sobie, że chcemy ustalić opłacalność eksploatacji różnych samochodów. W tym celu przemierzamy każdym z tych samochodów ustaloną trasę i mierzymy zużycie paliwa. Z każdym samochodem wiążemy listę liczb określających liczbę litrów paliwa zużytych na poszczególnych trasach. Aby porównać dwa samochody, musimy oczywiście zwracać uwagę na kolejność elementów listy, gdyż można porównywać tylko zużycie paliwa na tych samych trasach. Możemy założyć, że listy będą jednakowej długości, i że kolejność tras będzie stała. Zdefiniujemy predykat `zuzycie_paliwo`, taki, że cel `zuzycie_paliwo(C,R)` nie zawiedzie, gdy samochodem jest `C`, a listą tras `R`. Nasze dane będą więc miały postać:

```
zuzycie_paliwo(marnotrawny, [3.1, 10.4, 15.9, 10.3]).
zuzycie_paliwo(pozeracz, [3.2, 9.9, 13.0, 11.6]).
zuzycie_paliwo(pazerny, [2.8, 9.8, 13.1, 10.4]).
```

Mamy cztery trasy próbne. Gdybyśmy badali samochody na większej liczbie tras, listy byłyby dłuższe. Chcemy porównać samochody niezależnie od faktycznej długości list.

Przed wszystkim musimy zdecydować, kiedy traktować jedno zużycie paliwa jako lepsze od innego. To, ile paliwa się zużywa w czasie podróży, jest w pewnym stopniu

uwarunkowane przypadkowymi okolicznościami, więc przyjmiemy, że zużycie jest „nie gorsze” od innego zużycia, jeśli nie przekracza o więcej niż 5% średniej obu porównywanych zużyć. Wartość progową będziemy wyliczać jako 1/20 średniej, czyli 1/40 sumy, lepsze zużycie nie może przekroczyć przyjętego progu. Ostatecznie do porównania posłużymy nam operator `<`.

```
zuzycie_nie_gorsze(Dobre,Zle) :-
    Progowa is (Dobre + Zle) / 40,
    Najgorsze is Zle + Progowa,
    Dobre < Najgorsze.
```

Mamy zatem:

```
?- zuzycie_nie_gorsze(10.5, 10.7).
yes
?- zuzycie_nie_gorsze(10.7, 10.5).
yes
?- zuzycie_nie_gorsze(10.1, 10.7).
no
?- zuzycie_nie_gorsze(10.7, 10.1).
yes
```

Teraz możemy już stwierdzić, czy samochód `Auto1` jest lepszy od innego samochodu, `Auto2`:

```
lepszy(Auto1, Auto2) :-
    zuzycie_paliwo(Auto1, Zuzycie1),
    zuzycie_paliwo(Auto2, Zuzycie2),
    zawsze_lepszy(Zuzycie1, Zuzycie2).
```

Teraz musimy zdefiniować predykat `zawsze_lepszy`, który będzie sprawdzał, czy jedna lista zużyć paliwa jest lepsza od drugiej. Spodziewać się należy, że elementy list będą porównywane za pomocą `zuzycie_nie_gorsze`. Zgodnie ze swoją nazwą, predykat `zawsze_lepszy` nie zawodzi, jeśli dla danych dwóch list zużycia paliwa każdy element pierwszej listy jest nie gorszy od odpowiadającego mu elementu z drugiej listy. Oto potrzebna nam definicja:

```
zawsze_lepszy([], []).
zawsze_lepszy([Zuzycie1|O1], [Zuzycie2|O2]) :-
    zuzycie_nie_gorsze(Zuzycie1, Zuzycie2),
    zawsze_lepszy(O1, O2).
```

Najpierw zajmijmy się klauzulą rekurencyjną. Wynika z niej, że jedna lista jest zawsze lepsza od innej, jeśli jej głowa jest nie gorsza od głowy drugiej listy oraz ogon pierwszej listy jest zawsze lepszy od ogona drugiej listy. Ten drugi warunek sprawdzamy tym samym predykatem, stosując rekurencję. Kiedy zatem początkowo wywołujemy `zawsze_lepszy`, odrzucamy głowy list, porównujemy je i porównujemy ogony list. W tym wywołaniu rekurencyjnym porównywane są drugie elementy list początkowych, dalej badane są ogony ogonów. W ten sposób system przetwarza listy krok po kroku. Przetwarzanie kończy się, kiedy dojdziemy do końca obu list (jeśli są równej długości, obie kończą się jednocześnie). Wtedy uzgadniana jest pierwsza klauzula (warunek brzegowy) i cały cel jest uzgodniony. Jeśli w tym czasie zawiedzie którekolwiek porównanie `zuzycie_nie_gorsze`, zawiedzie też cały nasz cel `zawsze_lepszy`.

Jednak zastosujemy jeszcze inne warunki porównywania list zużycia paliwa. Interesuje nas też sytuacja, kiedy jedna lista jest *nie* gorsza od innej czasami.

```
czasami_lepszy([Zuzycie1|_], [Zuzycie2|_]) :-
    zuzycie_nie_gorsze(Zuzycie1, Zuzycie2).
czasami_lepszy([_|Zuzycie1], [_|Zuzycie2]) :-
    czasem_lepszy(Zuzycie1, Zuzycie2).
```

Definicja ta nieznacznie różni się od poprzedniej. Jak poprzednio, do analizy list używamy rekurencji. Teraz warunkiem końcowym jest stwierdzenie, że element z pierwszej listy jest lepszy od odpowiadającego mu elementu z listy drugiej; wtedy cel jest uzgodniony i dalsze badanie list jest zbędne. Jeśli dojdziemy do samego końca obu list, oznacza to, że nie znaleźliśmy żadnego dowodu wyższości pierwszej z nich. W takim wypadku predykat, zgodnie z oczekiwaniami, zawiedzie, gdyż obie klauzule jako argumenty muszą mieć niepuste listy. Klauzula rekurencyjna sugeruje, że jednym sposobem stwierdzenia, że jedna lista jest czasami lepsza od drugiej, jest znalezienie dowodu wyższości w ogonie listy — niezależnie od tego, jak mają się do siebie głowy list.

Ćwiczenie 3.1. Jeśli użyjemy definicji `czasami_lepszy`, prawie każdy samochód okaże się lepszy od pozostałych. Zmień program tak, aby samochód był uznawany za lepszy od innego wtedy, gdy przynajmniej jeden z wyników testów jest znacząco lepszy (Czytelnik zdecyduje, jak należy rozumieć określenie „znacząco”).

Łączenie struktur

Dołączenia dwóch list w jedną, nową listę używa się predykatu `append`. Nie zawiedzie na przykład wywołanie:

```
append([a,b,c],[3,2,1],[a,b,c,3,2,1]).
```

Predykat `append` najczęściej używany jest do tworzenia nowej listy będącej złożeniem dwóch innych list:

```
?- append([alfa,beta],[gamma,delta],X).
X = [alfa,beta,gamma,delta]
```

Można jednak użyć tego predykatu inaczej:

```
?- append(X,[b,c,d],[a,b,c,d]).
X = [a]
```

Predykat `append` definiuje się następująco:

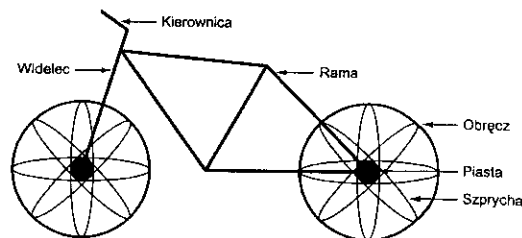
```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Warunkiem końcowym jest to, że pierwsza lista będzie pusta. W takim wypadku dowolna lista dołączona do listy pustej jest tą samą listą. W przeciwnym razie obowiązują następujące zasady:

1. Pierwszy element pierwszej listy X zawsze będzie pierwszym elementem listy połączonej.
2. Ogon pierwszej listy $L1$ zawsze będzie połączony z drugim argumentem $L2$ i razem staną się one ogonem połączonej listy $L3$.
3. Do wykonania złączenia opisanego w poprzednim punkcie używamy samego `append`.
4. W miarę pobierania głowy z pozostałości pierwszego argumentu, stopniowo redukujemy ten argument do listy pustej, dzięki czemu w końcu wystąpi warunek końcowy.

Więcej o `append` powiemy w dalszych przykładach. W kolejnych rozdziałach opiszemy różne właściwości i zastosowania predykatu `append`, a na razie użyjmy go w kolejnym prostym przykładzie zastosowania rekurencji.

Załóżmy, że pracujemy w fabryce rowerów, gdzie konieczne jest spisywanie części rowerowych. Jeśli chcemy zmontować rower, musimy wiedzieć, które części będą nam do tego potrzebne. Każda część roweru może mieć składniki (inne części), na przykład koło ma szprychy, obręcz i piastę. Piasta może się składać z osi i przekładni. Oto rysunek pokazujący te części:



Przyjrzyjmy się bazie danych w formie drzewa, w której możemy sprawdzać, jakie części są potrzebne do zmontowania roweru. W następnym podrozdziale poprawimy program tak, aby wyliczać, ile sztuk poszczególnych części jest potrzebnych.

Rower będziemy składać z dwóch rodzajów części: podzespołów i części nierozkładalnych. Każdy podzespół składa się z szeregu części nierozkładalnych, na przykład wspomniane koło zawiera szprychy, obręcz i piastę. Części nierozkładalne, zgodnie z nazwą, nie mają już elementów składowych, a same składają się na podzespoły.

Części nierozkładalne zapiszemy jako zwykłe fakty:

```
nierozkładalna(obrecz).
nierozkładalna(szprycha).
nierozkładalna(ramatylna).
nierozkładalna(kierownica).
nierozkładalna(przekladnia).
nierozkładalna(trzpień).
nierozkładalna(nakretka).
nierozkładalna(widelec).
```

Oczywiście nie są to wszystkie części składowe roweru, ale wystarczy nam do prezentacji.

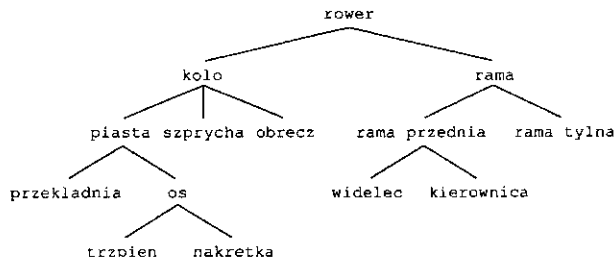
Podzespół będziemy zapisywać jako jego nazwę z listą elementów nierozkładalnych i ich liczbą. Oto przykładowy fakt opisujący montaż roweru z dwóch kół i ramy:

```
podzespól(rower,[kolo,kolo,rama]).
```

Oto baza danych z podzespółami naszego uproszczonego pojazdu:

```
podzespól(rower,[kolo,kolo,rama]).
podzespól(kolo,[szprycha,obrecz,piasta]).
podzespól(rama,[ramatylna,ramaprzednia]).
podzespól(ramaprzednia,[widelec,kierownica]).
podzespól(piasta,[przekladnia,os]).
podzespól(os,[trzpień,nakretka]).
```

Powyższe klauzule nie opisują oczywiście całego roweru, na przykład nie rozróżniają piasty przedniej i tylnej — obie mają przekładnię! Brak łańcucha i pedałów, nie ma też siodełka. Poza tym pominięto wskazówki dotyczące sposobu montowania części, jedynie podano te części, ułożone w następującą hierarchię:



Pamiętajmy, że powyższa struktura nie odzwierciedla kształtu struktury danych, a opisuje jedynie rower.

Teraz możemy napisać program, który, kiedy otrzyma część, wyliczy elementy potrzebne do zmontowania tej części. Jeśli część jest nierozkładalna, nie trzeba poza nią samą niczego więcej. Jeśli jednak chcemy mieć podzespół, musimy zmontować wszystkie jego składniki. Zdefiniujemy predykat `czesci`, którego będziemy używać w formie `czesci(X,Y)`, gdzie `X` to nazwa części, a `Y` to lista części nierozkładalnych potrzebnych do uzyskania danej części. W pierwszej wersji programu pominiemy liczbę potrzebnych części; bardziej dopracowaną wersję programu pokazemy w rozdziale 7.

Warunkiem końcowym jest to, że `X` jest częścią nierozkładalną — w takim wypadku jako listę zwracamy samo `X`:

```
czesci(X,[X]) :- nierozkladalna(X).
```

Następny przypadek jest taki, że `X` jest podzespółem. W takim wypadku musimy sprawdzić, czy w bazie danych znajduje się odpowiedni fakt `podzespól` i jeśli tak, użyć

predykatu `czesci` do każdej części składającej się na ten podzespół. Do tego ostatniego zadania użyjemy predykatu `listaczesci`:

```
czesci(X,P) :-
    podzespól(X,Podzesci),
    listaczesci(Podzesci,P).
```

Predykat `listaczesci` pobiera listę części (z drugiego argumentu faktu `podzespól`) i dla każdej wywołuje `czesci`. W predykanie `listaczesci` uzyskane części muszą być dołączone jako ogon do uzyskanej dotąd listy, do połączenia używamy predykatu `append`:

```
listaczesci([],[]).
listaczesci([P|Ogon],Wszytskie) :-
    czesci(P,Czesciglowy),
    listaczesci(Ogon,Czesciogona),
    append(Czesciglowy,Czesciogona,Wszytskie).
```

Lista stworzona przez `czesci` nie zawiera informacji o liczbie potrzebnych części, mogą natomiast pojawiać się w niej powtórzenia. W rozdziale 7. pokażemy pełniejszą wersję tego programu, w której te niedogodności zostaną usunięte.

Do generowania zwykłych zdań na podstawie części można podejść dwojako. Można zdania rozłożyć hierarchicznie: na frazę rzeczownikową, frazę czasownikową, frazę rzeczownikową zawierającą może liczebnik i rzeczownik, i tak dalej. Tak więc za pomocą części można opisać prostą gramatykę. Istnieje też inna możliwość. `listaczesci` elementy swojego pierwszego argumentu przegląda od lewej do prawej, tak samo wyniki dołączane są od lewej do prawej. Opisane właściwości predykatu `czesci` pozwalają użyć takiej samej metody do generowania zdań. Typowe „podzespóły” gramatyczne mogą wyglądać tak:

```
podzespól(zdanie,[fraza_rzeczownikowa,fraza_czasownikowa]).
podzespól(fraza_rzeczownikowa,[liczebnik,rzeczownik]).
podzespól(liczebnik,[jeden]).
podzespól(rzeczownik,[duchowny]).
podzespól(rzeczownik,[pojazd]).
```

zaś słowa mogą być opisywane jako części nierozkładalne:

```
nierozkladalna(duchowny).
nierozkladalna(pojazd).
```

Teraz warto zrobić samemu nieco ćwiczeń związanych z generowaniem zdań. Konieczne będzie podanie porządkowej gramatyki i słownictwa, na początek należy pominąć wszelkie kwestie odmiany słów. Jak zawsze, Prolog po podaniu każdego rozwiązania zatrzymuje się, więc w celu sprawdzenia wszystkich generowanych rozwiązań należy wcisnąć średnik.

Nie jest to oczywiście nasze ostatnie słowo w kwestii generowania zdań w języku naturalnym; cały rozdział 9. poświęcimy bardziej złożonej analizie języka polskiego w Prologu.

Akumulatory

Często konieczne jest przeglądanie struktur prologowych i wyliczanie wyników na podstawie wartości znalezionych w tych strukturach. Podczas analizy tych struktur pojawiają się wyniki pośrednie. W takim wypadku zwykle używa się w predykanie argumentu zawierającego „wartość dotąd uzyskaną”. Taki argument nazywany jest *akumulatorem*.

W następnym przykładzie pokażemy predykat `dllisty` bez akumulatora, a potem analogiczny predykat z akumulatorem. Cel `dllisty(L,N)` nie zawiedzie, jeśli długość listy `L` wynosi `N`. Niektóre implementacje Prologu mają predykat wbudowany `length` pełniący taką samą funkcję. Przystępujemy zatem do realizacji predykatu `dllisty` bez akumulatora. Użyjemy dwóch klauzul, jednej opisującej warunek końcowy, drugiej przypadek rekurencyjny. Warunkiem końcowym jest stwierdzenie, że lista pusta ma długość 0. Przypadek rekurencyjny mówi, że długość listy niepustej można wyliczyć przez dodanie wartości jeden do długości ogona tej listy:

```
dllisty([],0).
dllisty([_|_],N) :- dllisty([],N1), N is N1 + 1.
```

Alternatywnie moglibyśmy wykorzystać tę samą zasadę, ale odpowiedź zapisywać w akumulatorze w dodatkowym parametrze. Użyjemy predykatu pomocniczego `listaakum` będącego uogólnieniem `dllisty`. Wywołanie `listaakum(L,A,N)` oznacza, że długość listy `L` dodana do wartości akumulatora `A` wynosi `N`. Aby zatem wyznaczyć długość listy za pomocą `listaakum`, musimy jako drugi argument podać 0 (zero). Służy do tego pomocnicza klauzula określająca związek między `dllisty` a `listaakum`.

```
dllisty(L,N) :- listaakum(L,0,N).

listaakum([],A,A).
listaakum([_|_],A,N) :- A1 is A + 1, listaakum([],A1,N).
```

Predykat `listaakum` ma dwie klauzule: pierwsza opisuje listę pustą — mówi, że w takim wypadku długość jest taka, jaką zebrano dotąd w `A`. W drugiej klauzuli dodajemy do zebranej wartości `A` jedynkę i wywołujemy ten sam predykat na ogonie listy, podając jako wartość akumulatora `A1`.

Zwróćmy uwagę na to, że ostatni argument podcelu rekurencyjnego `N` jest taki sam, jak ostatni argument głowy klauzuli. Oznacza to, że długość całej listy będzie liczbą wyliczoną przez podcel rekurencyjny, więc wyznaczenie ostatecznego wyniku jest zlecane w całości do tego podcelu. Wszystkie dodatkowe informacje, jakie są potrzebne, przekazywane są w akumulatorze. Jeśli w podcelu rekurencyjnym ponownie zostanie użyta druga klauzula, znowu zostanie jej zlecone wyznaczenie wyniku (ale z inną wartością akumulatora). Tak więc otrzymujemy ciąg wywołań `listaakum`, za każdym razem jako lista przekazywany jest ogon listy z wywołania poprzedniego i taki sam ostatni argument, zaś akumulator jest większy o 1. Oto ciąg podcelów pozwalający wyznaczyć długość listy `[a,b,c,d,e]`:

```
listaakum([a,b,c,d,e],0,N)
listaakum([b,c,d,e],1,N)
listaakum([c,d,e],2,N)
```

```
listaakum([d,e],3,N)
listaakum([e],4,N)
listaakum([],5,N)
```

gdzie `N` jest tą samą zmienną. Ostatni cel spełnia warunek końcowy, więc odpowiada mu pierwsza klauzula `listaakum` i ostateczna wartość staje się równa aktualnej wartości akumulatora. Początkowo akumulator miał wartość 0, w każdym wywołaniu zwiększono jego wartość o 1, więc ostatecznie otrzymujemy 5, czyli faktyczną długość listy. Wszystkie cele `listaakum`, łącznie z pierwszym wywołaniem przez `dllisty`, miały taki sam ostatni argument, więc we wszystkich celach długość listy zostanie ukonkretniona tą samą wartością; w szczególności, `N` z klauzuli `dllisty` otrzyma wartość 5.

Akumulatory nie muszą być liczbami całkowitymi; jeśli potrzebny nam wynik jest listą, akumulator może zawierać utworzoną dotąd część listy. Jest to przydatne do zapisywania wartości tymczasowej, na przykład jeśli chcemy, aby wartości na liście się nie powtarzały. Użycie w takiej sytuacji akumulatora pozwala uniknąć kosztownego złączania struktur. Ogólnie rzecz biorąc, z uwagi na szybkość działania często staramy się uniknąć złączania struktur, gdyż są to operacje wolne. Jeśli na przykład łączymy listy za pomocą `append`, przetwarzanie trwa tak długo, aż pierwsza lista będzie pusta. Na każdym etapie tego przetwarzania tworzymy w trzecim argumentcie fragment listy. Kiedy w końcu dochodzimy do końca listy, ostatnią część wyniku ukonkretniamy listą z drugiego argumentu. Aby otrzymać listę wynikową kończącą się drugą listą wejściową, musimy stworzyć kopię pierwszej listy. Jeśli lista ta jest długa, oznacza to dużo pracy.

Zastanówmy się, co się stanie, jeśli w naszym katalogu części chcemy znaleźć elementy składające się na rower. Montaż roweru opisuje cel:

```
podzespól(rower,[koło,koło,rama]).
```

Aby określić części potrzebne do zmontowania roweru, używamy predykatu `listaczesci` do znalezienia składników podzespółów z listy `[koło,koło,rama]`. Z uwagi na sposób zdefiniowania `listaczesci`, trzeba:

- ♦ Znaleźć składniki `rama`.
- ♦ Połączyć powyższe z listą pustą, aby uzyskać części do `[rama]`.
- ♦ Znaleźć składniki `koło`.
- ♦ Dołączyć te składniki do listy części do `[rama]`, aby uzyskać części do `[koło,rama]`.
- ♦ Znaleźć składniki `koło`.
- ♦ Dołączyć te składniki do listy części do `[koło,rama]`, aby uzyskać listę części do `[koło,koło,rama]`.

Powyższa sekwencja operacji jest nadmiarowa, gdyż każda lista części podzespółów roweru jest tworzona dwukrotnie: najpierw kiedy jest po raz pierwszy ustalana, potem kiedy jest dołączana do listy uzyskanej wcześniej. Z uwagi na to, że niektóre części roweru same są podzespółami, ta nadmiarowość nawarstwia się podczas ich rozkładania.

Wszystkiego tego możemy uniknąć korzystając z akumulatorów. Tak jak w przypadku predykatu `dl`listy, wprowadzimy predykaty pomocnicze z akumulatorami i użyjemy klauzuli startowej, która wywoła te predykaty z wartościami początkowymi akumulatorów. Oto program zestawiający części, stworzony przy użyciu akumulatorów. Klauzule `czesc` i `podzespól` pozostały niezmiennie, więc pominęliśmy je tutaj. Zauważmy, że nie jest już używany `append`.

```
czesci(X,P) :- czesciakum(X,[],P).

czesciakum(X,A,[X|A]) :- czesc(X).
czesciakum(X,A,P) :-
    podzespól(X,Podczesci),
    listaczesciakum(Podczesci,A,P).

listaczesciakum([],A,A).
listaczesciakum([P|Ogon],A,Razem) :-
    czesciakum(P,A,CzesciGlowy),
    listaczesciakum(Ogon,CzesciGlowy,Razem).
```

Predykaty `czesciakum` i `listaczesciakum` zdefiniowano podobnie, jak poprzednie wersje `czesci` i `listaczesci`, tylko dodano akumulator jako drugi argument. Argument ten jest listą części (nierozkładalnych) znalezionych dotąd. Wobec tego wywołanie `czesciakum(X,A,P)` oznacza, że części obiektu `X` dodane do listy `A` dadzą listę `P`. Zauważmy podobieństwo tej interpretacji do `listaakum`. Jeśli chcemy użyć `czesciakum` do określenia części potrzebnych do zmontowania obiektu, musimy jako drugi argument podać listę pustą; robi to klauzula `czesci`.

Pierwsza klauzula `czesciakum` po prostu tworzy nową listę, której głowa to obiekt z pierwszego argumentu, a ogon to zbierana lista części; predykat nie zawiedzie, jeśli obiekt jest częścią nierozkładalną. Druga klauzula dotycząca obiektów będących podzespołami najpierw określa listę części składowych, a potem za pomocą `listaczesciakum` określa części składające się na poszczególne elementy z tej listy. Akumulator `A` został przekazany do `listaczesciakum`.

Pierwsza klauzula `listaczesciakum` to warunek końcowy, który powoduje zwrócenie jako wyniku zebranej dotąd listy części `A`. W przypadku rekurencyjnym, w celu określenia części następnego elementu z listy wywoływany jest predykat `czesciakum`, zaś cel rekurencyjny zajmuje się resztą listy. Zauważmy, że drugi argument drugiej klauzuli, `A`, używany jest jako akumulator dla celu `czesciakum`, a wynik celu `czesciakum`, `CzesciGlowy`, używany jest jako akumulator celu rekurencyjnego.

Akumulatorów używać będziemy w dalszej części książki; szczególną uwagę warto zwrócić na następny podrozdział oraz podrozdziały „Przeszukiwanie labiryntu”, „Przetwarzanie list” i „Użycie bazy danych: `random`, `gensym`, `findall`” z rozdziału 7.

Struktury różnicowe

W poprzednim podrozdziale pokazywaliśmy, jak za pomocą akumulatorów można uniknąć zbędnego złączania struktur. Pomińmy efekt uboczny takiego podejścia

jest tworzenie list, których elementy są w kolejności *odwrotnej* niż na liście wyjściowej. Czasami jednak potrzebne jest generowanie elementów w kolejności *takiej samej*, jak na liście oryginalnej. Służą do tego *struktury różnicowe* (w tym wypadku *listy różnicowe*).

Jeśli użyjemy naszego programu z zestawieniem części do znajdowania elementów roweru, wersja z akumulatorami działa równie dobrze jak nasza pierwsza wersja oraz działa szybciej. Jeśli jednak użyjemy jej do generowania zdań w języku naturalnym, natkniemy się na problem polegający na tym, że słowa będą pojawiały się w odwrotnej kolejności! Nie miałyby to znaczenia w przypadku części rowerowych, gdyż tam kolejność nie jest istotna, ale w przypadku zdań niewątpliwie kolejność jest ważna. Jeśli zastanowimy się nad tym, jak „części” są składane, nie powinno zaskakiwać, że ostatecznie mają one kolejność odwrotną. Zawsze, kiedy dochodzimy do części nierozkładalnych, tworzymy nowy akumulator mający tę część *przed* wszystkimi częściami znalezionymi dotąd.

W przypadku korzystania z akumulatorów tworzenie struktur wynikowych oparte jest na dwóch argumentach: jednym z nich jest wynik uzyskany dotąd, drugi to wynik ostateczny. W przypadku list różnicowych mamy też dwa argumenty i pierwszy z nich to wynik końcowy, ale drugi argument interpretowany jest jako „miejsce w wyniku końcowym, w które można wstawić dodatkowe informacje”. Sposób zapisu takiego miejsca w Prologu to po prostu zmienna powiązana ze składnikiem struktury. Wobec tego poniższe dwa terminy reprezentują listę wraz ze „zmienną rezerwującą miejsce” na dodatkowe informacje:

```
[a,b,c|X] X
```

Jeśli mamy listę z zarezerwowanym miejscem, możemy ją dalej wypełniać przekazując opisywaną zmienną jako argument do kolejnych celów, które ten argument będą ukonkretniać. Przede wszystkim interesuje nas miejsce na nowe informacje, wstawiane jeśli cel zostanie spełniony. Wobec tego wymagamy, aby cel przekazał dalej *nowe* wolne miejsce w innym argumentcie. Oto koniunkcja celów tworząca listę z wolnym miejscem, dodającą do listy elementy za pomocą predykatu `p` i następnie wypełniającą pozostałe miejsce listą `[z]`:

```
?- Res = [a,b,c|X], p(X,NoweMiejsce), NoweMiejsce = [z].
```

Możemy pozwolić `p` nie ukonkretniać dalej listy podając klauzulę, która spowoduje zwrócenie pierwotnie przekazanego miejsca jako miejsca nowego:

```
p(Miejsce,Miejsce).
```

Jeśli wybrana zostanie taka klauzula, po uzgodnieniu zapytania zmienna `Res` będzie miała wartość `[a,b,c,z]`. Można też podać klauzulę, powodującą ukonkretnienie pierwotnego miejsca strukturą zawierającą ową zmienną, która zostanie zwrócona jako nowe miejsce:

```
p([d|NoweMiejsce],NoweMiejsce).
```

Klauzula taka otrzyma uzyskane wyniki głównie dzięki wywołaniom podcelów. Jeśli wybrana zostanie pokazana wyżej klauzula, po uzgodnieniu celów zmienna `Res` będzie miała wartość `[a,b,c,d,z]`.

Oto nowa wersja programu z częściami rowerowymi, w której tym razem użyliśmy techniki list różnicowych:

```
czesci(X,P) :- czesciakum(X,P,Miejsce), Miejsce = [].
czesciakum(X,[X|Miejsce],Miejsce) :- czesc(X).
czesciakum(X,P,Miejsce) :-
    podzespol(X,Podczesci),
    listaczesciakum(Podczesci,P,Miejsce).
listaczesciakum([],Miejsce,Miejsce).
listaczesciakum([P|Ogon],Razem,Miejsce) :-
    czesciakum(P,Razem,Miejsce),
    listaczesciakum(Ogon,Miejsce,Miejsce).
```

Najpierw przyjrzyjmy się klauzuli `czesci`. Kiedy pierwszy raz jest wywoływana `czesciakum`, wynik jest tworzony w drugim argumente `P` i jest zwracany w zmiennej `Miejsce`. Jako że `czesci` wywołuje `czesciakum` tylko raz, musimy listę różnicową zakończyć ukonkretniając `Miejsce` listą pustą, `[]`. Warto zwrócić uwagę na równoważną definicję `czesci`:

```
czesci(X,P) :- czesciakum(X,P,[]).
```

Ta zwarta wersja zapewnia wypełnienie ostatniego miejsca pustą listą jeszcze przed ustaleniem zawartości listy.

Pierwsza klauzula `czesciakum` zwraca listę różnicową zawierającą obiekt w pierwszym argumente, wykonywana jest wtedy, gdy dany obiekt jest częścią nierozkładalną. Druga klauzula opisuje podzespół, umieszcza części na liście różnicowej ze zmiennych `Razem` i `Miejsce`. Rekurencyjny cel zwraca część listy różnicowej zaczynając się od `Miejsce`, kończąc się na `Miejsce`. Cały wynik, lista między `Razem` i `Miejsce`, jest wynikiem drugiej klauzuli `listaczesciakum`. Sposób tworzenia listy przez „splatanie” częściowych wyników można zobrazować następująco:

```
listaczesciakum([P|Ogon],Razem,Miejsce) :-
    czesciakum(P,Razem,Miejsce),
    listaczesciakum(Ogon,Miejsce,Miejsce).
```

Z list różnicowych skorzystamy znowu definiując predykat `quisortx` w rozdziale 7.

Rozdział 4.

Nawracanie i odcięcie

Podsumujmy naszą wiedzę z rozdziałów 1. i 2. dotyczącą tego, co może dzieć się z celem:

1. Następuje próba uzgodnienia celu. Uzgadniając cel, przeszukujemy bazę danych od początku. Możemy mieć do czynienia z jedną z dwóch sytuacji:

a) Powiedzie się unifikacja z faktem lub głową reguły. W takim przypadku cel został dopasowany; oznaczamy odnośne miejsce w bazie danych i ukonkretniamy zmienne jeszcze nieukonkretnione, które zostały też dopasowane. Jeśli dopasowywanie dotyczy reguły, najpierw trzeba spełnić podcele w tej regule zapisane. Jeśli cel zostanie uzgodniony, przechodzimy do uzgadniania następnego celu. W naszych diagramach cel taki jest następnym prostokątem pod strzałką. Jeśli pierwotny cel należał do koniunkcji, następnym celem będzie cel znajdujący się po prawej.

b) Żaden pasujący fakt ani głowa reguły nie zostaną znalezione. Wtedy mówimy, że cel zawiódł. W tej sytuacji staramy się inaczej spełnić cel z prostokąta powyżej strzałki. Jeśli pierwotny cel należał do koniunkcji, będzie to cel znajdujący się po lewej.

2. Następuje próba ponownego uzgodnienia celu. Przede wszystkim próbuje się uzgadniać kolejno wszystkie podcele, strzałka wraca w górę strony. Jeśli żaden podcel nie może zostać już spełniony, staramy się znaleźć alternatywną klauzulę dla samego celu. W tym wypadku wycofujemy ukonkretnienie wszystkich zmiennych ukonkretnionych w tym celu, następnie ponawiamy przeszukiwanie bazy danych, ale tym razem zaczynamy od tego miejsca w bazie danych, w którym był znacznik. Tak jak poprzednio, taki cel ponownie analizowany przez *nawracanie* może zostać uzgodniony lub zawieść, a wtedy odpowiednio korzystamy z powyższych podpunktów a) lub b).

W tym rozdziale dokładniej zajmiemy się nawracaniem. Przyjrzymy się też specjalnemu mechanizmowi programów prologowych, *odcięciu*. Odcięcie pozwala nakazać Prologowi, by nie wracał do wcześniej dokonanych wyborów.

Generowanie wielu rozwiązań

Najprościej jest wygenerować wiele rozwiązań zapytania na podstawie zbioru faktów wówczas, gdy wiele faktów pasuje do tego zapytania. Jeśli na przykład mamy w bazie następujące fakty `ojciec(X,Y)` oznaczające, że `Y` jest ojcem `X`:

```
ojciec(maria, jerzy).
ojciec(jan, jerzy).
ojciec(zuzanna, henryk).
ojciec(jerzy, edward).
```

Na zapytanie

```
?- ojciec(X,Y).
```

będzie istniało kilka odpowiedzi. Jeśli będziemy żądali za pomocą średnika kolejnych rozwiązań, otrzymamy taki oto wynik:

```
X=maria, Y=jerzy ;
X=jan, Y=jerzy ;
X=zuzanna, Y=henryk ;
X=jerzy, Y=edward
```

Dzięki nawracaniu znajdowane są wszystkie rozwiązania z bazy danych. Prolog nie analizuje zwracanych wyników i ich nie zapamiętuje, wobec czego, jeśli zadamy zapytanie

```
?- ojciec(_,X).
```

(czyli kto jest ojcem?) otrzymamy:

```
X=jerzy ;
X=jerzy ;
X=henryk ;
X=edward
```

`jerzy` pojawił się dwukrotnie, gdyż jest ojcem Marii i Jana. Jeśli Prolog może pokazać to samo w dwojaki sposób, traktuje obie możliwości jako różne rozwiązania.

Nawracanie ma miejsce także w sytuacji, kiedy istnieje więcej możliwych rozwiązań głębiej zagnieżdżonych. Przykładowa definicja reguły mówiącej, że „jednym z dzieci `X` jest `Y`”, może wyglądać tak:

```
dziecko(X,Y) :- ojciec(Y,X).
```

Wobec tego zapytanie

```
?- dziecko(X,Y).
```

da odpowiedź

```
X=jerzy, Y=maria ;
X=jerzy, Y=jan ;
X=henryk, Y=zuzanna ;
X=edward, Y=jerzy
```

Jako że `ojciec(Y,X)` ma cztery rozwiązania, tyleż rozwiązań ma `dziecko(X,Y)`. Co więcej, rozwiązania generowane są w takiej samej kolejności. Jedyną różnicą polega na odwrotnej kolejności argumentów, co wynika z definicji predykatu `dziecko`. Analogicznie, jeśli zdefiniujemy, że `ojciec(X)` oznacza, że `X` jest ojcem:

```
ojciec(X) :- ojciec(_,X).
```

to zapytanie

```
?- ojciec(X).
```

da w wyniku

```
X=jerzy ;
X=jerzy ;
X=henryk ;
X=edward
```

Jeśli mieszmamy fakty i reguły, możliwe rozwiązania pojawiają się w takiej kolejności, w jakiej prezentowane są dane. Tak więc możemy zapisać, że `adam` jest osobą, że `osoba` jest wszystko posiadające matkę, i że `ewa` jest osobą. Poza tym różni ludzie mogą mieć różne matki:

```
osoba(adam).
osoba(X) :- matka(X,Y).
osoba(ewa).
matka(kain,ewa).
matka(abel,ewa).
matka(jabal,ada).
matka(tubilkain,sybilla).
```

Jeśli teraz zadamy zapytanie:

```
?- osoba(X).
```

otrzymamy odpowiedzi:

```
X=adam ;
X=kain ;
X=abel ;
X=jabal ;
X=tubilkain ;
X=ewa ;
```

Zajmijmy się teraz ciekawszym przykładem, w którym mamy dwa cele i każdy z nich ma kilka rozwiązań. Wyobraźmy sobie, że planujemy prywatkę i chcemy zgadywać, kto będzie z kim tańczył. Możemy program napisać następująco:

```
ewentualna_para(X,Y) :- chlopiec(X), dziewczyna(Y).
```

```
chlopiec(jan).
chlopiec(marian).
chlopiec(bertrand).
chlopiec(karol).
```

```
dziewczyna(gryzelda).
dziewczyna(ermintruda).
dziewczyna(brunhilda).
```

Wyższa Szkoła
Zarządzania i Bankowości
 ul. Armii Krajowej 4, 30-150 Kraków
 tel. (012) 638-66-77 tel/fax (012) 637-33-47
 wpis do Rejestru MEN nr 55 z dnia 11.05.1996r
 Konto BGŚ OIKraków 1540115-12067-27006-00
 NIP 677-17-58-169 REGON 350814846

Program mówi, że X i Y mogą być parą, jeśli X jest chłopcem, a Y — dziewczyną. Oto uzyskane pary:

```
?- ewentualna_para(X,Y).
```

```
X = jan, Y = gryzelda ;
X = jan, Y = ermintruda ;
X = jan, Y = brunhilda ;
X = marian, Y = gryzelda ;
X = marian, Y = ermintruda ;
X = marian, Y = brunhilda ;
X = bertrand, Y = gryzelda ;
X = bertrand, Y = ermintruda ;
X = bertrand, Y = brunhilda ;
X = karol, Y = gryzelda ;
X = karol, Y = ermintruda ;
X = karol, Y = brunhilda
```

Musimy wyjaśnić sobie, dlaczego Prolog daje takie wyniki w takiej kolejności. Po pierwsze, spełniany jest cel `chlopiec(X)` — zostaje znaleziony Jan, pierwszy chłopiec. Następnie spełniany jest cel `dziewczyna(Y)`; pierwszą znalezioną dziewczyną jest Gryzelda. W tej chwili żądamy podania następnego rozwiązania. Prolog stara się spełnić ostatni cel, czyli `dziewczyna` i otrzymuje wartość `ermintruda`, więc drugą możliwą partnerką Jana jest `Ermintruda`. Podobnie Prolog generuje trzecie rozwiązanie, Jan i `brunhilda`. Kiedy Prolog znowu próbuje znaleźć inne rozwiązanie celu `dziewczyna(Y)`, okazuje się, że znacznik jest już na końcu bazy danych, więc cel ten zawodzi. Konieczne jest teraz inne spełnienie celu `chlopiec(X)`. Znacznik tego celu umieszczony jest na pierwszym fakcie `chlopiec`, więc następnym znalezionym rozwiązaniem jest `chlopiec(marian)`. Cel ten został zatem spełniony i Prolog sprawdza, jaki cel jest następny; okazuje się, że teraz trzeba od początku spełniać cel `dziewczyna(Y)`. Znajdowana jest najpierw `gryzelda`, potem pozostałe dwie dziewczyny. I znów niemożliwe jest spełnienie celu `dziewczyna`, więc pokazywany jest następny chłopiec i szukanie dziewczyn dla niego zaczyna się od początku. Sytuacja się stale powtarza, aż w końcu niemożliwe jest spełnienie ani celu `dziewczyna`, ani `chlopiec`, więc program więcej par nie może już wygenerować.

Pokazane tutaj przykłady były bardzo proste, gdyż mieliśmy po prostu wiele faktów i reguły z tych faktów korzystające. Dlatego właśnie możliwe było wygenerowanie jedynie skończonej liczby rozwiązań. Czasami chcielibyśmy generować dowolnie wiele rozwiązań, i to nie dlatego, że nas wszystkie interesują, ale dlatego, że nie wiemy z góry, ilu będziemy potrzebować. W takim wypadku konieczne jest użycie definicji rekurencyjnej, omawianej w poprzednim rozdziale.

Przyjrzyjmy się następującej definicji liczby całkowitej dodatniej¹. Cel `liczba_calkowita(N)` zostanie uzgodniony, jeśli N jest uokretniona liczbą całkowitą. Jeśli N jest nieukretniona, po wywołaniu naszego predykatu zostanie uokretniona liczbą całkowitą:

```
/* 1 */ liczba_calkowita(0).
/* 2 */ liczba_calkowita(X) :- liczba_calkowita(Y), X is Y+1.
```

¹ Przez dodatnią liczbę całkowitą rozumiemy liczbę całkowitą nie mniejszą od zera. Wystarczy powinno stwierdzenie, że liczb takich jest nieskończenie wiele.

Jeśli zadamy zapytanie

```
?- liczba_calkowita(X).
```

będziemy otrzymywać wszystkie możliwe liczby całkowite od 0, to jest 0, 1, 2, 3 i tak dalej. Za każdym razem, kiedy wymuszamy nawrót (na przykład wciskając średnik), `liczba_calkowita` generuje nową liczbę. Tak więc podana definicja generuje nieskończoną liczbę rozwiązań. Dlaczego? Ciąg zdarzeń, który do tego prowadzi, pokazano na rysunkach 4.1, 4.2 i 4.3.

Na każdym etapie najniższy punkt (1) jest miejscem dokonywania następnego wyboru. Początkowo wybieramy między faktem 1. a regułą 2. Jeśli wybierzemy fakt 1., nie trzeba już nic wybierać, otrzymujemy $X=0$. Jeśli wybierzemy regułę 2., mamy możliwość wyboru sposobu spełnienia celu z niej wynikającego. Jeśli wybierzemy fakt 1., otrzymamy $X=1$, zaś w przeciwnym razie do spełnienia podcelu używamy ponownie reguły. Za każdym razem Prolog najpierw wybiera fakt 1., zaś nawracanie powoduje zmianę ostatniego wyboru. Za każdym razem, kiedy się to dzieje, wracamy do miejsca ostatniego wybrania faktu 1. i zamiast niego korzystamy z reguły 2. Kiedy już tej reguły użyjemy, mamy do czynienia z nowym podcelem. Pierwszą możliwością spełnienia tego podcelu jest fakt 1.

W większości reguł Prologu możliwe jest znalezienie alternatywnych rozwiązań, o ile używane cele zawierają dużo nieukretnionych zmiennych. Oto predykat badający przynależność do listy (opisany w rozdziale 3.):

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

Powoduje on wygenerowanie wszystkich możliwych rozwiązań. Jeśli zapytamy:

```
?- member(a,X).
```

(zmienna X jest nieukretniona), otrzymamy jako wartości X kolejne częściowo uokretnione listy z pierwszym, drugim, trzecim i kolejnymi elementami. Warto spróbować przeanalizować, dlaczego tak się dzieje.

Innym przykładem wykorzystania przez ten sam predykat nawracania jest zapytanie

```
?- member(a,[a,b,r,a,k,a,d,a,b,r,a]).
```

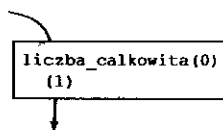
które uzgodni cel pięć razy. Istnieją też oczywiście przypadki, w których `member` powinien zostać uzgodniony co najwyżej raz, zaś pozostałe możliwości są zbędne. Możemy nakazać Prologowi odrzucenie dalszych rozwiązań przez użycie odcięcia.

Odcięcie

W tym podrozdziale zajmiemy się pewnym specjalnym mechanizmem programów prologowych: *odcięciem*. Odcięcie pozwala nakazać interpreterowi Prologu, by nie uwzględniał wcześniejszych możliwości wyboru podczas nawracania. Może to być istotne z dwóch powodów:

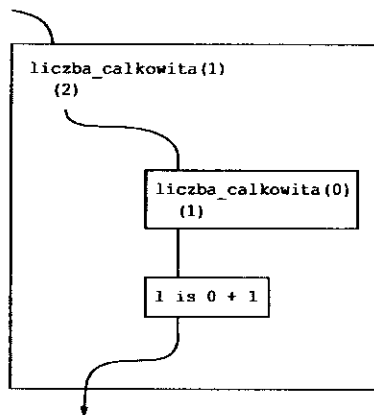
Rysunek 4.1.

Pierwsze rozwiązanie



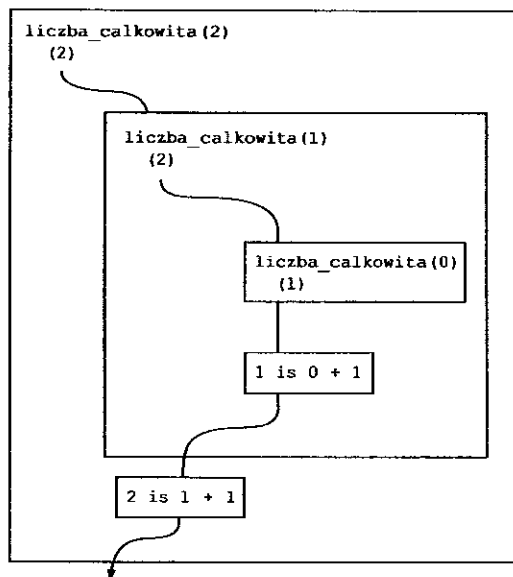
Rysunek 4.2.

Drugie rozwiązanie



Rysunek 4.3.

Trzecie rozwiązanie



- ♦ Program będzie działał szybciej, gdyż nie będzie się niepotrzebnie starał spełniać cele, których wynik uzgadniania jest z góry znany, i które wobec tego nie wnoszą do rozwiązania niczego nowego.
- ♦ Program zajmuje mniej pamięci komputera, gdyż pamięć jest zużywana oszczędniej z uwagi na to, że nie trzeba na później zapisywać punktów nawracania.

Czasami użycie odcięcia spowoduje też, że program, który dotąd działał, działać przestanie, albo odwrotnie.

Składniowo odcięcie w regule wygląda jak cel z predykatem ! bez argumentów. Cel ten zawsze jest natychmiast uzgadniany i nie może być ponownie uzgodniony. Jednak ma on też efekty uboczne polegające na zmianie sposobu nawracania. Miejsce tak zaznaczone jest niedostępne w tym sensie, że nie mogą być w nim umieszczane znaczniki.

Przyjrzyjmy się zastosowaniu odcięcia w praktyce. Załóżmy, że zarządzamy biblioteką i mamy zapisaną w Prologu bazę danych zawierającą dane o posiadanych książkach, osobach je pożyczających i terminach zwrotu. Jednym z zagadnień, które musimy uwzględnić jest dostępność poszczególnych usług bibliotecznych dla różnych grup ludzi. Pewne usługi — nazwijmy je podstawowymi — powinny być dostępne dla wszystkich. Przykładami takich usług jest korzystanie z katalogu czy czytelnia. Z kolei inne usługi mogą być dostępne tylko dla wybranych osób — wypożyczanie książek, korzystanie z pożyczek międzybibliotecznych. Jedną z reguł może mówić, że osoba przetrzymująca książki nie może korzystać z żadnych dodatkowych usług do czasu zwrócenia tych książek. Oto część odpowiedniego programu:

```
usluga(Osoba,Usluga) :-
    przetrzymana_książka(Osoba,Książka),
    !,
    usluga_podstawowa(Usluga).
usluga(Osoba,Usluga) :- dowolna_usluga(Usluga).
usluga_podstawowa(katalog).
usluga_podstawowa(czytelnia).
usluga_dodatkowa(wypożyczanie).
usluga_dodatkowa(pożyczki_miedzybiblioteczne).
dowolna_usluga(X) :- usluga_podstawowa(X).
dowolna_usluga(X) :- usluga_dodatkowa(X).
```

Potrzebna nam będzie baza danych klientów i informacji o korzystaniu z biblioteki; pokażemy jedynie dwa przykłady:

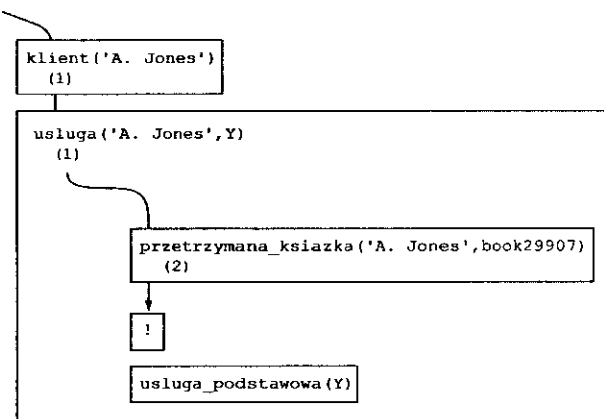
```
klient('A. Jones').
klient('W. Metesk').
przetrzymana_książka('C. Watzer',book10089).
przetrzymana_książka('A. Jones',book29907).
```

Po co użyto w tym programie odcięcia i jakie ma ono znaczenie? Załóżmy, że chcemy przejrzeć wszystkich korzystających z biblioteki i sprawdzić, które usługi są dla nich dostępne. Zadajemy zatem zapytanie:

```
?- klient(X), usluga(X,Y).
```

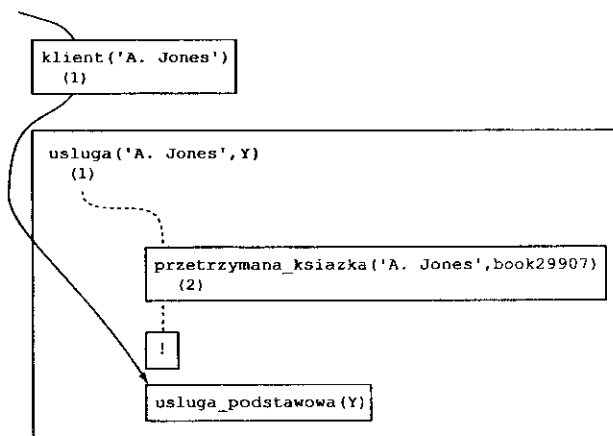
Prolog najpierw odnajduje pierwszą osobę, 'A. Jones'. Załóżmy, że ta osoba ma kilka książek, których termin zwrotu dawno minął. Aby dowiedzieć się, jakie usługi dla tej osoby są dostępne, najpierw znajdowana jest pierwsza klauzula `usługa`. Wprowadzany jest nowy cel: sprawdzenie, czy osoba ma jakieś przetrzymane książki. Po krótkim przeszukiwaniu faktów przetrzymana_książka znajduje pierwszy fakt dotyczący A. Jonesa (drugi fakt tego predykatu). Następny cel to odcięcie. Cel jest uzgadniany, jego efektem jest zatwierdzenie w systemie wszystkich decyzji pod chwili wybrania pierwszej klauzuli `usługa`. Tak więc sytuację istniejącą tuż przed odcięciem można zobrazować diagramem przedstawionym na rysunku 4.4:

Rysunek 4.4.
Tuż przed dojściem
do odcięcia



Kiedy zinterpretowane zostanie odcięcie, „odetnie” ono ścieżkę spełniania celów tak, że ewentualne nawracanie będzie musiało przejść poza odcięty punkt, jak na rysunku 4.5:

Rysunek 4.5.
Odcięcie zatwierdza
uzyskane rozwiązania,
ale jednocześnie
zmieniany jest sposób
uzgadniania w ten
sposób, że jeśli
`usługa_podstawowa`
zawiedzie,
nawracanie przejdzie
do ponownego
uzgadniania celu
`klient`



Efektom odcięcia w regule `usługa` (klauzula 1.) jest zatwierdzenie w systemie wszystkich wyborów dokonanych od chwili wybrania tej reguły. Ścieżka uzgadniania jest modyfikowana tak, aby pominąć wszystkie znaczniki między celem `usługa` a odcięciem, włącznie. Jeśli zatem nawracanie spowoduje wycofanie się poza ten punkt, cel `usługa` natychmiast zawiedzie.

Z powodu odcięcia system nie będzie uwzględniał innych możliwych uzgodnień celu `przetrzymana_książka` ('A. Jones', `Książka`). Jest to zgodne z naszymi oczekiwaniami, gdyż interesuje nas, czy dana osoba ma *jakikolwiek* przetrzymane książki. System nie będzie też uwzględniał klauzuli 2. predykatu `usługa`, gdyż wybór reguły, w której występuje odcięcie, jest także pomijany. To też jest zgodne z założeniami, gdyż nie chcemy generować rozwiązań twierdzących, że dla A. Jonesa dostępne są wszystkie usługi.

Podsumowując, efekt odcięcia w tym przykładzie można opisać następująco:

Jeśli okaże się, że osoba ma jakąś przetrzymaną książkę, może korzystać jedynie z usług podstawowych. Nie musimy przeglądać wszystkich przetrzymanych książek takiej osoby ani rozważać innych reguł dotyczących usług.

W naszym przykładzie odcięcie zatwierdziło wszystkie decyzje wstecz, aż do celu `usługa`. Cel ten jest nazywany *celem rodzicielskim* celu *odcięcia*, gdyż to ten cel spowodował użycie reguły zawierającej odcięcie. Na naszych diagramach cel rodzicielski zawsze jest celem, którego prostokąt jest najmniejszym prostokątem obejmującym prostokąt celu „!”. Formalna definicja wyniku działania odcięcia jest następująca:

Jeśli w celu znalezione zostanie odcięcie, system zatwierdza wszystkie decyzje podjęte od chwili wywołania celu rodzicielskiego. Wszystkie pozostałe możliwości są odrzucane, wobec czego wszelkie próby ponownego uzgodnienia któregośkolwiek celu znajdującego się między celem rodzicielskim a celem-odcięciem zawiodą.

Różnie można opisywać to, co dzieje się z wyborami, na które wpłynęło odcięcie. Można powiedzieć, że wybory są odcięte czy zamrożone, zaś system pozostaje przy dokonanych wyborach, albo że pozostałe możliwości są odrzucane. Można też spojrzeć na symbol odcięcia jako swego rodzaju rozdzielenie celów. W przypadku poniższej koniunkcji celów

`foo :- a, b, c, !, d, e, f.`

Prolog może swobodnie nawracać między celami a, b i c, póki nie zostanie uzgodniony cel c. Jego uzgodnienie spowoduje przejście do celu d. Teraz nawracanie może już zachodzić jedynie między celami d, e i f, możliwe jest wielokrotne uzgodnienie całej koniunkcji. Nie jest już jednak możliwe wykonanie nawrotu do celu c; jeśli cele za odcięciem zawiodą, zawiedzie od razu cała koniunkcja i zawiedzie też cel `foo`.

Zanim zajmiemy się praktycznymi zastosowaniami odcięcia, trzeba powiedzieć jeszcze o jednym. Jak wspomnieliśmy, jeśli odcięcie pojawia się w jakiejś regule i uzgodniony zostanie cel odcięcia, to Prolog zatwierdza wszystkie wybory dokonane do chwili wywołania celu rodzicielskiego. Oznacza to, że wybór z danej reguły i wszystkie

inne wybory są ustalone. Okaze się, że można podawać inne rozwiązania w jednej regule za pomocą predykatu wbudowanego: (oznaczającego logiczne *lub*). Wybory zrealizowane w taki właśnie sposób podlegają takim zasadom, jak osobne klauzule. Oznacza to, że kiedy spełniony zostanie cel odcięcia, wszystkie wybory wynikające z użycia *lub* też zostaną zamknięte, gdyż sposób uzgodnienia reguły jest już niezmienny.

Typowe zastosowania odcięcia

Typowe zastosowania odcięcia można podzielić na trzy grupy:

- ♦ Pierwsza grupa to przypadki, kiedy chcemy poinformować system, że dla danego celu już została znaleziona odpowiednia reguła. Odcięcie w takim wypadku mówi: „jeśli doszedłeś dotąd, wybrałeś już prawidłową regułę dla tego celu”.
- ♦ Druga grupa to sytuacje, kiedy chcemy poinformować Prolog, że dany cel ma natychmiast zawieść bez sprawdzania rozwiązań alternatywnych. W takim wypadku używamy odcięcia wraz z predykatem *fail* mówiąc: „jeśli dotarłeś tutaj, natychmiast zaprzestań prób uzgodnienia tego celu”.
- ♦ Trzecia grupa to przypadki, kiedy należy zaprzestać generowania rozwiązań alternatywnych przez nawracanie. W takim wypadku odcięcie mówi: „jeśli jesteś tutaj, znalazłeś już jedyne rozwiązanie problemu, wobec czego dalsze przeszukiwanie nie ma sensu”.

Przyjrzymy się teraz przykładom użycia odcięcia we wszystkich trzech sytuacjach. Trzeba jednak pamiętać, że znaczenie odcięcia jako takiego zawsze jest takie samo. Podział zastosowań na trzy grupy ma charakter wyłącznie dydaktyczny i ma uzmysłowić czytelnikowi, w jakich powodów odcięcie może być mu potrzebne w programach.

Potwierdzanie wyboru reguły

Bardzo często w programie napisanym w Prologu chcemy powiązać z tym samym predykatem wiele klauzul. Jedną z nich dotyczyć będzie argumentów w jednej postaci, drugą w innej i tak dalej. Często możemy wskazać regułę, która powinna być użyta do danego celu, za pomocą wzorców w głowie reguły pasujących do celów tylko w określonej postaci. Jeśli określenie zawczasu postaci argumentów nie jest możliwe lub nie można wskazać wszystkich wzorców, trzeba radzić sobie inaczej. W takim wypadku podaje się reguły dla poszczególnych typów argumentów i dodatkowo podaje się regułę, które ma być stosowana, jeśli nic innego nie będzie pasowało.

W ramach przykładu rozważmy następujący program: reguła definiująca predykat *suma_do* wywoływana jest jako cel *suma_do(N, X)*, gdzie *N* jest liczbą całkowitą. Zmienna *X* jest ukonkretniana sumą wszystkich liczb całkowitych od 1 do *N*, więc możemy predykatu tego użyć na przykład tak:

```
?- suma_do(5, X).
X = 15 ;
no
```

gdyż 1+2+3+4+5 równe jest 15. Oto program:

```
suma_do(1, 1) :- !.
suma_do(N, Razem) :-
    N1 is N - 1,
    suma_do(N1, Razem1),
    Razem is Razem1 + N.
```

Definicja ta jest rekurencyjna, warunkiem końcowym jest przekazanie liczby równej 1 — wtedy odpowiedzią też jest 1. Druga klauzula wykorzystuje rekurencyjne wywołanie celu *suma_do*, przy czym jako argument przekazywana jest liczba o jeden mniejsza od pierwotnej. Następny cel wywołany rekurencyjnie znowu będzie miał ten argument o jeden mniejszy, i tak dalej, aż zostanie osiągnięty warunek brzegowy (przy założeniu, że w pierwszym wywołaniu pierwszy argument nie był mniejszy niż 1).

Ciekawym aspektem tego programu jest sposób uwzględnienia dwóch przypadków: kiedy liczba jest jedynką i kiedy jest czymkolwiek innym. Kiedy definiowaliśmy predykaty do obsługi list, łatwo było rozróżnić dwa przypadki: kiedy lista jest pusta i kiedy ma postać *[A|B]*. W przypadku liczb rzecz nie jest tak prosta, gdyż nie można podać wzorca pasującego tylko do liczb innych niż 1. W tym przykładzie opisaliśmy sposób radzenia sobie z wartością 1 i dla wszystkich innych wartości przygotowaliśmy zwykłą zmienną. Wiedząc, jak Prolog przeszukuje bazę danych, jesteśmy pewni, że najpierw będzie starał się uzgodnić klauzulę z liczbą 1, dopiero potem drugą. Wobec tego druga reguła powinna być używana tylko wtedy, gdy liczba jest jedynką.

Jednak na tym nie koniec, gdyż Prolog może nawracać i ponownie dopasowywać regułę nawet wtedy, gdy zastosowano pierwszą jej klauzulę. Wówczas okazuje się, że druga reguła spełnia wszystkie zadane warunki. Z punktu widzenia samego systemu obie reguły mogą być zastosowane do celu *suma_do(1, X)*, wobec czego musimy Prolog poinformować, że jeśli raz została zastosowana pierwsza reguła, to druga nie powinna być stosowana. Jednym ze sposobów wykonania tego jest wstawienie odcięcia w pierwszej regule, co zrobiliśmy w naszym przykładzie. Prolog już „wie”, że jeśli doszedł do odpowiedniego miejsca tej reguły, nie wolno mu ponownie wracać do decyzji, której klauzuli *suma_do* użyć. Zdarzy się tak tylko wówczas, gdy liczba wynosi 1. Przyjrzyjmy się, jak wygląda spełnianie celu. Jeśli wywołamy *suma_do(1, X)* następująco:

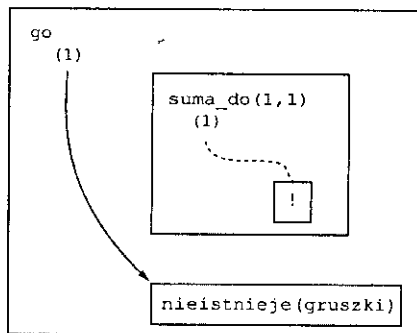
```
go :- suma_do(1, X), nieistnieje(gruszki).
?- go.
```

i cel *nieistnieje(gruszki)* zawiedzie, to mamy do czynienia z sytuacją jak na rysunku 4.6.

Kiedy Prolog stara się ponownie spełniać cele w odwrotnej kolejności, okazuje się, że dwa z nich nie mogą być już spełnione inaczej, gdyż ścieżka została zmieniona. Wobec tego nie będą sprawdzane alternatywne sposoby spełnienia celu *suma_do(1, X)*.

Rysunek 4.6.

Cel
nieistnieje(gruszki)
zawodzi



Ćwiczenie 4.1. Co się stanie, jeśli pominiemy odcięcie i nawracanie wróci do celu `suma_do`? Czy zostaną wygenerowane inne wyniki i dlaczego? Jeśli tak, to jakie?

W ostatnim przykładzie pokazaliśmy, jak użyć odcięcia, by zmusić Prolog do oczekiwanego przez nas zachowania nawet wówczas, gdy nie możemy uwzględnić wszystkich możliwych przypadków za pomocą wzorców w głowach reguł. Bardziej typową sytuacją, kiedy nie możemy wskazać odpowiednich wzorców, jest konieczność podania dodatkowych warunków w formie celów Prologu, które będą decydowały o odpowiedniości reguły. Przyjrzyjmy się następującej alternatywnej formie powyższego przykładu:

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

W tym wypadku informujemy, że pierwsza reguła ma być wybrana, jeśli podana liczba jest nie większa od jedynki. Jest to warunek nieco lepszy niż poprzednio, gdyż jeśli podana zostanie liczba 0 lub ujemna, program i tak zwróci wynik, a nie wpadnie w nieskończoną pętlę. Jeśli warunek jest prawdziwy, natychmiast uzyskamy wynik 1. Jeśli warunek ten nie jest spełniony, chcemy sprawdzić drugą regułę. Musimy poinformować Prolog, że jeśli ustalono, że $N \leq 1$, nie należy już sprawdzać tego warunku przy ewentualnym nawracaniu. Do tego właśnie potrzebne jest odcięcie.

Istnieje ogólna zasada mówiąca, że użycie odcięcia w celu poinformowania Prologu o tym, że jedyna odpowiednia reguła została już wybrana, może być zastąpione predykatem `not`. Jest to predykat wbudowany, czyli jego definicja znana jest systemowi już w chwili rozpoczynania sesji z Prologiem. Predykatów wbudowanych można używać w swoich programach bez konieczności ich definiowania; predykaty te dokładniej opiszemy w rozdziale 6. Predykat `not` jest zdefiniowany tak, że cel `not(X)` zostanie uzgodniony jedynie wtedy, kiedy `X` jako cel Prologu zawiedzie. Przykład zastąpienia odcięcia predykatem `not` mamy poniżej — przekształcamy pokazane wcześniej definicje `suma_do`:

```
suma_do(1,1).
suma_do(N,Razem) :-
    \+(N=1),
    N1 is N - 1,
    suma_do(N1,Razem1),
    Razem is Razem1 + N.
```

oraz

```
suma_do(N,1) :- N <= 1.
suma_do(N,R) :-
    \+(N <= 1),
    N1 is N - 1,
    suma_do(N1,R1),
    R is R1 + N.
```

Prolog zawiera zresztą predykaty wbudowane pozwalające zastąpić oba wystąpienia `not`. Zamiast `\+(N=1)` możemy napisać `N\=1`, a zamiast `\+(N<=1)` możemy zapisać `N>1`. W przypadku ogólnym nie zawsze będziemy mogli tak przekształcić zadane warunki z `not`.

Do dobrego stylu programowania należy zastępowanie odcięć predykatami `not`. Wynika to stąd, że programy zawierające dużo odcięć są trudniejsze do zrozumienia. Jeśli wszystkie odcięcia zostaną zastąpione predykatami `not`, program będzie czytelniejszy. Jednak definicja `not` zakłada próbę uzgodnienia celu, wobec czego, jeśli mamy program w ogólnej postaci:

```
A :- B, C.
A :- \+B, D.
```

Prolog może starać się spełnić `B` dwukrotnie. Najpierw robi to przy analizie pierwszej klauzuli, a jeśli potem dojdzie do nawracania i będzie wykonywana druga klauzula, ponownie nastąpi próba spełnienia `B`, aby sprawdzić, czy można spełnić `not(B)`. Działanie takie może być pracochłonne, jeśli warunek `B` jest skomplikowany. Problemu tego nie wystąpiłoby, gdyby definicja programu wyglądała tak:

```
A :- B, !, C.
A :- D.
```

Tak więc czasami trzeba dobrze się zastanowić, czy ważniejsza jest elegancja programu, czy szybkość jego działania. Dyskusja o szybkości działania wiedzie nas z kolei wprost do ostatniego przykładu użycia odcięcia, które miało uniemożliwić zmianę wybranej reguły. Przyjrzyjmy się definicji `append` z rozdziału 3.:

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).
```

Jeśli tego predykatu używamy zawsze do złączania dwóch znanych list, to strata czasu jest nawracanie w celu sprawdzenia drugiej klauzuli przy celu typu `append([], [a,b,c,d], X)`, gdyż druga klauzula w tym wypadku zawsze zawiedzie. Wiemy, że w takim wypadku pierwsza lista to `[]`, kiedy odpowiednia jest jedynie pierwsza klauzula. Naszą wiedzę możemy przekazać Prologowi za pomocą odcięcia. Ogólnie rzecz biorąc, implementacje Prologu są w stanie lepiej wykorzystać pamięć, jeśli mają tego typu informacje pozwalające pomijać zbędne dane o wyborach, przydatne przy nawracaniu. Zatem naszą definicję możemy zmienić następująco:

```
append([],X,X) :- !.
append([A|B].C,[A|D]) :- append(B,C,D).
```

Jeśli używamy naszego ograniczonego predykatu `append`, otrzymujemy takie same rozwiązania jak poprzednio, zaś nieco poprawia się szybkość działania i zmniejsza zużycie pamięci. Za to przestają działać inne sposoby wywołania `append`, czym zajmujemy się w podrozdziale „Niebezpieczeństwa wynikające ze stosowania odcięcia”.

Użycie odcięcia z predykatem `fail`

Drugie ważne zastosowanie odcięcia to użycie go wraz z predykatem wbudowanym `fail`. Predykat ten nie ma żadnych argumentów, co oznacza, że jego uzgodnienie lub to, że zawiedzie, są niezależne od jakichkolwiek zmiennych. Tak naprawdę `fail` jest zdefiniowany tak, że zawsze zawodzi i powoduje nawracanie. Jest to sytuacja podobna do próby spełnienia celu, którego predykat nie ma odpowiadających mu faktów ani reguł. Kiedy `fail` pojawia się po odcięciu, normalne działanie nawracania jest zmodyfikowane przez odcięcie. Wbrew pozorom, taka kombinacja jest bardzo przydatna w praktyce programistycznej.

Zastanówmy się, jak moglibyśmy użyć takiego połączenia w programie wyliczającym należne podatki. Jedno, co nas interesuje, to czy dana osoba jest „przeciętnym podatnikiem”; wtedy wyliczenia będą proste i nie będzie zbyt wielu różnych przypadków do rozważenia. Zdefiniujmy predykat `przecietny_podatnik`, taki, że `przecietny_podatnik(X)` oznacza, że X jest właśnie takim przeciętnym podatnikiem. Na przykład Fred Bloggs, żonaty i mający dwójkę dzieci, pracujący w fabryce rowerów, może być uważany za przeciętnego. Z kolei dyrektor koncernu naftowego zarabia zbyt dużo, a student zbyt mało, aby tak samo wyliczyć im podatek. Zaczniemy od przyjrzenia się możliwym przypadkom szczególnym. Przypadkiem szczególnym mogą być cudzoziemcy, mający zobowiązania także względem swojego kraju pochodzenia. Jeśli więc taka osoba jest nawet we wszech miar przeciętna, to jako podatnika musimy potraktować ją specjalnie. Zaczniemy od napisania odpowiednich reguł:

```
przecietny_podatnik(X) :- obcokrajowiec(X), fail.
przecietny_podatnik(X) :- ...
```

Wprawdzie to jeszcze nie wszystko, ale już widać, że chcemy zapisać, że „jeśli X jest obcokrajowcem, to cel `przecietny_podatnik(X)` zawodzi”. Druga reguła dotyczy ogólnych kryteriów bycia przeciętnym podatnikiem. Jeśli jednak zadamy zapytanie

```
?- przecietny_podatnik(widslawip).
```

o cudzoziemca określanego jako `widslawip`, cel `obcokrajowiec` z pierwszej klauzuli zostanie uzgodniony. Następnie `fail` spowoduje nawracanie i przy próbie ponownego uzgadniania celu `przecietny_podatnik` Prolog znajdzie drugą klauzulę i zacznie do `obcokrajowca` stosować kryteria ogólne. Istnieje duża szansa, że osoba ta spełni warunki, wobec czego zostanie niesłusznie uznana za przeciętnego podatnika. Wobec tego nasza pierwsza klauzula w ogóle nie odrzuci naszego przyjaciela.

Dlaczego? Wynika to stąd, że podczas nawracania Prolog stara się ponownie spełniać wszystkie cele, które zostały uzgodnione. W szczególności badane będą inne możliwe sposoby uzgodnienia celu

```
przecietny_podatnik(widslawip).
```

Aby to zachowanie wyłączyć, musimy odrzucić pozostałe wybory przed użyciem `fail`. W tym celu wystarczy przed `fail` użyć odcięcia. Nieco pełniejszą definicję przeciętnego podatnika pokazano poniżej:

```
przecietny_podatnik(X) :- obcokrajowiec(X), !, fail.
przecietny_podatnik(X) :-
    malzonek(X,Y),
    przychod_brutto(Y,Przychod),
    Przychod > 3000,
    !, fail.
przecietny_podatnik(X) :-
    przychod_brutto(X,Przychod),
    2000 < Przychod, 20000 > Przychod,
    przychod_brutto(X,Y) :-
        otrzymuje_rente(X,R),
        R < 5000,
        !, fail.
    przychod_brutto(X,Y) :-
        pobory_brutto(X,Z),
        przychody_z_inwestycji(X,W),
        Y is Z+W.
    przychody_z_inwestycji(X,Y) :- ...
```

W powyższym programie występuje kilka połączeń „odcięcia” i `fail`. W drugiej klauzuli predykatu `przecietny_podatnik` odrzucamy osoby, których małżonek zarabia więcej niż ustalona kwota. Poza tym w definicji predykatu `przychod_brutto` w pierwszej regule zakładamy, że osoby otrzymujące rentę nie większą niż ustalona wielkość, niezależnie od wszelkich innych okoliczności, traktowane są jako nie mające dochodów.

Interesującym zastosowaniem kombinacji odcięcie + `fail` jest definicja predykatu `\+`. W większości implementacji Prologu predykat ten jest wbudowany, ale warto zastanowić się, jak można go zdefiniować. Żądamy, aby cel `\+P`, gdzie P jest innym celem, był uzgodniony wtedy, gdy P zawodzi. Nie jest to do końca zgodne ze zdrowym rozsądkiem, gdyż niebezpieczne jest założenie, że jeśli czegoś nie możemy udowodnić, to nie jest to prawdą. Niemniej definicja wygląda tak:

```
\+P :- call(P), !, fail.
\+P.
```

Definicja `\+` zawiera wywołanie argumentu P jako celu przy użyciu predykatu `call`. Predykat `call` po prostu traktuje swój argument jako cel i stara się go uzgodnić. Chcemy, aby pierwsza klauzula była używana, gdy zachodzi P , a druga w przeciwnym razie. Wobec tego mówimy, że jeśli możliwe jest spełnienie celu `call(P)`, należy uznać cel `not` za niespełniony. Istnieje też możliwość, że Prolog nie będzie mógł uzgodnić `call(P)` — wtedy nie dojdziemy do odcięcia. Skoro `call(P)` zawiedzie, nawracanie spowoduje znalezienie drugiej klauzuli i cel `\+P` zostanie uzgodniony, jeśli niemożliwe będzie uzgodnienie P .

Wykazanie
Zarządca i Bankowcy
ul. Armii Krajowej 4, 00-150 Kraków
tel. (0-22) 630-05-77, tel. fax (0-22) 631-33-47
Wpis do Rejestru MKN nr 55 z dnia 11.05.1995r.
Konto BOS Orlanów 1540115-12607-27066-00
NIP 471.17.59.189 REGON 350814846

Tak jak w przypadku pierwszego użycia odcięcia, parę odcięcie + fail można zastąpić predykatem not. Wiąże się to z większymi modyfikacjami programu niż poprzednio, ale nie powoduje takiego pogorszenia szybkości. Jeśli mielibyśmy zmodyfikować nasz predykat przecietny_podatnik, zaczęlibyśmy tak:

```
przecietny_podatnik(X) :-
    \+obcokrajowiec(X),
    \+((malzonek(X,Y), przychod_brutto(Y,Przychod), Przychod>3000)),
    przychod_brutto(X,Przychod1).
```

Zauważmy, że w tym przykładzie w \+ zamknęliśmy całą koniunkcję celów. Aby pokazać jednoznacznie, że przecinki oznaczają koniunkcję, a nie oddzielają argumenty, całość ujęliśmy w dodatkowy nawias.

Kończenie generowania możliwych rozwiązań i ich sprawdzanie

Teraz przechodzimy do omówienia ostatniego sposobu użycia odcięcia: do kończenia generowania ciągu rozwiązań i następnie ich sprawdzania. Bardzo często fragmenty programu działają zgodnie z następującym modelem ogólnym: istnieje sekwencja celów, które mogą być uzgadniane w różnoraki sposób, i które generują za pomocą nawracania wiele możliwych rozwiązań. Następnie wszystkie cele są sprawdzane: czy rozwiązanie spełnia pewne przyjęte założenia. Jeśli cele zawiodą, nawracanie powoduje sprawdzanie kolejnego rozwiązania, które znowu jest sprawdzane, i tak dalej. Cały proces kończy się, kiedy znalezione zostanie rozwiązanie spełniające zadane warunki (sukces) lub kiedy niemożliwe jest znalezienie już jakichkolwiek rozwiązań. Cele generujące wszystkie możliwe rozwiązania nazwiemy „generatorem”, zaś cele sprawdzające, czy kolejne rozwiązania są do przyjęcia, nazwiemy „testerem”. Rozpatrzymy następujący przykład: program grający w kółko i krzyżyk. Kilka słów wyjaśnienia należy się osobom nie znającym tej gry (są takie?): dwaj gracze robią kolejno ruchy na planszy o wymiarach 3×3. Jeden z nich zaznacza pola kółkami (o), drugi krzyżykami (x). Oto przykładowy wygląd planszy podczas gry:

		o	
	x		o
	x		

Planszę będziemy zapisywać jako 9-składnikową strukturę b, stałymi x i o będziemy oznaczali ruchy graczy. Jeśli pole jest puste, oznaczmy je literą e. Elementy planszy można uporządkować wierszami od lewej do prawej, tak więc przykładowa plansza pokazana powyżej będzie zapisana jako b(e,o,e,e,x,o,e,x,e). Celem gry jest uzyskanie trzech kółek lub trzech krzyżyków ustawionych w jednym rzędzie (poziomo, pionowo lub po przekątnej) wcześniej niż drugi gracz. Cel ten można uzyskać na osiem sposobów. Rzędy planszy możemy zapisać jak pokazano poniżej, przy czym

linia(B,X,Y,Z) ukonkretnia trzy argumenty, X, Y i Z, trzema kwadratami tworzącymi na planszy linię:

```
linia(b(X,Y,Z,_,_,_,_,_,_),X,Y,Z).
linia(b(_,X,Y,Z,_,_,_,_),X,Y,Z).
linia(b(_,_,X,Y,Z,_,_,_,_),X,Y,Z).
linia(b(X,_,_,Y,_,_,Z,_,_),X,Y,Z).
linia(b(_,X,_,_,Y,_,_,Z,_,_),X,Y,Z).
linia(b(X,_,_,_,Y,_,_,_,Z,_,_),X,Y,Z).
linia(b(X,_,_,_,_,Y,_,_,_,Z,_,_),X,Y,Z).
```

Od razu powinno być widoczne, że zmienna X oznacza pole planszy, zaś x to stała będąca nazwą krzyżyków.

```
ruch_wymuszony(Plansza,Pole) :-
    linia(Kwadraty),
    zagrozenie(Kwadraty,Plansza,Pole),
    !.
```

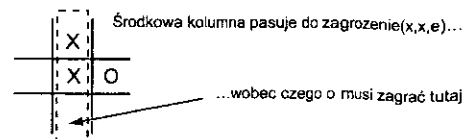
Program działa z punktu widzenia gracza używającego kółek. Predykat ruch_wymuszony odpowiada na pytanie „czy muszę wykonać ruch na narzuconym polu?” Dzieje się tak, jeśli gracz nie może natychmiast wygrać (tym się tu nie zajmujemy w ogóle), ale przeciwnik ma szansę wygrać w następnym ruchu.

```
ruch_wymuszony(Plansza) :-
    linia(Plansza,X,Y,Z),
    zagrozenie(X,Y,Z),
    !.
```

Zauważmy, że „odcięcie” oznacza tutaj, że pojedyncze zagrożenie jest wystarczające do wymuszenia ruchu. Teraz musimy ustalić, który rząd stanowi zagrożenie. Jest taki rząd, w którym mamy dwa pola z krzyżykami i trzecie puste. Takie zagrożone układy są trzy:

```
zagrozenie(e,x,x).
zagrozenie(x,e,x).
zagrozenie(x,x,e).
```

Na przykład w sytuacji pokazanej na rysunku poniżej grający kółkiem jest zmuszony do umieszczenia kółka we wskazanym polu, gdyż inaczej przeciwnik będzie mógł wygrać, wypełniając kolumnę drugą. Program stara się znaleźć rząd, w którym już dwa pola są zajęte przez przeciwnika, a trzecie jest jeszcze puste. Jeśli rząd taki zostanie znaleziony, zgłaszany jest ruch wymuszony.



W klauzuli `run_wymuszony` cel `linia(Plansza,X,Y,Z)` służy jako „generator” możliwych rzędów. Cel ten może zostać uzgodniony na różne sposoby, zmienne `X`, `Y` i `Z` są ukonkretniane różnymi możliwymi rzędami. Kiedy proponowany zostanie jeden z takich rzędów, trzeba sprawdzić, czy istnieje zagrożenie, że przeciwnik spróbuje taki rząd wypełnić. Do tego służy cel „tester”, `zagrozenie(X,Y,Z)`, który stara się znaleźć jeden z trzech wzorców stanowiących zagrożenie. Zasady działania programu są bardzo proste. Najpierw `linia` wskazuje jeden z rzędów, predykat `zagrozenie` sprawdza, czy dany rząd jest zagrożony. Jeśli tak, uzgodniony zostaje cel `run_wymuszony`. W przeciwnym razie zachodzi nawracanie i `linia` wskazuje następny rząd. Ten rząd jest sprawdzany tak samo, może znowu wystąpić nawracanie i tak dalej; kiedy zabraknie już rzędów do sprawdzenia (zawiedzie `linia`), zawiedzie też cały cel `run_wymuszony` — gracz może wykonać dowolny ruch.

A co się stanie, jeśli nasz program, będący częścią większego systemu, znajdzie `run_wymuszony`? Założymy, że gdzieś dalej jakiś cel zawodzi i nawracanie dochodzi do celu `run_wymuszony`. Nie chcemy, aby predykat `linia` znowu generował dalsze rzędy do sprawdzania. Jeśli jakiś zagrożenie zostało znalezione, to nic więcej już nie daje się zrobić: jeśli nie można zagrożeniu zaradzić, to i tak gra jest przegrana. Zwykle jednak innych rozwiązań nie będzie, `run_wymuszony` będzie na darmo sprawdzał kolejne rzędkę, aby w końcu zawieść. Wiemy też, że nawet jeśli istnieje inne zagrożenie, to i tak jego znajomość nic nie daje, jeśli nie można zaradzić zagrożeniu znalezionemu wcześniej. Możemy nakazać systemowi przerwać szukanie alternatywnych rozwiązań wstawiając odcięcie na końcu klauzuli. Wynikiem jest zamrożenie ostatnio uzgodnionego rzędu. Wstawienie odcięcia jest równoważne ze stwierdzeniem: „kiedy szukam ruchów wymuszonych, istotne jest tylko pierwsze rozwiązanie”.

Przyjrzyjmy się jeszcze jednemu przykładowi programu działającego w myśl zasady „generuj i sprawdzaj”. W rozdziale 2. omawialiśmy dzielenie całkowitoliczbowe. Większość wersji Prologu domyślnie je zawiera, ale można też napisać odpowiedni program przy użyciu jedynie dodawania i mnożenia:

```
dzielenie(N1,N2,Wynik) :-
    liczba_calkowita(Wynik),
    iloczyn1 is Wynik*N2,
    iloczyn2 is (Wynik+1)*N2,
    iloczyn1 =:= N1, iloczyn2 > N2,
    !.
```

Używamy predykatu `liczba_calkowita` (zdefiniowanego wcześniej w tym rozdziale), aby generować liczbę `Wynik` będącą wynikiem „dzielenia” `N1` przez `N2`. Na przykład wynikiem dzielenia 27 przez 6 jest 4, gdyż 4×6 jest mniejsze lub równe 27, zaś 5×6 jest większe od 27.

W regule predykatu `liczba_calkowita` używamy jako „generatora”, zaś dalsza część reguły to „tester”. Z góry wiemy, że jeśli mamy konkretne `N1` i `N2`, cel `dzielenie(N1,N2,Wynik)` nie zawiedzie tylko dla jednej wartości `Wynik`. Wprawdzie `liczba_calkowita` może generować dowolnie wiele liczb, lecz tylko jedna z nich spełni zadany warunek. Możemy tę naszą wiedzę zapisać poprzez umieszczenie na końcu reguły odcięcia. W ten sposób zaznaczamy, że jeśli wygenerowany zostanie `Wynik` spełniający zadane warunki, jest on wynikiem dzielenia i nie należy już wykonywać żadnych prób. W szczególności,

nigdy nie musimy ponownie rozważać wyborów związanych z szukaniem reguł dzielenia, liczba_calkowita i tak dalej. Kiedy znalezione zostanie jedyne rozwiązanie, nie ma powodu robić nic więcej. Gdybyśmy nie wstawili odcięcia, nawracanie mogłoby ponownie wymusić generowanie liczb przez `liczba_calkowita`, ale żadna z tych liczb nie spełniałaby warunków — nie byłaby wynikiem dzielenia, więc proces nigdy by się nie skończył.

Niebezpieczeństwa wynikające ze stosowania odcięcia

Widzieliśmy już, że musimy uwzględniać sposób przeszukiwania przez Prolog bazy danych, aby prawidłowo dobrać kolejność klauzul. W przypadku stosowania odcięcia trzeba tę wiedzę jeszcze bardziej pogłębić, bo o ile odcięcie zastosowane w jednej regule nie wpłynie na jej działanie lub nawet je poprawi, to w innej regule to samo odcięcie spowoduje niespodziewane efekty w przypadku innego niż założony sposobu jej użycia. Przyjrzyjmy się zmodyfikowanemu predykatowi `append`:

```
append([],X,X) :- !.
append([_],_,_) :- append(B,C,D).
```

Jeśli mamy cele:

```
append([a,b,c],[d,e],X)
```

czy

```
append([a,b,c],X,Y)
```

odcięcie jest jak najbardziej na miejscu. Jeśli pierwszy argument celu ma już ustaloną wartość, odcięcie jedynie potwierdza, że pierwsza klauzula będzie użyteczna jedynie dla wartości `[]`. Jeśli jednak cel ma postać:

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

to zostanie on dopasowany do głowy pierwszej reguły, więc otrzymamy

```
X=[], Y=[a,b,c]
```

i natknijemy się na odcięcie. Spowoduje to zamrożenie wszystkich dokonanych wyborów, więc na pytanie o następne rozwiązanie uzyskamy odpowiedź negatywną, no, choć zapytanie ma inne rozwiązania.

Oto inny ciekawy przykład: co się stanie, jeśli w nieprzewidziany sposób użyjemy reguły zawierającej odcięcie. Zdefiniujmy predykat `ilu_rodzicow` zawierający informację o tym, ile rodziców ma dana osoba. Predykat ten zdefiniujemy tak:

```
ilu_rodzicow(adam,0) :- !.
ilu_rodzicow(ewa,0) :- !.
ilu_rodzicow(X,2).
```

Zatem *adam* ani *ewa* nie mają rodziców, zaś wszyscy inni ludzie mają ich dwoje. Jeśli teraz stosujemy nasz predykat `ilu_rodzcicow` do określania liczby rodziców poszczególnych osób, wszystko działa prawidłowo:

```
?- ilu_rodzcicow(ewa,X).
X=0 ;
no
?- ilu_rodzcicow(jan,X).
X=2 ;
no
```

Odcięcie jest niezbędne, aby nawracanie nie spowodowało dotarcia do trzeciej klauzuli, jeśli osobą jest *adam* lub *ewa*. Jednak co będzie, jeśli zechcemy sprawdzić, czy dana osoba ma pewną liczbę rodziców?

```
?- ilu_rodzcicow(ewa,2).
yes
```

Nietrudno samemu dojść, dlaczego tak się dzieje. Po prostu jest to konsekwencja sposobu przeszukiwania bazy danych przez Prolog. Nasza implementacja „przeciwego wypadku” za pomocą odcięcia po prostu nie działa prawidłowo. Z problemem tym można sobie poradzić w dwojaki sposób:

```
ilu_rodzcicow(adam,N) :- !, N = 0.
ilu_rodzcicow(ewa,N) :- !, N = 0.
ilu_rodzcicow(X,2).
```

lub

```
ilu_rodzcicow(adam,0).
ilu_rodzcicow(ewa,0).
ilu_rodzcicow(X,2) :- \+(X=adam), \+(X=ewa).
```

Oczywiście nadal nie uzyskamy oczekiwanej odpowiedzi, jeśli będziemy chcieli wyliczyć wszystkie możliwości:

```
?- ilu_rodzcicow(X,Y).
```

Morał jest następujący:

Jeśli chcemy zastosować odcięcie w celu uzyskaniażądanego wyniku dla celów w jakiejś postaci, nie ma żadnej gwarancji, że uzyskamy sensownie odpowiedzi na cele w innej postaci.

Wynika stąd, że odcięcia można używać bezpiecznie tylko wtedy, gdy wie się, jak odpowiednie reguły będą używane. Jeśli zasady użycia tych reguł się zmieniają, trzeba przejrzeć wszystkie zastosowania odcięcia.

Rozdział 5.

Wejście i wyjście

Jak dotąd, informacje do programów wprowadzaliśmy korzystając z zapytań, zaś jedyną metodą określania wartości zmiennych było spełnienie celu, po którym Prolog wyświetlał odpowiedź w postaci „X = odpowiedź”. Zwykle takie bezpośrednie korzystanie zapytań wystarcza do zapewnienia prawidłowej pracy programu. Często jednak chcemy napisać program, który sam zainicjuje komunikację z użytkownikiem.

Założmy, że mamy bazę danych zawierającą kalendarium XVI wieku, w którym zapisane są daty i krótkie opisy. Na razie daty będziemy zapisywać jako liczby całkowite, zaś opisy jako listy atomów. Niektóre atomy ujmemy w cudzysłowy, gdyż zaczynają się wielkimi literami, a nie chcemy, aby były potraktowane jako zmienne:

```
zdarzenie(1505,['Euklides',przetlumaczony,na,lacine]).
zdarzenie(1510,['Reuchlin',kontra,'Pfefferkorn']).
zdarzenie(1523,['Chrystian','II',ucieka,z,'Danii']).
```

Jeśli teraz chcemy poznać historię jakiegoś zdarzenia, wystarczy zadać zapytanie:

```
?- zdarzenie(1505,X).
```

i Prolog odpowie:

```
X=[Euklides,przetlumaczony,na,lacine].
```

Zapisywanie nagłówków zdarzeń historycznych jako list atomów umożliwia potem wyszukiwanie dat, kiedy interesujące zdarzenia miały miejsce. Możemy na przykład zdefiniować predykat `kiedy`. Cel `kiedy(X,Y)` zostanie uzgodniony, jeśli `X` jest rokiem, w którym zaszło zdarzenie `Y`:

```
kiedy(X,Y) :- zdarzenie(Y,Z), member(X,Z).
?- kiedy('Chrystian',D).
D = 1523
```

Wygodniej byłoby w ogóle nie zadawać zapytań w opisany sposób, ale napisać program, który poprosi o datę i wyświetli odpowiedni nagłówek zdarzenia. Wtedy nagłówki mogą być dowolnie zapisywane. Prolog zawiera szereg predykatów wbudowanych, pozwalających wyświetlać teksty oraz pobierać je od użytkownika z klawiatury, a następnie ukonkretniać zmienne wczytanymi danymi. Tak oto program prologowy może prowadzić dialog z użytkownikiem: pobierać dane wejściowe i wyświetlać dane wyjściowe.

Kiedy program oczekuje na wprowadzenie jakichś danych, nazywamy to *czytaniem* danych wejściowych. Kiedy program jakieś dane wyświetla, nazywamy to *pisaniem*.¹

W tym rozdziale omówimy różne metody czytania i pisania. Jeden z naszych przykładów wyświetli nagłówki z historycznej bazy danych, rozdział zakończymy prezentacją programu, który będzie odczytywał zdania w języku naturalnym i następnie przekształcał je na stałe, które mogą być przetwarzane przez inne programy. Ten program konwertujący, czytaj, może być częścią programu analizującego język naturalny. Tego typu programy omówimy dalej, w rozdziale 9.

Cały czas trzeba pamiętać, że predykaty wejścia i wyjścia podane w standardzie Prologu różnią się pod pewnymi względami od predykatów, które używane były w poprzednich wydaniach niniejszej książki, a niejednokrotnie także od predykatów dostępnych w poszczególnych implementacjach języka. W przypisach poinformujemy Czytelników, jak owe predykaty mają się do starszych wersji. W dodatku C opisano, jak ułatwić sobie pisanie programów w standardzie Prologu nawet w systemach nie całkiem z tym standardem zgodnych.

Czytanie i pisanie termów

Czytanie termów

Predykat specjalny `read` odczytuje następny term wprowadzony z klawiatury. Za tym termem musi się znajdować kropka i znak niedrukowalny, jak *spacja* lub *Enter*. Jeśli `X` jest nieukonkretnioną, cel `read(X)` odczyta następny term i ukonkretni zmienną `X` jego wartością.

Jeśli argument jest ukonkretniony w chwili wywołania tego celu, odczytany zostanie następny term i będzie dopasowywany do argumentu `read`. Uzgodnienie celu zależy od tego, czy ta unifikacja się powiedzie. Predykat `read` można uzgodnić tylko raz. Przy każdej kolejnej próbie uzgodnienia go, predykat ten już zawodzi.

Używając predykatu `read`, możemy napisać program drukujący nagłówki historyczne z bazy danych:

```
zaczynamy1(Zdarzenie) :- read(Data), zdarzenie(Data, Zdarzenie).
```

Jeśli zadamy zapytanie:

```
?- zaczynamy1(X).
```

¹ Jest to oczywiście prawda w odniesieniu do aplikacji konsolowych. Trzeba jednak pamiętać, że dostępne obecnie interpretry Prologu (a także kompilatory) umożliwiają normalną interakcję za pośrednictwem okienek dialogowych. Systemy Prologu mogą zawierać predykaty do tworzenia i obsługi okienek, zwykle też jest możliwe łączenie się z innymi językami lepiej nadającymi się do oprogramowania interfejsu użytkownika, jak np. C++ — *przyjp. thum*.

Prolog będzie starał się uzgodnić cel `read`, wobec czego będzie czekał na odpowiedź. Wpiszmy na przykład

```
1523.
```

Pamiętajmy, że po 1523 musimy dopisać kropkę (.) i wcisnąć *Enter*. Cel `read` zostanie uzgodniony, `Data` będzie ukonkretniona wartością 1523. W bazie odszukane zostanie zdarzenie spod takiej daty, zaś zmienna `Zdarzenie` zostanie ukonkretniona listą atomów i Prolog odpowie:

```
X = ['Chrystian', 'II', ucieka, z, 'Danii'] ?
```

Jeśli zażądamy dalszych rozwiązań, możemy zobaczyć inne zdarzenia z roku 1523. Jednak `read` zawiedzie przy próbie ponownego uzgodnienia podczas nawracania, więc już nie będziemy pytani o inne daty.

Predykat `zaczynamy1` stanowi typowe przybliżenie się do programu wygodnego w użyciu, ale teraz podstawowym problemem jest nieelegancki sposób pokazywania zdarzeń. Potrzebny jest nam sposób, który pozwoli wymusić na Prologu pokazywanie danych w pożądanej postaci.

Pisanie termów

Zapewne najprzydatniejszym predykatem do wyświetlania termów na monitorze jest wbudowany predykat `write`. Jeśli zmienna `X` jest ukonkretniona termem, `write(X)` spowoduje wypisanie tego termu na monitorze. Jeśli `X` jest nieukonkretniona, pokazana zostanie zmienna z niepowtarzalnym numerem, na przykład `_253`. Tak jak w przypadku `read`, `write` udaje się uzgodnić tylko jeden raz.

Użycie `write` do pokazywania zdarzeń historycznych nie jest najlepszym pomysłem, gdyż `write` wyświetla dane w postaci standardowej dla list, z nawiasami kwadratowymi i przecinkami. Z drugiej jednak strony, jeśli zastosujemy `write` do poszczególnych składników list, to możemy uzyskać wynik bliższy naszym oczekiwaniom.

Omówimy jeszcze jeden predykat, potem pokażemy pierwszy przykład użycia `write`. Predykat wbudowany `nl` używany jest do wymuszenia przejścia do nowego wiersza. Tak jak `write`, `nl` można uzgodnić tylko raz.

Podczas drukowania listy dobrze jest pokazywać jej elementy tak, aby były one łatwe do zrozumienia. Listy zawierające inne listy zagnieżdżone są szczególnie trudne do odczytania, zwłaszcza jeśli mają jeszcze wewnątrz struktury. Zdefiniujemy predykat `pp`, taki, że cel `pp(X, Y)` wydrukuje listę (z argumentu `X`) w wygodny sposób. Nazwa `pp` to skrót od „pretty print” (elegancki wydruk). Drugi argument `pp` omówimy później.

Każdy autor programów elegancko drukujących dane ma własne poglądy dotyczące czytelności wyniku. Dla prostoty zastosujemy tutaj metodę, w której elementy listy są drukowane w kolumnie pionowej. Jeśli element sam jest listą, jego elementy są drukowane w kolumnie przesuniętej w prawo. Jest to wydruk podobny do diagramu wioronośli (omawianego w rozdziale 3.) położonego na boku. Na przykład lista `[1, 2, 3]` zostanie pokazana jako

```
1
2
3
```

zaś lista [1.2.[3.4].5.6] jako

```
1
2
3
4
5
6
```

Warto zauważyć, że usunęliśmy rozdzielające elementy przecinki i nawiasy kwadratowe. Jeśli element listy jest strukturą, potraktujemy go tak, jakby był atomem. W ten sposób nie musimy „zagłębiać” się w struktury i formatować ich treści.

Najpierw musimy nauczyć się stosować wcięcia używane w przypadku list zagnieżdżonych. Jeśli będziemy rejestrować głębokość zagnieżdżenia danej listy, powinniśmy przed elementami tej listy wyświetlić pewną liczbę spacji zależną. Zdefiniujemy nowy predykat, `spacje`, który wyświetli żądane wcięcie, pokazując spację tyle razy, ile nakażemy mu w argumencie:

```
spacje(0) :- !.
spacje(N) :- write(' '), N1 is N - 1, spacje(N1).
```

Oto program realizujący opisany tu elegancki wydruk:

```
pp([H|T],I) :- !, J is I+3, pp(H,J), ppx(T,J), n1.
pp(X,I) :- spacje(I), write(X), n1.
```

```
ppx([],_).
ppx([H|T],I) :- pp(H,I), ppx(T,I).
```

Widać, że drugi argument `pp` to licznik kolumn. Cel główny drukujący listę może wyglądać zatem tak:

```
..., pp(L,0), ...
```

czyli licznik kolumn inicjalizujemy wartością 0. Pierwsza klauzula `pp` obsługuje przypadek szczególny: kiedy pierwszy argument `pp` jest listą. W takim wypadku musimy przygotować nową kolumnę zwiększając licznik kolumn o pewną wielkość (tutaj: 3). Następnie musimy „elegancko wydrukować” głowę listy, gdyż to może też być lista. Następnie drukujemy poszczególne elementy z ogona listy w tej samej kolumnie — zajmuje się tym `ppx`. Z kolei `ppx` potrzebuje `pp` w każdym elemencie, gdyż elementy te też mogą być listami. Druga klauzula `pp` jest stosowana wówczas, gdy drukujemy coś, co nie jest listą. Po prostu robimy wcięcie o zadaną liczbę kolumn, za pomocą `write` drukujemy term i przechodzimy do nowego wiersza, dlatego w niej również pojawia się `n1`.

Przyjrzyjmy się faktom zdarzenie podanym na początku tego rozdziału. Jeśli poszczególne nagłówki mają formę listy atomów, możemy poszczególne atomy drukować za pomocą `write`, wstawiając między nie spację. Napiszemy odpowiedni predykat `phh`:

```
phh([]) :- n1.
phh([H|T]) :- write(H), tab(1), phh(T).
```

Tak więc następujące zapytanie wyświetli wszystkie nagłówki zdarzeń zawierające słowo „Anglia”:

```
?- zdarzenie(_L), member('Anglia',_L), phh(L).
```

Nawracanie pozwala nam przeszukiwać bazę danych: kiedy tylko cel `member` zawiedzie, następuje próba ponownego uzgodnienia celu zdarzenie, przez co cała baza jest przeglądana w poszukiwaniu zdarzeń zawierających atom `Anglia`.

Predykat `write` potrafi sam ładnie wydrukować przekazany mu term, gdyż uwzględnia zadeklarowane operatory. Jeśli na przykład atom zadeklarujemy jako operator infiksowy, term zawierający ten atom jako funktor wraz z dwoma argumentami zostaną wydrukowane z atomem pomiędzy tymi argumentami. Istnieje jeszcze jeden predykat wbudowany działający dokładnie tak jak `write`, ale pomijający deklaracje operatorów: jest `towrite_canonical`.²

Różnica między `write` a `write_canonical` pokazana została poniżej:

```
?- write(a+b*c*c), n1, display(a+b*c*c), n1.
a+b*c*c
+(a,*(b,c,c))
yes
```

Zwróćmy uwagę na to, że predykat `write_canonical` potraktował atomy `+` i `*` tak, jak wszystkie inne atomy. Zwykle nie interesuje nas taka postać struktur, gdyż zapis operatorowy zwykle zwiększa czytelność danych. Jednak `display` może być przydatny, jeśli nie jesteśmy pewni, jakie są priorytety operatorów.

Teraz, kiedy już umiemy użyć `read` i mamy predykat `phh`, możemy zestawić je w doskonalszy program wyświetlający zdarzenia historyczne:

```
zaczynamy2 :-
    phh(['Proszę', 'podać', 'żadaną', 'datę:']),
    read(D),
    zdarzenie(D,S),
    phh(S).
```

Zdefiniowaliśmy bezargumentowy predykat `zaczynamy2`. Kiedy zostanie on wywołany, `phh` wyświetla pytanie, `read` odczytuje datę (jak w `zaczynamy1`) i w końcu ponownie używamy `phh` do pokazania pobranego nagłówka. Zauważmy, że pierwsza klauzula w treści `zaczynamy2` także używa `phh`, mimo że nie służy do prezentacji zdarzenia historycznego. Po prostu `phh` może być użyty do pokazywania dowolnej listy atomów, niezależnie od jej interpretacji.

² W wielu systemach Prologu dostępny jest podobnie działający predykat `display`.

Czytanie i pisanie znaków

Najmniejszą porcją danych, którą można zapisać lub odczytać, jest znak. W standardzie Prologu znaki są atomami, które w nazwie mają dokładnie jeden element. Wobec tego znakami są 'a', '\n' czy ' ', zaś 'abc' już nie.³

Standardowy Prolog zawiera kilka wbudowanych predykatów, służących do czytania i pisania znaków, które mogą być przydatne do czytania danych nie będących w postaci termów lub w przypadku, kiedy potrzebujemy ścisłej kontroli nad danymi wynikowymi.

Czytanie znaków

Znak *X* można odczytać z klawiatury za pomocą celu `get_char(X)`⁴. Cel taki nigdy nie zawodzi, jeśli jego argument jest nieukonkretniony, nie można go spełnić po raz drugi. Spełnianie takiego celu wymusza na komputerze czekanie na wprowadzenie jakichś znaków. W zależności od używanego systemu, znaki mogą być dostępne dla Prologu od razu lub dopiero po zakończeniu wiersza klawiszem *Enter*. Jeśli zmienna *X* jest już ukonkretniona, cel `get_char(X)` porównuje wpisany znak z wartością *X*. To, czy `get_char` zawiedzie, zależy od wartości *X*.

Oto prosty program wykorzystujący `get_char` do sprawdzania poprawności danych.

```

sprawdz_wiersz(OK) :-
    get_char(X),
    reszta_wiersza('\n', X, OK).

reszta_wiersza(_, '\n', tak) :- !.
reszta_wiersza(ostatni, Aktualny, nie) :-
    blad_pisania(ostatni, Aktualny), !,
    get_char(Nowy),
    reszta_wiersza(Aktualny, Nowy, _).
reszta_wiersza(_, Aktualny, OK) :-
    get_char(Nowy),
    reszta_wiersza(Aktualny, Nowy, OK).

blad_pisania('q', 'w').
blad_pisania('c', 'v').

```

Przy wywołaniu `sprawdz_wiersz(X)` odczytywane są wszystkie wpisane dotąd znaki, aż do znaku nowego wiersza, '\n'. Kiedy badana linia zostanie wczytana, każda para kolejnych znaków jest porównywana ze znanymi błędami typograficznymi. Na przykład

³ Tak naprawdę szybkie operacje znakowe korzystają z małych liczb całkowitych będących kodami znaków, które jednak mogą być różne na różnych maszynach. Standard Prolog zawiera narzędzia pozwalające przetwarzać kody znaków, ale nie będziemy się tym zagadnieniem zajmować. W wielu starszych systemach Prologu dostępne są jedynie operacje na kodach znaków, a nie na samych znakach, ale w dodatku pokażemy, jak łatwo zdefiniować operacje na znakach za pomocą operacji na kodach.

⁴ Dawniej stosowano do tego `get0(X)`, które jednak przypisywało *X*.

„qw” i „cv” są uważane za błędy, gdyż normalnie nigdy nie występują w języku angielskim. sprawdź_wiersz ukonkretnia OK wartością tak lub nie — odpowiednio, kiedy tekst spełnia narzucone ograniczenia:

```

?- sprawdz_wiersz(X).
Please could you enter your comments on the proposal

```

X = nie

Powyższy program działa w ten sposób, że odczytuje z wiersza znak po znaku i zapamiętuje znak bieżący (właśnie odczytany) oraz poprzedni. Zawsze po odczytaniu znaku wywoływany jest predykat `reszta_wiersza`, przekazujemy mu jako pierwszy argument poprzedni znak, jako drugi — znak bieżący. Predykat ten w trzecim argumentie zwraca tak lub nie, w zależności od tego, czy dalszy ciąg wiersza (począwszy od dwóch wymienionych znaków) nie ma błędów. Początkowo zakładamy, że znakiem poprzednim było '\n', zaś znakiem bieżącym jest pierwszy odczytany znak. Przy odczytaniu każdego kolejnego znaku znak bieżący staje się poprzednim, zaś nowy znak staje się bieżącym. Kiedy `reszta_wiersza` wywołuje się rekurencyjnie, odpowiednio ustawiane są dwa pierwsze argumenty. Klauzule `reszta_wiersza` uwzględniają trzy różne sytuacje:

1. Osiągnięto koniec wiersza. Wtedy dana część wiersza (końcówka) nie ma błędów.
2. Poprzedni i bieżący znak pasują do znanego błędu typograficznego. Wtedy dany fragment wiersza zawiera błędy. Program sprawdza dane do końca wiersza, ale odpowiedzią będzie i tak no.
3. Nie znaleziono dotąd znanego programowi błędu. Odczytywany jest następny znak, program kontynuuje sprawdzanie.

Pisanie znaków

Jeśli zmienna *X* jest ukonkretniona znakiem, znak ten będzie wypisany po wywołaniu celu `put_char(X)`⁵.

Predykat `put_char` nigdy nie zawodzi, nie można go ponownie uzgodnić (przy takiej próbie zawodzi). Efektem ubocznym jego działania jest pokazanie argumentu na monitorze. Możemy na przykład słowo *hello* wyświetlić następująco:

```

?- put_char('h'), put_char('e'), put_char('l'), put_char('l'), put_char('o').
hello

```

Spełniając powyższą koniunkcję celów, Prolog wyświetla kolejno znaki *h*, *e*, *l*, *l* i *o*.

Wiemy już, że można przejść do nowego wiersza za pomocą bezargumentowego predykatu `nl`. Działanie `nl` polega na drukowaniu znaków kontrolnych w celu przejścia do następnego wiersza; zapytanie

```

?- put_char('h'), put_char('l'), nl, put_char('t'), put_char('h'), put_char('e').
put_char('n'), put_char('e'), .

```

⁵ W starszych wersjach Prologu używano `put(X)`, gdzie *X* było kodem znaku.

spowoduje wyświetlenie

```
hi
there
```

Opisanej metody wyświetlania znaków możemy użyć do tego, że nasz kontroler pisowni będzie poprawiał znalezione błędy. W poniższej, poprawionej wersji sprawdz_wiersz, nazwanej popraw_wiersz, każdy znany błąd jest wykrywany i poprawiany. Znak nie związane z żadnymi błędami są kopiowane bez zmian. Pokazane rozwiązanie jest mocno ograniczone, gdyż zakładamy, że każdy błąd typograficzny dotyczy dwóch sąsiednich znaków, a jego poprawienie polega na zastąpieniu błędnej pary pojedynczym znakiem. Informacje te są zapisywane w predykatcie korekta, którego dwa pierwsze argumenty to błędna para, a trzeci to wartość poprawna. Względnie łatwo byłoby tę metodę uogólnić.

```
popraw_wiersz :-
    get_char(X),
    popraw_reszte_wiersza('\n', X).

popraw_reszte_wiersza(C, '\n') :- !,
    put_char(C), nl.
popraw_reszte_wiersza(Ostatni, Aktualny) :-
    korekta(Ostatni, Aktualny, Korekta), !,
    get_char(Nowy),
    popraw_reszte_wiersza(Aktualny, Nowy).
popraw_reszte_wiersza(Ostatni, Aktualny) :-
    put_char(Ostatni),
    get_char(Nowy),
    popraw_reszte_wiersza(Aktualny, Nowy).

korekta('q', 'w', 'q').
korekta('c', 'v', 'c').
```

Procedura popraw_wiersz jest bardzo podobna do sprawdz_wiersz, ale nie ma trzeciego argumentu. Zamiast zwracać tak lub nie, pokazuje poprawiony tekst dla części wiersza zaczynającej się od znaku będącego pierwszym argumentem tej procedury.

Tak jak poprzednio, pierwszy argument to poprzedni znak, a drugi — znak bieżący. Znak można wyświetlić bez zmian dopiero wtedy, kiedy wczytany został za nim następny znak i znaki te nie są błędne. Wobec tego popraw_reszte_wiersza wyświetla znak poprzedni przed przejściem dalej.

Wczytywanie zdań

Pokażemy teraz program wczytujący zdania wpisane z klawiatury i przekształcający je w listę atomów⁶. W programie tym zdefiniujemy jednoargumentowy predykat wczytaj.

Program musi „wiedzieć”, kiedy poszczególne słowa się kończą i zaczynają następne. Zakładamy zatem, że słowo składa się z dowolnej liczby liter, cyfr i znaków specjalnych.

Litery i cyfry są takie same jak pokazaliśmy w rozdziale 2., zaś znaki specjalne to pojedynczy cudzysłów „'” i minus, „-”. Poza tym znaki: ‘,’, ‘,’, ‘,’, ‘,’ i ‘!’ traktowane są jako samodzielne słowa. Wszystkie inne znaki to przerwy między słowami. Zdanie musi kończyć się jednym ze słów: ‘,’, ‘?’ lub ‘!’. Wielkie litery są automatycznie zamieniane na małe, dzięki czemu to samo słowo zawsze daje ten sam atom. W wyniku takiej definicji program będzie działał następująco:

```
?- wczytaj(S).
Starszy pan, bardzo elegancko ubrany, mija teraz Janka.
S = [starszy, pan, ',', bardzo, elegancko, ubrany, ',', mija, teraz, janka, '.']
```

Wstawiliśmy tu dodatkowe pojedyncze cudzysłowy, aby było jasne, że znaki przestankowe stanowią osobne atomy.

W programie do odczytu znaków używamy predykatu get_char. Związany jest z tym problem polegający na tym, że kiedy znak został raz wczytany, już go nie ma i żadne następne wywołanie get_char lub próba ponownego uzgodnienia nie przywróci nam tego znaku. Musimy zatem unikać nawracania przez get_char, jeśli chcemy unikać „gubienia” wczytywanych znaków. Na przykład poniższy program wczytujący znaki i drukujący je, zamieniający przy tym a na b **nie działa**:

```
do_roboty :- przetworz_znak, do_roboty.
przetworz_znak :- get_char(X), X='a', !, put_char('b').
przetworz_znak :- get_char(X), put_char(X).
```

Kolejną wadą tego programu jest to, że działa on w nieskończoność. Zajmijmy się jednak próbą spełnienia celu przetworz_znak. Jeśli pierwsza klauzula przetworz_znak wczyta znak inny niż a, nawracanie spowoduje próbę uzgodnienia drugiej klauzuli.

Jednak cel get_char(X) z tej klauzuli ukonkretni zmienną X *następnym* znakiem po znaku już wczytanym. Wynika to stąd, że spełnienie pierwotnego celu get0 jest procesem nieodwracalnym. Wobec tego program nie będzie wyświetlał części znaków, a czasami nawet może pokazać a.

Jak więc napisać program wczytaj, aby poradzić sobie z problemem nawracania przez wczytywanie danych? Musimy zawsze wczytywać znak naprzód i sprawdzać w innej regule, jaki znak wczytaliśmy. Kiedy znaleziony zostanie znak i nie będzie można go użyć, zostanie przekazany do reguły, która już będzie mogła go zastosować. Wobec tego predykat odczytujący pojedyncze słowo, wczytajslowo, ma trzy argumenty. Pierwszy z nich to znaleziony przez get_char znak, który nie został dotąd nigdzie indziej użyty. Drugi to atom, który stanie się słowem. Ostatni argument to pierwszy znak odczytany po słowie.

Aby sprawdzić, gdzie słowo się kończy, konieczne jest odczytanie znaku zza tego słowa. Znak ten musi zostać zwrócony, gdyż może być pierwszym znakiem słowa następnego. Oto nasz program:

⁶ Oznacza to, że nie możemy używać polskich liter, gdyż te nie mogą należeć do atomów. Aby to zrobić, musielibyśmy skorzystać z listy łańcuchów znakowych, co zresztą jest powszechną praktyką — *przyj. tłum.*

```

/* Wczytaj zdanie */
wczytaj([W|Ws]) :-
    get_char(C), wczytajslowo(C,W,C1), ciagdalszy(W,C1,Ws).
/*
    Jeśli mamy słowo i znak występujący za nim, wczytujemy ciąg
    dalszy zdania.
*/
ciagdalszy(W,_,[]) :- ostatnieslowo(W), !.
ciagdalszy(W,C,[W1|Ws]) :-
    wczytajslowo(C,W1,C1), ciagdalszy(W1,C1,Ws).
/*
    Wczytaj pojedyncze słowo, znając jego pierwszy znak, zapamiętaj
    znak, który pojawi się za tym słowem.
*/
wczytajslowo(C,W,C1) :-
    pojedynczy_znak(C), !, name(W,[C]), get_char(C1).
wczytajslowo(C,W,C2) :-
    w_slowie(C.NewC),
    !,
    get_char(C1),
    resztaslowa(C1,Cs,C2),
    atom_chars(W,[NewC|Cs]).
wczytajslowo(C,W,C2) :- get_char(C1), wczytajslowo(C1,W,C2).
resztaslowa(C,[NewC|Cs],C2) :-
    w_slowie(C.NewC),
    !,
    get_char(C1), resztaslowa(C1,Cs,C2).
resztaslowa(C,[],C).
/*
    Znaki, które mogą występować w słowach. Druga klauzula w_slowie
    przekształca znaki na małe.
*/
w_slowie(C,C) :- litera(C,_). /* a b...z */
w_slowie(C,L) :- litera(L,C). /* A B...Z */
w_slowie(C,C) :- cyfra(C). /* 1 2...9 */
w_slowie(C,C) :- znak_specjalny(C). /* ' . - */

/* Znaki tworzące słowa samodzielnie */
pojedynczy_znak(' '). pojedynczy_znak(':').
pojedynczy_znak(' '). pojedynczy_znak('?').
pojedynczy_znak(';'). pojedynczy_znak('!').

/* Wielkie i małe litery */
litera(a,'A'). litera(n,'N').
litera(b,'B'). litera(o,'O').
litera(c,'C'). litera(p,'P').
litera(d,'D'). litera(q,'Q').
litera(e,'E'). litera(r,'R').
litera(f,'F'). litera(s,'S').
litera(g,'G'). litera(t,'T').
litera(h,'H'). litera(u,'U').
litera(i,'I'). litera(v,'V').
litera(j,'J'). litera(w,'W').
litera(k,'K'). litera(x,'X').
litera(l,'L'). litera(y,'Y').
litera(m,'M'). litera(z,'Z').

```

```

/* Cyfry */
cyfra('0'). cyfra('5').
cyfra('1'). cyfra('6').
cyfra('2'). cyfra('7').
cyfra('3'). cyfra('8').
cyfra('4'). cyfra('9').

/* Słowa kończące zdania */
ostatnieslowo(' ').
ostatnieslowo('!').
ostatnieslowo('?').

```

Predykat wbudowany `atom_chars` pozwala stworzyć atom na podstawie listy znaków (rozdział 6.).

Ćwiczenie 5.1. Objaśnij znaczenie poszczególnych zmiennych w powyższym programie.

Ćwiczenie 5.2. Napisz program wczytujący znaki bez końca i wyświetlający je, ale zmieniający przy tym „a” na „b”.

Czytanie z plików i pisanie do plików

Omówione wcześniej w tym rozdziale predykaty służą do czytania z terminala i pisania do niego, ale są one ogólniejsze. Zgodny ze standardem Prolog może pisać dane do *strumieni* i ze strumieni dane odczytywać. Strumień może odpowiadać klawiaturze lub monitorowi, może też odpowiadać *plikowi* będącemu ciągiem znaków na nośniku zewnętrznym. Pliki mogą znajdować się na różnych nośnikach, jakkolwiek obecnie najczęściej są to dyski magnetyczne. Każdy plik ma wyróżniającą go *nazwę pliku*. Aby ten podrozdział był zrozumiały, konieczna jest znajomość sposobu organizacji plików w używanym systemie operacyjnym i zasad nazewnictwa. W Prologu nazwy plików są atomami, ale trzeba liczyć się z dodatkowymi ograniczeniami związanymi z używanym systemem.

Plik ma pewną długość, czyli zawiera określoną liczbę znaków. Na końcu pliku znajduje się specjalny znacznik, *znacznik końca pliku*. Nie mówiliśmy o nim dotąd, gdyż nie występuje on w przypadku klawiatury ani monitora. Jeśli program odczytuje dane z pliku, znacznik końca pliku jest wykrywany niezależnie od tego, czy odczytujemy termy, czy znaki.

Jeśli znacznik ten zostanie wykryty przez `get_char(X)` lub `read(X)`, `X` zostanie ukonkretniona specjalnym atomem `end_of_file`⁷. Jeśli program próbuje czytać za końcem pliku, powstaje błąd.

⁷ Implementacje Prologu niezgodne ze standardem mogą zwracać inne wartości, ale są to i tak różne wartości specjalne oznaczające koniec pliku.

W Prologu istnieją wbudowane strumienie: wejściowy, `user_input`, oraz wyjściowy, `user_output`. Ustawienie strumienia wejściowego na `user_input` (domyślne) powoduje, że dane wejściowe będą pochodziły z klawiatury; ustawienie standardowego wyjścia na `user_output` (domyślne) powoduje, że dane wynikowe będą wysyłane na monitor. Tak normalnie działają systemy Prologu. Kiedy dane wejściowe pochodzą z klawiatury, koniec pliku możemy wygenerować, wciskając odpowiedni znak kontrolny; jego konkretna wartość zależy od używanego systemu. Wtedy `get_char` i `read` zachowują się tak, jakby natknęły się na koniec pliku.

Otwieranie i zamykanie strumieni

Standardowy Prolog rozpoznaje *bieżący strumień wejściowy*, z którego wczytywane są wszelkie dane. Dane wejściowe z `get_char` i `read` są pobierane właśnie z bieżącego strumienia wejściowego. Istnieje też *bieżący strumień wyjściowy*, do którego kierowane są dane wypisywane przez `put_char` i `write`.

Klawiatura komputera zwykle pełni funkcję bieżącego strumienia wejściowego, zaś funkcję strumienia wyjściowego pełni zwykle ekran, ale oba te strumienie można chwilowo zmieniać podczas działania programu.

Zanim możliwe będzie korzystanie z pliku, trzeba otworzyć nowy strumień z tym plikiem związany. Służy do tego predykat wbudowany `open`, którego argumentami są nazwa pliku oraz atom mówiący, czy plik ma być otwarty do czytania, czy do pisania. Trzeci argument jest ukonkretniany specjalnym termem będącym nazwą otwartego strumienia. Wobec tego przykładowo:

```
?- open('plik.pl', read, X).
```

ukonkretnia zmienną `X` nazwą strumienia, który będzie służył do czytania z pliku „plik.pl”. Z kolei

```
?- open('wyniki', write, X).
```

ukonkretnia `X` nazwą strumienia, który umożliwia pisanie do pliku o nazwie „wyniki”.

Zauważmy, że przy każdym otwarciu nowego strumienia związanego z danym plikiem, strumień ten wskazuje początek tego pliku. Można mieć w zasadzie wiele strumieni dla jednego pliku, ale strumienie te będą wskazywały różne miejsca tego pliku i rzadko coś takiego będzie naprawdę potrzebne. W zasadzie należy dążyć do posiadania pojedynczego strumienia dla każdego otwartego pliku, zatem `open` należy używać raz, tuż przed pierwszą próbą dostępu do pliku. Kiedy strumień przestaje być potrzebny, czy to z powodu odczytania wszystkich danych, czy zakończenia pisania danych wynikowych, predykat `close` pozwoli zakończyć używanie pliku. Predykat ten ma jeden argument, nazwę strumienia, otrzymany z `open`. Tak więc typowy program czytający plik powinien wyglądać tak:

```
program :-
    open('plik.pl', read, X),
    kod_odczytujujacy_dane(X),
    close(X).
```

przy czym `kod_odczytujujacy_dane(X)` to predykat wymagający danych wejściowych ze strumienia `X`. Analogicznie, typowy program piszący do pliku ma postać:

```
program :-
    open('plik.pl', write, X),
    kod_zapisujacy_dane(X),
    close(X).
```

przy czym `kod_zapisujacy_dane(X)` zapisuje dane w pliku. Zauważmy, że `kod_odczytujujacy_dane` i `kod_zapisujacy_dane` nie powinny zawodzić, gdyż wtedy odpowiednie strumienie nie zostałyby zamknięte. Jeśli używane tu predykaty mogą w pewnych sytuacjach zawieść, trzeba to zmienić, na przykład przez dodanie klauzuli obsługi przypadków nieuwzględnionych nigdzie indziej. W zasadzie predykaty czytające i piszące do plików trudno jest debugować, wobec czego dobrze jest sprawdzić ich działanie na monitorze, a dopiero potem użyć ich do pisania w pliku i czytania z niego.

Zmiana bieżącego strumienia wejściowego i wyjściowego

Postać nazwy strumienia zależy od używanego Prologu, jeśli więc nasze programy mają być przenośne, nie powinniśmy co do tej postaci czynić żadnych założeń. Ogólnie rzecz biorąc, kiedy program otwiera nowy strumień, otrzymuje jego nazwę w zmiennej `X` i dalej nie powinien robić z tą zmienną niczego poza:

1. Ustawieniem bieżącego strumienia wejściowego lub wyjściowego na `X` na jakiś czas.
2. Wywołaniem `close` w celu zamknięcia strumienia.
3. Przekazaniem strumienia (jako argumentu predykatu) w inne miejsce programu.

Zmiana bieżącego strumienia wejściowego i wyjściowego odbywa się za pomocą predykatów wbudowanych `set_input` i `set_output`. Każdy z nich spodziewa się jako argumentu nazwy strumienia (można też użyć atomów `user_input` i `user_output`). Wynikiem uzgodnienia takiego celu jest przełączenie bieżącego strumienia wejściowego lub wyjściowego do podanego strumienia, póki nie zostanie ponownie wywołany ten sam predykat. Trzeba zdawać sobie przy tym sprawę, że ewentualne ponowne przełączenie nie następuje przy próbie ponownego spełnienia celu; próba taka po prostu zawodzi.

Jako że predykaty `set_input` i `set_output` mają nieodwracalne działanie, dobrą praktyką jest każdorazowe jawne przypisywanie strumieni wejściowych i wyjściowych i zapewnienie, żeby miały one wartości zgodne z oczekiwaniami niezależnie od tego, jak program zadziała (jeśli na przykład któryś ważny cel zawiedzie). W szczególności, jeśli program zmienia strumień wejściowy lub wyjściowy, powinien następnie przywracać jego pierwotne ustawienie. W tym celu konieczne jest sprawdzenie na początku ustawienia strumieni.

Predykaty wbudowane `current_input` i `current_output` pozwalają na sprawdzenie, jak są aktualnie ustawione strumienie. Predykaty te ukonkretniają swój jedyny parametr nazwą bieżącego strumienia, odpowiednio, wejścia lub wyjścia.

Teraz możemy pokazać lepszą ogólną postać programu odczytującego dane z pliku:

```
program :-
    open('plik.pl', read, X),
    current_input(Strumien),
    set_input(X),
    kod_odczytujacy,
    close(X),
    set_input(Strumien).
```

Zauważmy, że teraz `kod_odczytujacy` nie musi już odwoływać się jawnie do `X`. Bieżącym strumieniem przed wywołaniem tego celu jest `X`, wobec czego można używać `get_char` i `read`, a one będą podawały dane z odpowiedniego pliku. Tak więc `kod_odczytujacy` może być też użyty w innych sytuacjach, do obsługi innych plików.

Zauważmy też, że para celów związanych ze strumieniem `Strumien` przywraca początkowy strumień wejściowy. Ogólna postać programu zapisującego dane wynikowe do pliku jest podobna:

```
program :-
    open('wyniki', write, X),
    current_output(Strumien),
    set_output(X),
    kod_zapisujacy,
    close(X),
    set_output(Strumien).
```

Konsultowanie

Operacje czytania z plików i pisania do nich są najbardziej przydatne, kiedy mamy do czynienia z większą liczbą termów niż chcemy wpisywać ręcznie, a musimy je wstawić do bazy danych. W Prologu plików używa się jednak głównie do przechowywania programów. Jeśli mamy tekst programu w pliku, możemy odczytać z tego pliku wszystkie klauzule i wstawić je do bazy danych w trakcie procesu nazywanego „konsultowaniem pliku”. Standard Prologu pozostawia otwartą kwestię dogodnego zaimplementowania odpowiedniej metody. W tym punkcie opiszemy metodę najczęściej stosowaną, choć nie można zakładać, że zadziała ona zawsze..

Wiele systemów Prologu ma predykat wbudowany `consult`. Jeśli `X` jest ukonkretniona nazwą pliku, `consult(X)` odczytuje klauzule i cele Prologu z tego pliku. Większość implementacji Prologu zawiera specjalny zapis `consult`, który pozwala konsultować od razu listę plików. Jeśli jako cel podana zostanie lista plików, Prolog będzie konsultował kolejno te pliki. Przykładowe użycie może wyglądać tak:

```
?- [plik1, mapper, expert].
```

Zachowanie to jest bardzo podobne do zachowania w przypadku wywołania celu `consultall(X)`, gdzie `X` jest listą plików podaną w zapytaniu, a `consultall` zdefiniowany jest na przykład tak:

```
consultall([]).
consultall([H|T]) :- consult(H), consultall(T).
```

Jednak skrócony zapis w formie listy ułatwia pracę, szczególnie jeśli programista na początku chce wywołać `consult` na rzecz listy plików zawierających najprzystatniejsze predykaty. Predykat `consult` automatycznie przerywa swoje działanie po napotkaniu końca pliku. W następnym rozdziale, w podrozdziale „Wprowadzanie nowych klauzul”, omówimy ten predykat dokładniej.

Deklarowanie operatorów

Operatory dołączono do tego rozdziału dlatego, że pozwalają przy czytaniu i pisaniu termów korzystać z wygodnej składni. Operatory zresztą do niczego innego nie służą. Wróćmy na chwilę do podrozdziału „Operatory” z rozdziału 2. i potem zajmijmy się ich deklarowaniem.

Zgodnie ze składnią Prologu, każdy operator ma trzy właściwości: położenie, priorytet i łączność. Położenie może być infiksowe, postfiksowe lub prefiksowe (operator dwuarumentowy może być między swoimi argumentami; operator jednoargumentowy może być przed lub za swoim argumentem). Priorytet to liczba całkowita, której konkretna wartość zależy od używanej wersji Prologu, ale zakłada się, że w większości systemów wartości mieszczą się między 1 a 1200. Priorytet używany jest do eliminacji niejednoznaczności z wyrażeń w przypadku, kiedy nie są stosowane nawiasy. Łączność też służy do usuwania niejednoznaczności w sytuacji, kiedy w wyrażeniu występują dwa operatory o takim samym priorytecie. W Prologu położenie i łączność operatora wskazuje specjalny atom. W przypadku operatorów infiksowych może on mieć wartości:

```
xfx xfy yfx yfy
```

Aby zrozumieć te oznaczenia, warto potraktować je jako przykłady możliwych sposobów stosowania tych operatorów. Litera `f` oznacza sam operator, `x` i `y` to jego argumenty. W powyższych przykładach operator musi pojawić się *między* argumentami, więc jest operatorem infiksowym. Zgodnie z tą konwencją

```
fx fy
```

opisują operatory prefiksowe (które znajdują się *przed* swoim argumentem), zaś

```
xf yf
```

opisują operatory postfiksowe. W pierwszej chwili może być niejasne, dlaczego do opisu argumentów używa się dwóch różnych liter. Wybór `x` lub `y` niesie informację o łączności. Jeśli w wyrażeniu nie ma nawiasów, użycie `y` oznacza, że argument może zawierać operatory o takim samym lub niższym priorytecie. Z kolei `x` oznacza, że argument może zawierać jedynie operatory o niższym priorytecie. Zastanówmy się zatem, co to oznacza dla operatora + zadeklarowanego jako `yfx`. Jeśli weźmiemy pod uwagę wyrażenie:

```
a + b + c
```

możemy je zinterpretować w dwojaki sposób:

```
(a + b) + c lub a + (b + c)
```


stosowane zwykle w różnych wersjach Prologu. Nowe klauzule można wpisywać z terminala lub nakazać mu pobierać je z przygotowanego wcześniej pliku. Z punktu widzenia Prologu obie te operacje nie różnią się od siebie, gdyż terminal jest po prostu plikiem o nazwie `user`. Istnieje jeden podstawowy predykat wbudowany, służący do dodawania klauzul: `consult`. Poza tym istnieje wygodny zapis pozwalający wczytywać klauzule z wielu plików: użycie listy. Osoby zainteresowane działaniem `consult` i `reconsult` znajdą więcej informacji na ich temat w podrozdziale „Przetwarzanie programów” w następnym rozdziale.

consult(X)

Predykat wbudowany `consult` służy do dodawania klauzul z pliku (lub wpisywanych z klawiatury) tak, aby *uzupełniały* klauzule już w bazie danych istniejące. Argumentem tego predykatu musi być atom określający nazwę pliku, z którego należy pobrać klauzule; oczywiście zasady tworzenia nazw plików zależą od używanego systemu. Oto przykłady celów `consult` odpowiednich dla różnych systemów operacyjnych:

```
?- consult(myfile).
?- consult('/usr/john/pl/chat').
?- consult('\\john\\pl\\chat').
?- consult('lib:iorout.pl').
```

Warto spojrzeć, czy któraś z powyższych notacji jest sposobem zapisywania nazw plików w używanym systemie. Znaki `\` muszą występować parami, jak zawsze w atomach Prologu.

Jeśli w pliku znalezione zostanie zapytanie, będzie potraktowane jak każde inne zapytanie. Zwykle nie ma sensu przeplatanie zapytań z nowymi klauzulami, chyba że definiowane są nowe operatory i wyświetlane komunikaty pomocnicze.

W przypadku czytania danych z wielu plików, jeśli okaże się, że w jednej z klauzul wystąpił błąd, można go poprawić bez konieczności ponownego wczytywania wszystkich plików. W tym celu wystarczy wykonać `consult` na pliku zawierającym poprawny zestaw klauzul kwestionowanego predykatu. Poprawne klauzule można wpisać z klawiatury (przy użyciu `consult(user)`) lub możemy poprawić plik, nie wychodząc z Prologu, po czym ponownie wywołać `consult` na poprawionym pliku. Oczywiście pierwsze rozwiązanie pozwoli poprawić zawartość bazy danych Prologu, ale w plikach błąd pozostanie! W podrozdziale „Poprawianie błędów” w rozdziale 8. pokażemy, jak używa się `consult` podczas pisania programu.

`Consult` w opisanej powyżej postaci nie pozwala rozdzielać definicji predykatu na wiele plików. Zwykle jest to ograniczenie naturalne, gdyż typowo wszystkie klauzule jednego predykatu trzyma się razem. Jeśli konieczne jest rozdzielenie tych klauzul między różne pliki, większość systemów Prologu umożliwia uprzednie zasygnalizowanie takiej potrzeby, poza tym dostępne mogą być predykaty podobne do `consult`, obsługujące taką sytuację.

Zapis w formie listy

Szereg implementacji Prologu obsługuje specjalną formę zapisu ułatwiającą wykonywanie `consult`, szczególnie gdy trzeba wczytać wiele plików. Wystarczy podać nazwy wszystkich żądanych plików (jako atomy) na liście i listę tę podać jako cel do spełnienia. Tak więc zapytanie

```
?- [plik1.plik2, 'fred.1', 'bill.2'].
```

jest równoważne z zapytaniem:

```
?- consult(plik1), consult(plik2),
   consult('fred.1'), consult('bill.2').
```

Zapis w formie listy jest jedynie konwencją i nie zawiera żadnych dodatkowych możliwości względem `consult`.

Sukces i porażka

Zwykle podczas wykonywania programu prologowego cel jest spełniany, jeśli można go uzgodnić, i zawodzi, kiedy jest to niemożliwe. Istnieją dwa predykaty ułatwiające wskazanie spełnienia lub niespełnienia celu: są to `true` i `fail`.

true

Ten cel nigdy nie zawodzi. Nie jest on niezbędny, gdyż klauzule i cele można tak uporządkować, aby jego użycie było zbędne. Niemniej jednak, dla wygody programistów, zdefiniowano go.

fail

Cel `fail` zawsze zawodzi. Przydatny jest w dwóch sytuacjach: po pierwsze, w połączeniu z „odcięciem”, co było opisane w rozdziale 4., w podrozdziale „Typowe zastosowania odcięcia”. Następujące połączenie celów:

```
..., !, fail.
```

pozwala powiedzieć: „jeśli doszedłeś dotąd, zrezygnuj z prób uzgadniania tego celu”. Cała koniunkcja zawodzi z uwagi na istnienie `fail`, zaś cel nadrzędny zawodzi z powodu odcięcia. Inne zastosowanie `fail` to przypadek, kiedy chcemy jawnie wskazać, że pewien cel ma generować kolejne rozwiązania przez nawracanie. Na przykład

```
?- zdarzenie(X,Y), phh(Y), fail.
```

pokaże wszystkie zdarzenia z bazy danych z poprzedniego rozdziału (i na koniec zawiedzie). Inne zastosowanie `fail` można zobaczyć w definicji `retractall` w podrozdziale „Przetwarzanie programów” rozdziału 7.

Klasyfikacja termów

Jeśli definiujemy predykaty, które mają potem być użyte z różnorodnymi argumentami, często potrzebne jest rozróżnienie w definicji, co z poszczególnymi rodzajami argumentów można zrobić. W najprostszym przypadku możemy chcieć odróżnić przetwarzanie atomu od przetwarzania liczby, możemy też inaczej obsługiwać argumenty ukonkretnione, a inaczej nieukonkretnione. Opisane dalej predykaty pozwalają programiście nałożyć takie warunki w klauzulach.

var(X)

Cel `var(X)` nie zawodzi, jeśli `X` jest zmienną *nieukonkretnioną*. Tak więc przykładowy dialog może wyglądać tak:

```
?- var(X).
yes
?- var(23).
no
?- X = Y, Y = 23, var(X).
no
```

Zmienna nieukonkretniona może być częścią struktury; przykładami są niezajęte pola w opisywanej w rozdziale 4. grze w kółko i krzyżyk, a także niewypełnione części posortowanego drzewa słownika z rozdziału 7. Kiedy badamy tego typu struktury, predykat `var` pozwala sprawdzić, co zostało już wypełnione, a co jeszcze nie. Dzięki temu można uniknąć „przypadkowego” ukonkretnienia zmiennej, podczas gdy zamierzano było sprawdzenie jej wartości. Na przykład w drzewie słownika przed stworzeniem nowego klucza chcemy wiedzieć, czy dany klucz już istnieje.

nonvar(X)

Cel `nonvar(X)` nie zawodzi, jeśli `X` *nie* jest zmienną ukonkretnioną. Predykat ten zatem jest przeciwieństwem `var` — można go zdefiniować następująco:

```
nonvar(X) :- var(X), !, fail.
nonvar(_).
```

atom(X)

Cel `atom(X)` nie zawodzi, jeśli `X` jest atomem Prologu — oto przykładowy dialog:

```
?- atom(23).
no
?- atom(gruszki).
yes
?- atom('/us/chris/pl.123').
yes
?- atom("to jest łańcuch").
no
?- atom(X).
no
?- atom(ksiązka(bronte,w_w,X)).
no
```

number(X)

Cel `number(X)` nie zawodzi, jeśli `X` jest liczbą. W podrozdziale „Odwzorowywanie struktur i przekształcanie drzew” w rozdziale 7. pokażemy, jak predykatu tego używa się przy definiowaniu prostych wyrażeń arytmetycznych, kiedy musimy wiedzieć, czy mamy do czynienia z liczbą.

atomic(X)

Cel `atomic(X)` nie zawodzi, jeśli `X` jest liczbą lub atomem. Predykat `atomic` można zdefiniować przy użyciu `atom` i `number`:

```
atomic(X) :- atom(X).
atomic(X) :- number(X).
```

Przetwarzanie klauzul jako termów

Prolog umożliwia programiście modyfikowanie programu (klauzul używanych do spełniania celów). Jest to tym prostsze, że na klauzule można spojrzeć jako na zwykłą strukturę Prologu. Prolog zawiera predykaty wbudowane, które pozwalają programiście:

- ♦ tworzyć strukturę reprezentującą klauzulę w bazie danych,
- ♦ dodawać do bazy danych klauzulę zapisaną jako taka struktura,
- ♦ usuwać z bazy danych klauzulę zapisaną jako taka struktura.

Większość operacji na bazie danych możemy wykonać korzystając z opisywanych predykatów i standardowych operacji Prologu pozwalających tworzyć i rozkładać struktury. W rozdziale 7., w podrozdziale „Użycie bazy danych: `random`, `gensym`, `findall`” pokazujemy dodawanie i usuwanie klauzul w ten sposób.

Zanim zajmniemy się odpowiednimi predykatami wbudowanymi, musimy wiedzieć, jak Prolog traktuje klauzule jako struktury. W przypadku zwykłych faktów struktura jest po prostu odpowiednim predykatem z argumentami. Zatem fakt

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

może być potraktowany jako struktura z funktorem `lubi` i dwoma argumentami, `jan` i `X`. Reguła z kolei to struktura, której funktorem jest `:-`, i która ma dwa argumenty. Funktor ten zdefiniowano jako operator infiksowy, pierwszy argument to głowa klauzuli, a drugi to jej treść. Zatem

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

to tak naprawdę

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

czyli najzwyklejsza struktura. Jeśli w regule jest więcej niż jeden cel, cele te wiąże funktor `(.)` (dwuargumentowy). Funktor ten także jest operatorem infiksowym. Wobec tego

```
dziadek_babcia(X,Z) :- rodzic(X,Y), rodzic(Y,Z).
```

to

```
'- '(dziadek_babcia(X,Z), ' ', '(rodzic(X,Y),rodzic(Y,Z))
```

Trzeba tutaj odnotować, że w standardowym Prologu poniższe predykaty wbudowane mogą nie współpracować z niektórymi predykatami używanymi w programie:

- ♦ `clause` (a także narzędzia takie jak `listing`) będą mogły znaleźć klauzule jedynie dla operatorów „publicznych” (dokładna interpretacja tego określenia zależy od konkretnej implementacji Prologu, ale chodzi o to, że na przykład klauzul predykatów wbudowanych nie można podejrzec; programista może wpływać na to, które jego predykaty będą „publiczne”).
- ♦ `asserta`, `assertz` i `retract` działają jedynie dla predykatów zadeklarowanych jako „dynamiczne”. Znowu chodzi tu o uniemożliwienie przypadkowej zmiany definicji. Predykat `jakis/4` można zadeklarować jako „dynamiczny” przez włączenie w odpowiednim pliku następującego zapisu (włączenie to musi się odbyć przed definicją `jakis/4`, a jeśli takiej definicji nie ma, przed kodem używającym `asserta` itd.):

```
:- dynamic jakis/4.
```

W jednym wierszu można zadeklarować dynamiczność wielu predykatów, jeśli tylko poda się je w formie listy rozdzielanej przecinkami, na przykład:

```
:- dynamic jakis/4, inny/3.
```

Oto predykaty służące do badania i modyfikacji klauzul.

listing(A)

Większość systemów Prologu pozwala podejrzec załadowane klauzule, choć standard tego nie wymaga. Typowym rozwiązaniem jest użycie predykatu wbudowanego `listing`; spełnienie celu `listing(A)`, gdzie zmienna `A` jest ukonkretniona atome, powoduje, że wypisywane są wszystkie klauzule predykatu o nazwie `A`. W ten sposób można sprawdzić, jak aktualnie wygląda definicja wybranego predykatu. Dokładna postać wyniku zależy od używanego systemu Prologu. Trzeba pamiętać, że pokazane zostaną wszystkie klauzule, których predykatem jest dany atom, niezależnie od liczby argumentów. Użycie `listing` pozwala czasem wykryć błędy w programie. Na przykład w poniższym dialogu programista stwierdza, że predykat `reverse` nie został zdefiniowany prawidłowo.

```
?- [test].
test consulted
yes
?- reverse([a,b,c,d],X).
no
?- listing(reverse).
reverse([],[]).
reverse([_44|_45],_38) :-
    reverse(_45,_47),
    append(_47,[_44],_38).
yes
```

Okazuje się, że nieprawidłowo zapisano atom `append` (jako `appendD`).

clause(X,Y)

Spełnienie celu `clause(X,Y)` powoduje, że `X` i `Y` dopasowywane są do głowy i treści klauzuli z bazy danych. `X` musi być ukonkretniona przynajmniej na tyle, aby znany był główny funktor klauzuli. Jeśli żadne klauzule nie zostaną znalezione, cel zawodzi. Jeśli istnieje więcej niż jedna pasująca klauzula, Prolog wybierze pierwszą z nich, zaś przy próbach ponownego spełnienia tego celu wybierane będą następne klauzule.

Warto zauważyć, że choć `clause` ma zawsze argument odpowiadający treści reguły, to nie każda klauzula ma treść. Jeśli tak nie jest, przyjmuje się, że treścią jest `true`; takie klauzule to „fakty”. Jeśli `X` i `Y` są w mniejszym czy większym stopniu ukonkretnione, możemy wyszukiwać wszystkie klauzule danego predykatu lub tylko te, które mają określoną liczbę argumentów, a także wybierać argumenty pasujące do jakiegoś wzorca. Oto przykład:

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).
?- clause(append(A,B,C),Y).
A=[], B=_, C=_, Y=true ;
A=[_3|_4], B=_5, C=[_3|_6], Y=append(_4,_5,_6) ;
```

no

Predykat `clause` jest bardzo ważny w przypadku tworzenia programów, które mają badać lub wykonywać inne programy.

asserta(X), assertz(X)

Predykaty `asserta` i `assertz` pozwalają dodać do bazy danych nowe klauzule. Oba działają tak samo, tyle że `asserta` dodaje klauzule *na początek* bazy danych, a `assertz` *na koniec*. Najłatwiej to zapamiętać, gdy uzmysłowimy sobie, że `a` jest pierwszą literą alfabetu, a `z` — ostatnią. W celu `asserta(X)` zmienna `X` musi być ukonkretniona dostatecznie, aby można było zapisać ją jako klauzulę. Sprowadza się to do tego, że musi być znany funktor główny.

Trzeba podkreślić, że dodawanie klauzuli do bazy danych *nie* jest wycofywane w przypadku nawracania, jeśli zatem użyto `asserta` lub `assertz`, klauzulę tę można usunąć jedynie jawnie (za pomocą `retract`). Przykłady użycia `asserta` znajdują się w następnym rozdziale, w podrozdziale „Użycie bazy danych: `random`, `gensym`, `findall`”.

retract(X)

Predykat wbudowany `retract` umożliwia programowi usuwanie klauzul z bazy danych. Predykat ten ma jeden argument odpowiadający termowi, którego klauzule są szukane. Term musi być ukonkretniony w dostatecznym stopniu, aby można było określić klauzulę (jak w przypadku `asserta`, `clause` i innych). Przy próbie spełnienia celu `retract(X)` zmienna `X` jest dopasowywana do pierwszej pasującej do niej klauzuli z bazy danych, następnie klauzula ta jest usuwana. Przy próbie ponownego spełnienia celu wyszukiwana jest następna klauzula i jeśli zostanie znaleziona, cały proces jest powtarzany. Trzeba pamiętać, że po usunięciu klauzula nie jest z powrotem wstawiana, nawet podczas nawracania. Jeśli nie można znaleźć klauzuli pasującej do klauzuli przeznaczonej do usunięcia, `retract` zawodzi.

Jako że argument X jest dopasowywany do usuwanej klauzuli, można sprawdzić, co dokładnie jest usuwane, o ile tylko X ma nieukonkretnione zmienne. Dzięki temu możemy użyć `retract` do zasymulowania `clause`. W ten sposób korzystamy z `retract` w definicji `gensym` (podrozdział „Użycie bazy danych: `random`, `gensym`, `findall`” w rozdziale 7.).

Tworzenie składników struktur i sięganie do nich

Kiedy chcemy sięgnąć do jakiejś struktury Prologu, po prostu odwołujemy się do odpowiedniej części tej struktury. Jeśli jednak predykat musi obsługiwać dużo różnych struktur pojawiających się jako argumenty, zwykle podaje się osobne klauzule dla każdego rodzaju struktur. Dobrym przykładem jest różniczkowanie symboliczne z rozdziału 7.: mamy osobne klauzule dla funktorów `+`, `-`, `*` i tak dalej. Przewidzieliśmy, jakie struktury mogą wystąpić i przygotowaliśmy odpowiednie klauzule.

Czasami jednak nie można przewidzieć, jakiego rodzaju struktury mogą się pojawić. Dzieje się tak na przykład kiedy piszemy program *eleganckiego wydruku* prezentujący struktury Prologu z wcięciami, podzielone na wiersze (o takim programie obsługującym listy mówiliśmy już w rozdziale 5.).

Przykładowo, *elegancki wydruk termu*

```
ksiazka(b29,autor(bronte,emily),wh)
```

może wyglądać następująco:

```
ksiazka
  b29
  autor
    bronte
    emily
  wh
```

Ważne jest to, że program może przetwarzać *dowolne* struktury. Można byłoby oczywiście tworzyć osobne klauzule dla wszystkich możliwych funktorów, ale jest to fizycznie niemożliwe. Do tego typu zadań używa się predykatów wbudowanych operujących na dowolnych strukturach. Opiszemy niektóre z tych predykatów: `functor`, `arg` i `=`. Opiszemy też predykat operujący na atomach, `atom_chars`.

`functor(T,F,N)`

Predykat `functor` zdefiniowano tak, że `functor(T,F,N)` oznacza, że T jest strukturą z funktorem F z N argumentami. Tego predykatu używa się w dwojaki sposób. T może być ukonkretnione; cel zawodzi, jeśli T nie jest atomem lub strukturą. Jeśli T jest atomem lub strukturą, to F jest funktorem T , zaś N jest liczbą argumentów. W takim wypadku atom uważa się za strukturę bez żadnych argumentów. Oto przykłady użycia predykatu `functor`:

```
?- functor(f(a,b,g(Z)),F,N).
Z = _23, F = f, N = 3
?- functor(a+b,F,N).
F = +, N = 2
?- functor([a,b,c],F,N).
F = ., N = 2
?- functor(gruszka,F,N).
F = gruszka, N = 0
?- functor([a,b,c],',',3).
no
?- functor([a,b,c],a,Z).
no
```

Zanim przejdziemy do następnego predykatu, `arg`, zastanówmy się nad innym zastosowaniem `functor` — polega ono na tym, że pozostawiamy wolny (nieukonkretniony) pierwszy argument, T . Wtedy ukonkretnione muszą być oba pozostałe argumenty, funktor i liczba argumentów tego funktora. Cel tego typu nigdy nie zawodzi, zaś T zostanie ukonkretniony strukturą z podanym funktorem i określoną liczbą argumentów. Jest to zatem metoda *budowania* dowolnych struktur. Argumenty tak budowanych struktur są zmiennymi nieukonkretnionymi, zatem takie struktury pasują do wszystkich innych struktur mających taki sam funktor i tyle samo argumentów.

Typowym zastosowaniem predykatu `functor` jest budowa nowych struktur jako „kopi” struktur istniejących, przy czym argumenty mają być nieukonkretnione. W tym celu właśnie zdefiniujemy predykat `kopiuj`:

```
kopiuj(Stara,Nowa) :- functor(Stara,F,N), functor(Nowa,F,N).
```

Mamy tu dwa sąsiadujące ze sobą cele `functor`. Jeśli cel `kopiuj` ma ukonkretniony pierwszy argument, a drugi nie, pierwszy cel `functor` będzie też miał ukonkretniony pierwszy argument. Wobec tego F i N zostaną ukonkretnione nazwą funktora i liczbą jego argumentów na podstawie przekazanej struktury. W drugim celu `functor` mamy w takim wypadku do czynienia z drugim sposobem jego użycia: pierwszy argument nie jest ukonkretniony, więc na podstawie wartości F i N zostanie *zbudowana* struktura `Nowa`. Struktura ta ma taki sam funktor i tyle samo argumentów, co struktura `Stara`, ale argumenty są zmiennymi nieukonkretnionymi. Tak więc dialog może wyglądać następująco:

```
?- kopiuj(zdanie(to(rz(jan)),cz(czeka)),X).
X = zdanie(_23,_24)
```

Definiując w rozdziale 7. predykat `consult`, użyjemy celów `functor`, podobnie jak to pokazaliśmy tutaj.

`arg(N,T,A)`

Używając predykatu `arg` jako celu zawsze musimy ukonkretnić dwa jego pierwsze argumenty. Predykat ten pozwala sięgnąć do wybranego argumentu struktury. Pierwszy argument `arg` wskazuje, który argument nas interesuje. Drugi argument to struktura, której argument wybieramy. Prolog odnajduje odpowiedni argument struktury i stara się dopasować go do trzeciego argumentu `arg`, więc `arg(N,T,A)` nie zawiedzie, jeśli N -tym argumentem T jest A . Przyjrzyjmy się kilku przykładom:

```
?- arg(2,krewny(jan,matka(janina)),X).
X = matka(janina)
?- arg(1,a+(b+c),X).
X = a
?- arg(2,[a,b,c],X).
X = [b,c]
?- arg(1,a+(b+c),b).
no
```

Wpływ na nawracanie

Istnieją dwa predykaty wbudowane, które wpływają na normalne działanie Prologu podczas nawracania. Odcięcie, `!`, uniemożliwia ponowne spełnienie celów, zaś `repeat` tworzy nowe punkty nawracania tam, gdzie ich wcześniej nie było.

!

Na symbol odcięcia można patrzeć jako na predykat wbudowany zatwierdzający pewne wybory dokonane przez interpreter Prologu. Więcej na ten temat powiedzieliśmy w rozdziale 4. i nie będziemy tutaj tego powtarzać.

repeat

Predykat wbudowany `repeat` to dodatkowa metoda generowania wielu rozwiązań przez nawracanie. Wprawdzie jest to predykat wbudowany, ale łatwo można go zdefiniować jako:

```
repeat.  
repeat :- repeat.
```

Jakie znaczenie ma wstawienie `repeat` do którejś z reguł? Cel taki oczywiście nie wiadzie, a to dzięki pierwszej klauzuli `repeat`. Po drugie, jeśli w trakcie nawracania dojdziemy do `repeat`, Prolog będzie mógł podjąć poszukiwanie rozwiązania alternatywnego — użyta zostanie druga klauzula. Wygenerowany zostanie kolejny cel `repeat`, nastąpi uzgodnienie pierwszej klauzuli tego nowego celu i możliwe będzie przejście dalej. Po ponownym nawrocie predykat `repeat` zostanie użyty tak samo, jak to opisaliśmy przed chwilą. Predykat `repeat` może być wykonywany dowolnie wiele razy. Warto zwrócić uwagę na to, że istotna jest kolejność klauzul — co by się stało, gdyby je zamienić?

Po co w ogóle używać celu, który przy nawracaniu zawsze jest uzgadniany? Dzięki temu można generować nowe rozwiązania także w regułach, które same w sobie takich rozwiązań już nie mają, a w ten sposób można generować coraz to nowe wartości.

Przjrzyjmy się wbudowanemu predykatowi `get_char`, opisanemu w rozdziale 5. Kiedy Prolog usiłuje spełnić cel `get_char(X)`, pobiera następny znak (literę, cyfrę, spację lub cokolwiek innego) i przypisuje ten znak zmiennej `X`. Jeśli znak pasuje do `X`, cel jest uzgodniony; w przeciwnym razie `get_char` zawodzi. Nie ma żadnych punktów wyboru. Predykat `get_char` zawsze pobiera jeden następny znak. Przy kolejnym wywołaniu znowu zostanie odczytany jeden znak, ale znów nie będzie punktu wyboru. Możemy zdefiniować jednak predykat `nowy_get`:

```
nowy_get(X) :- repeat, get_char(X).
```

Predykat `nowy_get` ma tę właściwość, że generuje wartości kolejnych znaków jako rozwiązania alternatywne. Dlaczego? Kiedy pierwszy raz wywołujemy `nowy_get(X)`, podcel `repeat` jest uzgadniany i tak samo uzgadniany jest `get_char(X)`. Podczas nawracania najbliższym punktem wyboru jest `repeat`, więc Prolog „zapomina” o wszystkich

wcześniejszych ustaleniach i znów uzgadnia `repeat`, następnie ponownie wywołuje `get_char(X)`. Teraz odczytywany jest znak następujący za znakiem odczytanym poprzednio, `X` przybiera nową wartość.

Możemy wykorzystać nasz predykat `nowy_get` w innym przydatnym predykanie, który pobijać będzie dane wejściowe, póki nie zostanie znaleziony czarny znak. Kiedy Prolog natknie się na cel `get_non_space(X)`, będzie odczytywał kolejne znaki tak długo, aż znaleziony zostanie czarny znak (czyli nie spacja). Wtedy podejmowana jest próba uzgodnienia kodu znaku z `X`. Możemy zapisać podobny predykat następująco:

```
get_non_space(X) :- nowy_get(X), \+X = ' '.
```

Co się zatem dzieje, kiedy próbujemy spełnić cel `get_non_space(X)`? Przede wszystkim, `nowy_get(X)` dopasowuje `X` do następnego odczytanego znaku. Jeśli wartość ta jest równa `' '`, następny cel zawodzi i `nowy_get` generuje następny znak jako możliwe rozwiązanie. Ten nowy znak też jest porównywany z `' '`, i tak dalej. Kiedy znaleziony zostanie w końcu czarny znak, porównanie się powiedzie i jako wynik `get_non_space` zwrócono zostanie odczytany znak.

Ćwiczenie 6.1. Powyższa definicja `get` nie zawsze zadziała, jeśli wywołany zostanie cel `get_non_space(X)`, w którym argument `X` będzie już ukonkretniony. Dlaczego?

Niedogodnością związaną z `repeat` jest to, że zawsze istnieje w tym predykanie punkt nawrotu, więc niemożliwy jest nawrót przed ostatnim `repeat`, chyba że zostanie odpowiednio użyte odcięcie. Z tego powodu powyższą definicję należy zmodyfikować następująco:

```
nowy_get(X) :- repeat, get_char(X).  
get_non_space(X) :- nowy_get(X), \+ X = ' ', !.
```

Zauważmy, że definicja ta nadal działa tylko wtedy, gdy argument `X` predykatu `get_non_space` nie jest ukonkretniony. Z uwagi na problemy związane z nawracaniem przez `repeat`, przy każdym użyciu `nowy_get` trzeba zatroszczyć się o odcięcie tego predykatu najszybciej, jak to możliwe.

Tworzenie celów złożonych

Jeśli reguły i zapytania mają postać `X :- Y` lub `?- Y`, term `Y` może być pojedynczym celem lub koniunkcją celów, albo też alternatywą. Poza tym celami mogą być zmienne, może występować także zaprzeczenie celów, not. Predykaty opisane w tym podrozdziale umożliwiają tworzenie złożonych celów.

X, Y

Operator `,` definiuje koniunkcję celów. Operator ten przedstawiliśmy w rozdziale 1. Jeśli `X` i `Y` są celami, cel `X, Y` nie zawiedzie, jeśli nie zawiedzie `X` ani `Y`. Jeśli `X` zostanie uzgodniony, a `Y` zawiedzie, podejmowana jest próba ponownego uzgodnienia `X`.

Jeśli zawiedzie X , zawiedzie też cała koniunkcja. Na tym właśnie opiera się nawracanie. „ $;$ ” jest wewnętrznie zadeklarowany jako prawostronnie łączny operator infiksowy, więc zapis $X ; Y ; Z$ jest równoważny z zapisem $X ; (Y ; Z)$.

$X ; Y$

Operator $;$ to alternatywa (*lub*) celów. Jeśli X i Y są celami, cel $X ; Y$ nie zawiedzie, jeśli nie zawiedzie X lub Y . Jeśli zawiedzie X , następuje próba spełnienia Y . Jeśli zawiedzie też Y , zawodzi cała alternatywa. Operatora $;$ można użyć do zapisu alternatyw w ramach tej samej klauzuli. Załóżmy na przykład, że obiekt jest osobą, jeśli jest Adamem lub Ewą lub ma matkę. Możemy to zapisać w jednej regule następująco:

```
osoba(X) :- (X=adam; X=ewa; matka(X,Y)).
```

W regule tej wskazaliśmy trzy możliwości. Jednak z punktu widzenia Prologu następuje rozbieżność na dwie alternatywy, z których jedna znowu rozbijana jest na dwie kolejne alternatywy. Z uwagi na to, że $;$ zadeklarowany jest wewnętrznie jako prawostronnie łączny operator infiksowy, powyższą klauzulę należy odczytać następująco:

```
osoba(X) :- ' ; '(X=adam. ' ; '(X=ewa,matka(X,Y)) ) .
```

Tak więc pierwsza możliwość polega na tym, że X to adam. Druga możliwość, to albo X to ewa, albo X ma matkę.

Alternatywy można wstawiać wszędzie tam, gdzie można wstawiać inne rodzaje celów Prologu. Dobrze jednak dodać nawiasy, aby uniknąć wątpliwości związanych z połączeniem operatorów $;$ i $;$. Zwykle alternatywę można zastąpić przez zapisanie kilku faktów i reguł, ewentualnie zdefiniowanie dodatkowego predykatu. Przykładowo, powyższy predykat jest równoważny z następującym:

```
osoba(adam).
osoba(ewa).
osoba(X) :- matka(X,Y).
```

Ta wersja jest bliższa temu, do czego się przyzwyczailiśmy, i jest łatwiejsza do zinterpretowania. Nie zaleca się nadmiernie eksploatować $;$. W rozdziale 8. przedstawimy kilka ostrzeżeń demonstujących, jak użycie średnika ($;$) może prowadzić do nieczytelności programów.

call(X)

Zakłada się, że X jest ukonkretnionym termem, który może zostać zinterpretowany jako cel. Cel $call(X)$ nie zawiedzie, jeśli nie zawiedzie X i zawiedzie, jeśli zawiedzie próba spełnienia X . Na pierwszy rzut oka predykat $call$ może wydawać się zbędny, bo czyż sam argument $call$ nie mógłby być po prostu celem? Na przykład cel

```
.... call(member(a,X)), ....
```

można byłoby zastąpić

```
.... member(a,X), ....
```

Jeśli jednak tworzymy cele za pomocą predykatu $=..$ lub jemu podobnych, można wywołać cele mające funktor *nieznany* w chwili tworzenia programu. Na przykład w definicji `consult` w podrzdziale „Przetwarzanie programów” rozdziału 7. chcemy mieć możliwość traktowania dowolnego termu $zza ?-$ jako celu. Zakładając, że P , X i Y są ukonkretnione odpowiednio jako funktor i argumenty, możemy użyć `call` następująco:

```
.... Z =.. [P,X,Y], call(Z), ....
```

Powyższy wiersz można potraktować jako pewnego rodzaju wywołanie, które jest *niezgodne ze składnią* standardowego Prologu, używaną w tej książce:

```
.... P(X,Y), ....
```

$\backslash + X$

Predykat $\backslash +$ (czytany jako *not*) zadeklarowany został jako operator przedrostkowy. Zakłada się, że argument X jest ukonkretniony termem, który może zostać zinterpretowany jako cel. Cel $\backslash + X$ jest uzgadniany, jeśli X zawodzi i zawodzi, jeśli X można uzgodnić. Przez to $\backslash +$ działa podobnie jak `call`, ale odwrócona jest interpretacja uzgodnienia i zawiedzenia. Czym różnią się poniższe zapytania?

```
?- member(X,[a,b,c]), write(X).
?- \+ \+ member(X,[a,b,c]), write(X).
```

Można byłoby zgadywać, że niczym, gdyż w drugim zapytaniu

```
member(X,[a,b,c]) nie zawodzi, więc
```

```
\+ member(X,[a,b,c]) zawodzi, zatem
```

```
\+ \+ member(X,[a,b,c]) nie zawodzi.
```

Częściowo jest to prawda, ale pierwsze zapytanie spowoduje wypisanie atomu `a`, a drugie spowoduje wypisanie *zmiennej nieukonkretnionej*. Oto co się dzieje przy próbie uzgodnienia pierwszego celu drugiego z powyższych zapytań:

1. Cel `member` nie zawodzi, X ukonkretniane jest termem `a`.
2. Próba uzgodnienia pierwszego $\backslash +$ zawodzi, gdyż cel `member`, argument naszego $\backslash +$, nie zawodzi. Wobec wszystkich zmiennych wycofywane jest ukonkretnienie, w szczególności zwalniana jest zmienna X .
3. Podejmowana jest próba uzgodnienia drugiego celu $\backslash +$, próba ta się udaje, gdyż argument $\backslash +$, czyli $\backslash + \text{member}(\dots)$, zawiódł. Niemniej jednak X pozostaje nieukonkretniona.
4. Następuje próba uzgodnienia celu `write`, ale zmienna X jest nieukonkretniona. Zgodnie z opisem w podrzdziale „Wejście i wyjście”, zmienna ta jest pokazywana w specjalny sposób.

Równość

W tym podrozdziale zajmiemy się pokrótce różnymi predykatami pozwalającymi porównywać i zrównywać obiekty.

X = Y

Kiedy Prolog natyka się na cel $X = Y$, stara się zrównać X i Y dopasowując je do siebie. Jeśli jest to możliwe, cel jest uzgodniony (przy tym X i Y mogą zostać bardziej ukonkretnione). W przeciwnym razie cel zawodzi. Dokładniej zagadnienie to omówiliśmy w podrozdziale „Równość i dopasowywanie” w rozdziale 2. Definicja predykatu równości wygląda następująco:

```
X = X.
```

Warto spróbować samemu zrozumieć, jak taka definicja działa.

X == Y

Predykat `==` powoduje znacznie dokładniejsze porównanie niż `=`. Jeśli nie zawodzi $X == Y$, to nie zawiedzie też $X = Y$, ale nie odwrotnie. Predykat `==` znacznie dokładniej analizuje zmienne. Predykat `=` traktuje zmienną nieukonkretnioną jako równą dowolnej wartości, zaś dla predykatu `==` zmienna nieukonkretniona jest równa jedynie zmiennej z nią związanej. Tak więc otrzymujemy następujące odpowiedzi:

```
?- X = Y.
no
?- X == X.
X = _23
?- X=Y, X==Y.
X = _23, Y = _23
?- append([A|B],C) == append(X,Y).
no
?- append([A|B],C) == append([A|B],C).
A = _23, B = _24, C = _25
```

Wejście i wyjście

Predykaty pozwalające odczytywać i zapisywać znaki i termy omówiliśmy już w rozdziale 5., teraz jeszcze je krótko przypomnimy.

get_char(X)

Cel `get_char(X)` nie zawodzi, jeśli X można dopasować do następnego znaku z bieżącego strumienia wejściowego. `get_char` może być uzgodniony tylko raz (nie można go uzgodnić ponownie). Operacja przejścia do następnego znaku jest nieodwracalna podczas nawracania, gdyż znaku nie można z powrotem wstawić do strumienia wejściowego.

read(X)

Cel ten odczytuje następny term z bieżącego strumienia wejściowego, następnie dopasowuje ten term do X . `read` uzgadniany jest tylko raz. Odczytywany term musi być zakończony kropką, która nie jest częścią tego termu, oraz przynajmniej jednym znakiem niedrukowalnym. Kropka jest usuwana z bieżącego strumienia wejściowego.

put_char(X)

Cel `put(X)` wypisuje do bieżącego strumienia wyjściowego znak X . `put` może być uzgodniony tylko raz. Jeśli argument X nie jest ukonkretniony, występuje błąd.

nl

Wypisuje do bieżącego strumienia wyjściowego sekwencję powodującą przejście do nowego wiersza. Wszystkie dalsze znaki są wyświetlane lub drukowane w następnym wierszu. `nl` może być uzgodniony tylko raz.

write(X)

Cel `write(X)` powoduje zapisanie termu X do bieżącego strumienia wyjściowego. `write` może być uzgodniony tylko raz. Wszystkie zmienne nieukonkretnione pojawiające się w X wypisane są jako jednoznacznie ponumerowane zmienne, których nazwy zaczynają się od podkreślenia — na przykład `_239`. Jeśli jakieś zmienne są ze sobą powiązane, mają taki sam numer. Predykat `write` podczas pokazywania termu uwzględnia deklaracje operatorów, więc na przykład operatory infiksowe będą pokazywane między swoimi argumentami.

write_canonical(X)

Predykat `write_canonical` zachowuje się tak samo jak `write`, ale nie uwzględnia deklaracji operatorów: w jego przypadku przy wyświetlaniu struktur najpierw pokazywany jest funktor, a za nim lista argumentów ujętych w nawiasy.

op(X,Y,Z)

Cel `op` pozwala zadeklarować operator o priorytecie X , położeniu i łączności określonych argumentem Y i o nazwie Z . Położenie i łączność wyrażane są jednym z poniższych atomów:

```
fx  fy  xf  yf  xfx  xfy  yfx  yfy
```

Jeśli deklaracja operatora jest poprawna, `op` nie zawodzi. Więcej informacji na ten temat znajduje się w rozdziale 5., w podrozdziale „Deklarowanie operatorów”.

**Wyższa Szkoła
Zarządzania i Bankowości**
ul. Armii Krajowej 4, 30-150 Kraków
tel. (012) 638-66-77 tel.fax (012) 637-33-47
wpis do Rejestru MKN nr 55 z dnia 11.05.1995r.
Konto GOS O/Kraków 15-01115-12087-27006-00
NIP 677-17-58-109 REGON 350814846

Obsługa plików

Predykaty umożliwiające zmianę bieżących strumieni wejściowego i wyjściowego omówiliśmy w rozdziale 5., teraz krótko je przypomnimy.

open(X,Y,Z)

Cel ten otwiera plik o nazwie *X* (atom). Jeśli *Y* ma wartość *read*, plik jest otwierany do czytania, w przeciwnym razie plik jest otwierany do pisania. *Z* jest ukonkretniane specjalnym termem będącym nazwą strumienia, którego należy używać podczas odwoływania się do pliku. Jeśli *X* nie jest ukonkretnione lub jest nazwą nieistniejącego pliku, występuje błąd.

close(X)

Predykat używany do zamknięcia strumienia o nazwie *X*. Strumień staje się niedostępny.

set_input(X)

Ustawia bieżący strumień wejściowy na strumień wskazany przez *X*. *X* jest termem zwracającym jako trzeci argument *open* lub atomem *user_input*, nakazującym pobieranie danych z klawiatury.

set_output(X)

Ustawia bieżący strumień wyjściowy na strumień wskazany przez *X*. *X* jest termem zwracającym jako trzeci argument *open* lub atomem *user_output*, nakazującym wypisywanie danych na monitorze.

current_input(X)

Cel jest uzgodniony, jeśli nazwa bieżącego strumienia wejściowego pasuje do *X*, a zawodzi w przeciwnym razie.

current_output(X)

Cel jest uzgodniony, jeśli nazwa bieżącego strumienia wyjściowego pasuje do *X*, a zawodzi w przeciwnym razie.

Wyliczanie wyrażeń arytmetycznych

Arytmetykę Prologu omawialiśmy już w rozdziale 2., teraz podsumujemy użycie predykatu *is* i przypomnimy dostępne funktory służące do budowy wyrażeń arytmetycznych.

X is Y

Argument *Y* musi być ukonkretniony strukturą, którą można zinterpretować jako wyrażenie arytmetyczne zgodne z opisem z podrozdziału „Operatory” w rozdziale 2. Najpierw struktura odpowiadająca *Y* jest wartościowana w celu określenia jej *wyniku* — liczby całkowitej. Wynik jest dopasowywany do *X* i w zależności od tego, czy dopasowanie się uda, cały cel zawodzi lub nie. W strukturze występującej po prawej stronie *is* można używać następujących operatorów:

X + Y

Operator dodawania. Przy wartościowaniu go predykatem *is* daje sumę swoich argumentów. Argumenty muszą być liczbami lub strukturami dającymi w wyniku liczbę.

X - Y

Operator odejmowania. Przy wartościowaniu go predykatem *is* daje różnicę swoich argumentów. Argumenty muszą być liczbami lub strukturami dającymi w wyniku liczbę.

X * Y

Operator mnożenia. Przy wartościowaniu go predykatem *is* daje iloczyn swoich argumentów. Argumenty muszą być liczbami lub strukturami dającymi w wyniku liczbę.

X / Y

Operator dzielenia zmiennoprzecinkowego. Przy wartościowaniu go predykatem *is* daje iloraz swoich argumentów (zwykle nie jest to liczba całkowita). Argumenty muszą być liczbami lub strukturami dającymi w wyniku liczbę.

X // Y

Operator dzielenia całkowitoliczbowego. Przy wartościowaniu go predykatem *is* daje iloraz swoich argumentów w formie liczby całkowitej, największej liczby mniejszej od *X/Y*. Argumenty muszą być liczbami lub strukturami dającymi w wyniku liczbę.

X mod Y

Operator mnożenia *modulo*. Przy wartościowaniu go predykatem *is* daje resztę z całkowitoliczbowego dzielenia swoich argumentów. Argumenty muszą być liczbami lub strukturami dającymi w wyniku liczbę.

Poszczególne implementacje Prologu mogą zawierać dodatkowe operacje arytmetyczne, np. potęgowanie. W przykładach z tej książki używamy jedynie operatorów wymienionych powyżej.

Porównywanie termów

Predykaty pozwalające porównywać termy omówiliśmy w rozdziale 2. Wszystkie one są dwuargumentowymi operatorami infiksowymi.

$X = Y$

Równość opisana wcześniej w tym rozdziale, w podrozdziale „Równość”. Operator ten nie zawodzi, jeśli dwa argumenty całkowitoliczbowe są takie same.

$X < Y$

Predykat mniejszości nie zawodzi, jeśli lewy argument jest mniejszy od argumentu prawego. Oba argumenty muszą być ukonkretnione.

$X > Y$

Predykat większości nie zawodzi, jeśli lewy argument jest większy od argumentu prawego. Oba argumenty muszą być ukonkretnione.

$X \geq Y$

Predykat większości bądź równości nie zawodzi, jeśli lewy argument jest większy od argumentu prawego lub mu równy. Oba argumenty muszą być ukonkretnione.

$X \leq Y$

Predykat mniejszości bądź równości nie zawodzi, jeśli lewy argument jest mniejszy od argumentu prawego lub mu równy. Oba argumenty muszą być ukonkretnione. Warto zauważyć, że operator ten zapisywany jest jako `=<` zamiast `<=`, dzięki czemu można użyć tego ostatniego jako operatora w kształcie strzałki.

Poza tym w standardzie Prologu uwzględniono predykaty pozwalające porównywać dwa dowolne termy. Co jednak to ma oznaczać — czy na przykład `f(X)` jest mniejszy od 123? Zwykle porównywane są termy tego samego typu, na przykład atomy (zobacz słownik w formie drzewa, rozdział 7.). Jednak czasami przydatne bywa porównywanie także różnych typów danych. Oto zasady decydujące o tym, który z danych termów jest mniejszy:

- ♦ Wszystkie zmienne nieukonkretnione są mniejsze od wszystkich liczb zmiennoprzecinkowych, te są mniejsze od wszystkich liczb całkowitych, te są mniejsze od wszystkich atomów, a te są mniejsze od jakichkolwiek struktur.
- ♦ W przypadku dwóch niepowiązanych zmiennych nieukonkretnionych zawsze jedna z nich będzie mniejsza od drugiej (która, zależy już od konkretnej implementacji Prologu).
- ♦ Porównywanie liczb zmiennoprzecinkowych i całkowitych odbywa się zgodnie z intuicją.

- ♦ Atom jest mniejszy od innego, jeśli występuje przed nim w zwykłym porządku słownikowym. Ściśle rzecz biorąc, uporządkowanie to zależy od kodów znaków, choć alfabet łaciński zwykle jest uporządkowany zgodnie z intuicją.
- ♦ Jedna struktura jest mniejsza od innej, jeśli jej funktor ma mniej argumentów. Jeśli dwie struktury mają tyle samo argumentów, mniejsza jest ta, której funktor jest mniejszy (funktor porównuje się jak atomy). Jeśli dwie struktury mają tyle samo argumentów i takie same funktory, bada się kolejno ich argumenty: pierwszy z pierwszym, drugi z drugim i tak dalej.

Oto przykłady niezawodzących porównań dowolnych termów:

```
?- g(X) @< f(X,Y).
?- f(Z,b) @< f(a,A).
?- 123 @< 124.
?- 123.5 @< 2.
```

$X @< Y$

Predykat nie zawodzi, jeśli jego lewy argument jest mniejszy od argumentu prawego przy takim uporządkowaniu, jakie opisano powyżej.

$X @> Y$

Predykat nie zawodzi, jeśli jego lewy argument jest większy od argumentu prawego przy takim uporządkowaniu, jakie opisano powyżej.

$X @>= Y$

Predykat nie zawodzi, jeśli jego lewy argument jest większy od argumentu prawego przy takim uporządkowaniu, jakie opisano powyżej lub jeśli oba argumenty są takie same.

$X @<= Y$

Predykat nie zawodzi, jeśli jego lewy argument jest mniejszy od argumentu prawego przy takim uporządkowaniu, jakie opisano powyżej lub jeśli oba argumenty są takie same.

Badanie działania Prologu

W tym podrozdziale opiszemy predykaty wbudowane, które umożliwiają obserwację wykonywania programu. W standardzie Prologu nie ustalono odpowiedniego zestawu predykatów tego typu, predykaty tutaj wymienione mają służyć jako prezentacja dostępnych możliwości. Opiszemy jedynie predykaty wbudowane, zaś w rozdziale 8. więcej miejsca poświęcimy usuwaniu błędów i śledzeniu wykonywania programu.

trace

Wynikiem uzgodnienia `trace` jest włączenie dokładnego raportowania wykonania programu. Od tej chwili wszystkie cele programu będą generowały informacje na wszystkich czterech głównych portach.

notrace

Uzgodnienie celu `notrace` powoduje wyłączenie dokładnego raportowania wykonywania programu, choć nadal może być realizowane śledzenie wynikające z obecności punktów śledzenia wstawionych za pomocą predykatów `spy`.

spy P

Predykatu `spy` używa się, jeśli konieczne jest zwrócenie dokładniejszej uwagi na cele zawierające wybrane predykaty. Na takich predykatkach ustawia się *punkty śledzenia*. Predykat `spy` zdefiniowano jako operator prefiksowy, więc argumentu nie trzeba ujmować w nawias. Argument może mieć jedną z trzech postaci:

- ◆ Atom. Punktami śledzenia stają się wszystkie predykaty o danej nazwie, niezależnie od liczby ich argumentów. Gdyby zatem istniały klauzule `sort` z dwoma i trzema argumentami, zarówno jedno, jak i drugie byłyby śledzone.
- ◆ Struktura w formie `Nazwa/Liczba`, gdzie `Nazwa` to atom, a `Liczba` oznacza liczbę argumentów. Taki zapis oznacza nakaz śledzenia predykatu o funktorze `Nazwa`, mającego `Liczba` argumentów. Wobec tego cel `spy sort/2` spowoduje ustawienie punktów śledzenia na dwuargumentowe klauzule predykatu `sort`.
- ◆ Lista. Lista musi być zakończona `[]`, każdy element tej listy musi być poprawnym argumentem `spy`. Prolog będzie śledził wszystkie punkty wskazane na liście. Jeśli zatem użyjemy celu `spy [sort/2, append/3]`, śledzone będą dwuargumentowe klauzule `sort` i trzyargumentowe klauzule `append`.

debugging

Predykat wbudowany `debugging` pozwala sprawdzić, jakie punkty śledzenia zostały ustawione. Pokazanie listy tych punktów jest efektem ubocznym spełnienia celu `debugging`.

nodebug

Cel `nodebug` powoduje usunięcie wszystkich punktów śledzenia.

nospy

Tak jak `spy`, `nospy` jest operatorem prefiksowym. `nospy` działa podobnie jak `nodebug`, ale jest precyzyjniejszy, gdyż pozwala wskazać, które punkty śledzenia mają być usunięte. Tak więc `nospy [reverse/2, append/3]` spowoduje usunięcie punktów śledzenia z dwuargumentowych klauzul `reverse` i trzyargumentowych klauzul `append`.

Rozdział 7.

Przykładowe programy

Każdy podrozdział tego rozdziału poświęcimy innemu zastosowaniu Prologu. Warto zapoznać się z każdym z nich. Nie należy zanadto przejmować się, jeśli niejasny będzie cel stawiany przed programem z powodu nieznamości danego zastosowania — na przykład tylko osoby zaznajomione z rachunkiem różniczkowym będą mogły docenić wagę przedstawionego różniczkowania symbolicznego. Warto jednak także ten podrozdział przeczytać, gdyż program wyznaczający symboliczne pochodne pokazuje, jak można użyć dopasowywania wzorca do przekształcania jednych struktur (wyrażeń arytmetycznych) w inne. Ważne jest zrozumienie zastosowanych technik programowania niezależnie od ich konkretnego użycia.

Mamy nadzieję, że niniejszy rozdział zawiera dość aplikacji, aby zaspokoić różne gusta naszych czytelników. Oczywiście wszystkie proponowane przykłady obejmują zagadnienia, które ze swojej natury są zgodne z filozofią Prologu, wobec czego na przykład nie pokazujemy, jak można wyliczyć przepływ ciepła przez kwadratową rurę z metalu. Rozwiązanie takiego problemu w Prologu jest możliwe, ale siła tego języka nie ujawnia się w przypadku programów, w których konieczne jest robienie powtarzalnych obliczeń na macierzach. Chcielibyśmy omówić duże systemy pisane w Prologu, jak choćby systemy używane w sztucznej inteligencji na przykład do rozpoznawania języka naturalnego. Niestety, w książkach takich jak ta konieczne jest ograniczenie się do programów mieszczących się mniej więcej na jednej stronie oraz unikanie programów, które będą interesujące jedynie dla wąskiej grupy specjalistów.

Sortowany słownik w formie drzewa

Załóżmy, że mamy zestaw powiązanych ze sobą danych, które chcemy w miarę potrzeb pobierać. Na przykład, w Słowniku języka polskiego mamy słowa powiązane z ich definicjami, w słowniku polsko-angielskim mamy słowa powiązane z ich odpowiednikami w języku angielskim. Wiemy już, jak można taki słownik utworzyć: przy użyciu faktów.

Gdybyśmy chcieli stworzyć słownik zawierający informacje o osiągnięciach koni podczas wyścigów w Anglii w 1938 roku, moglibyśmy po prostu użyć faktów `wygrane(X,Y)`, gdzie X byłoby imieniem konia, a Y liczbą gwinei zdobytych przez konia. Oto przykładowy fragment takiej bazy danych:

```
wygrane(abaris,582).
wygrane(careful,17).
wygrane(jingling_silver,300).
wygrane(maloja,356).
```

Gdybyśmy chcieli dowiedzieć się teraz, ile wygrał koń `maloja`, wystarczyłoby zadać odpowiednie zapytanie:

```
?- wygrane(maloja,X).
X=356
```

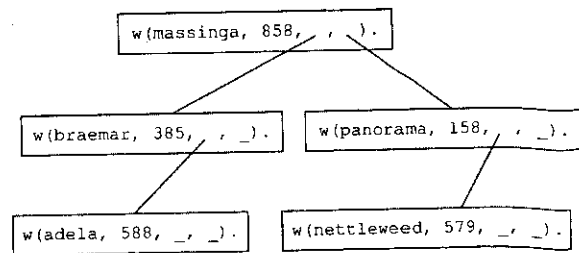
Pamiętajmy, że przeszukując bazę danych Prolog przegląda ją od początku do końca. Oznacza to, że jeśli nasz słownik jest ułożony alfabetycznie, zostaną szybko znalezione informacje o koniu `ablaze`, a trochę dłużej będzie trwało szukanie informacji o koniu `zoltan`. Wprawdzie Prolog potrafi takie informacje odszukać znacznie szybciej niż my zrobilibyśmy to w papierowym zestawieniu, ale nierozważne jest szukanie od początku danych o koniu, o którym wiemy, że znajduje się pod koniec zestawienia.

Wprawdzie Prolog projektowano tak, aby szybko przeszukiwał bazę danych, jednak nie zawsze szybkość ta jest zadowalająca. W zależności od wielkości zestawienia i od ilości danych o poszczególnych koniach, w pewnych sytuacjach oczekiwanie na odpowiedź może być nieprzyjemnie długie.

Opisane wyżej przyczyny — choć nie tylko one — spowodowały, że informatycy tak wiele czasu poświęcili zapisywaniu informacji w formie indeksów i słowników. Sam Prolog używa wewnętrznie tego typu struktur do przechowywania faktów i reguł, ale czasami dobrze jest też z tych metod skorzystać w programie. Opiszemy jedną z takich metod zapisu słownika w formie *posortowanego drzewa*. Drzewo takie przeszukuje się szybko, poza tym może służyć jako ilustracja przydatności list.

Posortowane drzewo składa się ze struktur nazywanych *węzłami*, z których jeden jest węzłem początkowym całego drzewa. Każdy węzeł ma cztery składniki. Dwa elementy zawierają dane, jak w powyższym predykanie `wygrane`. Pierwszy z tych elementów to *klucz*, którego zawartość decyduje o miejscu w słowniku (u nas jest to imię konia). Drugi z nich może zawierać dowolne informacje, w naszym przykładzie jest to wielkość wygranej. Poza tym każdy węzeł zawiera *ogon* (podobny do ogona listy) wskazujący węzeł z kluczem *mniejszym* od klucza bieżącego (alfabetycznie). Poza tym każdy węzeł zawiera jeszcze jeden węzeł, który wskazuje węzeł z kluczem *większym* od klucza danego węzła.

Użyjemy struktury `w(K,W,P,N)`; w to skrót od `wygrane`, K oznacza konia (jest to atom), W to wygrane w gwineach (liczba całkowita), P to struktura z poprzednim koniem („mniejszym”), zaś N — z następnym. Jeśli nie ma struktur odpowiadających P i N , użyjemy zmiennych nieukonkretnionych. Jeśli mamy niewiele koni, odpowiedni słownik w formie drzewa będzie wyglądał następująco:



Gdy zapiszemy tę samą strukturę w Prologu, stosując niewielkie wcięcia, otrzymamy:

```
w(massinga, 858,
  w(braemar, 385,
    w(adela, 588, _, _),
    _),
  w(panorama, 158,
    w(nettleweed, 579, _, _),
    _),
  _)
```

Teraz, kiedy mamy tego typu strukturę, możemy odszukiwać imiona koni i sprawdzać, ile gwinei konie te wygrały w 1938 roku. Struktura ma format `w(K,W,P,N)`, jak pokazano powyżej. Warunkiem końcowym jest sytuacja, kiedy imię szukanego konia to K . W przeciwnym razie używamy predykatu `mniejszy` pokazanego w rozdziale 3., aby podjąć decyzję, która gałąź drzewa ma być przeszukiwana rekurencyjnie: P czy N . Korzystając z takiego właśnie opisu definiujemy predykat `odszukaj`, gdzie `cel odszukaj(K,S,G)` oznacza, że koń K zgodnie ze strukturą S wygrał G gwinei:

```
odszukaj(K, w(K,G,_,_) , G) :- !.
odszukaj(K, w(K1,_,Poprzedni,_) , G) :-
    K @< K1,
    odszukaj(K, Poprzedni, G).
odszukaj(K, w(K1,_,Nastepny,_) , G) :-
    K @> K1,
    odszukaj(K, Nastepny, G).
```

Używając tego predykatu do przeszukiwania posortowanego drzewa, przede wszystkim musimy sprawdzić mniej koni, niż byłoby to konieczne w przypadku przeglądania listy od początku do końca.

Predykat `odszukaj` ma pewną ciekawą właściwość: jeśli szukamy konia, którego w naszej strukturze nie ma, wszystkie informacje podane do `odszukaj` w celu zostaną w strukturze ukonkretnione. Na przykład interpretacja `odszukaj` w zapytaniu

```
?- odszukaj(ruby_vintage,S,X).
```

jest następująca:

Stwórz strukturę S , taką, aby koń `ruby_vintage` był powiązany z X .

Wobec tego odszukaj wstawia nowe składniki do częściowo ukonkretnionej struktury, więc za pomocą kolejnych wywołań odszukaj możemy utworzyć cały słownik. Na przykład

```
?- odszukaj(abaris,X,582), odszukaj(małoja,X,356).
```

Spowoduje ukonkretnienie X posortowanym drzewem zawierającym dwie pozycje. To, jak odszukaj dodaje i pobiera dane, możemy wyjaśnić korzystając z przedstawionej wiedzy o Prologu, więc warto samemu popracować. Jeszcze tylko odpowiedź: kiedy w koniunkcji celów używamy odszukaj(K, S, G), zmiany w S obowiązują jedynie tam, gdzie S jest dostępna.

Ćwiczenie 7.1. Poeksperymentuj z odszukaj, aby sprawdzić, jak na ostateczny efekt wpływa kolejność wstawiania elementów. Czy na przykład wstawienie koni w kolejności: massinga, braemar, nettleweed, panorama da inny wynik niż wstawienie ich w kolejności: adela, braemar, nettleweed, massinga?

Przeszukiwanie labiryntu

Jest ciemna, burzowa noc. Kiedy przejeżdżasz przez opuszczony teren, samochód się psuje. Okazuje się, że obok drogi jest wspaniały pałac. Stajesz na jego schodach, podchodzisz do drzwi — otwarte. Wchodzisz, rozglądasz się w poszukiwaniu telefonu. Jak należy poruszać się po pałacu, aby się nie zgubić, a jednocześnie mieć pewność, że odwiedzone zostaną wszystkie komnaty? Jak znaleźć najkrótszą drogę do telefonu? Właśnie w takich sytuacjach przydatne są metody przeszukiwania labiryntów.

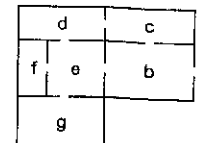
W wielu programach komputerowych, takich jak właśnie programy przeszukujące labirynty, zapamiętuje się wszystkie zebrane dotąd dane i w razie potrzeby przegląda je. Jeśli na przykład chcemy szukać w pałacu telefonu, warto mieć listę sprawdzonych dotąd pokoi, aby nie krążyć stale po kilku tych samych. Jeśli danego pokoju nie mamy zanotowanego, dopisujemy go do pokoi odwiedzonych i wchodzimy do niego, i tak dalej, aż znajdziemy telefon. Metodę tę można nieco udoskonalić, czym zajmiemy się omawiając przeszukiwanie grafów. Na razie jednak zapiszmy podstawowy algorytm, abyśmy wiedzieli, jak mamy postępować:

1. Podejdź do drzwi dowolnej komnaty.
2. Jeśli numer tej komnaty jest na liście pokoi już odwiedzonych, wróć do kroku 1. Jeśli nie ma żadnych innych drzwi, cofnij się do pokoju odwiedzanego ostatnio i tam wykonaj krok 1.
3. Jeśli numeru komnaty nie ma na liście, umieść ją tam.
4. Sprawdź, czy w pokoju jest telefon.
5. Jeśli nie ma telefonu, wróć do kroku 1. Jeśli telefon jest, to już koniec algorytmu — lista odwiedzonych pokoi zawiera ścieżkę prowadzącą do niego.

Zakładamy, że numery pokoi są stałe, choć mogą to być liczby czy atomy — nie ma to znaczenia. Najpierw rozwiążemy problem przeszukiwania pokoi na kartce papieru; skorzystamy z predykatu member pokazanego w rozdziale 3. Przyjrzyjmy się małemu pałacykowi — mamy jego plan, a pokoje są oznaczone literami, jak na rysunku 7.1.

Rysunek 7.1.
Plan domu
i opisujące go fakty

```
d(a,b).
d(b,e).
d(b,c).
d(d,e).
d(c,d).
d(e,f).
d(g,e).
```



Przerwy w ścianach oznaczają drzwi, zaś a to po prostu oznaczenie obszaru poza domem. Drzwi łączą a z b , c z d , f z e i tak dalej. Fakt istnienia tych drzwi możemy zapisać jako fakty Prologu.

Informacje o drzwiach nie powtarzają się — wprawdzie moglibyśmy stwierdzić, że oprócz drzwi z pokoju g do e istnieją drzwi z e do g , ale nie dopisaliśmy faktu $d(e,g)$.

Aby rozwiązać problem dwukierunkowości drzwi, moglibyśmy powtórzyć każdy fakt d dla każdych drzwi ze zmienioną kolejnością argumentów. Moglibyśmy też poinformować program, że przez drzwi można przechodzić w dowolnym kierunku — i tę właśnie możliwość wybraliśmy.

Aby przejść z jednego pokoju do drugiego, musimy stwierdzić jedno z dwojga:

- ♦ Jesteśmy w pokoju, do którego chcemy się dostać.
- ♦ Przeszliśmy przez drzwi i znowu powtarzamy sprawdzanie jednego z tych dwóch przypadków (rekurencyjnie).

Weźmy pod uwagę cel $go(X,Y,T)$, który nie zawodzi, jeśli można przejść z pokoju X do pokoju Y . Trzeci argument, T , to niesiony przez nas skrawek papieru, na którym zapisujemy odwiedzone dotąd pokoje.

Warunkiem końcowym przejścia z X do Y jest fakt, że już jesteśmy w pokoju Y , czyli $X = Y$. Opisuje to klauzula

```
go(X,X,T).
```

Jeśli nie mamy do czynienia z warunkiem końcowym, wybieramy jakiś pokój sąsiedni, niech to będzie Z , i sprawdzamy, czy byliśmy w nim wcześniej. Jeśli nie, używamy go do przejścia z Z do Y i dodajemy Z do listy. Klauzula ta ma postać:

```
go(X,Y,T) :- d(X,Z), \+ member(Z,T), go(Z,Y,[Z|T]).
```

Klauzulę tę opisowo możemy scharakteryzować tak:

Aby przejść z X do Y , nie przechodząc przez pokoje z listy T , musimy znaleźć drzwi z X do dowolnego Z spoza listy i przejść z Z do Y , dodając Z do listy.

Reguła ta może zawieść na trzy sposoby. Po pierwsze, w X może nie być żadnych drzwi. Po drugie, wybrane drzwi mogą być już na liście. Po trzecie, nie możemy przejść do Y z Z , co się okaże dopiero w dalszych wywołaniach rekurencyjnych. Jeśli zawiedzie pierwszy cel $d(X, Z)$, zawiedzie go. Na najwyższym poziomie (wywołaniu nierekurencyjnym) oznacza to, że nie istnieje droga z X do Y . Na wszystkich następnych poziomach oznacza to konieczność cofnięcia się i znalezienia innych drzwi.

Program w pokazanej postaci traktuje wszystkie drzwi jako „jednokierunkowe”. Jeśli założymy, że istnienie drzwi z a do b jest równoważne z istnieniem drzwi z b do a , musimy rzecz tę wyraźnie zapisać. Zamiast duplikować fakty d z przestawionymi argumentami, możemy dodać następującą regułę:

```
go(X, X, T).
go(X, Y, T) :- d(X, Z), \+ member(Z, T), go(Z, Y, [Z|T]).
go(X, Y, T) :- d(Z, X), \+ member(Z, T), go(Z, Y, [Z|T]).
```

Możemy też użyć predykatu : (czyli alternatywy):

```
go(X, X, T).
go(X, Y, T) :-
    (d(X, Z) ; d(Z, X)),
    \+ member(Z, T),
    go(Z, Y, [Z|T]).
```

Teraz przejdźmy do szukania telefonu. Zdefiniujmy cel `jesttelefon(X)`, który nie zawodzi, jeśli w pokoju X jest telefon. Jeśli chcemy zaznaczyć, że telefon jest w pokoju g , dopisujemy do bazy danych po prostu fakt

```
jesttelefon(g).
```

Załóżmy, że zaczynamy od pokoju a — możemy nakazać szukanie telefonu za pomocą zapytania:

```
?- go(a, X, []). jesttelefon(X).
```

Zapytanie to należy do grupy zapytań „generuj możliwe odpowiedzi i sprawdzaj je”: znajdowane są wszystkie pokoje, w których mógłby być telefon, następnie sprawdza się, czy telefon tam faktycznie jest. Inną możliwością jest użycie najpierw celu `jesttelefon(X)`, a potem szukanie drogi do X :

```
?- jesttelefon(X), go(a, X, []).
```

Jest to metoda szybsza, ale opieramy się w niej na tym, że z góry wiemy, gdzie jest interesujące nas urządzenie.

Inicjalizacja trzeciego argumentu pustą listą oznacza, że poszukiwanie zaczynamy od pustej kartki. Można to oczywiście zmienić i zadać zapytanie: „Znajdź telefon nie wchodząc do pokoi d i f ”:

```
?- jesttelefon(X), go(a, X, [d, f]).
```

Kiedy dalej w tym rozdziale zajmiemy się przeszukiwaniem grafów, pokażemy między innymi program znajdujący w grafie najkrótszą drogę.

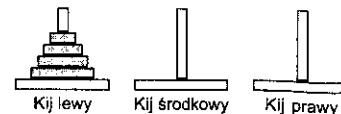
Ćwiczenie 7.2. Zmodyfikuj powyższy program tak, aby wyświetlał komunikaty typu „wchodzę do pokoju Y ” czy „w pokoju Y znaleziono telefon”.

Ćwiczenie 7.3. Czy pokazany program może znaleźć alternatywną drogę? Jeśli tak, gdzie należy wstawić odcięcie, aby żadna inna droga nie była znajdowana?

Ćwiczenie 7.4. Co decyduje o kolejności sprawdzania pokoi?

Wieże Hanoi

Wieże Hanoi to gra, w której używa się trzech kijów i zestawu krążków. Krążki różnią się średnicą, pośrodku mają dziurki pozwalające nawlec je na kijki. Początkowo wszystkie dyski znajdują się na lewym kiju, zaś celem gry jest przeniesienie ich na kijek środkowy. Prawy kijek to kijek pomocniczy, można na nim krążki umieszczać tymczasowo. W każdym ruchu trzeba pamiętać o dwóch zasadach: można przenieść tylko jeden, górny krążek na raz, poza tym dysku nie można nigdy kłaść na dysku od niego mniejszym.



Wiele osób grających w tę grę nie potrafi odkryć prostej strategii, która pozwala wygrać przy użyciu trzech kijów i N krążków. Aby zaoszczędzić czytelnikom wyważania otwartych drzwi, od razu podamy potrzebny algorytm:

- ♦ Warunkiem końcowym jest brak krążków na początkowym (lewym) kiju.
- ♦ Należy przenieść $N-1$ krążków z kija lewego na prawy przy użyciu kija środkowego jako pośredniego. Jest to krok rekurencyjny.
- ♦ Przenieść ostatni krążek z kija początkowego na docelowy.
- ♦ Przenieść $N-1$ krążków z kija pośredniego na docelowy, przy użyciu kija początkowego jako pośredniego.

W Prologu opisany algorytm można zaimplementować następująco: definiujemy jednoargumentowy predykat `hanoi`, taki, że `hanoi(N)` oznacza wydrukowanie ruchów pozwalających przenieść N krążków. Dalej mamy dwie klauzule `ruch`, z których pierwsza opisuje warunek końcowy, a druga realizuje rekurencję. Predykat `ruch` ma cztery argumenty. Pierwszy to liczba krążków do przeniesienia, pozostałe to kolejne kijki: początkowy, docelowy i pomocniczy. Predykat `informuj` za pomocą `write` pokazuje nazwy kijów, na które przenoszone są krążki.

```
hanoi(N) :- ruch(N, lewy, srodkowy, prawy).
ruch(0, _, _, _) :- !.
ruch(N, A, B, C) :-
    N is N-1,
    ruch(M, A, C, B), informuj(A, B), ruch(M, C, B, A).
informuj(X, Y) :-
    write([przenies, dysk, z, kija, X, na, kijek, Y]),
    nl.
```


że za X i Y mogą na liście występować jeszcze jakieś elementy — do tego właśnie służy zmienna anonimowa na ogon listy.

```
nextto(X,Y,[X,Y]_).
nextto(X,Y,[_Z]) :- nextto(X,Y,Z).
```

Łączenie list: przykład ten znamy już z rozdziału 3. Cel `append(X,Y,Z)` nie zawiedzie, jeśli Z jest listą powstałą z połączenia list X i Y , na przykład

```
?- append([a,b,c],[d,e,f],Q).
Q=[a,b,c,d,e,f]
```

Predykat ten definiujemy następująco:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Warunkiem końcowym jest to, że pierwszy argument jest listą pustą — dodanie listy pustej nie zmienia wyniku. Do warunku końcowego stopniowo zbliżamy się dzięki usuwaniu głowy pierwszego argumentu w każdym wywołaniu rekurencyjnym.

Ukonkretnione mogą być dowolne dwa argumenty `append`, wtedy trzeci argument zostanie ukonkretniony odpowiednią wartością, resztą cecha ta jest charakterystyczna dla wielu predykatów z tego podrozdziału. Dzięki elastyczności `append`, możemy na jego bazie zdefiniować różne inne predykaty:

```
last(E1,Lista) :- append(_,E1,Lista).
next_to(E1,E2,Lista) :- append(_,E1,E2|_,Lista).
member(E1,Lista) :- append(_,E1|_,Lista).
```

Odwracanie listy: cel `rev(L,M)` nie zawiedzie, jeśli lista M powstaje przez odwrócenie kolejności elementów listy L . Używamy standardowej techniki polegającej na przyłączeniu głowy do odwróconego ogona listy, zaś ogon odwracamy oczywiście za pomocą predykatu `rev`. Warunkiem końcowym jest zredukowanie pierwszego argumentu do listy pustej — wtedy wynikiem też jest lista pusta.

```
rev([],[]).
rev([H|T],L) :- rev(T,Z), append(Z,[H],L).
```

Zauważmy, że H jako drugi argument `append` zamknięto w nawiasie kwadratowym. Wynika to stąd, że wcześniej H pobraliśmy jako głowę pierwszego elementu, zaś głowa ta nie musi być wcale listą; ogon listy jest listą.

Można zdefiniować szybciej działającą wersję `rev`, w której złączenie zrealizujemy bezpośrednio w samym `rev`:

```
rev2(L1,L2) :- revzap(L1,[],L2).
revzap([X|L],L2,L3) :- revzap(L,[X|L2],L3).
revzap([],L,L).
```

Drugi argument `revzap` zawiera „odповідь uzyskaną dotąd”, czyli jest to akumulator. Zawsze, kiedy znajdujemy nowy fragment odpowiedzi (X), jako akumulator do następnego wywołania przekazywana jest nowa wartość połączona z dotychczasowym akumulatorem. Kiedy algorytm się kończy, wartość akumulatora zwracana jest jako ostateczna odpowiedź.

Usuwanie pojedynczego elementu: cel `efface(X,Y,Z)` usuwa pierwsze wystąpienie elementu X z listy Y , Z to skrócona lista. Jeśli w liście Y nie ma elementu X , predykat zawodzi. Warunkiem końcowym jest znalezienie elementu — w przeciwnym razie przetwarzany jest ogon Y .

```
efface(A,[A|L],L) :- !.
efface(A,[B|M],B[M]) :- efface(A,L,M).
```

Nietrudno dodać klauzulę powodującą, że predykat nie zawiedzie, jeśli drugi argument stanie się listą pustą. Nowa klauzula będzie miała postać:

```
efface(_,[],[]).
```

Usuwanie wszystkich wystąpień elementu: cel `delete(X,L1,L2)` tworzy listę $L2$, taką samą jak $L1$, ale z usuniętymi wszystkimi wystąpieniami elementu X . Warunek końcowy to lista $L1$ równa liście pustej, co oznacza, że przetworzona została cała lista wejściowa. Jeśli $L1$ nie jest pusta, jeśli X znajduje się na liście, wynikiem jest ogon tej listy pozbawiony pozostałych wystąpień X . Jeśli głową przekazanej listy nie jest X , po prostu wywołujemy następny krok rekurencyjny.

```
delete(_,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L1],[Y|L2]) :- delete(X,L1,L2).
```

Podstawienie: ta operacja podobna jest do `delete`, ale zamiast usuwać wskazany element, zamieniamy go na inny. Cel `subst(X,L,A,M)` tworzy nową listę M składającą się z elementów listy L poza elementami X , które są zastępowane elementami A . Mamy tu do czynienia z trzema przypadkami: pierwszy to warunek końcowy, tak samo jak w `delete`. Drugi przypadek polega na tym, że X jest głową drugiego argumentu, zaś trzeci przypadek to głowa inna niż X .

```
subst(_,[],_,[]).
subst(X,[X|L],A,[A|M]) :- !, subst(X,L,A,M).
subst(X,[Y|L],A,[Y|M]) :- subst(X,L,A,M).
```

Sublisty: lista X jest sublistą listy Y , jeśli wszystkie elementy z X występują w Y kolejno i w takiej samej kolejności. Nie zawiedzie na przykład taki cel:

```
sublist([naszego,klubu],[spotkania,naszego,klubu,przewidziano,na,czwartki]).
```

Program `sublist` wymaga zastosowania dwóch predykatów: jednego znajdującego dopasowanie do pierwszego elementu, drugiego zapewniającego, że reszta pierwszego argumentu pasuje do reszty drugiego argumentu:

```
sublist([X|L],[X|M]) :- prefix(L,M), !.
sublist(L,[_M]) :- sublist(L,M).
prefix([],_).
prefix([X|L],[X|M]) :- prefix(L,M).
```

Usuwanie duplikatów: predykat `remdup` przeszukuje listę i tworzy nową listę, która zawiera takie same elementy jak lista wejściowa, ale żaden element się nie powtarza. Cel `remdup(L,M)` nie zawiedzie, jeśli L jest listą wejściową, a M listą z tych samych elementów, ale bez powtórzeń. W definicji użyto predykatu pomocniczego `dupacc`, w którym drugim argumentem jest akumulator (omawiany w rozdziale 3.), początkowo będący listą pustą. Używamy też predykatu `member` z trzeciego rozdziału.

```
remdup(L,M) :- dupacc(L,[],M).
dupacc([],A,A).
dupacc([H|T],A,L) :- member(H,A), dupacc(T,A,L).
dupacc([H|T],A,L) :- \member(H,A), dupacc(T,[H|A],L).
```

Predykat `dupacc` ma trzy klauzule. Warunek końcowy mówi, że kiedy lista wejściowa jest pusta, wynikiem jest wartość akumulatora. Druga klauzula sprawdza, czy następny element listy występuje w liście z akumulatora; jeśli tak, po prostu przechodzimy do przetwarzania ogona. Jeśli nie, używamy ostatniej klauzuli i też przetwarzamy ogon, ale do akumulatora dodajemy nowy element, `H`.

Odwzorowania: bardzo silne narzędzie programistyczne pozwalające przekształcać jedną listę w inną w wyniku zastosowania pewnej funkcji do każdego elementu listy wejściowej, zaś otrzymywane wyniki stają się kolejnymi elementami drugiej listy. Nasz program z rozdziału 3. zmieniający jedno zdanie w inne był przykładem właśnie odwzorowania — odwzorowywaliśmy jedno zdanie w inne.

Odwzorowania są na tyle istotne, że zasługują na osobny podrozdział. Co więcej, jako że listy w Prologu są po prostu przypadkiem szczególnym struktur, odłożymy ich omówienie na koniec tego rozdziału, do podrozdziału „Odwzorowywanie struktur i przekształcanie drzew”. Z odwzorowaniami mamy też do czynienia w podrozdziale poświęconym różniczkowaniu symbolicznemu: jedne wyrażenia algebraiczne przekształcamy w inne.

!apis i przetwarzanie zbiorów

Zbiór to jedna z najważniejszych struktur danych w matematyce, zaś operacje na zbiorach znajdują zastosowanie w informatyce. Zbiór to, podobnie jak lista, zestaw elementów, ale w zbiorze nieważna jest kolejność elementów ani ich powtarzanie się. Tak więc zbiór $\{1,2,3\}$ jest takim samym zbiorem, jak $\{2,3,1\}$, gdyż istotne jest jedynie to, czy element w zbiorze występuje, czy nie. Elementami zbiorów mogą być też inne zbiory. Najbardziej podstawową operacją związaną ze zbiorami jest sprawdzanie, czy dany element występuje w zbiorze.

Naturalne wydaje się, że zbiory wygodnie jest zapisywać jako listy. Lista może zawierać dowolną liczbę elementów, w tym także inne listy, nietrudno zdefiniować też predykat sprawdzający, czy element należy do listy. Przyjmijmy jednak, że każdy element listy może na liście wystąpić tylko raz, gdyż uprosi to później niektóre operacje, takie jak usuwanie elementów. W przypadku opisywanych dalej predykatów będziemy właśnie zakładać, że używane listy nie mają duplikatów.

Zwykle w przypadku przetwarzania zbiorów używa się następującego zestawu działań:

Przynależność do zbioru: $X \in Y$

X należy do zbioru Y , jeśli X jest jednym z elementów Y .

Przykład: $0 \in \{k,o,t\}$

Podzbiór: $X \subseteq Y$

Zbiór X jest podzbiorem zbioru Y , jeśli każdy element zbioru X jest jednocześnie elementem zbioru Y . Y może poza tym zawierać elementy, które nie należą do X .

Przykład: $\{x,r,u\} \subseteq \{p,q,r,s,t,u,v,w,x,y,z\}$

Przecięcie: $X \cap Y$

Przecięcie zbiorów X i Y to zbiór zawierający te elementy X , które są jednocześnie elementami Y .

Przykład: $\{d,l,a,c,z,e,g,o\} \cap \{c,z,e,m,u\} = \{c,z,e\}$

Suma: $X \cup Y$

Suma zbiorów X i Y to zbiór zawierający wszystkie elementy, które należą choć do jednego z tych zbiorów.

Przykład: $\{a,b,c\} \cup \{c,d,e\} = \{a,b,c,d,e\}$

Teraz, kiedy już znamy najważniejsze operacje na zbiorach, możemy napisać odpowiadające im programy w Prologu. Pierwsza podstawowa operacja to przynależność do zbioru — działa ona podobnie jak `member`, choć nie zawiera odcięcia po warunku końcowym; jest to konieczne do wygenerowania wszystkich elementów listy przez nawracanie.

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

Następny predykat to `subset`: cel `subset(X,Y)` nie zawiedzie, jeśli X jest podzbiorem Y . Druga klauzula definicji służy do zapisania reguły, że zbiór pusty jest podzbiorem dowolnego zbioru. W Prologu sprowadza się to do zapisania warunku końcowego na pierwszy argument.

```
subset([A|X],Y) :- member(A,Y), subset(X,Y).
subset([],Y).
```

Następny przykład, przecięcie, jest bardziej skomplikowany. Cel `intersection(X,Y,Z)` nie zawiedzie, jeśli Z jest przecięciem (częścią wspólną) zbiorów X i Y . Zakładamy przy tym, że listy reprezentujące zbiory nie zawierają duplikatów.

```
intersection([],X,[]).
intersection([X|R],Y,[X|Z]) :-
    member(X,Y),
    !,
    intersection(R,Y,Z).
intersection([X|R],Y,Z) :- intersection(R,Y,Z).
```

I na koniec została nam jeszcze suma zbiorów. Cel `union(X,Y,Z)` nie zawiedzie, jeśli Z jest sumą mnogościową zbiorów X i Y . Warto zauważyć, że predykat ten przypomina trochę `intersection`, a trochę `append`:

```
union([],X,X).
union([X|R],Y,Z) :- member(X,Y), !, union(R,Y,Z).
union([X|R],Y,[X|Z]) :- union(R,Y,Z).
```

To już wszystkie operacje dotyczące zbiorów. Wprowadź zbiory nie w każdej sytuacji są przydatne, lecz warto przeanalizować podane przykłady, aby dobrze zrozumieć, jak można wykorzystać rekurencję i nawracanie.

Sortowanie

Czasami dobrze jest ustawić elementy listy w jakiejś kolejności. Jeśli elementy te są liczbami całkowitymi, do ich porównywania można użyć predykatu `<`. Lista `[1,2,3]` jest posortowana, gdyż predykat `<` nie zawiedzie dla każdej pary kolejnych liczb znajdujących się na niej. Jeśli elementy są atomami, możemy użyć predykatu „`@=<`”, omówionego na początku tego rozdziału. Lista `[alfa, beta, gamma]` jest posortowana, gdyż predykat `@=<` nie zawiedzie dla kolejnych par jej elementów.

W informatyce istnieje wiele technik sortowania list w kolejności określonej za pomocą jakiegoś predykatu. Pokażemy cztery programy służące do tego: sortowanie naturalne, sortowanie przez wstawianie, sortowanie bąbelkowe i *quicksort*. W każdym programie użyjemy predykatu kolejność, który można zdefiniować za pomocą predykatu `<`, `@=<` lub innego, który w danej sytuacji będzie najodpowiedniejszy. Zakładamy, że cel `kolejnosc(X,Y)` nie zawiedzie, jeśli obiekty `X` i `Y` są w zadanej kolejności, czyli `X` jest w jakiś sposób mniejszy od `Y`.

Jednym ze sposobów rosnącego uporządkowania elementów jest wygenerowanie jakiejś ich permutacji i następnie sprawdzenie, czy uzyskana lista jest uporządkowana rosnąco. Jeśli nie, generujemy inną permutację. Metodę tę nazywamy sortowaniem naturalnym.

```
sort(L1,L2) :- permutacja(L1,L2), posortowana(L2), !,
permutacja([], []).
permutacja(L,[H|T]) :-
    append(V,[H|U],L),
    append(V,U,W),
    permutacja(W,T),
posortowana([]),
posortowana([X]),
posortowana([X,Y|L]) :- kolejnosc(X,Y), posortowana([Y|L]).
```

Predykat `append` prezentowaliśmy już wielokrotnie wcześniej. W powyższym programie użyto predykatów: `sort(L1,L2)` wskazuje, że `L2` jest posortowaną wersją listy `L1`. `permutacja(L1,L2)` informuje, że `L2` jest listą składającą się z takich samych elementów jak `L1`, ale w różnej kolejności; zgodnie z terminologią z rozdziału 4. jest to *generator*. Predykat `posortowana(L)` sprawdza, czy lista przekazana mu jako argument jest posortowana rosnąco; jest to więc warunek.

Zadanie przedstawionego wyżej programu polega na generowaniu permutacji zestawu elementów i sprawdzaniu, czy otrzymana lista jest posortowana. Jeśli tak, znaleziona została jedyna możliwa odpowiedź. W przeciwnym razie generowane są kolejne permutacje. Opisana metoda sortowania list jest bardzo niewydajna.

Metoda *sortowania przez wstawianie* operuje na kolejnych elementach listy pierwotnej — każdy z nich wstawiany jest w odpowiednie miejsce nowej listy. Osoby grające w karty zapewne znają tę metodę z autopsji; tak zwykle porządkuje się karty w trzymanym w ręku wachlarzu. Cel `insort(X,Y)` nie zawiedzie, jeśli `Y` jest posortowaną listą `X`. Z głowy pierwotnej listy są usuwane kolejne elementy i przekazywane do `insortx`, predykatu wstawiającego dany element do listy i zwracającego uzyskaną listę:

```
insort([], []).
insort([X|L],M) :- insort(L,N), insortx(X,N,M),
insortx(X,[A|L],[A|M]) :-
    kolejnosc(A,X), !, insortx(X,L,M),
insortx(X,L,[X|L]).
```

Wygodnym sposobem zapisania ogólnego sortowania przez wstawianie jest podanie predykatu określającego porządek jako argumentu `insort`. Do procedury sortującej dodajemy trzeci argument, a następnie korzystamy z omawianego w rozdziale 6. predykatu `=..` do stworzenia celu, który potem wywołamy:

```
insort([], [], _).
insort([X|L],M,0) :- insort(L,N,0), insortx(X,N,M,0),
insortx(X,[A|L],[A|M],0) :-
    P =.. [O,A,X],
    call(P), !,
    insortx(X,L,M,0),
insortx(X,L,[X|L],0).
```

Teraz możemy wywoływać cele typu `insort(A,B,'<')` czy `insort(A,B,@=<)` bez konieczności odwoływania się do predykatu o nazwie `kolejnosc`. Tę samą technikę można zastosować do innych algorytmów przedstawianych w tym podrozdziale.

Sortowanie bąbelkowe polega na sprawdzaniu, czy dwa sąsiednie elementy są odpowiednio uporządkowane. Jeśli nie, ich kolejność jest zamieniana. Proces trwa tak długo, aż nie są już potrzebne żadne zmiany. O ile sortowanie przez wstawianie powoduje umieszczanie elementów w odpowiednich miejscach, sortowanie bąbelkowe powoduje, że elementy stopniowo „przepływają” na swoje miejsca.

```
busort(L,S) :-
    append(X,[A,B|Y],L),
    kolejnosc(B,A), !,
    append(X,[B,A|Y],M),
    busort(M,S),
busort(L,L),
append([],L,L),
append([H|T],L,[H|V]) :- append(T,L,V).
```

Predykat `append` jest taki sam jak poprzednio, ale tym razem musi on umożliwiać nawracanie przez kolejne rozwiązania — dlatego w pierwszej klauzuli nie pojawia się odcięcie. Jest to zatem kolejny przykład programowania niedeterministycznego, gdyż używamy `append` do wybierania dowolnych elementów listy. To predykat `append` zapewnia, że wybierane są wszystkie możliwości z listy.

Algorytm *quicksort* to bardziej wymyślna metoda sortowania stworzona przez C.A.R. Hoare'a. Implementacja tego algorytmu wymaga podzielenia listy o głowie *H* i ogonie *T* na dwie listy *L* i *M*, takie, że:

- ♦ wszystkie elementy *L* są mniejsze od *H*,
- ♦ wszystkie elementy *M* są większe lub równe *H*,
- ♦ kolejność elementów w *L* i *M* jest taka sama jak w *[H|T]*.

Kiedy lista zostanie już podzielona, każdą z uzyskanych list znowu przetwarzamy za pomocą tego samego algorytmu (rekurencyjnie), na koniec z powrotem złączamy *M* z *L*. Cel podzieli(*H*,*T*,*L*,*M*) dzieli listę *[H|T]* na listy *L* i *M* zgodnie z podanymi powyżej wyrażeniami:

```
podziel(H,[A|X],[A|Y],Z) :-
    kolejnosc(A,H), podziel(H,X,Y,Z).
podziel(H,[A|X],Y,[A|Z]) :-
    \+(kolejnosc(A,H)), podziel(H,X,Y,Z).
podziel(_,[],[],[]).
```

Oto implementacja samego algorytmu *quicksort*:

```
quicksort([],[]).
quicksort([H|T],S) :-
    podziel(H,T,A,B),
    quicksort(A,A1),
    quicksort(B,B1),
    append(A1,[H|B1],S).
```

Można też przerobić *append* na predykat sortujący, dzięki czemu uzyskamy szybciej działający program:

```
quicksortx([],X,X).
quicksortx([H|T],S,X) :-
    podziel(H,T,A,B),
    quicksortx(A,S,[H|Y]),
    quicksortx(B,Y,X).
```

W tym wypadku trzeci argument używany jest jako tymczasowy obszar roboczy, początkowo jest inicjalizowany listą pustą.

Więcej informacji o sortowaniu można znaleźć w tomie 3. (Sortowanie i wyszukiwanie) książki „Sztuka programowania” Donalda Knutha, wydanej w 1973 roku przez wydawnictwo Addison-Wesley. Algorytm *quicksort* opisał jego autor, Hoare, w czasopiśmie „Computer Journal” w numerze 5 (1962), strony 10 – 15.

Cwiczenie 7.5. Jeśli dana jest lista *L1*, sprawdź, czy permutacja(*L1*,*L2*) wygeneruje wszystkie możliwe permutacje *L1* jako możliwe wartości *L2*. W jakiej kolejności generowane są te rozwiązania?

Cwiczenie 7.6. Algorytm *quicksort* najlepiej sprawdza się w przypadku dużych list, gdyż szybko zbliża się do rozwiązania. Jednak każda iteracja wymaga wykonania większej ilości pracy niż w pozostałych algorytmach, a to dlatego, że konieczne jest wywołanie *podziel*. Wobec tego w przypadku niedużych list wywołania rekurencyjne

quicksort można zastąpić wywołaniami innych metod sortowania, na przykład sortowaniem przez wstawianie. Stwórz hybrydowy program sortujący, który za pomocą *quicksort* będzie rekurencyjnie sortował duże części (listy uzyskiwane w wyniku działania predykatu *podziel*), zaś kiedy rozmiar części będzie dostatecznie mały, będzie używana inna metoda sortowania. Wskazówka: wobec tego, że *podziel* musi przeglądać wszystkie kolejne elementy listy, można użyć tego predykatu do określania długości listy.

Użycie bazy danych

We wszystkich omawianych dotąd programach bazy danych używaliśmy jedynie do zapisywania faktów i reguł opisujących predykaty. W bazie danych można też zapisywać zwykłe struktury, np. struktury tworzone w trakcie wykonywania programu. Aż do teraz struktury takie między predykatami przekazywaliśmy jako argumenty. Powodem, dla którego warto przechowywać informacje w bazie danych zamiast przekazywać je jako argumenty, jest ich dostępność w wielu miejscach programu oraz możliwość skrócenia wywołań predykatów. Innym powodem jest możliwość zachowania danych także w trakcie nawracania. W tym podrozdziale opiszemy trzy predykaty, w których będziemy korzystać z bazy danych, aby przechować w niej struktury, dostępne dłużej niż byłoby to możliwe w przypadku zastosowania zmiennych. Te trzy predykaty to *random*, generujący podczas kolejnych wywołań pseudolosowe liczby całkowite, *findall*, podający listę wszystkich struktur, dla których dany predykat nie zawodzi, oraz *gensym*, generujący atomy o niepowtarzalnych nazwach.

random

Cel *random(R,N)* ukonkretnia *N* losowo wybraną liczbą całkowitą z zakresu od 1 do *R*. Dobieranie losowej liczby polega na użyciu kongruencji oraz posilkowaniu się z góry zadaną liczbą jako „zarodkiem”. Przy każdym wywołaniu predykatu dobierana jest nowa liczba na podstawie istniejącego zarodka, po czym określany i zapamiętywany na następny raz jest nowy zarodek. Zarodek ten jest przechowywany w bazie danych; po użyciu *random* stary zarodek z bazy usuwamy, a po wyliczeniu nowego, dodajemy go do bazy. Początkowa wartość zarodka zapisana jest po prostu jako dynamiczny fakt z jednoargumentowym funktorem *zarodek*.

```
:- dynamic zarodek/1.
```

```
zarodek(13).
random(R,N) :-
    zarodek(S),
    N is (S mod R) + 1,
    retract(zarodek(S)),
    NowyZarodek is (125 * S + 1) mod 4096,
    asserta(zarodek(NowyZarodek)).
```

Możemy skorzystać z cech *retract*, aby nieco uprościć definicję predykatu *random*, pobierając ziarno i jednocześnie od razu je usuwając:


```

random(R,N) :-
    retract(zarodek(S)),
    N is (S mod R) + 1,
    NowyZarodek is (125 * S + 1) mod 4096,
    asserta(zarodek(NowyZarodek)), !.

```

Aby pokazać pięć liczb losowych z zakresu od 1 do 10, wystarczy wywołać:

```
?- repeat, random(10,X), write(X), nl, X = 5.
```

gensym

Predykat `gensym` umożliwia generowanie nowych atomów Prologu. Jeśli mamy program przyswajający sobie wiedzę o świecie zewnętrznym (na przykład interpretujący zdania w języku naturalnym), pewnym problemem staje się natknięcie się na nowy obiekt. Naturalnym sposobem zapisu obiektu jest atom. Jeśli obiekt nie był dotąd znany, trzeba zapewnić, że przypisany mu atom nie będzie w konflikcie z innymi obiektami. Oznacza to, że potrzebne jest nam wygenerowanie nowego atomu. Możemy też zażądać, aby generowany atom był w jakiś sposób mnemonikiem, co ułatwiłoby interpretację odpowiedzi systemu. Jeśli na przykład opisujemy studentów, pierwszy student może być identyfikowany przez `student1`, drugi przez `student2`, trzeci przez `student3` i tak dalej. Jeśli oprócz tego mamy rejestrować wykładowców, naturalnymi atomami będą `wykladowca1`, `wykladowca2`, `wykladowca3` i tak dalej.

Predykatu `gensym` używa się do generowania danych atomów na podstawie danych szkieletów (jak `student` czy `wykladowca`). Dla każdego szkieletu zapamiętywany jest ostatnio nadany numer, a przy generowaniu nowego atomu numer ten jest sprawdzany, dzięki czemu mamy gwarancję, że identyfikator się nie powtórzy. Wobec tego pierwsze wywołanie

```
?- gensym(student,X).
```

da w odpowiedzi

```
X = student1
```

Przy następnym wywołaniu uzyskamy odpowiedź `X = student2` i tak dalej. Warto zauważyć, że kolejne rozwiązania nie są generowane dzięki nawracaniu (`gensym(X,Y)` nie może być ponownie spełniony), ale przez kolejne wywołania tego samego predykatu.

W definicji `gensym` korzystamy z predykatu pomocniczego `aktualny_numer`, wstawiając go do bazy (i usuwając, kiedy przestaje być potrzebny), dzięki czemu zawsze wiemy, jaki następny numer należy wykorzystać. Fakt `aktualny_numer(Szkielet,Numer)` oznacza, że ostatnim numerem użytym dla Szkieletu był Numer, więc ostatnio wygenerowany atom składał się ze znaków atomu Szkielet i z liczby Numer. Normalnie, przy próbie spełnienia celu `gensym`, z bazy danych usuwany jest fakt `aktualny_numer` dotyczący danego szkieletu, do ostatniego numeru dodawane jest 1, po czym do bazy wstawiany jest nowy fakt `aktualny_numer`, a jednocześnie nowy numer jest używany

do stworzenia następnego atomu. Zapis aktualnego numeru w bazie danych jest bardzo wygodny; innym rozwiązaniem mogłoby być jedynie dodanie argumentów do każdego predykatu bezpośrednio lub pośrednio używającego `gensym`. Oto program:

```

gensym(Szkielet,Atom) :-
    daj_liczbe(Szkielet,Liczba),
    atom_chars(Szkielet,Nazwa1),
    number_chars(Liczba,Nawa2),
    append(Nazwa1,Nawa2,Nazwa),
    atom_chars(Atom,Nazwa),
    daj_liczbe(Szkielet,Liczba) :-
    retract(aktualny_numer(Szkielet,Liczba1)), !,
    Liczba is Liczba1 + 1,
    asserta(aktualny_numer(Szkielet,Liczba)),
    daj_liczbe(Szkielet,1) :- asserta(aktualny_numer(Szkielet,1)).

```

Predykat `get_num` służy do pobierania następnej liczby, która będzie używana z danym szkieletem. Jeśli ze szkieletem związana jest już liczba (pierwsza klauzula), zwracana jest liczba następna i baza danych jest aktualizowana. Jeśli liczby takiej dotąd nie było (druga klauzula), zaczynamy od rekordu z jedynką. Predykat `gensym` jedynie ogranicza szkielet i liczbę znaków za pomocą odpowiednich predykatów wbudowanych, po czym łączy listy i z listy wynikowej tworzy potrzebny atom.

findall

W niektórych zastosowaniach przydatne jest określenie wszystkich termów spełniających dany predykat. Możemy na przykład chcieć stworzyć listę wszystkich dzieci Adama i Ewy, korzystając z opisanego w rozdziale 1. predykatu `rodzice` (zakładając, że mamy bazę danych zawierającą fakty `rodzice`). Możemy w tym celu użyć predykatu `findall`, który jest już dostępny we wszystkich implementacjach Prologu zgodnych ze standardem. Standard Prologu zawiera także podobnie działający predykat `setof`. Jako że predykat `findall` dobrze pokazuje, jak w Prologu używa się bazy danych, pokażemy teraz, jak można go zdefiniować.

Cel `findall(X,G,L)` tworzy listę `L` składającą się z wszystkich obiektów `X`, takich, że cel `G` jest spełniony. Zakłada się, że argument `G` jest ukonkretniony zwykłym termem, zaś `findall` traktuje ten term jako cel Prologu. Poza tym `X` musi wystąpić w `G`. Dzięki temu `G` można ukonkretnić dowolnie złożonym celem.

Oto przykład znajdowania wszystkich dzieci Adama i Ewy:

```
?- findall(X, rodzice(X,ewa,adam), L).
```

Zmienna `L` zostanie ukonkretniona listą wszystkich wartości `X` spełniających cel `rodzice(X,ewa,adam)`. Predykat `findall` po prostu stara się wielokrotnie spełnić drugi argument i przy każdej udanej próbie dopisuje odpowiedź do bazy danych. Kiedy próba spełnienia drugiego argumentu w końcu zawiedzie, zbieramy wszystkie uzyskane odpowiedzi z bazy danych. Uzyskana lista jest zwracana jako trzeci argument. Jeśli próba spełnienia drugiego argumentu nie powiedzie się ani razu, trzeci argument zostanie ukonkretniony pustą listą. Do wstawiania odpowiedzi do bazy danych używany

predykatu wbudowanego `asserta`, który wstawia nowy fakt *przed* pierwszy fakt z takim samym funktorem. Uzyskane odpowiedzi zapisujemy jako fakty znalezione. Oto treść predykatu `findall`:

```
:- dynamic znalezione/1.

findall(X,G,_):-
    asserta(znalezione(zaznacz)),
    call(G),
    asserta(znalezione(wynik(X))),
    fail.
findall(_,_,L):- zbierz_znalezione([],M), !, L = M.
zbierz_znalezione(S,L):-
    weznastepny(X),
    !,
    zbierz_znalezione([X|S],L).
zbierz_znalezione(L,L).
weznastepny(X):- retract(znalezione(X)), !, X \== zaznacz.
```

Predykat `findall` najpierw dodaje specjalny fakt `znalezione(zaznacz)`, dzięki temu mamy w bazie danych wskazane miejsce, przed które będziemy wstawiać wartości `X` spełniające `G`. Każdy inny argument predykatu `znalezione` ma postać `wynik(X)`, gdzie `X` jest znalezioną wartością. Następnie próbujemy uzgodnić cel `G` i za każdym razem, kiedy się to udaje, do bazy danych wstawiany jest fakt `znalezione(X)`. Użycie `fail` wymusza nawracanie, następuje próba ponownego spełnienia `G` (`asserta` nie ma innych rozwiązań). Kiedy w końcu `G` zawiedzie, nawracanie spowoduje, że zawiedzie cała pierwsza klauzula `findall` i nastąpi próba uzgodnienia drugiej klauzuli. Druga klauzula wywołuje `zbierz_znalezione` usuwając z bazy danych kolejne fakty `znalezione` i wstawiając je jednocześnie na listę. Predykat `zbierz_znalezione` wstawia kolejne elementy na listę „dotąd zebraną”; taki sposób działania programu objaśnialiśmy w punkcie poświęconym predykatowi `gensym`. Kiedy zostanie znaleziony znacznik `zaznacz` (lub coś nie mającego postaci `wynik(X)`), `weznastepny` zawodzi i spełniana jest druga klauzula `zbierz_znalezione`, która wiąże drugi swój argument (`wynik`) z pierwszym (uzyskując dotąd listę).

Warto zauważyć, że obecność w bazie danych faktu `znalezione(zaznacz)` identyfikuje użycie `findall`, dzięki czemu można używać `findall` rekurencyjnie: wszystkie wystąpienia `findall` w drugim argumentcie innego wywołania `findall` zostaną prawidłowo obsłużone.

W następnym podrozdziale stworzymy program, w którym użyjemy `findall` do stworzenia listy wszystkich potomków danego węzła grafu — potrzebne jest to do przeszukiwania grafu wszerz.

Ćwiczenie 7.7. Zakoduj w Prologu predykat `wybierz_losowo`, taki, że cel `wybierz_losowo(L,E)` ukonkretni `E` losowo wybranym elementem listy `L`. Wskazówka: użyj generatora liczb losowych i zdefiniuj predykat zwracający `N`-ty element listy.

Ćwiczenie 7.8. Jeśli mamy cel `findall(X,G,L)`, co się stanie, jeśli w `G` będą istniały nieukonkretnione zmienne niewystępujące w `X`?

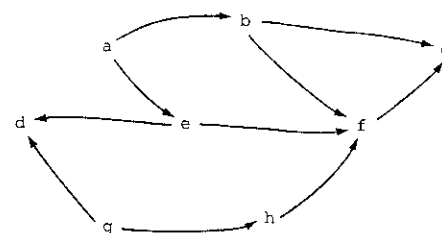
Przeszukiwanie grafów

Graf to zestaw węzłów połączonych łukami. Przykładem grafu może być mapa, na której węzłami są miejscowości, a łukami łączące je drogi. Jeśli chcemy znaleźć najkrótszą drogę między dwiema miejscowościami, musimy rozwiązać problem najkrótszej drogi między węzłami grafu.

Najprostszym sposobem zapisywania grafu jest stworzenie bazy danych faktów opisujących łuki (krawędzie) między węzłami. Przykładowy graf z rysunku 7.2 można zapisać w postaci faktów podanych na tym samym rysunku.

Rysunek 7.2.
Reprezentacja grafu skierowanego

```
a(g,h).
a(g,d).
a(e,d).
a(h,f).
a(e,f).
a(a,e).
a(a,b).
a(b,f).
a(b,c).
a(f,c).
```



Zauważmy, że nazwą predykatu opisującego łuki jest `a`, istnieje też węzeł nazwany `a`. Nie stanowi to problemu, gdyż predykat `a` zawsze ma dwa argumenty, zaś nazwa węzła `a` jest stałą. Aby przejść z węzła `a` do `c`, możemy skorzystać z drogi `a, e, f, c` lub z innej drogi wyznaczonej strzałkami. Predykat `a` interpretujemy tak, że `a(X,Y)` oznacza, że istnieje łuk z `X` do `Y` (co *nie implikuje* istnienia łuku z `Y` do `X`).

Najprostszy program przeszukujący powyższy graf może mieć postać:

```
go(X,X).
go(X,Y):- a(X,Z), go(Z,Y).
```

Program ten jest nieco bardziej ograniczony niż program z podrozdziału „Przeszukiwanie labiryntu”, gdyż ruch po łukach odbywa się jedynie w kierunku wyznaczonym przez strzałki. Tak jak poprzednio, program ten może wpaść w pętlę; wystarczy dodać do powyższego grafu łuk

```
a(d,a).
```

Wtedy przedstawiony na rysunku 7.1. graf stanie się grafem cyklicznym. Dlatego właśnie, podobnie jak poprzednio, powinniśmy w liście `T` zapamiętywać trasę składającą się z odwiedzonych dotychczas węzłów:

```
go(X,X,T).
go(X,Y,T):- a(X,Z), poprawny(Z,T), go(Z,Y,[Z|T]),
    poprawny(X,[ ]).
poprawny(X,[H|T]) :- \+ X = H, poprawny(X,T).
```

Predykat `poprawny` to nic innego jak sprawdzenie, czy węzeł nie jest elementem listy.

Przedstawiony program realizuje przeszukiwanie w głąb, gdyż najpierw analizowany jest tylko jeden węzeł sąsiadujący z węzłem bieżącym. Reszta sąsiednich węzłów jest pomijana, póki któryś krok nie zawiedzie, co spowoduje nawrót i badanie innego sąsiada.

Załóżmy teraz, że graf jest *nieskierowany*, czyli wszystkie jego łuki są dwukierunkowe. Wtedy konieczne jest dwukrotne wykorzystanie informacji o poszczególnych łukach i otrzymujemy program podobny do programu przeszukującego labirynt:

```
go(X,X,T).
go(X,Y,T) :-
    ( a(X,Z) ; a(Z,X) ),
    poprawny(Z,T), go(Z,Y,[Z|T]).
```

Zastanówmy się teraz, jak moglibyśmy wykorzystać w praktyce przeszukiwanie grafów. Załóżmy, że mamy zaplanować trasę łączącą szereg miast. Informacje o miastach i drogach w północnej Anglii mamy w postaci bazy danych (liczby to odległości):

```
a(newcastle,carlisle,58).
a(carlisle,penrith,23).
a(darlington,newcastle,40).
a(penrith,darlington,52).
a(workington,carlisle,33).
a(workington,penrith,39).
```

Chwilowo zapomnijmy o odległościach i zdefiniujmy nowy predykat a:

```
a(X,Y) :- a(X,Y,Z).
```

Nie będzie problemu z rozróżnianiem tych dwóch predykatów, gdyż każdy z nich ma inną liczbę argumentów. Kiedy mamy taką definicję a, nasza dotychczasowa procedura przeszukiwania grafu, go, znajdzie wszystkie możliwe drogi w takim grafie. Jednak go ma pewną wadę: nie informuje, jaka droga ostatecznie pozwoliła spełnić cel, zaś nasze minimalne wymagania to podanie listy odwiedzanych miejsc uszeregowanych we właściwej kolejności. Program ma już nawet przejrzaną trasę, ale jest ona w kolejności odwrotnej. Możemy użyć predykatu rev z podrödziału „Przetwarzanie list”. Oto nowa definicja go, które tym razem zwraca wybrane trasy w trzecim argumentcie:

```
go(Start,Koniec,Trasa) :-
    go0(Start,Koniec,[],R),
    rev(R,Trasa).
go0(X,X,T,[X|T]).
go0(Miejsce,Y,T,R) :-
    poprawnywezel(Miejsce,T,Nastepny),
    go0(Nastepny,Y,[Miejsce|T],R).
poprawnywezel(X,Trasa,Y) :-
    (a(X,Y) ; a(Y,X)), poprawny(Y,Trasa).
```

Zwróćmy uwagę na to, że użyliśmy predykatu poprawnywezel, aby zapisać, jakie przejście jest z węzła poprawne, zaś poprawność samego węzła jest zdefiniowana tak samo jak poprzednio.

Oto przykład użycia naszego programu do znalezienia drogi z Darlington to Workington:

```
?- go(darlington,workington,X).
X=[darlington,newcastle,carlisle,penrith,workington]
```

Nie jest to być może najlepsza droga, ale nawracanie pozwoli nam znaleźć inne możliwości.

Program ten ma szereg niedociągnięć. Nie w pełni kontrolujemy, która droga ma być badana dalej, gdyż nigdy nie wykonujemy pełnego przeglądu istniejących możliwości. Inne możliwe opcje rozwiązania są dostępne niejawnie, przez mechanizm nawracania, zaś naturalniejsze byłoby ich jawne udostępnienie programowi. Oto poprawiona wersja, mająca szersze spektrum zastosowań. Warto zauważyć, że niewielkie modyfikacje programu pozwalają znacznie zmienić sposób wyszukiwania.

```
go(Start,Koniec,Trasa) :-
    go1([Start],Koniec,R),
    rev(R,Trasa).
go1([Pierwszy|Reszta],Koniec,Pierwszy) :- Pierwszy = [Koniec,_].
go1([Ostatni|Dalsze]|Inne,Koniec,Trasa) :-
    findall([Z,Ostatni|Dalsze],poprawnywezel(Ostatni,Dalsze,Z),Lista),
    append(Lista,Inne,NoweTrasy),
    go1(NoweTrasy,Koniec,Trasa).
```

Predykat poprawnywezel jest taki sam jak poprzednio. Predykat go1 otrzymuje listę rozważanych dróg wraz z miejscem docelowym; zwraca w ostatnim argumentcie znaną trasę. Lista rozważanych tras to po prostu wszystkie drogi, które były dotąd rozważane, od punktu startowego. Liczymy na to, że którąś z tras uda się przedłużyć tak, aby doprowadziła nas do miejsca docelowego. Drogi zapisujemy jako listy miejsc w kolejności odwrotnej do wymaganej.

Kiedy zaczynamy badanie grafu, istnieje tylko jedna droga, którą możemy chcieć przedłużyć: zawiera ona po prostu sam węzeł początkowy. Jeśli zaczynamy wycieczkę w Darlington, będzie to droga [darlington]. Następnie badamy drogi z Darlington do miast sąsiednich; w tym wypadku są to [newcastle, darlington] oraz [penrith, darlington]. Pośród tych miast nie ma Workington, więc musimy teraz zdecydować, którą drogę będziemy przedłużać. Jeśli zdecydujemy się na pierwszą, znajdziemy tylko jeden możliwy następny węzeł za Newcastle (ostatnie miasto tej drogi). Mamy zatem, poza Darlington-Penrith, nową drogę: [carlisle, newcastle, darlington].

Nasz predykat przeszukujący, go1, zapamiętuje wszystkie drogi, którymi ewentualnie warto się zainteresować. Jak podejmowana jest decyzja, która droga ma być pierwsza? Po prostu wybierana jest droga pierwsza w kolejności, następnie znajdowane są możliwe przedłużenia tej drogi o kolejne miasto (za pomocą findall tworzymy listę takich przedłużonych dróg) i wstawiana jest na początek listy.

W wyniku tego go1 sprawdzi wszystkie możliwe drogi stanowiące przedłużenie pierwszej ścieżki przed podjęciem próby przeanalizowania rozwiązania alternatywnego, więc mamy do czynienia z przeszukiwaniem w głąb. Przypadkowo go1 sprawdza trasy w takiej samej kolejności, jak go0. Czytelnikom zostawiamy samodzielne przeanalizowanie, dlaczego tak jest.

Jeśli interesuje nas najkrótsza droga z Darlington do Workington, pokazany program nie wydaje się być najlepszym rozwiązaniem. Pierwsze znalezione rozwiązanie nie jest najkrótsze (a w tym wypadku nawet jest najdłuższe). Musimy zmodyfikować nasz program tak, aby znajdował najpierw najkrótszą drogę (długość drogi określamy zliczając

występujące w niej miasta). Uzyskany program będzie realizował przeszukiwanie wszerz. Wystarczy wstawić inne alternatywy na-koniec listy możliwości, a nie na początek, jak to robiliśmy poprzednio. Po prostu modyfikujemy drugą klauzulę `go1`:

```
go1([[Ostatni|Dalsze]|Inne,Koniec,Trasa) :-
    findall([Z,Ostatni|Dalsze], poprawnywez(0statni,Dalsze,Z),Lista),
    append(Inne,Lista,NoweTrasy),
    go1(NoweTrasy,Koniec,Trasa).
```

Tak poprawiony program znajduje teraz możliwe drogi z Darlington do Workington:

```
[darlington,penrith,workington]
[darlington,newcastle,carlisle,workington]
[darlington,penrith,carlisle,workington]
[darlington,newcastle,carlisle,penrith,workington]
```

Możemy ten program mocno uprościć, jeśli jesteśmy pewni, że zapytanie zawsze ma odpowiedź i jeśli interesuje nas tylko pierwsze rozwiązanie. W tego typu sytuacjach nie musimy kontrolować pętli za pomocą `poprawnywez`. Czytelnicy mogą sami sprawdzić, czy potrafią zrozumieć, dlaczego tak jest.

Niestety, nie zawsze droga z najmniejszą liczbą miast jest najkrótsza. Na razie pomialiśmy informacje o odległościach między miastami. Jeśli dodamy do naszego grafu kilka fikcyjnych miast, otrzymamy:

```
a(newcastle,carlisle,58).
a(carlisle,penrith,23).
a(miastoB,miastoA,15).
a(penrith,darlington,52).
a(miastoB,miastoC,10).
a(workington,carlisle,33).
a(workington,miastoC,5).
a(workington,penrith,39).
a(darlington,miastoA,25).
```

Najkrótsza droga zostanie pokazana jako ostatnia, gdyż zawiera wiele miast. Wobec tego trzeba wraz z każdą ścieżką zapamiętywać dotychczasową długość ścieżki. Teraz będziemy kontynuować przetwarzanie ścieżki najkrótszej do danej chwili. Tego typu przeszukiwanie nazywamy przeszukiwaniem *od najlepszego*.

Teraz drogi na liście będziemy zapisywać jako struktury `r(M,P)`, gdzie `M` to całkowita długość drogi w milach, zaś `P` to liczba odwiedzonych miast. Zmodyfikowany predykat `go3` znajduje najkrótszą drogę na liście możliwości. Predykat `najkrotszy` zwraca najkrótszą drogę z listy i zwraca pozostałe na liście drogi. Kiedy mamy drogę dotąd najkrótszą, predykat `przetwarzaj` znajduje wszystkie poprawne przedłużenia tej drogi i dodaje je do listy. To z kolei wymaga stworzenia nowej wersji predykatu `poprawnywez`, który dodaje odległość do następnego miasta do wyliczonej dotąd odległości. Oto cały program:

```
go3(Trasy,Koniec,Trasa) :-
    najkrotszy(Trasy,Najkrotsza,ResztaTras),
    przetwarzaj(Najkrotsza,Koniec,ResztaTras,Trasa).
przetwarzaj(r(Odleglosc,Trasa),Koniec,_,Trasa) :-
    Trasa = [Koniec|_].
```

```
przetwarzaj(r(Odleglosc,[Ostatni|Dalsze]),Odleglosc,Trasy,Trasa) :-
    findall(
        r(D1,[Z,Ostatni|Dalsze]),
        poprawnywez(Ostatni,Dalsze,Z,Odleglosc,D1),
        Lista),
    append(Lista,Trasy,NoweTrasy),
    go3(NoweTrasy,Koniec,Trasa),
    najkrotsza([Trasa|Trasy],Najkrotsza,[Trasa|Reszta]) :-
        najkrotsza(Trasy,Najkrotsza,Reszta),
        krotsza(Najkrotsza,Trasa),
        !,
        najkrotsza([Trasa|Reszta],Trasa,Reszta).
krotsza(r(M1,_),r(M2,_)) :- M1 < M2.
poprawnywez(X,Dalsze,Y,Odleglosc,NowaOdleglosc) :-
    (a(X,Y,Z) ; a(Y,X,Z)),
    poprawny(Y,Dalsze),
    NowaOdleglosc is Odleglosc + Z.
```

Aby tego programu użyć, będziemy starali się spełnić predykat `go`:

```
go(Poczek,Koniec,Trasa) :-
    go3([r(0,[Poczek])],Koniec,R),
    rev(R,Trasa).
```

Nowy program generuje możliwe drogi przejścia według ich długości. Można go jeszcze zmodyfikować tak, aby wraz z poszczególnymi drogami podawane były też ich długości.

Dopiero zaczęliśmy poznawać możliwe sposoby przeszukiwania grafu. Więcej informacji o przeszukiwaniu grafów za pomocą metod wydajniejszych od przeszukiwania od najlepszego, omówiono w książkach poświęconych sztucznej inteligencji. Można tu wymienić następujące pozycje: „Podstawy sztucznej inteligencji” Nilsa Nilssona wydaną w 1982 roku przez wydawnictwo Springer-Verlag, „Sztuczna inteligencja” Patricia Winstona (wydanie drugie, Addison-Wesley 1984) czy „Sztuczna inteligencja dzisiaj” Stuarta Russella i Petera Norviga wydaną w 1995 roku przez Prentice-Hall.

Odsiej Dwójki i odsiej Trójki

*Odsiej Dwójki i odsiej Trójki:
Sito Erastotenesa.
Kiedy odpadną wielokrotności,
zostaną same liczby Pierwsze.*

Anonim

Liczba pierwsza to liczba, która nie ma dzielników innych niż jeden i ta liczba. Przykładowo, 5 jest liczbą pierwszą, ale 15 już nie, gdyż 3 jest dzielnikiem 15. Jedną z metod wskazywania liczb pierwszych to sito Erastotenesa. Metoda ta, pozwalająca odsiać liczby pierwsze nie większe od N , działa następująco:

1. Wstaw do „sita” wszystkie liczby od 2 do N .
2. Wybierz z sita najmniejszą pozostałą liczbę, usuń ją stamtąd.
3. Wybrana w poprzednim kroku liczba jest liczbą pierwszą.
4. Przejdź przez sito usuwając wszystkie wielokrotności dodanej liczby pierwszej.
5. Jeśli sito nie jest puste, powtarzaj kroki od 2. do 5.

Aby przekształcić powyższy algorytm w program prologowy, definiujemy predykat `liczbycałkowite` generujący listę liczb całkowitych, predykat `sito` sprawdzający kolejne elementy sita i predykat `usun` tworzący nowe sito przez usunięcie z sita wielokrotności wskazanej liczby. Nowe sito jest z powrotem przekazywane do predykatu `sito`. Predykat pierwszy zdefiniowano tak, aby cel pierwsze(N,L) ukonkretniał L listą liczb pierwszych z zakresu od 2 do N włącznie:

```
pierwsze(Limit,Pierwsze) :-
    liczbycałkowite(2,Limit,Is),
    sito(Is,Pierwsze).
liczbycałkowite(Dolna,Gorna,[Dolna|Reszta]) :-
    Dolna <= Gorna,
    !,
    M is Dolna+1,
    liczbycałkowite(M,Gorna,Reszta).
liczbycałkowite(_,_,[]).
sito([],[]).
sito([I|Is],[I|Pierwsze]) :-
    usun(I,Is,Nowa),
    sito(Nowa,Pierwsze).
usun(P,[],[]).
usun(P,[I|Is],[I|Nis]) :-
    \+ 0 is I mod P,
    !,
    usun(P,Is,Nis).
usun(P,[I|Is],Nis) :-
    0 is I mod P,
    !,
    usun(P,Is,Nis).
```

Czasami lepszy program możemy uzyskać, nie traktując podanego algorytmu zbyt dosłownie. Liczby pierwsze można znaleźć prościej: cel `pierwsze(I,L,P)` przegląda listę liczb całkowitych I w celu podania liczb pierwszych P , korzystając przy tym z akumulatora L na liczby znalezione dotąd. Każdy element I musi zostać sprawdzony, czy dzieli się przez jakiś element z L ; jeśli nie, można go do L dodać. Kiedy dochodzimy do końca listy, wszystkie liczby pierwsze mamy w akumulatorze.

```
pierwsze([],P,P).
pierwsze([H|T],P,Z) :-
    poprawna(H,P),
    !,
    pierwsze(T,[H|P],Z).
pierwsze([H|T],P,Z) :- pierwsze(T,P,Z).
```

```
/* X jest poprawna na L, jeśli nie jest podzielna
   przez żaden element L */
poprawna(X,[]).
poprawna(X,[H|_]) :-
    0 is X mod H,
    !,
    fail.
poprawna(X,[_|L]) :- poprawna(X,L).
```

Kontynuując te rozważania arytmetyczne, pokażemy program rekurencyjnie znajdujący największy wspólny dzielnik i najmniejszy wspólny mianownik za pomocą algorytmu Euklidesa. Cel `nwd(I,J,K)` nie zawiedzie, jeśli K jest największym wspólnym dzielnikiem liczb I i J . Cel `nww(I,J,K)` nie zawodzi, jeśli K jest najmniejszą wspólną wielokrotnością I i J :

```
nwd(I,0,I).
nwd(I,J,K) :- R is I mod J, nwd(J,R,K).
nww(I,J,K) :- nwd(I,J,R), K is (I*J)/R.
```

Zwróćmy uwagę, że ze względu na sposób wyliczania reszt, predykaty nie są „odwracalne”. Zmienne I i J muszą być ukonkretnione.

Ćwiczenie 7.10. Trzy liczby x , y i z nazywamy *trójką pitagorejską*, jeśli kwadrat z jest sumą kwadratów x i y (czyli $z^2 = x^2 + y^2$). Napisz program generujący trójki pitagorejskie. Zdefiniuj predykat `trojkipitag`, taki, aby zapytanie

```
?- trojkipitag(X,Y,Z).
```

dało trójkę pitagorejską, i aby kolejne nawroty dawały dalsze odpowiedzi. Wskazówka: skorzystaj z predykatów takich jak `liczba_calkowita` z rozdziału 4.

Różniczkowanie symboliczne

W matematyce przez różniczkowanie symboliczne rozumiemy przekształcenie jednego wyrażenia algebraicznego na inne, nazywane *pochoďną* pierwszego. Załóźmy, że U oznacza wyrażenie algebraiczne ze zmienną x . Pochoďną U względem x zapisujemy jako

$$\frac{dU}{dx}.$$

wyznaczamy ją stosując rekurencyjnie pewne zasady przekształcania wyrażeń do wyrażenia U . Poniżej najpierw podajemy dwa warunki graniczne, zaś strzałkę interpretujemy jako „jest przekształcane na”. U i V oznaczają wyrażenia, zaś c oznacza stałą:

$$\frac{dc}{dx} \rightarrow 0$$

$$\frac{dx}{dx} \rightarrow 1$$

$$\begin{aligned}\frac{d(-U)}{dx} &\rightarrow -\left(\frac{dU}{dx}\right) \\ \frac{d(U+V)}{dx} &\rightarrow \frac{dU}{dx} + \frac{dV}{dx} \\ \frac{d(U-V)}{dx} &\rightarrow \frac{dU}{dx} - \frac{dV}{dx} \\ \frac{d(cU)}{dx} &\rightarrow c\left(\frac{dU}{dx}\right) \\ \frac{d(UV)}{dx} &\rightarrow U\left(\frac{dV}{dx}\right) + V\left(\frac{dU}{dx}\right) \\ \frac{d(U/V)}{dx} &\rightarrow \frac{d(UV^{-1})}{dx} \\ \frac{d(U^c)}{dx} &\rightarrow cU^{c-1}\left(\frac{dU}{dx}\right) \\ \frac{d(\log_x U)}{dx} &\rightarrow U^{-1}\left(\frac{dU}{dx}\right)\end{aligned}$$

Powyższy zestaw reguł łatwo przekształcić w klauzule Prologu, gdyż wyrażenia algebraiczne możemy zapisać jako struktury, a operatorów użyć jako funktorów. Możemy też korzystać z dopasowywania wzorców, dopasowując cele do głów reguł.

Cel $d(E, X, F)$ nie zawiedzie, jeśli pochodną wyrażenia E względem zmiennej X jest wyrażenie F . Wprawdzie operatory $+$, $-$, $*$ i $/$ są już wbudowane, ale musimy jeszcze zadeklarować operator $^$, taki, że X^Y oznaczać będzie x^y . Deklaracje operatorów ułatwią nam po prostu odczytywanie wyrażeń. Oto przykłady uzyskiwanych odpowiedzi systemu:

```
?- d(x+1,x,X).
X = 1+0
?- d(x*x-2,x,X).
X = x*1+1*x-0
```

Zwróćmy uwagę na to, że przekształcanie jednego wyrażenia w inne niekoniecznie musi dawać wynik w formie maksymalnie uproszczonej, ale upraszczanie jest opisane w następnym podrozdziale. Program różniczkujący składa się z deklaracji dodatkowych operatorów oraz powyższych reguł w formie predykatów:

```
?- op(300,yfx,^).
```

```
d(X,X,1) :- !.
d(C,X,0) :- atomic(C).
d(-U,X,-A) :- d(U,X,A).
d(U+V,X,A+B) :- d(U,X,A), d(V,X,B).
d(U-V,X,A-B) :- d(U,X,A), d(V,X,B).
```

```
d(C*U,X,C*A) :- atomic(C), \+ C = X, d(U,X,A), !.
d(U*V,X,B*U+A*V) :- d(U,X,A), d(V,X,B).
d(U/V,X,A) :- d(U*V^(-1),X,A).
d(U^C,X,C*U^(C-1)*W) :- atomic(C), \+ C = X, d(U,X,W).
d(log(U),X,A*U^(-1)) :- d(U,X,A).
```

Zauważmy, że w dwóch miejscach występuje odcięcie. Pierwsze z nich zapewnia, że wyznaczanie pochodnej zmiennej względem niej samej odbędzie się jedynie przy użyciu pierwszej klauzuli. Drugie odcięcie wynika z istnienia dwóch klauzul dla mnożenia, z których pierwsza obsługuje przypadek szczególny. Jeśli zachodzi przypadek szczególny, z przypadku ogólnego należy zrezygnować.

Jak wspomniano wcześniej, generowane rozwiązania nie są wyrażeniami uproszczonymi, gdyż $x*1$ można byłoby zapisać jako x , a przykładowo $x*1+1*x-0$ jako $2*x$. W następnym podrozdziale opiszemy program upraszczający wyrażenia algebraiczne, bardzo podobny do powyższego, wyliczającego pochodne.

Odwzorowywanie struktur i przekształcanie drzew

Jeśli kopiujemy strukturę do innej element po elemencie, *odwzorowujemy* jedną strukturę na inną. Często poszczególne elementy nieznacznie podczas kopiowania modyfikujemy, jak to robiliśmy ze zdaniem w rozdziale 3. W tamtym przykładzie czasami chcieliśmy słowo skopiować identycznie, a czasami chcieliśmy zastosować inne słowo. Oto program *odwzorowujący* pierwszy argument na drugi:

```
zamien([],[]).
zamien([A|B],[C|D]) :- zmien(A,C), zamien(B,D).
```

Takie odwzorowanie jest operacją ogólną, więc możemy zdefiniować predykat *odwzorujliste*, taki, że *odwzorujliste(P,L,M)* spowoduje zastosowanie predykatu P do każdego elementu listy L w celu stworzenia nowej listy M . Założmy, że P ma dwa argumenty, z których pierwszy jest elementem wejściowym, a drugi elementem zmodyfikowanym, wstawianym do M :

```
odwzorujliste(_,[],[]).
odwzorujliste(P,[X|L],[Y|M]) :-
    Q =... [P,X,Y], call(Q), odwzorujliste(P,L,M).
```

W tej definicji warto zwrócić uwagę na kilka rzeczy. Po pierwsze, najpierw mamy klauzulę z warunkiem końcowym, potem klauzulę z przypadkiem rekurencyjnym. W drugiej klauzuli użyto operatora $=...$, czyli *univ*, który tworzy cel z danego predykatu P , elementu wejściowego X i zmiennej Y zwracanej przez P jako wynik. Dalej staramy się spełnić cel Q , przez co ukonkretniona zostanie zmienna Y tworząca głowę drugiego argumentu tego wywołania *odwzorujliste*. Na koniec wywołanie rekurencyjne powoduje odwzorowanie ogona.

Predykat zamien można zastąpić predykatem odwzorujliste. Jeśli zmien zdefiniowano tak, jak w rozdziale 3., użycie odwzorujliste będzie wyglądać następująco:

```
?- odwzorujliste(zmien,[ty,jestes,komputerem],Z).
Z=[[ja,nie].jestem,komputerem]
```

Uprosczenie odwzorujliste da nam w wyniku predykat robliste, który po prostu stosuje pewien predykat (mający z założenia jeden argument) do każdego elementu listy. Nie jest tworzona żadna nowa lista:

```
robliste(_,[_]).
robliste(P,[X|L]) :-
    Q =.. [P,X], call(Q), robliste(P,L).
```

Uprosczenie robliste daje nam roblisteput, który stosuje pewien (jednoargumentowy) predykat do wszystkich elementów danej listy. Nie jest przy tym tworzona żadna nowa lista:

```
roblisteput(_,[_]).
roblisteput(P,[X|L]) :-
    Q =.. [P,X], call(Q), roblisteput(P,L).
```

Przykładem zastosowania takiego predykatu może być następująca alternatywna definicja phh z rozdziału 5.:

```
phh(Lista) :- roblisteput(write_space,Lista).
write_space(X) :- write(X), spaces(1).
```

Odwzorowywanie nie ogranicza się tylko do list, ale można je zdefiniować dla dowolnych struktur. Przyjrzyjmy się na przykład wyrażeniom algebraicznym zawierającym funktory typu * czy +, wszystkie dwuargumentowe. Założmy, że chcemy odwzorowywać jedne wyrażenia na inne usuwając mnożenia przez 1. Jednym ze sposobów opisania takiego uproszczenia algebraicznego jest zdefiniowanie predykatu s, takiego, że s(Op,La,Pa,Odop) zamienia wyrażenie z operatorem Op, lewym argumentem La i prawym argumentem Ra na wyrażenie Odp. Fakty opisujące usuwanie mnożenia przez 1 uwzględniałyby dwa przypadki dotyczące łączności mnożenia:

```
s(*,X,1,X).
s(*,1,X,X).
```

Jeśli mielibyśmy zatem wyrażenie 1*X, uprościlibyśmy je jako X niezależnie od konkretnej wartości X. Zastanówmy się teraz, jak to zaimplementować w naszym programie.

Aby uprościć wyrażenie E za pomocą tablicy reguł upraszczania, musimy najpierw uprościć lewy argument E, potem prawy, a następnie sprawdzić, czy otrzymane wyrażenie znajduje się w tablicy uproszczeń. Jeśli tak, tworzymy nowe wyrażenie zgodnie z instrukcjami z tablicy. Jeśli „liście” drzewa wyrażenia są liczbami lub atomami, powinniśmy użyć predykatu wbudowanego atomic, aby sprawdzić, czy osiągnięto warunek końcowy. Jak poprzednio, możemy użyć =.. do rozdzielenia E na funktor i jego składniki:

```
uproszc(E,E) :- atomic(E), !.
uproszc(E,F) :-
    E =.. [Op,La,Pa],
```

```
uproszc(La,X).
uproszc(Pa,Y).
s(Op,X,Y,F).
```

Tak więc uproszc odwzorowuje wyrażenie E na wyrażenie F korzystając z faktów z tablicy uproszczeń s. Co się stanie, jeśli nic nie da się uprościć? Aby s(Op,X,Y,F) nie zawodziło, trzeba zdefiniować ogólną regułę dla każdego możliwego operatora. Poniższa tablica zawiera reguły opisujące upraszczanie dodawania i mnożenia oraz pokazuje dodatkowe reguły ogólne:

```
s(+,X,0,X).
s(+,0,X,X).
s(+,X,Y,X+Y). /* przypadek ogólny dodawania */
s(*,_,0,0).
s(*,0,_,0).
s(*,1,X,X).
s(*,X,1,X).
s(*,X,Y,X*Y). /* przypadek ogólny mnożenia */
```

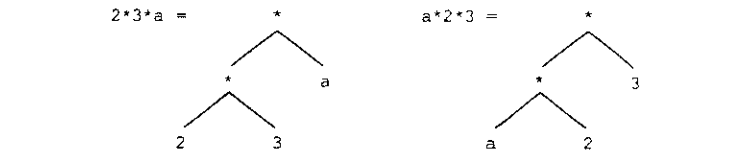
Kiedy mamy już reguły opisujące przypadki ogólne, możemy przystąpić do upraszczania wyrażeń. Na przykład, jeśli mamy wyrażenie 3+0, może zostać użyta pierwsza klauzula lub przypadek ogólny dodawania. Wskutek przyjętej kolejności faktów, zawsze przed przypadkami ogólnymi stosowane będą reguły szczegółowe. Podawane rozwiązanie będzie uproszczone, ale potem rozwiązania alternatywne już niekoniecznie.

Inny rodzaj upraszczania uwzględnianego w komputerowej arytmetyce to wydzielenie stałych. Operację 3*4+a można zapisać jako 12+a. Do powyższych reguł możemy dodać odpowiednie klauzule dotyczące stałych; w przypadku dodawania będzie to

```
s(+,X,Y,Z) :- integer(X), integer(Y), Z is X+Y.
```

Reguły dla innych operatorów będą podobne.

W przypadku operatorów zachowujących prawo łączności, jak mnożenie i dodawanie, opisane uproszczenia mogą dawać inne wyniki dla równoważnych, lecz inaczej zapisanych wyrażeń wejściowych; jeśli na przykład mamy regułę dotyczącą stałych przy mnożeniu, predykat uproszc odwzoruje 2*3*a jako 6*a, ale już a*2*3 czy 2*a*3 pozostaną niezmienione:



W pierwszym drzewie najniższe mnożenie można zamienić z 2*3 na 6, ale w drugim drzewie żadne poddrzewo nie może być uproszczone. Mnożenie jest łączne, więc opisany problem można rozwiązać przez dodanie do tablicy reguły:

```
s(*,X*Y,W,X*Z) :- number(Y), number(W), Z is Y*W.
```

Jednak aby rozwiązać problem w przypadku ogólnym, a nie tylko dla danego typu wyrażenia, zamiast rozszerzać tablicę uproszczeń, należy stworzyć bardziej rozbudowany system algebraiczny. Techniki upraszczania oraz ogólniejsze omówienie przekształcania wyrażeń Czytelnik znajdzie w literaturze podanej na końcu tego rozdziału.

Przetwarzanie programów

Wiele predykatów wbudowanych, omawianych wcześniej w tej książce, można zdefiniować w samym Prologu za pomocą prostszych predykatów wbudowanych. Kilka takich przykładów pokazemy w tym rozdziale. Mogą być one przydatne dla osób używających uboższych implementacji Prologu, ale przede wszystkim warto je poznać, aby poznać ciekawe techniki programowania. Ich analiza może być inspiracją do samodzielnego tworzenia innych wersji tych predykatów.¹

listing

Za pomocą `clause` można zdefiniować predykat `listing`. Zdefiniujmy predykat `list1` tak, że spełnienie celu `list1(X)` pokaże klauzule z bazy danych, których głowy pasują do `X`. Jako że w `list1` skorzystamy z `clause` i prześlemy mu `X` jako pierwszy argument, będziemy wymagali, aby `X` był ukonkretniony przynajmniej na tyle, aby znany był funktor główny. Oto definicja `list1`:

```
list1(X) :-
    clause(X,Y),
    wypisz_klauzule(X,Y), write(' '). nl, fail.
list1(X).
wypisz_klauzule(X,true) :- !, write(X).
wypisz_klauzule(X,Y) :- write((X :- Y)).
```

Kiedy próbujemy uzgodnić cel `list1(X)`, pierwsza klauzula powoduje wyszukanie klauzuli, której głowa pasuje do `X`. Jeśli klauzula taka zostanie znaleziona, jest ona pokazywana i wywoływana jest `fail`. Nawracanie spowoduje osiągnięcie celu `clause` i ewentualnie odszukanie pozostałych pasujących klauzul. Kiedy nie będzie już żadnych pasujących klauzul, zawiedzie w końcu także cel `clause` i użyta zostanie druga klauzula `list1`. Efektem ubocznym działania predykatu będzie wydrukowanie kolejnych klauzul. Jeśli zachodzi przypadek szczególny i treść to `true`, po prostu pokazujemy głowę. W przeciwnym razie pokazujemy głowę i treść połączone funktorem `:-`. Odcięcie powoduje, że w przypadku treści o wartości `true` jedynie pierwsza klauzula zostanie zastosowana. Cały przykład oparty jest na nawracaniu, więc odcięcie to jest bardzo istotne.

¹ Pamiętać jednak trzeba, że implementacje Prologu niezgodne ze standardem spośród wszystkich predykatów operujących na predykatach dynamicznych mogą zawierać jedynie predykat `clause`.

Interpreter Prologu

Predykat wbudowany `clause` może być też użyty do napisania interpretera Prologu, czyli jeden program w Prologu może wykonywać inny program w Prologu. Oto definicja predykatu `interpretuj`, takiego, że cel `interpretuj(X)` nie zawiedzie, jeśli nie zawiedzie `X` jako cel. Predykat ten jest podobny do wbudowanego `call`, ale jego funkcjonalność jest nieco bardziej ograniczona, gdyż nie są uwzględniane odcięcia i predykaty wbudowane.

```
interpretuj(true) :- !.
interpretuj((G1,G2)) :- !, interpretuj(G1), interpretuj(G2).
interpretuj(Cel) :-
    clause(Cel,InneCele), interpretuj(InneCele).
```

Pierwsze dwie klauzule obsługują przypadki szczególne: cel `true` i cel będący koniunkcją innych celów. Ostatnia klauzula obsługuje zwykły cel. Działanie predykatu polega na znalezieniu klauzuli, której głowa pasuje do celu, a następnie interpretacji celów z treści klauzuli. Zauważmy, że taka definicja nie wystarcza do programów, w których użyto predykatów wbudowanych, gdyż takie predykaty nie mają klauzul.

retractall

Jako przykład użycia predykatu `retract`, pokażemy definicję bardzo przydatnego predykatu `retractall`. Podczas spełniania celu `retractall(X)`, z bazy danych usuwane są wszystkie klauzule, których głowy pasują do `X`. Jako że w definicji tej używamy `retract`, `X` nie może być nieukonkretniona, gdyż inaczej nie można byłoby ustalić predykatu klauzuli. W naszej definicji musimy uwzględnić dwa przypadki: kiedy `X` pasuje do faktu i kiedy `X` pasuje do reguły. W obu wypadkach podamy inne argumenty `retract`. W definicji korzystamy z tego, że `retract` podczas kolejnych nawrotów usuwa kolejne klauzule, aż któraś z nich daje się dopasować do `X`.

```
retractall(X) :- retract(X), fail.
retractall(X) :- retract((X :- Y)), fail.
retractall(_).
```

consult

Jako zastosowanie powyższego predykatu `retractall`, pokażemy definicję omawianego w rozdziale 6. predykatu `consult`, wczytującego z pliku klauzule i usuwającego z bazy danych klauzule, których definicje są w pliku. Oczywiście `consult` i inne tego typu narzędzia występują właściwie we wszystkich implementacjach Prologu, ale warto zobaczyć, jak je można zdefiniować. Definicja ta jest niepełna, gdyż nie obsługuje ona prawidłowo dyrektyw², poza tym może być niezgodna ze standardem Prologu, gdyż do klauzul używany jest predykat `assertz`, bez zachowania uprzedniego deklarowania predykatów jako dynamicznych.

² Dyrektywa to specjalny predykat wbudowany, który jest zwykle wywoływany podczas ładowania kodu z pliku (i który jakoś na to ładowanie wpływa), a nie podczas wykonywania programu. Dyrektywy można wywoływać jako cele w postaci `:- G`, zamiast standardowego `?- G`; jedyną praktyczną różnicą polega na tym, że w pierwszym wypadku nie mamy odpowiedzi `yes` ani `no`, nie mamy też możliwości pytania o kolejne odpowiedzi. Jedynie dyrektywy omawiane w tej książce to `op/3` oraz `dynamic/1`.


```
consult(Plik) :-
    retractall(gotowe(_)),
    current_input(Stare),
    open(Plik, read, Strumien),
    repeat,
    read(Term),
    przetwarzaj(Term),
    close(Strumien),
    set_input(Stare),
    !.

przetwarzaj(end_of_file) :- !. % odczytano znacznik końca pliku
przetwarzaj((?- Cele)) :- !, call(Cele), !, fail.
przetwarzaj((:- Cele)) :- !. % pomijamy dyrektywy
przetwarzaj(Klauzula) :-
    glowa(Klauzula, Głowa),
    rekord_gotowy(Głowa),
    assertz(Klauzula),
    fail.

:- dynamic gotowy/1.

rekord_gotowy(Głowa) :- gotowy(Głowa), !.
rekord_gotowy(Głowa) :-
    functor(Głowa, Funktor, IleArg),
    functor(Proc, Funktor, IleArg),
    asserta(gotowy(Proc)),
    retractall(Proc),
    !.

glowa((A :- B), A) :- !.
glowa(A, A).
```

W powyższej definicji warto zwrócić uwagę na kilka rzeczy. Po pierwsze, cel `current_input(Stare)` i jego odpowiednik `set_input(Stare)` zapewniają, że po wykonaniu `consult` nie ulegnie zmianie aktualny strumień wejściowy. Celem predykatu `przetwarzaj` jest wykonanie odpowiedniej akcji dla każdego termu odczytanego z pliku. Cel `przetwarzaj` nie zawiedzie tylko wtedy, gdy jego argumentem jest znacznik końca pliku. W przeciwnym razie zawsze zawodzi, dzięki czemu nawracanie powoduje powrót do celu `repeat`. Zwróćmy uwagę, jak istotne jest odcięcie na końcu definicji `consult`. Odcięcie to powoduje odrzucenie punktu nawracania w `repeat`.

I na koniec: jeśli odczytany z pliku term to zapytanie (druga klauzula `przetwarzaj`), za pomocą predykatu `call` staramy się uzgodnić odpowiedni cel (więcej o `call` w rozdziale 6.). Jeśli zostanie odczytane wywołanie dyrektywy, jest ono pomijane. Dyrektywy muszą być wywoływane bezpośrednio z kodu programu, którego dotyczą (a nie pośrednio, przez `przetwarzaj`); nie jesteśmy w stanie w naszych programach zasymulować ich działania.

Kiedy w pliku pojawia się pierwsza klauzula danego pliku, wszystkie klauzule tego predykatu z bazy danych muszą być usunięte. Nie możemy usunąć dalszych klauzul, które pojawiają się dla tego predykatu, więc musimy jakoś sprawdzić, czy dana klauzula jest pierwszą. W tym celu zapisujemy w bazie danych informacje o predykatkach,

dla których wystąpiły już jakieś klauzule; służy do tego predykat `zrobione`. Kiedy z bazy danych odczytujemy pierwszą klauzulę dwuargumentowego predykatu, dajmy na to, predykat, istniejące klauzule predykatu są usuwane i do bazy danych dodawana jest nowa, właśnie odczytana klauzula. Poza tym do bazy danych dodawany jest fakt

```
zrobione(predykat(_, _)).
```

Kiedy odczytywane będą kolejne klauzule predykatu, będziemy wiedzieli, że stare klauzule już usunięto. W definicji tej istotne jest użycie zmiennych anonimowych, a nie na przykład

```
zrobione(predykat(a, X)).
```

gdyż wtedy mogłyby wystąpić problemy z dopasowaniem klauzuli predykatu. Cele

```
..., functor(Głowa, Funktor, IleArg), functor(Proc, Funktor, IleArg), ...
```

ukonkretniają `Proc` strukturą z takim samym funktorem jak `glowa Głowa`, ale argumenty stają się zmiennymi (więcej o predykanie `functor` w rozdziale 6.).

Literatura

Większe programy prologowe z komentarzami można znaleźć w książce „The Practice of Prolog” pod redakcją Leona Sterlina, wydanej przez MIT Press w 1994 roku.

Poza tym analizę pewnych konkretnych przypadków zastosowania Prologu w nietypowych rozwiązaniach, jak szybka transformata Fouriera, znajdziemy w książce „Clause and Effect” Williama Clocksina, wydanej przez Springer Verlag w 1997 roku.

Rozdział 8.

Usuwanie błędów w programach prologowych

Aż do teraz używaliśmy różnych programów przykładowych i modyfikowaliśmy je, ale w końcu czytelnicy tej książki będą tworzyli swoje własne programy. Trzeba zastanowić się, co robić, jeśli programy te będą działały niezgodnie z oczekiwaniami. Czasami proces usuwania błędów z programów nazywa się „odpluskwianiem” (ang. *debugging*). Autorzy wierzą, że dobrym rozwiązaniem jest „bezpieczne programowanie”. Zgodnie ze starą zasadą programistyczną, im staranniej napisany program, tym szybciej można usunąć zeń wszystkie błędy. W tym rozdziale zajmiemy się pewnymi technikami poprawiania programów, ale najpierw powiemy, jak można ograniczyć ryzyko wystąpienia błędów w tworzonych programach. Zdajemy sobie sprawę, że problem tak postawiony w przypadku ogólnym jest w ogóle nierozwiązywalny, ale z drugiej strony można wskazać pewne techniki, które są przydatne dla osób programujących w języku Prolog.

Tak jak w przypadku wszelkiej działalności twórczej — czy to komponowania muzyki, pisania powieści czy architektury — w programowaniu można korzystać z różnych metod *reprezentowania* i *przetwarzania* obiektów i relacji związanych z jakimś zagadnieniem. Ogólnie rzecz biorąc, te same informacje można różnie zapisywać i przetwarzać. Zawsze, kiedy programista decyduje się na jakieś konkretne rozwiązanie, podejmuje *decyzję projektową*.

Kiedy osoby dopiero zaczynające naukę programowania stają przed koniecznością podjęcia decyzji projektowych, zwykle czują się zagubione. Zrozumienie, jakie decyzje można podjąć, pomoże wybrać najlepsze rozwiązanie, poza tym trzeba w ogóle znać istniejące techniki programowania. Sztuka podejmowania słusznych decyzji projektowych jest sama w sobie złożonym zagadnieniem. Staraliśmy się dać przedsmak tego już w pierwszym rozdziale, kiedy omawialiśmy różne sposoby interpretowania klauzul. Dobór jednej z interpretacji jest kwestią *reprezentowania* obiektów

i związków między nimi. Poza tym w rozdziale 7., kiedy omawialiśmy różne metody sortowania list, znów mieliśmy możliwość wyboru jednego z wielu rozwiązań. W tym wypadku była to kwestia *przetwarzania* obiektów i związków między nimi.

Mamy nadzieję, że książka ta pomoże czytelnikom rozwinąć umiejętność podejmowania właściwych decyzji projektowych. Po pierwsze, pokażemy szereg przykładowych programów, co powinno dać pogląd, jakie rozwiązania są powszechnie używane w programowaniu w ogóle. Po drugie, ten rozdział zawiera dużo wiadomości dotyczących konkretnie języka Prolog.

Układ programów

Gdy programista zdecydował już, jak zamierza reprezentować i przetwarzać obiekty i relacje między nimi, następnym krokiem jest zapewnienie takiego układu programu, aby jego działanie było jasne i czytelne. Zbiór klauzul danego predykatu nazywamy *procedurą*. W przykładach w tej książce każda klauzula procedury zaczyna się w nowym wierszu, poza tym między procedurami warto wstawiać wiersz odstępu. Na przykład jednym ze sposobów zapisu predykatu równości zbiorów (zbiory reprezentujemy jako listy) jest użycie trzech predykatów, każdy składający się z dwuwierszowej procedury:

```
rownezbior(X,X) :- !.
rownezbior(X,Y) :- rownelisty(X,Y).

rownelisty([],[]).
rownelisty([X|L1],L2) :- usun(X,L2,L3), rownelisty(L1,L3).

usun(X,[X|Y],Y).
usun(X,[X|L1],[Y|L2]) :- usun(X,L1,L2).
```

Nie jest to może najlepsza definicja równości zbiorów, ale za to dobrze pokazuje, jak powinny wyglądać procedury. Warto zauważyć, że klauzule poszczególnych procedur są zgrupowane razem, a procedury są oddzielone pustymi wierszami. Poza tym treści poszczególnych reguł są na tyle krótkie, że mieszczą się w pojedynczym wierszu. Inną konwencją stosowaną przez wielu programistów Prologu jest wypisywanie klauzuli w pojedynczym wierszu, o ile się w nim mieści. W przeciwnym razie w pierwszym wierszu pisze się głowę klauzuli i `:-`, a każdy cel koniunkcji zapisuje się w kolejnym wierszu. Oto przykładowy program generujący wszystkie permutacje listy:

```
permutuj([],[]).
permutuj(L,[H|T]) :-
    append(V,[H|U],L),
    append(V,U,W),
    permutuj(W,T).
```

W definicji tej korzystamy z nawracania w `append`, dzięki czemu kolejne permutacje `X` są generowane przy każdej próbie ponownego uzgodnienia celu `permutuj(X,Y)`. Warto zauważyć, jak składniki koniunkcji są ułożone w kolejnych wierszach.

Najważniejsze jest zdecydowanie się na jednolite konwencje, zaś to, jakie konwencje zostaną przyjęte, ma już mniejsze znaczenie. Dobrze jest umieszczać w programach komentarze, grupować odpowiednio terminy, w przypadku niejasności co do priorytetów operatorów stosować nawiasy, wstawiać dużo białych znaków (spacji i pustych wierszy). Komentarze powinny objaśniać strukturę i sposób interpretacji argumentów oraz ich kolejność. Poza tym dobrze jest poinformować w komentarzach, jak mają być ukonkretnione zmienne po uzgodnieniu klauzuli.

Jeśli chodzi o ogólniejsze zasady tworzenia programów, dobrze jest podzielić program na względnie niezależne części, na przykład wszystkie procedury przetwarzające listy można umieścić w pojedynczym pliku. Procedura Prologu, w której użyto więcej niż dziesięciu reguł może być nieczytelna, więc warto zastanowić się nad jej rozbięciem na predykaty opisujące poszczególne fragmenty zadania. Jeśli w programie używa się wielu faktów, takich jak reguły upraszczania wyrażeń z poprzedniego rozdziału, wszystkie te fakty powinny znaleźć się w jednym pliku. Ogólnie rzecz biorąc, duży zbiór faktów jest łatwiejszy do czytania niż duży zbiór reguł — o ile nawet względnie krótkie reguły mogą być trudne do zrozumienia, wiele stron jednego typu faktów można zrozumieć właściwie na pierwszy rzut oka.

Inne zagadnienie, które wpływa na czytelność programów języka Prolog, to używanie alternatywy w postaci średnika i używanie odcięć. Jakie kłopoty wiążą się z nadmiernym stosowaniem odcięć, powiedzieliśmy w rozdziale 4. Jeśli chodzi o użycie średnika, zawsze należy się zastanowić, czy nie lepiej byłoby zamiast tego zdefiniować osobne klauzule. Na przykład poniższy program:

```
bezpogladu(X) :-
    sprawdz(X,Funktor,IleArg,A), !,
    ( punktkontrolny(_,Funktor,A), !,
      { zabron(punktkontrolny(Glowa,Funktor,IleArg),_),
        podglad(Glowa,Tresc), zabron(Glowa,Tresc),
        write('Punkt kontrolny na '), pokazterm(Funktor,IleArg),
        write(' został usunięty.'), nl,
        fail : true ) : write('Na '), write(X),
        write(' nie ma punktu kontrolnego'), put(46), nl ), !.
```

jest przykładem, jak *nie należy* postępować. Jest on znacznie trudniejszy do zrozumienia niż:

```
bezpogladu(X) :-
    sprawdz(X,Funktor,IleArg,A), !,
    sprobuj_usunac(X,Funktor,IleArg,A).
sprobuj_usunac(_,Funktor,IleArg,A) :-
    punktkontrolny(_,Funktor,A), !,
    usun_punkt(Funktor,IleArg,A).
sprobuj_usunac(X,_,_) :-
    write('Na '), write(X), write(' nie ma punktu kontrolnego'),
    put(46), nl ), !.
usun_punkt(Funktor,IleArg,A) :-
    zabron(punktkontrolny(Glowa,Funktor,IleArg),_),
    podglad(Glowa,Tresc),
    zabron(Glowa,Tresc),
    write('Punkt kontrolny na '),
    pokazterm(Funktor,IleArg),
    write(' został usunięty.'), nl, fail.
usun_punkt(_,_,_).
```

który realizuje te same funkcje. Jeśli naprawdę chcesz używać średnika (;) jako alternatywy, dobrze jest ustawić koniunkcję celów tak, aby alternatywa była wyróżniona. Poza tym za pomocą nawiasów należy jawnie określić zakres obowiązywania operatora (;).

W tej książce wielokrotnie podkreślaliśmy konieczność myślenia o różnych problemach z punktu widzenia warunków końcowych i reguły zasadniczej. Jeśli tylko jest to możliwe, należy warunek końcowy umieszczać przed jakimikolwiek innymi klauzulami procedury — dzięki temu łatwiej później ten warunek odnaleźć, poza tym łatwiej uchronić się w ten sposób przed definicjami cyklicznymi. Czasami jednak dobrze jest warunek końcowy umieścić na końcu procedury. Oczywiście jest na przykład, że reguły obejmujące wszystkie przypadki poza jawnie wyliczonymi muszą być umieszczane na końcu procedury.

Podczas odczytywania procedury Prologu trzeba zawsze zwracać uwagę na następujące rzeczy:

- ♦ Sprawdzić poprawność zapisu wszystkich predykatów i zmiennych; często problemy wynikają z pomyłkowego wpisania nazwy.
- ♦ Sprawdzić liczbę składników poszczególnych funktorów; sprawdzić, czy liczba tych składników i ich kolejność są zgodne z przyjętymi założeniami projektowymi.
- ♦ Odnaleźć w procedurze operatory i sprawdzić ich priorytety, łączność oraz zlokalizować argumenty. Można skorzystać z deklaracji operatorów i ewentualnie nawiasów, jeśli je umieszczono. W razie wątpliwości należy dodać nawiasy. Poza tym trzeba sprawdzić, czy operatory zachowują się zgodnie z oczekiwaniami, obejrzeć za pomocą `write_canonical` przykładowe termy.
- ♦ Sprawdzić zakres poszczególnych zmiennych oraz odnaleźć wszystkie podobnie nazwane zmienne w danym zakresie. Należy zwrócić uwagę na to, że zmienne mogą być ze sobą powiązane. Sprawdzić, czy zmienne z głowy klauzuli występują potem w jej treści.
- ♦ Postarać się określić, które zmienne w chwili użycia klauzuli są ukonkretnione, a które nie.
- ♦ Odnaleźć klauzulę (lub klauzule), która definiuje warunki końcowe. Sprawdzić, czy wszystkie możliwe warunki końcowe zostały uwzględnione.

Kiedy procedura zostanie w opisany sposób przeanalizowana, na pewno stanie się bardziej zrozumiała.

Typowe błędy

W tym podrozdziale wskażemy szereg problemów, z jakimi stykają się początkujący i zaawansowani programiści Prologu. Problemy te mogą być dwójakiego rodzaju: błędy *składniowe* oraz błędy *sterowania*.

Kiedy już wiemy, jaki program chcemy stworzyć i jaki będzie jego układ, pojawia się problem wpisania tego programu do pliku. Najważniejsza kwestia to wpisanie go bez błędów składniowych. Oto szereg typowych błędów *składniowych*. Jeśli błędy te nie zostaną zauważone przez programistę, Prolog może w chwili próby wczytania programu przy użyciu predykatu `consult` wyświetlić komunikat o błędzie.

- ♦ Często zdarza się zapomnieć o kropce na końcu klauzuli. Kropka musi znajdować się za każdym termem wczytywanym za pomocą predykatu `read`, poza tym za kropką musi pojawić się co najmniej jeden biały znak. Trzeba więc pamiętać też o tym, aby za ostatnią kropką w pliku coś się pojawiło, zwykle po prostu przejście do nowego wiersza.
- ♦ Niektóre znaki specjalne występują parami. Przykładem są okragłe nawiasy grupujące termy, nawiasy kwadratowe stosowane przy zapisie list oraz nawiasy klamrowe stosowane w notacji reguł gramatyki (rozdział 9). Poza tym parami używane są podwójne cudzysłowy do zapisywania łańcuchów, pojedyncze cudzysłowy do zapisu atomów. Nawiasy złożone, /* i */, otaczają komentarze. Zawsze trzeba sprawdzić, czy zgadza się liczba poszczególnych rodzajów nawiasów.
- ♦ Wiele błędów bierze się z nieprawidłowego wpisania słów, szczególnie nazw predykatów wbudowanych. Może to powodować niespodziewane nawroty, gdyż błędnie wpisanym predykatom nie będą odpowiadały żadne klauzule z bazy danych. Czasami może też okazać się, że do nieprawidłowego operatora pasują całkiem inne klauzule niż programista miał na myśli.
- ♦ Źródłem możliwych błędów są operatory. W razie wątpliwości należy stosować nawiasy, aby jawnie wskazać łączność. Za pomocą predykatu `write_canonical` można sprawdzić działanie zdefiniowanych operatorów.

Podczas sprawdzania sposobu zapisania liczb, warto odpowiedzieć sobie na poniższe pytania:

- ♦ Jak dopasowują się $[a, b, c]$ i $[X|Y]$? (X jest ukonkretniony a , a, Y jest ukonkretnione $[b, c]$).
- ♦ Czy $[a]$ i $[X|Y]$ można dopasować? (Tak. X jest ukonkretnione a , $a, Y = []$).
- ♦ Czy można dopasować $[]$ i $[X|Y]$? (nie).
- ♦ Czy zapis $[X, Y|Z]$ jest prawidłowy? (tak).
- ♦ Czy zapis $[X|Y, Z]$ jest prawidłowy? (nie).
- ♦ Czy zapis $[X|[Y|Z]]$ jest prawidłowy? (Tak, jest on równoważny z $[X, Y|Z]$).
- ♦ Jak dopasowują się $[a, b]$ i $[A|B]$? (A jest ukonkretnione a , a, B jest ukonkretnione $[b]$).
- ♦ Czy powyższe można dopasować w więcej niż jeden sposób? (nie).

Kiedy mamy do czynienia z listami lub innymi strukturami tego typu, należy podkreślić przydatność diagramów w formie drzew, które omawialiśmy w rozdziale 2.

Zarządzanie i konsultacje
ul. Armii Krajowej 4 30-150 Kraków
tel. (012) 656-68-77 tel.fax (012) 657-53-47
wypr. do Rejestru MEN nr 16 z datą 11.05.1990
Konto BOS O/Rachunek 1540115-12687-2/106-06
NIP 677-17-58-169 REGON 350914-1

Nawet jeśli w programie nie ma żadnych błędów składniowych, program ten nadal może działać nieprawidłowo. Typowe objawy to: działanie programu bez żadnych przerw (nieskończona pętla), niespodziewana odpowiedź *no*, inne niż oczekiwano wartości ukonkretnianych zmiennych. Zwykle źródłem tego typu błędów są:

- ◆ Zapętlone definicje, o których wspomniano w rozdziale 3.
- ◆ Niewystarczające warunki końcowe lub niepełne opisanie zagadnienia.
- ◆ Zbędne procedury nadpisujące predykaty wbudowane.
- ◆ Podanie funktorowi nieprawidłowej liczby parametrów. Nie jest to błąd składniowy, gdyż liczba argumentów funktora może zależeć od sposobu użycia tego funktora.
- ◆ Nieoczekiwany koniec pliku podczas działania predykatu *read*.

Jeden z najtrudniejszych do wychwycenia błędów zilustrowano poniżej. w programie porównującym listy:

```
eq([], []).
eq([X|L], M) :- del(X, M, N), eq(L, N),

del(X, [X|Y], Y).
del(X, [Y|L1], [Y|L2]) :- del(X, L1, L2).
```

Gdzie jest błąd? Druga klauzula *eq* kończy się przecinkiem. Na pierwszy rzut oka widać, że jest to błąd, ale program jest *poprawny składniowo*, bo następny term traktowany jest jako następny cel koniunkcji. Powyższy program jest równoważny z poniższym, w oczywisty sposób nie sprawdzającym równości list:

```
eq([], []).
eq([X|L], M) :- del(X, M, N), eq(L, N), del(X, [X|Y], Y).

del(X, [Y|L1], [Y|L2]) :- del(X, L1, L2).
```

Inny błąd podobnego typu zaprezentowano niżej:

```
eq([], []).
eq([X|L], M) :- del(X, M, N), eq(L, N).

del(X, [X|Y], Y).
del(X, [Y|L1], [Y|L2]) :- del(X, L1, L2).
```

Druga klauzula *eq* zawiera kropkę rozdzielającą jej podcele, podczas gdy powinien być to przecinek. Program znów jest poprawny składniowo; jest równoważny programowi:

```
eq([], []).
eq([X|L], M) :- del(X, M, N),
eq(L, N).

del(X, [X|Y], Y).
del(X, [Y|L1], [Y|L2]) :- del(X, L1, L2).
```

Trzeba też pamiętać o pułapkach związanych z nawracaniem:

- ◆ Jednym z powodów istnienia nawracania jest to, że Prolog może powrócić do poprzedniego dopasowania i zmienić je na inne. Tak naprawdę, kiedy Prolog przeszukuje bazę danych w celu znalezienia dopasowania celu do faktu lub głowy reguły z bazy danych, dopasowanie albo się powiedzie, albo zawiedzie. Prolog nie nawraca do dopasowania i nie modyfikuje tego dopasowania, gdyż dopasowywane mogą być jedynie cel i klauzula z bazy danych.
- ◆ Zapis list w formie $[X|Y]$ może być dopasowany do dowolnego fragmentu listy, w ten sposób można listę rozkładać w różnoraki sposób. Dzięki temu `append(X, Y, [a, b, c, d])` pokazuje możliwe sposoby rozłożenia listy na części. Tak naprawdę w przypadku $[X|Y]$ zmienna *X* pasuje jedynie do głowy listy, zaś *Y* do jej ogona. Cele `append` mogą zwracać różne części listy dzięki nawracaniu, nie dzięki dopasowywaniu.

Śledzenie programu

Można różnie patrzeć na sposób spełniania celów przez Prolog. Omówiliśmy to zagadnienie opierając się na „kolejności spełniania celów” i rysując prostokąty symbolizujące cele. Tym razem zaprezentujemy model używany w wielu narzędziach śledzących Prologu, w szczególności w narzędziu *trace*. Model ten w dużej mierze powstał dzięki Lawrence’owi Byrdowi, stąd jego nazwa „model prostokątów Byrda”. Różne systemy Prologu oferują różne narzędzia do śledzenia programów (a standard Prologu nie wskazuje, które z nich są obowiązkowe), dlatego dalszy opis dość dobrze będzie opisywał możliwości większości systemów Prologu.

Kiedy używa się *trace*, Prolog wyświetla informacje o tym, jakie cele są kolejno wykonywane. Aby jednak zrozumieć te komunikaty, trzeba zrozumieć, kiedy i dlaczego poszczególne informacje są prezentowane. W proceduralnych językach programowania najważniejsze są punkty wejścia do funkcji i wyjścia z nich. Jednak w Prologu można tworzyć programy niedeterministyczne, co jest przyczyną złożoności nawracania. Do klauzul nie tylko się wchodzi i z nich wychodzi, bo nawracanie może ponownie wywołać te same klauzule w celu znalezienia innych rozwiązań. Co więcej, odciecie oznaczane wykrzyknikiem (!) wskazuje, że cel jest zatwierdzony jako mający tylko jedno rozwiązanie. Dla nowicjuszy jednym z najtrudniejszych zagadnień jest zrozumienie, co się dzieje, kiedy jakiś cel zawodzi i system zaczyna nawracanie. Mamy nadzieję, że wyjaśniliśmy to dość dokładnie we wcześniejszych rozdziałach. Omówiliśmy nie tylko przepływ sterowania, ale też ukonkretnianie zmiennych, dopasowywanie celów do głów klauzul z bazy danych oraz sposób spełniania podcelów. Opisując model śledzenia w Prologu, musimy omówić cztery *zdarzenia*:

CALL. Zdarzenie *CALL* pojawia się wtedy, kiedy Prolog stara się spełnić cel. Na diagramach odpowiada to strzałce wchodzącej do prostokąta z góry.

EXIT. Zdarzenie *EXIT* pojawia się wtedy, kiedy jakiś cel zostanie spełniony. Na diagramach odpowiada to strzałce wychodzącej z dołu prostokąta.

REDO. Zdarzenie REDO występuje wówczas, gdy system wraca do danego celu i stara się go ponownie spełnić. Na diagramach odpowiada to strzałce, która wraca do prostokąta od dołu.

FAIL. Zdarzenie FAIL występuje wtedy, kiedy cel zawodzi. Na diagramach odpowiada to strzałce wracającej z prostokąta w górę.

Podczas śledzenia programu użytkownik jest informowany, które z powyższych czterech zdarzeń zachodzi dla poszczególnych celów. Aby można było odróżnić, które zdarzenia z którymi celami są powiązane, celom nadaje się identyfikator liczbowy określany mianem *numeru wywołania*. Poniżej pokażemy zestaw celów z ujętymi w nawiasy kwadratowe ich numerami wywołania.

Zajmijmy się teraz konkretnym przykładem. Oto definicja predykatu potomek:

```
potomek(X,Y) :- potomstwo(X,Y).
potomek(X,Z) :- potomstwo(X,Y), potomek(Y,Z).
```

Program ten pokazuje potomków danej osoby. Będziemy używać następującej bazy danych faktów:

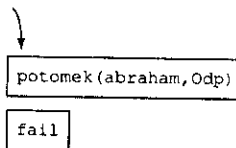
```
potomstwo(abraham,izmael).
potomstwo(abraham,izaak).
potomstwo(izaak,ezaw).
```

Pierwsza klauzula potomek mówi, że Y jest potomkiem X, jeśli należy do jego potomstwa. Druga klauzula mówi, że Z jest potomkiem X, jeśli Z jest potomkiem potomstwa X. Przyjrzyjmy się zapytaniu:

```
?- potomek(abraham,Odp), fail.
```

Będziemy badali sterowanie programu i sprawdzali zachodzące zdarzenia. Śledząc wykonanie programu, musimy myśleć w kategoriach przechodzenia przez prostokąty odpowiadające celom. Co jakiś czas będziemy pokazywać aktualny stan programu w formie diagramu.

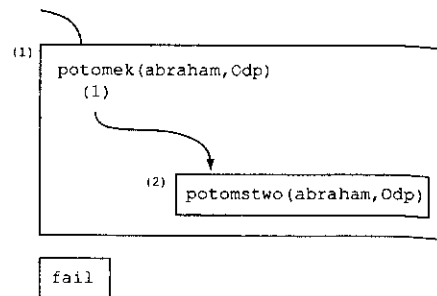
Za pierwszym celem w naszym zapytaniu występuje fail, które będzie wymuszało nawracanie po celu potomek. Zapytanie jako całość nie może być więc nigdy spełnione, ale naszym celem jest obserwacja nawracania. Zaczynamy od dwóch prostokątów celów jeszcze przed wejściem do któregośkolwiek z nich:



Pierwsze zdarzenie to wywołanie (CALL) celu potomek. Wywołanie to otrzyma numer 1 (w nawiasach kwadratowych).

```
[1] CALL: potomek(abraham,Odp)
[2] CALL: potomstwo(abraham,Odp)
```

Dopasowywana jest pierwsza klauzula procedury potomek, więc otrzymujemy CALL i sytuacja przedstawia się następująco:



Idziemy dalej:

```
[2] EXIT: potomstwo(abraham,izmael)
```

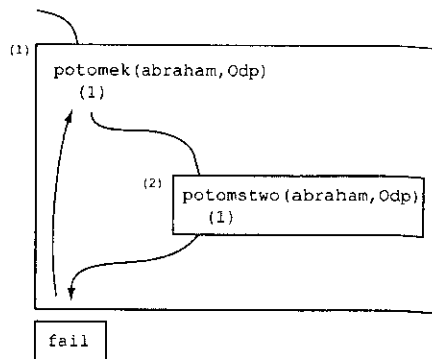
Pierwsza klauzula została od razu uzgodniona, więc otrzymujemy EXIT.

```
[1] EXIT: potomek(abraham,izmael)
```

Spełniona została zatem pierwsza klauzula potomek.

```
[3] CALL: fail
[3] FAIL: fail
[1] REDO: potomek(abraham,izmael)
```

Chcieliśmy spełnić cel fail, który oczywiście zawodzi (FAIL). Strzałka wraca z prostokąta fail do prostokąta potomek powyżej. Oto odpowiedni diagram; strzałka jest skierowana w górę, z prostokąta fail.



Idziemy dalej:

```
[2] REDO: potomstwo(abraham,izmael)
[2] EXIT: potomstwo(abraham,izaak)
```

Wybrana została inna klauzula celu potomstwo, więc strzałka znów wychodzi z prostokąta w dół.

```
[1] EXIT: potomek(abraham,izaak)
[4] CALL: fail
[4] FAIL: fail
[1] REDO: potomek(abraham,izaak)
```

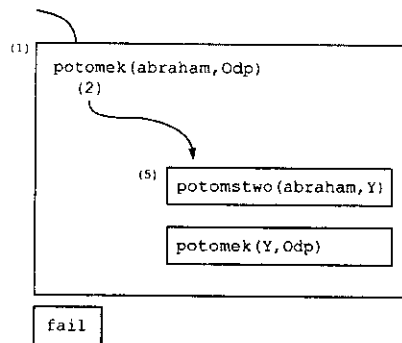
Znów fail powoduje odrzucenie rozwiązania i nawracanie. Zauważmy, że jest to całkiem nowe wywołanie fail (całkiem od nowa weszliśmy doń „z góry”).

```
[2] REDO: potomstwo(abraham,izaak)
[2] FAIL: potomstwo(abraham,Odp)
```

Tym razem potomstwo nie ma już żadnych innych dopasowań, więc nawracanie jest kontynuowane i strzałka cofając się dalej, wychodzi poza prostokąt potomstwo.

```
[5] CALL: potomstwo(abraham,Y)
```

Tym razem wybrana została druga klauzula potomek, więc mamy całkiem nowe wywołanie potomstwo odpowiadające pierwszemu podcelowi:



Strzałka znów przemieszcza się w dół:

```
[5] EXIT: potomstwo(abraham,izmael)
[6] CALL: potomek(izmael,Odp)
```

W ten sposób dochodzimy do rozwiązania, w którym rekurencyjnie wywołujemy predykat potomek otrzymując nowe wywołanie tego ostatniego.

```
[7] CALL: potomstwo(izmael,Odp)
[7] FAIL: potomstwo(izmael,Odp)
[8] CALL: potomstwo(izmael,Y2)
[8] FAIL: potomstwo(izmael,Y2)
[6] FAIL: potomek(izmael,Odp)
```

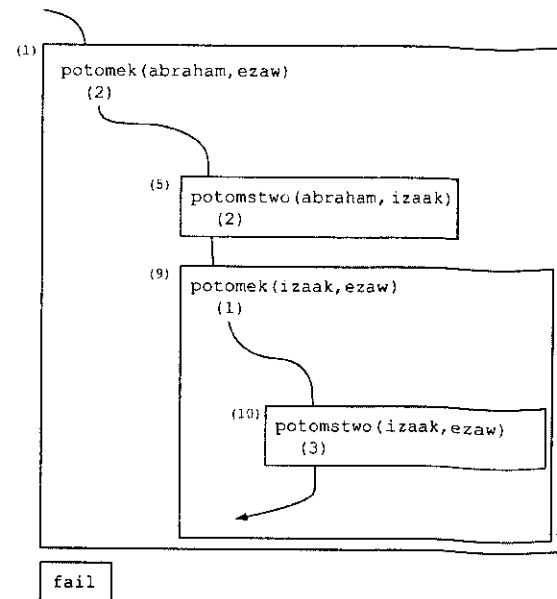
Izmael nie ma już potomstwa (w naszym przykładzie), więc podcele potomstwo w obu klauzulach potomek zawodzą, zatem zawodzi i cel potomek.

```
[5] REDO: potomstwo(abraham,izmael)
```

Wracamy szukając innego rozwiązania.

```
[5] EXIT: potomstwo(abraham,izaak)
[9] CALL: potomek(izaak,Odp)
[10] CALL: potomstwo(izaak,Odp)
[10] EXIT: potomstwo(izaak,ezaw)
```

Otrzymujemy nowe wywołanie potomek, udaje się uzgodnić podcel potomstwo.



I dalej:

```
[9] EXIT: potomek(izaak,ezaw)
[1] EXIT: potomek(abraham,ezaw)
[11] CALL: fail
[11] FAIL: fail
[1] REDO: potomek(abraham,ezaw)
[9] REDO: potomek(izaak,ezaw)
```

I tak oto dochodzimy do ostatecznego rozwiązania początkowego zapytania, ale fail wymusza ponowny nawrót, więc znów pojawia się REDO.

```
[10] REDO: potomstwo(izaak,ezaw)
[10] EXIT: potomstwo(izaak,jakub)
[9] EXIT: potomek(izaak,jakub)
[1] EXIT: potomek(abraham,jakub)
```

Podcel potomstwo ma inne rozwiązanie, które daje inny wynik dla pierwotnego celu potomek. Jak widać, otrzymujemy w ten sposób ostatniego znanego potomka Abrahama, choć przetwarzanie na tym się jeszcze nie kończy. Teraz następują nawroty do samego początku.

```
[12] CALL: fail
[12] FAIL: fail
[1] REDO: potomek(abraham,jakub)
[9] REDO: potomek(izaak,jakub)
[10] REDO: potomstwo(izaak,jakub)
[10] FAIL: potomstwo(izaak, Odp)
[13] CALL: potomstwo(izaak,Y3)
```

Teraz próbujemy skorzystać z drugiej klauzuli potomek.

```
[13] EXIT: potomstwo(izaak,ezaw)
[14] CALL: potomek(ezaw,Odp)
```

I znowu.

```
[15] CALL: potomstwo(ezaw,Odp)
[15] FAIL: potomstwo(ezaw,Odp)
[16] CALL: potomstwo(ezaw,Y4)
[16] FAIL: potomstwo(ezaw,Y4)
[14] FAIL: potomek(ezaw,Odp)
[13] REDO: potomstwo(izaak,ezaw)
[13] EXIT: potomstwo(izaak,jakub)
[17] CALL: potomek(jakub,Odp)
```

Sprawdzamy Jakuba.

```
[18] CALL: potomstwo(jakub,Odp)
[18] FAIL: potomstwo(jakub,Odp)
[19] CALL: potomstwo(jakub,Y5)
[19] FAIL: potomstwo(jakub,Y5)
[17] FAIL: potomek(jakub,Odp)
[13] REDO: potomstwo(izaak,jakub)
[13] FAIL: potomstwo(izaak,Y3)
[9] FAIL: potomek(izaak,Odp)
[1] FAIL: potomek(abraham,Odp)
no
```

I to już wszystko. Mamy nadzieję, że tak obszerny przykład pozwoli dokładnie zrozumieć, jak wykonywany jest program Prologu. Nietrudno zauważyć, że dla każdego celu może wystąpić tylko raz CALL i FAIL, choć wielokrotnie może występować REDO i EXIT. W następnym podrozdziale przyjrzymy się śledzeniu bardziej złożonego przykładu, append.

Ćwiczenie 8.1. W powyższym modelu nie wspomniano w ogóle, jak obsługiwane jest odcięcie. Rozszerz model tak, aby uwzględnić odcięcie.

Śledzenie i punkty kontrolne

Kiedy okazuje się, że napisany program nie działa (zgłaszany jest błąd, udzielana jest odpowiedź no lub uzyskiwana odpowiedź jest nieprawidłowa), chcemy szybko znaleźć błędy i je poprawić. W tym podrozdziale opiszemy predykaty wbudowane, które pozwalają „podglądać” wykonywanie programu. Dzięki nim można zadać programowi to samo zadanie i sprawdzić, co działa niezgodnie z oczekiwaniami. Podczas śledzenia programu będziemy analizować poszczególne zdarzenia, jak to pokazano w poprzednim podrozdziale. Konkretnie rozwiązania pozwalające na śledzenie programów zależą od używanej implementacji Prologu, ale cechy opisane dalej są typowe i w takiej czy innej postaci powinny być zawsze dostępne. Przed rozpoczęciem korzystania z tych możliwości warto zajrzeć do dokumentacji używanego systemu.

Podstawowa zasada związana ze śledzeniem programu to to, że programista jest informowany o spełnieniu pewnych celów. To programista decyduje, o których dokładnie celach ma być informowany oraz w jakim stopniu chce wpływać na spełnianie tych celów. Pierwsza decyzja pozwala wybrać, jak szczegółowe ma być śledzenie. Zasadniczo możliwe jest śledzenie *wszystkich* celów, zaś użycie punktów kontrolnych pozwala obserwować wykonywanie jedynie wybranych predykatów. Obie możliwości można ze sobą swobodnie mieszać. W podrozdziale „Badanie działania Prologu” w rozdziale 6. wymieniliśmy predykaty wbudowane przeznaczone do śledzenia programu. Aby ustawić na predykanie punkt kontrolny, używamy spy (aby punkt usunąć, używamy nospy). Aby zacząć dokładne śledzenie wszystkich celów, używamy trace; aby je zakończyć, wywołujemy notrace.

Druga decyzja związana ze śledzeniem wskazuje, na ile ściśle ma być kontrola nad wykonaniem programu. Jeśli żąda się ścisłej kontroli, wszystkie informacje są wyświetlane, w każdej chwili programista proszony jest o podjęcie decyzji co do sposobu dalszego działania. Kiedy nie żąda się ścisłej kontroli nad programem, wyświetlane są wszystkie informacje, ale program działa dalej bez interwencji programisty. Możliwa jest zmiana poziomu szczegółowości śledzenia, modyfikacja standardowego przepływu sterowania i inne. Używany system Prologu może umożliwiać śledzenie i kontrolowanie następujących czterech zdarzeń:

- ♦ Próby pierwszego spełnienia celu: kiedy cel jest spotykany po raz pierwszy (zdarzenie CALL).
- ♦ Spełnienia celu (zdarzenie EXIT).
- ♦ Próby ponownego spełnienia celu (zdarzenie REDO).
- ♦ Zawiedzenia celu (zdarzenie FAIL).

Dobrze jest wybrać kontrolowanie zdarzeń CALL i REDO, zaś pominąć zdarzenia EXIT i FAIL. Dokładniej zdarzenia te opisano w poprzednim podrozdziale.

Przyjrzymy się teraz, jakie informacje są przekazywane użytkownikowi, kiedy zachodzi zdarzenie dotyczące interesującego go celu. Przede wszystkim pokazywany jest sam cel wraz z rodzajem zdarzenia i ewentualnie numerem wywołania. Jeśli śledzenie danego typu zdarzenia jest związane z ingerencją użytkownika, użytkownik

dotatkowo proszony jest o wskazanie następnego kroku. Sesja z pełnym śledzeniem bez ingerencji użytkownika może wyglądać następująco:

```
?- [user].

append([],Y,Y).
append([A|B],C,[A|D]) :- append(B,C,D).
/* tu należy wpisać znak końca pliku */
yes
?- append([a],[b],X).
CALL append([a],[b],_43)
CALL append([],[b],_103)
EXIT append([],[b],[b])
EXIT append([a],[b],[a,b])
X = [a,b] ;
REDO append([a],[b],[a,b])
REDO append([],[b],[b])
FAIL append([],[b],_103)
FAIL append([a],[b],_43)
no
?- append(X,Y,[a]).
CALL append(_37,_38,[a])
EXIT append([],[a],[a])
X = [], Y = [a] ;
REDO append([],[a],[a])
CALL append(_93,_38,[])
EXIT append([],[],[])
EXIT append([a],[a],[a])
X = [a], Y = [] ;
REDO append([a],[a],[a])
REDO append([],[],[])
FAIL append(_93,_38,[])
FAIL append(_37,_38,[a])
no
```

W tym wypadku pokazywane były wszystkie cztery zdarzenia dla wszystkich celów. Jednak programista nie miał możliwości przzerwania działania programu, zmiany zakresu śledzenia czy zmiany sposobu jego wykonywania w jakikolwiek inny sposób. Możliwości te daje ściśła kontrola wykonywania programu.

Zanim zajmiemy się ściśłą kontrolą wykonywania programu, powiedzmy jeszcze, jak Prolog pokazuje cele podczas śledzenia. Sposób pokazywania celów podczas śledzenia nie musi być taki sam, jak w przypadku użycia write. Wynika to stąd, że można podać własne definicje opisujące sposób prezentacji celów. Dzięki temu można pewne typowe struktury danego programu pokazywać w sposób bardziej zrozumiały czy bardziej zwarty niż zwykle robi to write. Normalnie do pokazywania celów używa się jednoargumentowego predykatu wbudowanego print, który działa następująco:

```
print(X) :- portray(X), !.
print(X) :- write(X).
```

Predykat portray nie jest predykatem wbudowanym, więc można samemu go zdefiniować. Jeśli będą istniały klauzule użytkownika pozwalające wywołać portray(X) dla danego celu X, zakłada się, że wystarczą one do prezentacji tego celu. W przeciwnym razie użyte zostanie zwykle write. Jeśli z jakiegoś powodu nie chcielibyśmy widzieć trzeciego argumentu celów append, moglibyśmy sobie to zapewnić używając klauzuli:

```
portray(append(A,B,C)) :-
    write('append('), write(A), write(','),
    write(B), write(','),
    write('<nic>')).
```

Zawsze, kiedy pojawia się cel zawierający append, klauzula ta spowoduje, że nie zawiedzie portray(X) i cel będzie w taki właśnie sposób przedstawiony. W przypadku celów zawierających wszelkie inne predykaty, portray(X) zawiedzie, wobec czego cel zostanie wypisany za pomocą write. Jeśli w bazie danych byłaby pokazana powyższej klauzula, początek powyższej przykładowej sesji wyglądałby następująco:

```
?- append([a],[b],X).
CALL append([a],[b],<nic>)
CALL append([],[b],<nic>)
EXIT append([],[b],<nic>)
EXIT append([a],[b],<nic>)
X = [a,b] ;
REDO append([a],[b],<nic>)
REDO append([],[b],<nic>)
FAIL append([],[b],<nic>)
FAIL append([a],[b],<nic>)
no
```

Teraz przejdźmy do śledzenia pod nadzorem. Jeśli śledzone mają być zdarzenia jakiegoś typu, to przy ich wystąpieniu system będzie prosił o wskazanie następnej akcji. Może to mieć postać:

```
?- append([a],[b],X).
CALL append([a],[b],_43) ?
```

Program czeka teraz na odpowiedź użytkownika. Jeśli wskazana opcja nakazuje kontynuację wykonywania programu, będzie się on wykonywał aż do następnego śledzonego zdarzenia i wtedy pojawi się znowu pytanie:

```
CALL append([],[b],_103) ?
```

Zwykle jedna z opcji, jakie może wybrać użytkownik, wyświetla wszystkie dostępne opcje. Oto niektóre z nich:

Sprawdzanie celu

Pierwszy zestaw opcji zawiera sprawdzanie celu w różnoraki sposób. Jak widzieliśmy, standardem jest pokazanie celu za pomocą print, co pozwala użyć klauzuli portray do zastosowania specjalnego formatowania. Jednak można zacząć się zastanawiać, czy klauzule te napisano prawidłowo lub możemy chcieć zobaczyć cel w standardowej postaci. Dlatego Prolog pozwala bieżący cel pokazać za pomocą write lub write_canonical. W takim wypadku program nie będzie się wykonywał dalej, lecz poprosi o wskazanie następnej akcji. Może to wyglądać tak:

```
?- append([a],[b],X).
CALL append([a],[b],<nic>) ? write
CALL append([a],[b],_103) ?
```

**Wyższa Szkoła
Zarządzania i Bankowości**
ul. Armii Krajowej 4, 30-150 Kraków
tel. (012) 638-65-77, tel./fax (012) 631-33-47
Wpis do Rejestru MEB nr 55 z dnia 11.05.1999
Konto BOS Główny 13401115-12027-27036-00
NIP 677-17-58-169 REGON 365619666

Zwykle używa się jedynie `write`. `write_canonical` można zastosować, jeśli cel zawiera dużo operatorów i nie pamiętamy, jakie są ich priorytety; wtedy `write_canonical` pokazuje operatory jako funktory, których zagnieżdżenia pozwolą jednoznacznie odczytać wyrażenie.

prawdzenie przodków

Przodkowie celu to te cele, których spełnienie zależy od spełnienia celu bieżącego. Wobec tego każdy cel ma przodka, który jest jednym z celów pierwotnego zapytania. Poza tym zawsze, kiedy używana jest reguła, każdy z celów z treści reguły ma przodka wśród celów z głowy reguły. Przyjrzyjmy się jakimś przykładom przodków. Oto pierwszy program odwracający listę z poprzedniego rozdziału:

```
rev([], []).
rev([H|T], L) :- rev(T, Z), append(Z, [H], L).
append([], X, X).
append([A|B], C, [A|D]) :- append(B, C, D).
```

Jeśli zadamy zapytanie:

```
?- rev([a,b,c,d], X).                                     (A)
```

To z uwagi na istnienie drugiej klauzuli `rev` będzie trzeba spełnić dwa podcele. Każdy z nich ma bezpośredniego przodka w naszym zapytaniu; podcele to:

```
rev([b,c,d], Z)                                           (B)
append(Z, [a], X)                                         (C)
```

Druga klauzula ponownie jest używana do spełnienia (B), więc znów mamy dwa podcele:

```
rev([c,d], Z1)                                           (D)
append(Z1, [a], Z)                                       (E)
```

(A) i (B) są przodkami tak (D), jak i (E). Cel (C) nie jest ich przodkiem, gdyż wpływają one na spełnienie (B), które wpływa na spełnienie (A). Cele (D) i (E) nie wpływają w żaden sposób na spełnienie (C). Kiedy próba spełnienia zapytania będzie już dostatecznie zaawansowana, pojawi się cel:

```
append([c],[b], Y)
```

Na tym etapie cel i jego przodków można zapisać następująco:

```
rev([a,b,c,d], _46)                                     (Cel A)
rev([b,c,d],[d]_50)                                     (Cel B)
append([d,c],[b],[d]_51)
append([c],[b], _52)
```

Zanim przejdiesz do dalszego ciągu, musisz wiedzieć, dlaczego są to wszyscy przodkowie celu i dlaczego nie ma już więcej przodków. W zaprezentowanym sposobie pokazywania przodków istnieje pewna cecha szczególna, która może znaleźć odbicie w niektórych systemach Prologu. Przodka można pokazać w dwojaki sposób: w formie, jaką miał podczas pierwszej próby jego spełnienia i w postaci aktualnej, z ukonkretnionymi

wszystkimi zmiennymi. Tutaj pokazaliśmy drugą ewentualność. Kiedy pierwszy raz natykamy się na cel (B), drugi argument `rev` jest nieukonkretniony, ale jego wartość pokazano na liście przodków. Wynika to stąd, że obecnie zmienna ta została ukonkretniona i wiemy już, że pierwszy element odwrócenia `[b,c,d]` to `d`.

Przeglądając się przodkom bieżącego celu, możemy łatwo zorientować się, jak działa program i dlaczego. Jedną z możliwości w przypadku śledzenia z pełną kontrolą jest pokazanie wybranych przodków. Jeśli zatem któraś część programu wykonuje się bardzo długo i podejrzewamy, że możemy mieć do czynienia z pętlą, dobrze jest przerwać wykonywanie programu, włączyć pełne śledzenie i przejrzeć przodków wybranego celu, aby sprawdzić, gdzie doszliśmy.

Zmiana poziomu śledzenia

Kolejny zestaw opcji związany jest ze zmianą zakresu śledzenia. Oto niektóre bardziej całościowe możliwości:

- ♦ Usunięcie wszystkich punktów kontrolnych. Działa tak samo, jak wywołanie celu `nodebug`.
- ♦ Wyłączenie pełnego śledzenia. Działa tak samo, jak wywołanie celu `notrace`.
- ♦ Włączenie pełnego śledzenia. Działa tak samo jak wywołanie celu `trace`.

Cele `nodebug`, `notrace` i `trace` były omawiane w rozdziale 6.

Wszystkie powyższe opcje powodują kontynuację wykonywania programu aż do znalezienia celu, który ma być teraz śledzony. W zależności od używanej wersji Prologu, dostępnych może być więcej opcji o charakterze lokalnym, które pozwalają szybko przejść przez mniej interesujące fragmenty programu i skoncentrować się na tych częściach, w których podejrzewa się istnienie błędów. Opcjami takimi mogą być:

- ♦ `creep`: kontynuacja wykonywania programu z pełnym śledzeniem aż do następnego punktu, w którym użytkownik ma podjąć decyzję.
- ♦ `skip`: kontynuacja wykonywania programu bez wyświetlania komunikatów śledzenia aż do następnego zdarzenia związanego z bieżącym celem.
- ♦ `leap`: kontynuacja wykonywania programu bez wyświetlania komunikatów śledzenia aż do następnego punktu kontrolnego lub do wystąpienia zdarzenia związanego z bieżącym celem.

Pierwsza z powyższych opcji jest używana wówczas, gdy chcemy od danej chwili dokładnie śledzić program. Kiedy sposób spełnienia danego celu nie ma znaczenia, a chcemy sprawdzić, co dzieje się dalej, używamy drugiej opcji. Trzeciej opcji używa się wtedy, kiedy podczas spełniania celu w zasadzie nic ciekawego się nie dzieje, ale coś takiego (oznaczonego punktem śledzenia) może jednak mieć miejsce. Wobec tego nie interesuje nas nic aż do takiego punktu lub aż do spełnienia lub nie aktualnego celu. Oto przykład użycia `creep` i `skip`. Załóżmy, że w programie sortującym z podrzdziału „Sortowanie” w rozdziale 7, występuje błąd, ale wiemy, że permutacje generowane są poprawnie. Definicja `sort` zaczynała się następująco:

```
sort(X,Y) :- permutacja(X,Y), posortowana(Y), !.
```

Możemy użyć opcji skip, by uniknąć zaglądania we wszystkie nudne szczegóły działania predykatu permutacja, ale śledzić pozostałą część programu:

```
CALL sort([3,6,2,9,20],_45) ? creep
CALL permutacja([3,6,2,9,20],_45) ? skip
EXIT permutacja([3,6,2,9,20],[3,6,2,9,20]) ? creep
CALL posortowana([3,6,2,9,20]) ? creep
CALL posortowana([0,[3,6,2,9,20]]) ? creep
CALL 0<3 ?
```

... i tak dalej.

Zmiana sposobu spełnienia celu

Poniższe opcje pozwalają zmienić sposób działania programu. Można ich użyć do wielokrotnego powtarzania podejrzanych części programu bez konieczności zajmowania się przypadkami w danej chwili nieistotnymi oraz do wymuszenia na programie przetwarzania w sposób, który normalnie nie jest uwzględniany. Pozwala to znacznie przyspieszyć usuwanie błędów, gdyż można wielokrotnie wykonywać najtrudniejsze fragmenty programu zamiast zawsze powtarzać wszystko od nowa.

- ♦ **retry**: jeśli po wystąpieniu jakiegoś zdarzenia użytkownik odpowie retry, Prolog wraca do punktu, w którym cel został wywołany (CALL). Wszystko jest przywracane do stanu pierwotnego (wyjątkiem są modyfikacje bazy danych). Teraz można spróbować spełnić ten sam cel ponownie. Typową techniką jest łączenie opcji retry i skip. Jeśli nie jesteśmy pewni, czy błąd występuje podczas spełniania pewnego celu, możemy najpierw cel ten wywołać bez śledzenia (skip). Dzięki temu, jeśli cel zostanie spełniony poprawnie, można będzie uniknąć mnóstwa zbędnego śledzenia kodu. Jeśli wystąpi jakiś błąd i cel zawiedzie lub da nieprawidłowy wynik, możemy użyć opcji retry, aby się cofnąć i wykonując ten sam cel ponownie dokładniej mu się przyjrzeć.
- ♦ **or**: opcja ta działa podobnie jak użycie średnika wymuszające szukanie rozwiązań alternatywnych. Jeśli zdarzeniem danego celu jest EXIT, możemy w ten sposób poprosić o odszukanie rozwiązań alternatywnych. Jeśli wiemy z góry, że pierwsze znalezione rozwiązanie nie wystarczy do wykonania reszty programu, możemy natychmiast wymusić znalezienie następnego rozwiązania. Dzięki temu można szybciej dotrzeć do części programu zawierającej błąd. Rozwiązaniem alternatywnym byłoby, po znalezieniu pierwszego rozwiązania, oczekiwanie, aż któryś z dalszych celów zawiedzie.
- ♦ **fail**: opcja ta jest używana głównie przy zdarzeniu CALL. Jeśli wiemy z góry, że cel zawiedzie, a cel ten nie jest w danej chwili istotny, możemy wymusić, by zawiódł, natychmiast.

Oto przykład użycia wymienionych wyżej opcji podczas próby spełnienia zapytania:

```
?- member(X,[a,b,c]), member(X,[d,c,e]).
```

```
CALL member(_44,[b,c]) ? creep
EXIT member(a,[a,b,c]) ? or
REDO member(a,[a,b,c]) ? creep
CALL member(_44,[b,c]) ? fail
FAIL member(_44,[b,c]) ? creep
FAIL member(_44,[a,b,c]) ? retry
CALL member(_44,[a,b,c]) ? creep
EXIT member(a,[a,b,c]) ? creep
CALL member(a,[d,c,e]) ? fail
FAIL member(a,[d,c,e]) ? creep
REDO member(a,[a,b,c]) ? creep
CALL member(_44,[b,c]) ? creep
EXIT member(b,[b,c]) ? or
REDO member(b,[b,c]) ? creep
CALL member(_44,[c]) ? fail
FAIL member(_44,[c]) ? retry
CALL member(_44,[c]) ? creep
EXIT member(c,[c]) ? creep
EXIT member(c,[b,c]) ? creep
EXIT member(c,[a,b,c]) ? creep
CALL member(c,[d,c,e]) ? creep
CALL member(c,[c,e]) ? creep
EXIT member(c,[c,e]) ? creep
EXIT member(c,[d,c,e]) ? or
REDO member(c,[d,c,e]) ? creep
REDO member(c,[c,e]) ? creep
CALL member(c,[e]) ? creep
CALL member(c,[]) ? creep
FAIL member(c,[]) ? creep
FAIL member(c,[e]) ? creep
FAIL member(c,[c,e]) ? retry
CALL member(c,[c,e]) ? creep
EXIT member(c,[c,e]) ? creep
EXIT member(c,[d,c,e]) ? creep
```

Inne opcje

Oto pozostałe opcje, które mogą być użyte przy zatrzymaniu się programu:

- ♦ **break**: powoduje, że wykonywanie programu zostaje zawieszone i udostępniana jest nowa kopia interpretera Prologu. W ten sposób można sprawdzać, jakie są dostępne klauzule, ustawiać punkty kontrolne i tak dalej. Podczas zamykania interpretera (przez wpisanie znaku końca wiersza), Prolog wraca do zawieszonego wcześniej programu.
- ♦ **abort**: powoduje porzucenie aktualnie uruchomionych programów i powrót do interpretera Prologu.
- ♦ **halt**: powoduje natychmiastowe opuszczenie Prologu. Można z tej opcji skorzystać zaraz po znalezieniu błędu, aby uruchomić edytor i poprawić plik zawierający ten błąd.

Podsumowanie

Możemy teraz podsumować omówione dotąd zagadnienia. Kiedy zaczynamy sprawdzać sposób wykonywania programu, trzeba pamiętać o trzech rzeczach:

1. Który cel chcemy dokładnie zbadać? Jeśli będziemy badać wszystko (za pomocą trace), szybko zgubimy się w zalewie informacji. Z drugiej strony, jeśli będziemy sprawdzać jedynie kilka predykatów (ustawiając punkty kontrolne za pomocą spy), możemy przegapić miejsce, w którym pojawia się błąd. Najlepsze jest zwykle rozwiązanie pośrednie, w którym za pomocą spy ograniczamy ostrożnie badany zakres programu, a w celu ostatecznego znalezienia błędu używamy już trace.
2. Na ile chcemy ingerować w wykonywanie programu? Jeśli nie ustawimy żadnych punktów przerywania, program szybko się wykona, co utrudni nam odnalezienie wątpliwych miejsc. Z drugiej strony, jeśli program będzie przerywany przy każdym zdarzeniu, wskazywanie co chwila następnego kroku będzie bardzo uciążliwe.
3. Czy interesuje nas jakiś szczególny sposób pokazywania stanu realizacji programu i jego celów? Jest to przydatne, jeśli niektóre cele zawierają ogromne struktury, które są w danej chwili nieistotne i tylko zaciemniają obraz. W takim wypadku warto skorzystać z portray, aby pozbyć się niechcianych informacji.

Poprawianie błędów

Kiedy już sprawdziliśmy, jak działa nasz niepoprawny program i wiemy, gdzie są błędy, będziemy chcieli te błędy poprawić i ponownie przetestować program. Jeśli program nie jest zbyt mały, zwykle przechowywany jest w formie plików na dysku. Do zmiany zawartości tych plików używamy oczywiście edytora, po czym stajemy przed wyborem:

1. Jeśli używana wersja Prologu pozwala korzystać z edytora i potem wrócić do takiego stanu bazy danych, jaki był poprzednio, możemy potrzebne poprawki zrobić bezpośrednio, korzystając z predykatów wbudowanych. Niektóre systemy pozwalają zapisać stan systemu w osobnym pliku i potem odtworzyć go z tego pliku. W takim wypadku zapisujemy stan, opuszczamy Prolog, edytujemy pliki, ponownie uruchamiamy Prolog i odtwarzamy stan systemu. Jeśli wróciliśmy do Prologu po poprawieniu jednego lub kilku plików, wystarczy wywołać reconsult, aby zamienić definicje z odpowiednich plików.
2. Jeśli system nie pozwala wrócić do stanu początkowego, po zmianie plików z kodem źródłowym trzeba ponownie uruchomić Prolog i za pomocą consult ponownie wczytać wszystkie pliki programu.

Cały ten proces można sobie ułatwić poprzez utworzenie pojedynczego pliku zawierającego polecenia Prologu wczytujące potrzebne pliki; wtedy wystarczy wywołać consult dla tego jednego pliku. Jeśli przykładowo nakazemy za pomocą consult wczytać plik o następującej zawartości:

```
?- [plik1,plik2,plik3].
?- [plik4,plik5,plik6].
```

to wczytane zostaną pliki plik1, plik2, plik3, plik4, plik5 i plik6.

Czasami zmiany w programie mogą być tak niewielkie, że najprościej będzie wprowadzić je z klawiatury, za pomocą celów consult(user) lub reconsult(user). Jednak takie rozwiązanie nie powinno stać się normą, gdyż łatwo o tak wprowadzonych zmianach zapomnieć i potem wielokrotnie poprawiać te same błędy, które poprawiane były już wcześniej. Poza tym, jako że zmiany i tak trzeba zwykle wprowadzić do kodu programu w plikach, modyfikowanie programu w interpreterze Prologu jest niecelowe. Nie należy więc ulegać pokusie takiego ułatwiania sobie pracy.

Aby pokazać, jak za pomocą consult można zmienić program bezpośrednio z klawiatury, pokażemy prosty przykład sesji wyjątkowo nieuważnego programisty. Początkowo w bazie danych nie ma żadnych klauzul, więc piszemy z klawiatury...

```
?- append([a,b,c,d].[e],X).
no
?- consult(user).
append([A|B],C,[A|D]) :- append(A,C,D).
append([],X,X).
reverse([],[]).
reverse([A|B],C) :- reverse(B,D), append(D,[A],C).
/* tutaj znak końca pliku */
yes
```

Teraz badamy kilka celów...

```
?- reverse([a,b,c,d,e],X).
no
?- append([a,b,c,d,e],[f],X).
no
?- append([], [a,b,c],X).
X = [a,b,c]
yes
```

Chyba nie działa. Spróbujmy poprawić append.

```
?- consult(user).
append([A|B],C,[A|D]) :- append(B,C,D).
/* tutaj znak końca pliku */
yes
```

Jeszcze raz spróbujmy...

```
?- reverse([a,b,c,d],X).
no
```

Nie działa reverse. Spróbujmy dłuższej definicji...

```
consult(user).
append([],X,X).
/* tutaj znak końca pliku */
yes
```

Spróbujmy jeszcze raz...

```
?- reverse([a,b,c,d,e],X).
X = [e,d,c,b,a]
yes
```

W tej sesji programista zaczyna od wprowadzenia z terminala klauzul predykatów `append` i `reverse`. Oczywiście, można by było wpisać je wcześniej do pliku i za pomocą `consult` nakazać Prologowi je stamtąd odczytać, ale plik ten byłby bardzo mały, więc szkoda zachodu. Niestety, w pierwszej klauzuli występuje pomyłka: `append` zamiast `B` zawiera `A`. Błąd ten wychodzi na jaw, kiedy system nie może odpowiedzieć na zapytania zawierające cele `append` i `reverse`. Nasz programista jakoś sobie o tym przypomina (tak naprawdę dojście do tego wniosku wymagałoby zapewne użycia narzędzi opisanych wcześniej w tym rozdziale), więc postanawia zamienić starą definicję na nową za pomocą `consult`. Niestety, tym razem programista zapomina o warunku końcowym, tutaj dla listy pustej i program nadal nie działa. Pierwotna, dwuklauzulowa definicja `append` została zastąpiona definicją jednoklauzulową. Programista znów dostrzega swoją pomyłkę i postanawia dodać brakującą klauzulę za pomocą `consult`. Teraz program działa już poprawnie.

Reasumując, kiedy poprawiamy program, trzeba zachować taką ostrożność, jak przy pisaniu jego pierwszej wersji. Trzeba pamiętać o postępowaniu zgodnie z przyjętymi konwencjami, o tym, które zmienne powinny być kiedy ukonkretnione i które argumenty do czego służą. Przy okazji warto jeszcze raz przyjrzeć się programowi: mogą być w nim jeszcze inne błędy, dotąd niedostrzeżone!

Rozdział 9.

Użycie reguł gramatycznych w Prologu

Parsowanie

Zdania zapisane w językach naturalnych nie są ciągami przypadkowych słów, gdyż wybierając zestaw słów z jakiegoś zbioru nie otrzymamy zwykle prawidłowego zdania. Wymaganiem minimalnym jest zgodność z regułami gramatyki danego języka.¹

Gramatyka języka to zestaw reguł wskazujących, jakie ciągi słów uważane są za prawidłowe zdania w danym języku. Gramatyka mówi, jak słowa powinny być pogrupowane w zwiazki i jaka może być kolejność tych zwiazków (fraz). Jeśli mamy gramatykę danego języka, możemy przeanalizować ciąg słów tego języka i sprawdzając, czy ciąg ten jest zgodny z gramatyką, powiedzieć, czy jest to prawidłowe zdanie. Jeśli zdanie jest prawidłowe, proces sprawdzania tego faktu pozwala pogrupować słowa, czyli określić strukturę zdania.

Szczególnie prostym rodzajem gramatyki jest gramatyka *bezkontekstowa*. Nie będziemy podawać tutaj ścisłej definicji, ale po prostu pokażemy prosty przykład. Oto początek gramatyki zdań języka angielskiego:

```
zdanie --> fraza_rzecz, fraza_czas
faza_rzecz --> przedimek, rzeczownik.
faza_czas --> czasownik, fraza_rzecz.
faza_czas --> czasownik.
przedimek --> [the].
rzeczownik --> [apple].
```

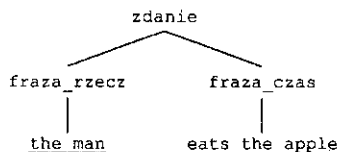
¹ W tym rozdziale analizować będziemy zdania zapisane w języku angielskim. Język angielski jest praktycznie nieodmienny, wobec czego jego analiza jest znacznie prostsza niż na przykład języka polskiego. Analiza zdań polskich z uwzględnieniem deklinacji i kontygacji wykracza poza zakres tej książki. Czytelnicy zainteresowani analizą zdań zapisanych w języku polskim powinni sięgnąć do literatury przedmiotu — *przyp. tłum.*

```

rzeczownik --> [man].
czasownik --> [eats].
czasownik --> [sings].

```

Gramatyka to zbiór reguł, tutaj jedna w każdym wierszu. Każda reguła wskazuje dopuszczalną postać pewnej frazy. Pierwsza reguła mówi, że zdanie składa się z frazy fraza_rzecz i znajdującej się za nią frazy fraza_czas. Te dwie frazy są podmiotem i orzeczeniem zdania, zaś ich układ przedstawiono w formie poniższego drzewka:



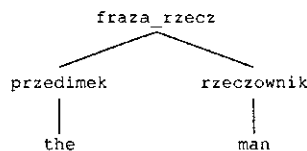
Jeśli w gramatyce bezkontekstowej mamy regułę „X --> Y”, oznacza to, że „X może mieć postać Y”, zaś zapis „X.Y” oznacza „X, a za nim Y”. Tak więc pierwszą regułę odczytujemy:

Zdanie może mieć postać: fraza_rzecz, a za nią fraza_czas.

Wszystko dobrze, ale co to jest fraza_rzecz i fraza_czas? Skąd mamy wiedzieć, jak one wyglądają i z czego się składają? Na te pytania odpowiadają reguły druga, trzecia i czwarta. Na przykład

fraz_a_rzecz może mieć postać: przedimek, a za nim rzeczownik.

Fraza rzeczownikowa to grupa słów określająca nazwę rzeczy. Fraza taka zawiera jedno słowo — „rzeczownik” — które daje nazwę całej klasie słów. „the man” to nazwa człowieka, „the program” to nazwa programu i tak dalej. Poza tym, zgodnie z pokazaną gramatyką, rzeczownik poprzedzony jest frazą „przedimek”:



Analogicznie, strukturę wewnętrzną frazy fraza_czas opisują reguły. Zauważmy, że istnieją dwie reguły opisujące tę frazę — wynika to stąd, że (zgodnie z używaną gramatyką) fraza czasownikowa może mieć dwie postacie. Może ona zawierać frazę_czas, jak w wyrażeniu „the man eats the apple”, lub nie, jak w „the man sings”.

Do czego służą pozostałe reguły gramatyki? Opisują one, jak poszczególne frazy mogą składać się z konkretnych słów, a nie z innych fraz. To, co znajduje się w nawiasach kwadratowych, to słowa opisywanego języka, więc regułę

```

przedimek --> [the].

```

należy odczytać jako

Przedimek może mieć postać: słowa the.

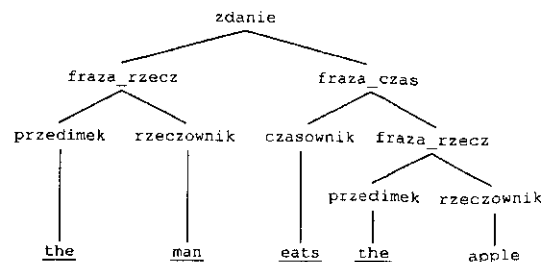
Teraz, kiedy przebrnęliśmy już przez całą gramatykę, możemy zacząć sprawdzać, jak słowa składają się w niej na zdania. Gramatyka ta jest bardzo prosta i konieczne jest znaczne jej rozszerzenie, szczególnie z tego powodu, że dopuszcza ona zdania składające się z ledwie pięciu różnych słów. Jeśli chcemy sprawdzić, czy dany ciąg słów jest zdaniem spełniającym podane warunki, musimy zastosować pierwszą regułę i nasz problem upraszcza się do następującego:

Czy zdanie składa się z dwóch fraz, z których pierwsza spełnia warunki nałożone na fraza_rzecz, a druga na fraza_czas?

Następnie, aby sprawdzić, czy pierwsza fraza jest frazą rzeczownikową, musimy zastosować drugą regułę pytając:

Czy fraza ta składa się z przedimek i dalej rzeczownik?

I tak dalej. Na koniec, jeśli uda nam się znaleźć wszystkie potrzebne frazy i subfrazy zdania zgodnie z gramatyką, otrzymamy strukturę w postaci:



Pokazany diagram zawierający strukturę fraz zdania nazywamy *drzewem parsowania* zdania.

Widzieliśmy, jak gramatyka języka pozwala tworzyć drzewa parsowania pokazujące strukturę zdań danego języka. Problem tworzenia drzewa parsowania zdania przy danej gramatyce nazywamy *problemem parsowania*. Program tworzący takie drzewo nazywamy *parserem*.

W tym rozdziale pokażemy, jak można zapisać problem parsowania w Prologu oraz pokażemy formalizm reguł gramatycznych Prologu, który ułatwia pisanie w tym języku parserów. Wprowadzie parser formalnych reguł gramatycznych (nazywanych skończonymi gramatykami klauzul, DCG) nie jest częścią standardu Prologu, ale jest on włączany do szeregu implementacji. Przydatność DCG nie ogranicza się jedynie do składni języków naturalnych, gdyż te same techniki można zastosować do każdego problemu, w którym mamy uporządkowany ciąg składników zestawianych w grupy, a grupy te można opisać za pomocą zbioru reguł. Jednak w imię prostoty w dalszej części tego rozdziału skoncentrujemy się na problemie parsowania zdań języka angielskiego, zaś uogólnienie tego na inne dziedziny zostawimy czytelnikom.

Problem parsowania w Prologu

Podstawowa struktura, którą posługujemy się omawiając parsowanie jest ciąg słów, którego strukturę należy określić. Chcemy wydzielić z całości podciagi będące różnymi frazami dopuszczonymi przez gramatykę oraz na koniec pokazać, że całość jest poprawną frazą typu zdanie. Z uwagi na to, że ciagi zwykle zapisujemy jako listy, dane wejściowe będziemy przekazywać właśnie w formie listy. Co ze sposobem zapisu poszczególnych słów? W tej chwili wydaje się, że nie ma powodu uwzględniać struktury wewnętrznej słów; chcemy jedynie porównywać jedno słowo z innymi, więc rozsądnym wyborem byłoby użycie po prostu atomów.

Stworzymy teraz program sprawdzający, czy dany ciąg słów jest zdaniem zgodnie z pokazaną powyżej gramatyką. W tym celu musimy ustalić sposób zapisu struktury zdania. Potem zastanowimy się, jak poprawić program, aby zapamiętywał strukturę zdania i ją pokazywał, ale na razie zapomnijmy o tym wymaganiu. Program obejmuje sprawdzanie, czy coś jest zdaniem, więc zdefiniujmy predykat zdanie. Predykat ten będzie miał tylko jeden argument:

zdanie(X) oznacza, że:

X jest ciągiem słów tworzących poprawne gramatycznie zdanie.

Wobec tego oczekujemy, że jeśli zadamy zapytanie

```
?- zdanie([the,man,eats,the,apple]).
```

to uzyskamy odpowiedź twierdzącą tylko wtedy, gdy „the man eats the apple” jest zdaniem zgodnym z przyjętą gramatyką.

Niedogodne jest podawanie zdań w sztucznej, bądź co bądź, formie list atomów. W prawdziwych zastosowaniach konieczne jest interpretowanie normalnych zdań wpisywanych jako tekst. W rozdziale 5. pokazaliśmy, jak można zdefiniować predykat wczytaj, który przekształci tekst w listę atomów Prologu. Moglibyśmy oczywiście już teraz stworzyć nasz parser tak, aby umożliwiał naturalniejsze wprowadzanie danych przez użytkownika. Jednak potraktujmy tego typu zagadnienia jako „kosmetykę” i skoncentrujmy się raczej na rozwiązaniu zasadniczego problemu — parsowania.

Co trzeba zrobić, aby sprawdzić, czy ciąg słów jest zdaniem? No cóż, zgodnie z pierwszą regułą gramatyki, musimy znaleźć na początku zdania frazę rzeczownika (frazę_rzecz), a reszta powinna dać frazę czasownika (frazę_czas). Powinniśmy użyć przy tym wszystkie słowa zdania. Zdefiniujmy zatem predykaty fraza_rzecz i fraza_czas opisujące właściwości fraz rzeczownikowych i czasownikowych:

frazę_rzecz(X) oznacza, że: ciąg X jest frazą rzeczownikową.

Poza tym

frazę_czas(X) oznacza, że: ciąg X jest frazą czasownikową.

Możemy teraz zestawić definicję zdanie z powyższych fraz: ciąg X jest zdaniem, jeśli można go rozłożyć na dwa podciagi Y i Z, gdzie Y spełnia warunki fraza_rzecz, a Z — fraza_czas. Sekwencje są listami, więc już mamy predykat append pozwalający rozkładać listy na części.

Wobec tego możemy zapisać:

```
zdanie(X) :-
    append(Y,Z,X), fraza_rzecz(Y), fraza_czas(Z).
```

Analogicznie

```
frazę_rzecz(X) :-
    append(Y,Z,X), przedimek(Y), rzeczownik(Z).
frazę_czas(X) :-
    append(Y,Z,X), czasownik(Y), fraza_rzecz(Z).
frazę_czas(X) :- czasownik(X).
```

Zwróćmy uwagę na to, że dwóm regułom fraza_czas odpowiadają dwie klauzule predykatu, każda odpowiadająca innej definicji frazy czasownikowej. W końcu, bez problemu zapiszemy reguły opisujące pojedyncze słowa:

```
przedimek([the]).
rzeczownik([apple]).
rzeczownik([man]).
czasownik([eats]).
czasownik([sings]).
```

Nasz program jest już gotowy. Program ten prawidłowo odpowie na pytanie, czy dany ciąg słów zgodnie z przyjętą gramatyką jest zdaniem. Zanim jednak uznamy zadanie za skończone, powinniśmy się przyjrzeć, co się stanie, jeśli zadamy zapytanie o jakieś przykładowe zdanie. Weźmy pod uwagę klauzulę zdanie:

```
zdanie(X) :-
    append(Y,Z,X), fraza_rzecz(Y), fraza_czas(Z).
```

oraz zapytanie:

```
?- zdanie([the,man,eats,the,apple]).
```

Zmienna X z reguły zostanie ukonkretniona listą [the,man,eats,the,apple], ale Y i Z będą nieukonkretnione, wobec czego naszym celem będzie wygenerowanie możliwych par wartości Y i Z, takich, że kiedy je połączymy, otrzymamy z powrotem X. W trakcie nawracania generowane będą wszystkie takie pary. Cel fraza_rzecz zostanie spełniony, jeśli wartość Y jest prawidłową frazą rzeczownikową. W przeciwnym razie cel ten zawiedzie, zaś append będzie musiało wygenerować kolejny możliwy podział listy. Tak więc nasz program będzie się wykonywał początkowo następująco:

1. Celem jest zdanie([the,man,eats,the,apple]).
2. Lista wejściowa jest rozkładana na dwie listy, Y i Z. Dostępne możliwości to:

```
Y = [], Z = [the,man,eats,the,apple]
Y = [the], Z = [man,eats,the,apple]
Y = [the,man], Z = [eats,the,apple]
Y = [the,man,eats], Z = [the,apple]
Y = [the,man,eats,the], Z = [apple]
Y = [the,man,eats,the,apple], Z = []
```

3. Wybieramy kolejne możliwe wartości Y i Z z powyższego zestawu i sprawdzamy, czy Y jest frazą rzeczownikową — czyli próbujemy spełnić cel fraza_rzecz(Y).

4. Jeśli Y spełnia warunki fraza_rzecz, to możemy sprawdzić spełnianie przez Z warunków fraza_czas. Jeśli Y tych warunków nie spełnia, wracamy do poprzedniego kroku i sprawdzamy kolejną parę wartości.

Wydaje się, że przy takim podejściu ma miejsce mnóstwo zbędnego przeszukiwania. Cel `append(Y, Z, X)` generuje szereg rozwiązań, z których większość z punktu widzenia szukania fraz rzeczownikowych jest całkiem nieprzydatna. Musi być jakiś prostszy sposób znalezienia rozwiązania. Zgodnie z naszą gramatyką, fraza_rzecz musi mieć dokładnie dwa słowa, więc można byłoby pominąć znaczną część generowanych przez `append` wariantów rozkładu listy słów. Jednak tego typu właściwości nie zawsze muszą być prawdziwe po zmianie gramatyki. Nawet niewielka zmiana w regułach przedimek może wpłynąć na dopuszczalne długości frazy rzeczownikowej i przez to na sposób sprawdzania, czy lista słów jest taką frazą. Zmiana pojedynczej klauzuli nie powinna powodować takiego zamieszania w całym programie.

Wobec powyższego opisane ustalenia dotyczące długości frazy rzeczownikowej są zbyt szczegółowe, aby je wbudowywać w nasz program. Niemniej jednak, możemy je uważać za szczególny przypadek ogólnej zasady. Jeśli chcemy wybrać podciąg będący frazą rzeczownikową, możemy przejrzeć właściwości takich fraz, aby ograniczyć sprawdzane listy wyrazów. Jeśli definicja frazy rzeczownikowej ma się zmienić, nie możemy opierać się na żadnych szczególnych właściwościach tych fraz, chyba że zapewnimy ich obsługę bezpośrednio w klauzulach fraza_rzecz. Klauzule te opisują właściwości fraz rzeczownikowych, więc czemu nie pozwolić im decydować, jakie listy mają być rozważane? Zažadajmy, aby to fraza_rzecz decydowała, jak duża część listy ma być pobierana i co ma zostać, jako dane wejściowe dla fraza_czas.

Rozważania te sugerują nam stworzenie nowej definicji predykatu fraza_rzecz, tym razem definicji dwuargumentowej:

fraza_rzecz(X, Y) jest prawdą, jeśli
na początku ciągu X znajduje się fraza rzeczownikowa
oraz część ciągu za tą frazą to Y .

Zatem możemy się spodziewać, że nie zawiodą zapytania:

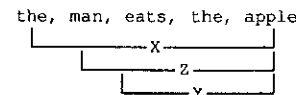
```
?- fraza_rzecz([the,man,saw,the,cat],[saw,the,cat]).
?- fraza_rzecz([the,apple,sings],[sings]).
?- fraza_rzecz([the,man,eats,the,apple],X).
?- fraza_rzecz([the,apple,sings],X).
```

Jednocześnie w dwóch ostatnich przypadkach zmienna X zostanie ukonkretniona listą z fraza_rzecz.

Musimy teraz skorygować definicję fraza_rzecz, aby uwzględnić zmianę założeń. Robiąc to musimy zdecydować, jak rozkładać ciąg związany z frazą rzeczownikową na ciąg zawierający przedimek i rzeczownik. Możemy znów przesunąć problem, ile danych z listy brać na przedimek i ile na rzeczownik, do klauzul wywoływanych już w ramach frazy:

```
fraza_rzecz(X,Y) :- przedimek(X,Z), rzeczownik(Z,Y).
```

Tak więc na początku zdania X mamy frazę rzeczownikową, jeśli na początku X znajduje się przedimek (reszta to Z), a na początku Z jest rzeczownik. Część ciągu znajdująca się za całą frazą rzeczownikową jest częścią znajdującą się za rzeczownikiem. Na diagramie wygląda to tak:



Aby zrealizować nasze zadanie, w odniesieniu do fraz przedimek i rzeczownik zastosujemy taki sam mechanizm, jak do frazy rzeczownikowej, fraza_rzecz.

Klauzula ta informuje nas, jak problem znalezienia ciągu odpowiadającego frazie rzeczownikowej rozkłada się na znalezienie podciągów odpowiadających przedimkowi i dalej rzeczownikowi. Analogicznie, problem znajdowania zdania rozkładamy na znalezienie frazy rzeczownikowej i za nią frazy czasownikowej. Wszystko to jest bardzo abstrakcyjne; nigdzie nie powiedziano, ile słów zużywa przedimek, fraza rzeczownikowa czy zdanie. Informacje te muszą być tworzone na podstawie użytej wersji reguł definiujących angielskie słowa. Możemy reguły te zapisać jako klauzule Prologu, ale tym razem musimy dodać jeszcze argument, na przykład

```
przedimek([the|X],X).
```

Reguła ta wyraża to, że przedimek występuje na początku zdania zaczynającego się słowem the. Co więcej, przedimek powoduje wzięcie pierwszego słowa zdania, reszta zostaje.

Tak naprawdę nowy argument możemy dodać do każdego predykatu, który rozpoznaje jakąś frazę, aby poinformować, że dany rodzaj frazy „zużywa” określoną liczbę słów, zostawiając resztę. W szczególności, aby zapewnić spójność rozwiązania, dobrze byłoby analogicznie postąpić w przypadku predykatu zdanie. Jak zatem będzie wyglądał początkowy cel, który zadajemy programowi? Musimy określić dwa argumenty przekazywane do zdanie. Argumenty zawierają początkowy ciąg i ciąg, który zostanie po wydzieleniu zdania. Pierwszy z tych argumentów jest taki sam, jak poprzednio. Co więcej, ponieważ dążymy do tego, aby zdanie zajmowało cały przekazany ciąg, nie chcemy, aby poza zdaniem cokolwiek zostało. Wobec tego odpowiedni będzie cel:

```
?- zdanie([the,man,eats,the,apple],[]).
```

Przyjrzyjmy się teraz postaci naszej gramatyki po przepisaniu jej zgodnie z powyższymi wytycznymi:

```
zdanie(S0,S) :-
    fraza_rzecz(S0,S1),
    fraza_czas(S1,S).
fraza_rzecz(S0,S) :- przedimek(S0,S1), rzeczownik(S1,S).
fraza_czas(S0,S) :- czasownik(S0,S).
fraza_czas(S0,S) :- czasownik(S0,S1), fraza_rzecz(S1,S).
przedimek([the|S],S).
rzeczownik([man|S],S).
```



```

rzeczownik([apple|S],S).
czasownik([eats|S],S).
czasownik([sings|S],S).

```

Tak więc mamy już sprawniej działającą wersję naszego programu rozpoznającego zdania zgodne z pewną gramatyką. Niestety, kod wygląda gorzej niż w wersji poprzedniej. Wszystkie dodatkowe argumenty wydają się być tu całkiem zbędne — i teraz tym się zajmiemy.

Notacja reguł gramatyki

Zapis w formie reguł gramatyki stworzono jako pomoc dla osób tworzących parsery za pomocą opisanych wyżej technik. Zapis taki jest czytelniejszy, a to z uwagi na pominięcie wszystkich nieistotnych informacji oraz jest bardziej zwarty od zwykłego Prologu, więc mniejsze są szanse na popełnienie pomyłki.

Wprawdzie zapis reguł gramatyki jest samodzielnym narzędziem, jednak trzeba zdawać sobie sprawę z faktu, że jest to jedynie skrócona forma zapisu kodu prologowego. Reguł gramatyki można użyć, gdyż są one wbudowane już w Prolog lub mają postać dodatkowej biblioteki (jak Appendix ***), którą trzeba odpowiednio wczytać (consult). Tak czy inaczej, sposób obsługi reguł gramatyki polega na rozpoznawaniu ich w danych wejściowych i następnie przekształcaniu ich na zwykły kod prologowy. Skoro więc nasze reguły gramatyczne ostatecznie otrzymują postać zwykłego kodu Prologu, praca systemu polega potraktowaniu tych reguł jako punktu wyjścia do przekształcania na zwykły Prolog, niezależnie od niecodziennej postaci początkowej.

Używana praktycznie notacja oparta jest na notacji gramatyk bezkontekstowych. Tak naprawdę, jeśli prześlemy Prologowi poniższe reguły gramatyki, otrzymamy przetłumaczenie klauzul zgodnie z ostateczną wersją programu zaprezentowaną wcześniej:

```

zdanie --> fraza_rzecz, fraza_czas.
faza_rzecz --> przedimek, rzeczownik.
faza_czas --> czasownik, fraza_rzecz.
faza_czas --> czasownik.
przedimek --> [the].
rzeczownik --> [apple].
rzeczownik --> [man].
czasownik --> [eats].
czasownik --> [sings].

```

Reguły gramatyczne są strukturami Prologu, których główny funktor to --> zadeklarowany jako operator infiksowy. Wszystko, co musi zrobić Prolog, to sprawdzenie, czy wczytany term (wczytany za pomocą consult lub podobnie) ma taki właśnie funktor i jeśli tak, to przekształcenie tego termu na odpowiednią klauzulę.

Co się z tym przekształceniem wiąże? Przede wszystkim wszystkie atomy będące nazwami fraz muszą być przekształcone na dwuargumentowe predykaty. Jeden z tych argumentów to przetwarzany ciąg, drugi to część tego ciągu pozostała po oderwaniu danej części. Następnie, kiedy według reguł gramatyki znajdują się kolejne frazy, muszą

one zostać tak ułożone, aby było jasne, że część pozostała po pobraniu pierwszej frazy będzie danymi wejściowymi frazy następnej. Na koniec, kiedy z reguły gramatyki wynika, że frazę można zapisać jako ciąg podfraz, argumenty muszą uwzględniać to, że liczba słów zużytych łącznie przez wszystkie te podfrazy musi być równa liczbie słów zużytych przez całą frazę z lewej strony operatora -->. Przy spełnieniu tych wszystkich warunków, przykładowo

```
zdanie --> fraza_rzecz, fraza_czas.
```

przekształcane jest na:

```

zdanie(S0,S) :-
    fraza_rzecz(S0,S1), fraza_czas(S1,S).

```

czyli, po polsku:

Między S0 a S mamy zdanie, jeśli między S0 a S1 występuje fraza rzeczownikowa, a między S1 a S — fraza czasownikowa.

Na koniec, system musi umieć przekształcić reguły opisujące poszczególne słowa elementarne. Oznacza to konieczność wstawienia słów na listy tworzące argumenty predykatów, na przykład

```
przedimek --> [the].
```

przekształcane jest na

```
przedimek([the|S],S).
```

Kiedy nasz program parsujący zapisaliśmy w formie reguł gramatyki, jak wskazywać cele, które chcemy wykonać? Wiemy już, jak reguły gramatyczne przekształcane są na zwykły program prologowy, więc i cele możemy wyrażać w Prologu, podając sami dodatkowe argumenty. Pierwszy argument to lista słów, która ma być sprawdzana, drugi to lista słów pozostających po parsowaniu; zwykle powinna to być lista pusta, []. Zatem cele podajemy w postaci:

```

?- zdanie([the,man,eats,the,apple],[]).
?- fraza_rzecz([the,man,sings],X).

```

Poza tym niektóre implementacje Prologu zawierają predykat wbudowany phrase, który po prostu dodaje brakujące argumenty; predykat ten zdefiniowany jest następująco:

phrase(P,L) jest spełnione, jeśli listę L można sparsować jako zdanie typu P.

Moglibyśmy więc zastąpić pierwszy z powyższych celów zapytaniem:

```
?- phrase(zdanie,[the,man,eats,the,apple]).
```

Zwróćmy uwagę na to, że w definicji phrase zakłada się parsowanie całej listy, kiedy pozostaje lista pusta. Wobec tego drugiego celu z pokazanych powyżej nie można zastąpić przytoczonym phrase.

Jeśli używana implementacja Prologu nie zawiera predykatu phrase, łatwo ten predykat zdefiniować:

```
phrase(P,L) :- Cel = .. [P,L,[]], call(Cel).
```

Zwróćmy jednak uwagę na to, że taka definicja nie będzie odpowiednia przy rozważaniu ogólniejszych reguł gramatyki, którymi zajmiemy się w następnym podrozdziale.

Dodatkowe argumenty

Omawiane dotąd reguły gramatyczne są dość ograniczone. W tym podrozdziale zajmujemy się przydatnym ich rozszerzeniem, które pozwoli podawać we frazach dodatkowe argumenty. To „rozszerzenie” też jest częścią standardowy reguł gramatycznych Prologu.

Wdzieliśmy, jak wystąpienie w regule frazy jest przekształcane na wystąpienie predykatu z dwoma dodatkowymi argumentami; dlatego pokazanym dotąd regułom odpowiadają predykaty dwuargumentowe. Predykaty Prologu mogą mieć dowolną liczbę argumentów, czasami w naszych parserach potrzebne są nam dodatkowe argumenty, niezależnie od argumentów związanych z użyciem wejściowych ciągów słów. Takie dodatkowe argumenty są uwzględniane w prologowym zapisie reguł gramatyki.

Przyjrzyjmy się przykładowi, w którym będą przydatne dodatkowe argumenty. Zastanówmy się nad zgodnością liczby podmiotu i orzeczenia w zdaniu. Angielskie zdania typu

★ The boys eats the apple.

★ The boy eat the apple.²

nie są poprawne gramatycznie, choć niewielka poprawka przedstawionej gramatyki wystarczyłaby, aby je zapisywać poprawnie (używamy gwiazdki, aby oznaczyć zdanie niezgodne ze stosowaną gramatyką). Zdania te nie są zgodne z gramatyką, gdyż jeśli podmiot jest w liczbie pojedynczej, orzeczenie też musi być w liczbie pojedynczej, i analogicznie w przypadku liczby mnogiej. Moglibyśmy zapisać to w formie reguł gramatycznych opisujących zdania w liczbie pojedynczej i w liczbie mnogiej. Zdania w liczbie pojedynczej muszą zaczynać się frazą rzeczownikową w liczbie pojedynczej i muszą zawierać czasownik również w liczbie pojedynczej, i tak dalej. Na koniec otrzymalibyśmy następujący zestaw reguł:

```
zdanie --> zdanie_pojedyncze.
zdanie --> zdanie_mnogie.
fraza_rzecz --> fraza_rzecz_pojedyncza.
fraza_rzecz --> fraza_rzecz_mnoga.
zdanie_pojedyncze -->
    fraza_rzecz_pojedyncza, fraza_czas_pojedyncza,
    fraza_rzecz_pojedyncza -->
        przedimek_pojedynczy, rzeczownik_pojedynczy.
fraza_czas_pojedyncza --> czasownik_pojedynczy, fraza_rzecz.
fraza_czas_pojedyncza --> czasownik_pojedynczy.
przedimek_pojedynczy --> [the].
rzeczownik_pojedynczy --> [boy].
czasownik_pojedynczy --> [eats].
```

² Najbliższymi ich odpowiednikami w języku polskim są „Chłopcy je jabłko” i „Chłopiec jedzą jabłko” — przyp. tłum.

Dalej byłyby analogiczne reguły fraz liczby mnogiej. Nie jest to rozwiązanie zbyt eleganckie, poza tym nie znajduje w nim odbicia fakt, że zdania w liczbie pojedynczej i mnogiej są bardzo podobne do siebie. Lepszym rozwiązaniem byłoby powiązanie z typem frazy dodatkowego argumentu, który wskazywałby, czy jest to liczba pojedyncza, czy mnoga. Tak więc zdanie(X) oznacza zdanie w liczbie X. Reguły dotyczące zgodności liczby zapisywane są przez użycie odpowiednich wartości dodatkowego argumentu. Liczba frazy rzeczownika stanowiącego podmiot zdania musi być taka sama, jak liczba frazy czasownikowej i tak dalej. Zmieniwszy odpowiednio gramatykę otrzymamy:

```
zdanie --> zdanie(X).
zdanie(X) --> fraza_rzecz(X), fraza_czas(X).
fraza_rzecz(X) --> przedimek(X), rzeczownik(X).
fraza_czas(X) --> czasownik(X), fraza_rzecz(Y).
fraza_czas(X) --> czasownik(X).
rzeczownik(pojedynczy) --> [boy].
rzeczownik(mnogi) --> [boys].
przedimek(_) --> [the].
czasownik(pojedynczy) --> [eats].
czasownik(mnogi) --> [eat].
```

Zwróćmy uwagę na sposób zapisania liczby słówka the. Słowo to może otwierać zarówno frazę w liczbie pojedynczej, jak i mnogiej, stąd nie musi być zachowana jego zgodność co do liczby. Poza tym druga reguła fraza_czas zawiera dwie zmienne po prawej — świadectwo tego, że liczba frazy czasownikowej pochodzi od czasownika, zaś ewentualne dopełnienie może być w dowolnej liczbie.

Argumentów możemy użyć do zapisania innych informacji, równie ważnych, jak zgodność co do liczby. Przykładowo, możemy tak zapisywać składniki, które znajdują się poza ich „normalnym” położeniem; taką cechę gramatyki nazywamy „przeniesieniem”. W argumentach możemy też zapisywać rzeczy istotne semantycznie, na przykład jak znaczenie frazy budowane jest ze znaczeń podfraz. Nie będziemy się tym zagadnieniem zajmować tutaj dokładnie, choć w podsumowaniu pokażemy prosty przykład włączenia do parsera semantyki. O jednym jednak trzeba wspomnieć: dla lingwistów interesujące może być to, że kiedy dodajemy do reguł gramatyki argumenty, nie można zagwarantować, że język tak zdefiniowany nadal będzie niezależny od kontekstu (choć często jest).

Ważnym zastosowaniem dodatkowych argumentów jest zwracanie drzewa parsowania będącego wynikiem analizy. W rozdziale 3. pokazywaliśmy, jak drzewa można zapisywać w Prologu. Teraz z tego skorzystamy rozszerzając parser tak, aby potrafił zwrócić drzewo parsowania. Drzewa te są przydatne, gdyż zawierają strukturalny opis zdania. Jest to wygodne przy pisaniu programów przetwarzających takie struktury podobnie, jak w rozdziale 7. przekształcaliśmy struktury wyrażeń algebraicznych i listy. Nowy program, kiedy otrzyma poprawne zdanie typu

The man eats the apple.

wygeneruje strukturę:

```
zdanie(
    fraza_rzecz(
        przedimek(the),
```

```

    rzeczownik(man)),
    fraza_czas(
        czasownik(eats),
        fraza_rzecz(
            przedimek(the),
            rzeczownik(apple))
        )
    )

```

Aby uzyskać taki wynik, wystarczy dodać do wszystkich predykatów jeden argument, który przekaże drzewo poszczególnych fraz do drzewa całego zdania. Tak więc pierwszą regułę możemy przekształcić na:

```

zdanie(X, zdanie(FR, FC)) -->
    fraza_rzecz(X, FR), fraza_czas(X, FC).

```

Ten zapis mówi, że jeśli znajdziemy ciąg składający się na frazę rzeczownikową o drzewie FR, dalej ciąg składający się na frazę czasownikową o drzewie FC, to cały ciąg jest zdaniem o drzewie zdanie(FR, FC). Z proceduralnego punktu widzenia, aby sparsować zdanie, trzeba znaleźć frazę rzeczownikową, dalej frazę czasownikową i połączyć ich drzewa funktorem zdanie.

Kwestią przypadku jest to, że użyliśmy reguły zdanie i jednocześnie węzła drzewa zdanie. Jako węzła drzewa moglibyśmy użyć na przykład S. Argumenty X to po prostu argumenty związane ze zgodnością liczby, które omawialiśmy wcześniej, zaś umieszczenie argumentu na drzewo na końcu jest kwestią przyjętej konwencji. Jeśli zrozumienie opisanego rozszerzenia jest trudne, warto zwrócić uwagę, że jest to po prostu skrótowy zapis zwykłej klauzuli Prologu:

```

zdanie(X, zdanie(FR, FC), S0, S) :-
    fraza_rzecz(X, FR, S0, S1),
    fraza_czas(X, FC, S1, S).

```

gdzie S0, S1 i S to części ciągu wejściowego. Argumenty na tworzenie drzewa można wprowadzić do gramatyki tak jak zwykle. Oto wyjątek z tego, co uzyskamy, jeśli zastosujemy opisanie rozszerzenie (aby zwiększyć czytelność przykładu, pominiemy argumenty związane ze zgodnością liczby):

```

zdanie(X, zdanie(FR, FC)) -->
    fraza_rzecz(X, FR), fraza_czas(X, FC),
    fraza_czas(fraza_czas(C)) --> czasownik(C),
    rzeczownik(rzeczownik(man)) --> [man],
    czasownik(czasownik(eats)) --> [eats].

```

Mechanizm przekształcający reguły gramatyki z takimi dodatkowymi argumentami na klauzule jest prostym rozszerzeniem mechanizmu opisanego wcześniej. Poprzednio dla każdego rodzaju frazy korzystaliśmy z nowego predykatu, który miał dwa argumenty wskazujące sposób zużycia ciągu wejściowego. Teraz musimy tworzyć predykaty z dwoma dodatkowymi argumentami, które są używane w regułach. Zwykle te dodatkowe argumenty dodaje się na koniec (choć różne implementacje Prologu mogą się pod tym względem różnić). Tak więc reguła:

```

zdanie(X) --> fraza_rzecz(X), fraza_czas(X)

```

zostanie przekształcona na klauzulę:

```

zdanie(X, S0, S) -->
    fraza_rzecz(X, S0, S1), fraza_czas(X, S1, S).

```

Kiedy chcemy wywołać cele zawierające reguły gramatyczne bezpośrednio z interpretera lub ze zwykłych reguł Prologu, musimy te dodatkowe argumenty podać jawnie. Wobec tego prawidłowymi celami korzystającymi z definicji zdanie będą:

```

?- zdanie(X, [a, c, l, e, r, g, y, m, a, n, e, a, t, s, a, c, a, k, e], []).
?- zdanie(X, [e, v, e, r, y, b, i, r, d, s, i, n, g, s, a, n, d, p, i, g, s, c, a, n, f, l, y], []).

```

Ćwiczenie 9.1. To ćwiczenie może okazać się trudne. Zdefiniuj predykat przekształc, taki, aby przekształc(X, Y) nie zawodziło, jeśli X jest regułą gramatyczną (taką jakie pokazywaliśmy wyżej), a Y jest termem opisującym klauzulę odpowiadającą tej regule.

Ćwiczenie 9.2. Napisz nową wersję predykatu phrase, taką, która pozwoli na użycie reguł gramatycznych zawierających dodatkowe argumenty, aby możliwe było użycie celów w postaci:

```

?- phrase(zdanie(X), [the, man, sings]).

```

Dodatkowe warunki

Jak dotąd jedynym, o co nasz parser dbał za pośrednictwem reguł, był sposób korzystania z ciągu wejściowego. Każdy element reguły miał dwa dodatkowe argumenty dodawane podczas przekształcania tych reguł, więc wszystkie stworzone w ten sposób klauzule uwzględniały zużywanie ciągu wejściowego. Czasami chcemy wskazać Prologowi cele w innej postaci i nie broni nam tego formalizm reguł gramatycznych. Wszystkie cele ujęte w nawiasy klamrowe ({}), są podczas przekształcania zostawiane bez zmian.

Przyjrzyjmy się teraz kilku przykładom, w których możemy skorzystać z opisanej właściwości. Poprawimy „słownik” parsera, czyli jego zdolność rozumienia słów. Najpierw zastanówmy się, na ile skomplikowane jest wprowadzanie do programu nowych słów wraz z dodatkowymi argumentami. Jeśli chcemy na przykład dodać nowy rzeczownik, banana, musimy dodać przynajmniej taką regułę:

```

rzeczownik(pojedyncza, rzeczownik(banana)) --> [banana].

```

której odpowiada klauzula

```

rzeczownik(pojedyncza, rzeczownik(banana), [banana[S], S]).

```

Dla pojedynczego rzeczownika trzeba wprowadzić mnóstwo informacji, zwłaszcza że wiadomo, iż każdy rzeczownik będzie reprezentowany na liście wejściowej jako pojedyncze słowo i w drzewie pojawi się z funktorem rzeczownik. Znacznie bardziej ekonomicznym rozwiązaniem byłoby zapisanie wszystkich niezmiennych informacji o rzeczownikach w jednym miejscu i o poszczególnych rzeczownikach gdzie indziej.

Możemy to osiągnąć łącząc reguły gramatyki ze zwykłymi klauzulami Prologu. Informacje ogólne o miejscu rzeczowników zapiszemy w formie reguł gramatyki, zaś informacje o konkretnych słowach będących rzeczownikami w zwykłych klauzulach. Rozwiązanie, które otrzymamy, będzie miało następującą postać:

```
rzeczownik(S,rzeczownik(R)) --> [R]. {is_rzeczownik(R,S)}.
```

gdzie zwykły predykat `is_rzeczownik` wskazuje, jakie słowa są rzeczownikami i czy są w liczbie pojedynczej, czy mnogiej:

```
is_rzeczownik(banana,pojedyncza).
is_rzeczownik(bananas,mnoga).
is_rzeczownik(man,pojedyncza).
```

Przyjrzyjmy się dokładnie znaczeniu takiej reguły gramatyki. Mówi ona, że fraza typu rzeczownik może mieć postać pojedynczego słowa `R` (zmienna z listy) z pewnym ograniczeniem. Ograniczenie to polega na tym, że `R` musi być w zbiorze `is_rzeczownik` i mieć liczbę `S`. W tym wypadku liczbą frazy także jest `S`, zaś w drzewie parsowania pojawia się po prostu słowo `R` jako jedyny argument węzła rzeczownik. Czemu `is_rzeczownik(R,S)` trzeba wstawiać w nawias klamrowy? Gdyż term ten opisuje relację nie mającą nic wspólnego z ciągiem wejściowym. Gdybyśmy te nawiasy pominęli, otrzymalibyśmy cel typu `is_rzeczownik(R,S.S1.S2)`, który nie pasowałby do żadnej klauzuli `is_rzeczownik`. Umieszczenie tego termu w nawiasie powoduje, że nie ingeruje mechanizm przekształcający, dzięki czemu ostatecznie cała reguła będzie miała postać:

```
rzeczownik(S,rzeczownik(R),[R|Ciag],Ciag) :- is_rzeczownik(R,S).
```

Mimo tych zmian sposób zapisu pojedynczych słów nadal jest daleki od ideału. Problem w tym, że konieczne będzie zapisanie dwóch klauzul `is_rzeczownik` dla każdego rzeczownika: jednej dla liczby pojedynczej, drugiej dla liczby mnogiej. Jest to zbędne, gdyż większość rzeczowników w języku angielskim tworzy formę liczby mnogiej w sposób regularny:

Jeśli `X` jest formą liczby pojedynczej rzeczownika, słowo stworzone przez dodanie na koniec litery `s` jest formą liczby mnogiej rzeczownika `X`.

Tej reguły opisującej formy rzeczowników możemy użyć do poprawienia definicji rzeczownik. Poprawki te dadzą nam nowy zestaw warunków, które słowo `N` musi spełnić, abyśmy uznali je za rzeczownik. Jako że warunki te związane są ze strukturą wewnętrzną tego słowa i nie mają nic wspólnego z korzystaniem z ciągu wejściowego, ujmijmy je w nawias klamrowy. Słowa angielskie zapisujemy jako atomy Prologu, więc musimy poświęcić nieco uwagi kwestii rozkładania słów na znaki. W naszej definicji konieczne będzie użycie predykatu `name`; poprawiona reguła przybierze postać:

```
rzeczownik(mnoga,rzeczownik(FormaPodst)) -->
[N].
{(atom_chars(N,NazwaMn),
 append(NazwaPod,[s],NazwaMn),
 atom_chars(FormaPodst,NazwaPod)),
 is_rzeczownik(FormaPodst,pojedyncza)} .
```

Oczywiście, taka ogólna reguła dotycząca liczby mnogiej rzeczowników nie zawsze jest prawdziwa, na przykład liczba mnoga rzeczownika „fly” nie brzmi „flys”. Zawsze trzeba będzie uwzględnić wyjątki.³ Teraz wystarczy podać klauzule `is_rzeczownik` form liczby pojedynczej rzeczowników regularnych. Zwróćmy uwagę na to, że zgodnie z powyższą definicją pozycje wstawiane do drzewa parsowania będą formami podstawowymi rzeczowników, a nie formami użytymi w zdaniu. Może to być przydatne przy dalszym przetwarzaniu drzewa. Zwróćmy też uwagę na sposób użycia nawiasów klamrowych. W nawiasach klamrowych można podać dowolny cel lub ciąg celów Prologu, które mają wystąpić w treści klauzuli.

Niezależnie od nawiasów klamrowych, większość programów przekształcających prologowe reguły gramatyczne „wie”, że pewnych celów nie należy przekształcać w zwykły sposób. Dzięki temu nie trzeba ujmować w nawias wykrzyknika (!) czyli odcięcia ani średnika (;) oznaczającego alternatywę.

Podsumowanie

Podsumujmy teraz, co wiemy o składni reguł gramatyki, a potem pokażemy pewne możliwe rozszerzenia i ciekawe zastosowania gramatyki. Najlepszym sposobem opisanie składni reguł gramatyki jest użycie tych reguł do opisanie ich samych. Oto ich nieformalna definicja; nie jest ona całkiem ścisła, gdyż pominięto wpływ priorytetów operatorów na składnię.

```
gramatyka_regula --> gramatyka_glowa, ['-->'], gramatyka_tresc.
gramatyka_glowa --> nie_koncowe.
gramatyka_glowa --> nie_koncowe, [''], koncowe.
gramatyka_tresc --> gramatyka_tresc, [''], gramatyka_tresc.
gramatyka_tresc --> gramatyka_tresc, [''], gramatyka_tresc.
gramatyka_tresc --> gramatyka_pozycja_tresci.
gramatyka_pozycja_tresci --> [''].
gramatyka_pozycja_tresci --> [''], cele_prologu, [''].
gramatyka_pozycja_tresci --> nie_koncowe.
gramatyka_pozycja_tresci --> koncowe.
```

Taka definicja wymaga objaśnienia. Są to definicje zapisane w języku polskim. `nie_konc` oznacza frazę, która może być częścią ciągu wejściowego. Ma ona postać struktury Prologu, w której funktor nazywa rodzaj frazy, zaś argumenty zawierają informacje dodatkowe, np. liczbę, znaczenie i tak dalej. `koniec` oznacza liczbę słów, które mogą być częścią ciągu wejściowego. W zapisie używana jest lista (może to być [] lub dowolna lista o ustalonej długości). Pozycje listy są elementami Prologu, które pasują do słów w ustalonej ich kolejności. `cele_prologu` to cele zapisane w Prologu, które mogą być stosowane do zapisu dodatkowych warunków i ograniczeń oraz mogą wskazywać sposób podziału wyników złożonych na prostsze.

³ W niektórych implementacjach dostępne są wersje `atom_chars`, dające na podstawie atomu listę kodów znaków, a nie same znaki. W takim wypadku cel `append` trzeba poprawić, aby podawał listę zawierającą kod s jako drugi argument. W niektórych implementacjach z kolei lista może być podana jako s w podwójnych cudzysłowach.

Kiedy przekształcamy reguły gramatyczne na klauzule, cele prologu pozostają niezmienione, zaś nie-koncowe za argumentami pojawiającymi się jawnie mają dwa argumenty dodatkowe, odpowiadające ciągowi wejściowemu oraz części tego ciągu zza opisywanej frazy. Koncowe występują w dodatkowych argumentach nie_koncowe. Kiedy predykat opisany regułą gramatyczną jest wywoływany bezpośrednio z interpretera lub ze zwykłej reguły Prologu, owe dodatkowe argumenty muszą zostać podane jawnie.

Druga reguła gramatyka_glowa zawiera pewien rodzaj reguły gramatycznej, z którą jeszcze się nie zetknęliśmy. Aż do teraz nasze elementy końcowe i niekończące były definiowane z punktu widzenia części ciągu wejściowego, którego potrzebowały. Czasami przydatne byłoby zdefiniowanie pewnych elementów tak, aby do ciągu wejściowego wstawiane były nowe pozycje (przetwarzane za pomocą innych reguł). Możemy na przykład analizować następujące zdanie rozkazujące:

Eat your supper.⁴

tak, jakby uzupełniono je słowem you:

You eat your supper.⁵

Wtedy mielibyśmy pełną strukturę z frazą rzeczownikową i frazą czasownikową, zgodną z naszą dotychczasową wiedzą o zdaniach. W tym celu możemy użyć następującej gramatyki (tu jej część):

```
zdanie --> rozkaznik, fraza_rzecz, fraza_czas.
rozkaznik, [you] --> [].
rozkaznik --> [].
```

Warto tu wspomnieć tylko o jednej regule — pierwszej, która jest przekształcana na:

```
rozkaznik(L, [you|L]).
```

Mamy więc przypadek, w którym zwracany ciąg jest dłuższy od ciągu pierwotnego. Ogólnie rzecz biorąc, lewa strona reguły gramatyki może składać się z ciągu pozycji niekończących oddzielanych od listy słów przecinkiem. Znaczenie takiej konstrukcji polega na tym, że podczas parsowania słowa są wstawiane do ciągu wejściowego po tym, jak cele z prawej strony mogą ewentualnie pobrać słowa na swoje potrzeby.

Ćwiczenie 9.3. Definicja podanych reguł gramatycznych, nawet jeśli zostanie uzupełniona, nie opisuje jeszcze praktycznie przydatnego parsera przetwarzającego ciąg pozycji wejściowych. Dlaczego?

Przekształcanie języka na logikę

Aby pokazać, jak można użyć DCG do bardziej złożonej analizy języka, zaprezentujemy przykład (pochodzący z artykułu Pereiry i Warrena z pisma „Artificial Intelligence” numer 13.) reguł gramatycznych używanych do bezpośredniego pozyskiwania

⁴ „Zjedz kolację” — *przyp. tłum.*

⁵ „Ty zjedz kolację” — *przyp. tłum.*

znaczenia zdań, bez konieczności korzystania z pośrednictwa drzewa parsowania. Poniższe reguły przekształcają (ograniczone) zdania angielskie na ich zapis w formie rachunku predykatów. Więcej o rachunku predykatów powiemy w rozdziale 10. Aby pokazać ten program w działaniu, powiedzmy, że znaczeniem zdania „every man loves a woman” jest struktura

```
all(X, (man(X) -> exists(Y, (woman(Y) & loves(X, Y)))))6
```

Oto obiecane reguły gramatyki:

```
?- op(500, xfy, &).
?- op(600, xfy, ->).
zdanie(P) -->
    fraza_rzecz(X, P1, P), fraza_czas(X, P1),
    fraza_rzecz(X, P1, P) -->
        przedimek(X, P2, P1, P),
        rzeczownik(X, P3, P),
        klauzula_wzgl(X, P3, P2),
    fraza_rzecz(X, P, P) --> rzeczownik_wlasny(X),
    fraza_czas(X, P) -->
        czasownik_przech(X, Y, P1), fraza_rzecz(Y, P1, P),
    fraza_czas(X, P) --> czasownik_nieprzech(X, P),
    klauzula_wzgl(X, P1, (P1&P2)) -->
        [that], fraza_czas(X, P2),
    klauzula_wzgl(_, P, P) --> [],
    przedimek(X, P1, P2, all(X, (P1->P2))) --> [every],
    przedimek(X, P1, P2, exists(X, (P1&P2))) --> [a],
    rzeczownik(X, man(X)) --> [man],
    rzeczownik(X, woman(X)) --> [woman],
    rzeczownik_wlasny(john) --> [john],
    czasownik_przech(X, Y, loves(X, Y)) --> [loves],
    czasownik_nieprzech(X, lives(X)) --> [lives].
```

W podanym programie za pomocą argumentów budujemy strukturę opisującą znaczenie fraz. Dla każdej frazy jej ostatni argument opisuje jej znaczenie, choć znaczenie tej frazy może zależeć od innych czynników, podanych w innych argumentach. Przykładowo, czasownik *lives* implikuje istnienie formy *lives(X)*, gdzie *X* jest żyjącą osobą. Znaczenie *lives* nie może być podane z góry, niezależnie od *X*. Znaczenie to, aby było do czegośkolwiek przydatne, trzeba przyłożyć do pewnego konkretnego obiektu. Kontekst, w którym używa się czasownika, określa rodzaj obiektu. Wobec tego z definicji wynika, że dla dowolnego *X*, jeśli omawiany czasownik zostanie doń zastosowany, jako znaczenie należy przyjąć *lives(X)*. Takie słowa jak *every* sprawiają znacznie więcej kłopotu. W tym wypadku znaczenie musi być przyłożone do zmiennej i dwóch stwierdzeń zawierających tę zmienną. Wynikiem jest stwierdzenie, że jeśli podstawienie obiektu pod zmienną w pierwszym zdaniu da prawdę, to podstawienie tego samego obiektu pod zmienną w drugim stwierdzeniu także da prawdę.

Ćwiczenie 9.4. Przeczytaj ze zrozumieniem powyższy program. Wypróbuj jego działanie, podając cele typu

```
?- zdanie(X, [every, man, loves, a, woman], []).
```

⁶ Zdanie oryginalne to „każdy mężczyzna kocha jakąś kobietę”. Jego znaczenie to *każdy(X, (mężczyzna(X) -> istnieje(Y, (kobieta(Y) & kocha(X, Y)))))* — *przyp. tłum.*

Jakie znaczenie przypisze program zdaniom „every man that lives loves a woman” czy „every man that loves a woman lives”? Zdanie „Every man loves a woman” jest tak naprawdę niejednoznaczne: czy istnieje jedna kobieta, którą kocha każdy mężczyzna, czy też może być wiele różnych kobiet, które kochają poszczególni faceci? Czy program wygeneruje oba te znaczenia jako rozwiązania alternatywne? Jeśli nie, to dlaczego? Jakie proste założenie dotyczące sposobu tworzenia znaczenia zdań przyjęto?

Ogólniejsze zastosowanie reguł gramatyki

Notacja reguł gramatyki może być użyta w przypadku ogólniejszym do ukrycia dodatkowej pary argumentów stosowanych jako akumulator lub struktura różnicowa. Niezależnie od obsługi danych ostatniego poziomu (listy konkretnych słów), dwa dodatkowe argumenty mechanizmu tranlacji reguł gramatyki pozwalają śledzić sytuację dotyczącą poszczególnych fragmentów informacji, zmieniając się w miarę postępów przetwarzania. Tak więc bardziej uniwersalne odczytanie

```
fraza_rzecz(X,Y) :- przedimek(X,Z), rzeczownik(Z,Y).
```

będzie brzmiało „fraza_rzecz jest prawdą w sytuacji X, jeśli przedimek jest prawdą w sytuacji X oraz w sytuacji wynikającej z (Z) prawdą jest rzeczownik. Sytuacja po fraza_rzecz jest taka, jak to wynika z rzeczownik(Y)”. W przypadku reguł gramatycznych, „sytuacja” to zwykle lista słów pozostających do przetworzenia. Jak się zaraz przekonamy, istnieją też inne możliwości.

Wystąpienie słowa w regule gramatyki oznacza przejście z jednej „sytuacji” do innej. W przypadku zwykłego użycia reguł gramatyki, jest to przejście od jednej listy nieprzetworzonych słów do tej samej listy bez pierwszego słowa. Ostatecznie wszystkie zmiany „sytuacji” prowadzą do redukcji tak danego ciągu (jedyna dostępna metoda poruszania się po liście słów to znajdowanie kolejnych słów w układach wynikających z reguł gramatyki).

Aby uogólnić użycie reguł gramatyki, dobrze jest zdefiniować predykat mówiący, jak odnalezienie słowa powoduje przejście z jednej sytuacji do innej. Stosując różne definicje tego predykatu, możemy spowodować, że nasze reguły będą zachowywały się standardowo lub w sposób całkiem odmienny. Predykat taki często nazywany jest 'C' /3 (cudzysłowy są konieczne, gdyż mamy do czynienia z wielką literą), jego definicja dla zwykłych reguł gramatyki jest następująca:

```
% 'C'(Prev, Terminal, New)
%
% Jest uzgadniany, jeśli słowo Terminal powoduje przejście
% z sytuacji Prev do sytuacji New.
'C'([W|Ws], W, Ws).
```

Jeśli zatem sytuacja (lista niewykorzystanych słów) to [W|Ws] i słowo W jest następne w bieżącej regule gramatyki, możemy przejść do nowej sytuacji, w której listą nieużytych słów jest Ws.

W przypadku przekształcania na Prolog, słowa w regułach gramatyki mogą zostać wyrażone w kontekście 'C', a nie bezpośrednio jako wartości listy argumentów. Faktycznie, wiele systemów Prologu przekształca reguły gramatyczne w kontekście 'C', czyli reguła

```
przedimek --> [the]
```

przekształcana jest na

```
przedimek(In,Out) :- 'C'([In,the],Out).
```

zamiast na

```
przedimek([the|S],S).
```

jak to było powyżej. Jeśli mamy definicję 'C' jak powyżej, obie te definicje przedrostka w Prologu dadzą takie same wyniki (można sprawdzić to samemu). Kiedy więc używamy reguł gramatyki zgodnie ze standardem, zbędna jest nam wiedza o użytej metodzie przekształcania.

Użycie zmodyfikowanej definicji 'C' /3 pozwala na użycie reguł gramatyki do zapisywania w regułach danych innych niż zredukowana krok po kroku lista. Przykładowo, jeśli wyznaczamy długość listy, można zapisać liczbę znalezionych dotąd pozycji. Oto kod z rozdziału 3, w wersji reguł gramatyki:

```
listlen(L,N) :- lenacc(L,0,N).

lenacc([]) --> [].
lenacc([H|T]) --> [1], lenacc(T).
```

W takiej sytuacji natknięcie się na słowo 1 powoduje dodanie 1 do dotychczasowej sumy. Wobec tego definicja 'C' teraz będzie wyglądała następująco:

```
'C'([0|_],X,New) :- New is 0|_ + X.
```

Innym przykładem może być wyznaczanie listy części dotąd uzyskanych. Oto jak mogłyby wyglądać odpowiednie reguły:

```
czesci(X,P) :- czesci2(X,[],P).
czesci2(X) --> [X], {nierozkładalna(X)}.
czesci2(X) --> {podzesp01(X Podczesci)}, listaczesci2(Podczesci).

listaczesci2([]) --> [].
listaczesci2([P|Ogon]) --> czesci2(P), listaczesci2(Ogon).
```

W tym wypadku potrzebna będzie następująca definicja 'C':

```
'C'([0|_], X, [X|0|_]).
```

Ważna uwaga: predykat phrase/2 zakłada, że użytkownika interesuje ostateczna sytuacja mająca postać []. Jeśli zmienimy definicję 'C', możemy potrzebować własnej zmodyfikowanej wersji phrase/2, która nie będzie takiego założenia robiła.

**Wyższa Szkoła
Zarządzania i Bankowości**
ul. Armii Krajowej 4, 30-150 Kraków
tel. (012) 638-66-77, tel./fax (012) 637-33-47
wpis do Rejestru MEN nr 55 z dnia 11.05.1995
konto BOŚ O/M Kraków 15401115-12087-27000-00
NIP 677-17-56-169 REGON 350814545

Rozdział 10.

Prolog a logika

Język programowania Prolog został wymyślony przez Alaina Colmerauera i jego współpracowników około roku 1970. Była to pierwsza próba zaprojektowania praktycznie użytecznego języka programowania, który umożliwiałby programiście logiczny opis zadania, a nie za pomocą standardowych konstrukcji programistycznych mówiących *co i kiedy* komputer ma zrobić. Zamiary twórców Prologu dobrze wyraża nazwa tego języka, pochodząca od wyrażenia **Programming in Logic** (Programowanie logiczne).

W tej książce skoncentrowaliśmy się przede wszystkim na użyciu Prologu jako narzędzia do wykonywania praktycznych zadań, natomiast nie omawialiśmy tego, jak Prolog zbliża się do ostatecznego celu bycia systemem „programowania logicznego”. W tym rozdziale postaramy się tę lukę w teorii nadrobić i omówimy powiązanie Prologu z logiką oraz powiemy, na ile język ten umożliwia „programowanie logiczne”.

Krótkie wprowadzenie do rachunku predykatów

Jeśli chcemy omawiać powiązanie Prologu z logiką, najpierw musimy powiedzieć sobie, co rozumiemy przez logikę. Pierwotnie logikę stworzono jako sposób zapisu formy argumentów, dzięki czemu możliwe jest formalne sprawdzanie, czy argumenty te są poprawne, czy nie. Tak oto można za pomocą logiki zapisać twierdzenia, związki między twierdzeniami i sposoby poprawnego *wnioskowania* jednych twierdzeń z innych. Konkretna postać logiki, której będziemy używać, to rachunek predykatów. Tematowi temu będziemy mogli poświęcić niewiele miejsca. Dostępnych jest szereg dobrych książek poświęconych logice, których lektura pomoże w zrozumieniu omawianych w tym rozdziale zagadnień.

Jeśli chcemy zapisać jakieś twierdzenie opisujące świat, musimy być w stanie zapisać obiekty, których twierdzenie to dotyczy. W rachunku predykatów używamy do tego *termów*. Term ma jedną z następujących postaci:

- ♦ *Symbol stałej.* Jest to symbol oznaczający pojedynczy byt lub pojęcie. Można to traktować jako atom Prologu; będziemy używać właśnie składni Prologu. Przykładami symboli stałych są grecki, agata czy porozumienie.
- ♦ *Symbol zmiennej.* Jest to symbol, który może oznaczać w różnych chwilach różne byty. Zmienne są tak naprawdę używane jedynie w połączeniu z kwantyfikatorami, które omówimy dalej. Można je utożsamiać ze zmiennymi Prologu i dalej też będziemy używać składni Prologu. Przykładami symboli zmiennych są X, Człowiek czy Grecki.
- ♦ *Term złożony.* Term złożony składa się *symbolu funkcyjnego* oraz uporządkowanego zbioru termów będących *argumentami*. Term złożony może reprezentować byt, który zależy od bytów opisanych jego argumentami. Symbol funkcyjny wskazuje, jak definiowany byt zależy od swoich argumentów. Przykładowo, moglibyśmy użyć dwuargumentowego symbolu funkcyjnego opisującego odległość. W takim wypadku term złożony oznacza różnicę między obiektami reprezentowanymi przez swoje argumenty. O termie złożonym możemy myśleć jako o strukturze prologowej, w której symbol funkcyjny jest funktorem. Termy złożone rachunku predykatów będziemy zapisywać stosując składnię Prologu, więc przykładowo `zona(henryk)` może oznaczać żonę Henryka, `odleglosc(punkt1, X)` może oznaczać odległość między danym punktem a innym, potem wskazywanym miejscem przestrzeni, zaś `klasy(maria, dzienpo(W))` może oznaczać klasy, w których Maria prowadzi zajęcia dzień po dniu W.

Tak więc rachunek predykatów pozwala zapisywać obiekty tak, jak robi się to w Prologu.

Aby wyrazić pewne twierdzenie o obiektach, musimy wyrazić związek między obiektami. Używamy do tego *symboli predykatów*. *Twierdzenie atomowe* składa się z symbolu predykatu oraz ciągu termów będących jego argumentami — tego typu wyrażenia mogą być też celami w Prologu. Oto przykład twierdzenia atomowego:

```
czlowiek(maria)
lubi(czlowiek, wino)
posiada(X, osiol(X))
```

W Prologu struktura może być celem lub argumentem innej struktury lub jednocześnie jednym i drugim. W przypadku rachunku predykatów jest inaczej, gdyż istnieje zasadnicze rozróżnienie symboli funkcyjnych używanych do tworzenia argumentów i symboli predykatów pełniących funkcję funktorów tworzących twierdzenia.

Twierdzenia złożone tworzymy z twierdzeń atomowych w różny sposób. Od tej chwili zaczynamy omawiać zagadnienia, które nie mają bezpośrednich odpowiedników w Prologu. Możemy tworzyć twierdzenia złożone za pomocą *złączeń logicznych*. Możemy używać znanych już operatorów „nie”, „oraz”, „lub”, „wynika” i „jest równoważne z”. W poniższej tabeli zestawiono złączenia i ich znaczenie. α i β oznaczają dowolne twierdzenia. Podajemy składnię z klasycznego rachunku predykatów (RP) oraz składnię używaną w programowaniu z uwagi na łatwość zapisu w komputerze.

Łącznik	Składnia RP	Nasza składnia	Znaczenie
Negacja	$\neg \alpha$	$\sim \alpha$	„nie α ”
Koniunkcja	$\alpha \wedge \beta$	$\alpha \& \beta$	„ α i β ”
Alternatywa	$\alpha \vee \beta$	$\alpha \# \beta$	„ α lub β ”
Implikacja	$\alpha \supset \beta$	$\alpha \rightarrow \beta$	„z α wynika β ”
Równoważność	$\alpha \equiv \beta$	$\alpha \leftrightarrow \beta$	„ α jest równoważne z β ”

Tak więc

```
mezczyzna(fred) # kobieta(fred)
```

może być zapisem twierdzenia, że Fred jest mężczyzną lub Fred jest kobietą.

```
mezczyzna(jan) -> czlowiek(jan)
```

może być zapisem twierdzenia, że z tego, iż Jan jest mężczyzną, wynika, że Jan jest człowiekiem. Zapisy wynikania i równoważności są czasem w pierwszej chwili trudne do zrozumienia. Mówimy, że z α wynika β , jeśli z prawdziwości α wynika prawdziwość β . Mówimy, że α i β są ze sobą równoważne, jeśli α jest prawdziwe zawsze wtedy, gdy prawdziwe jest β . Powyższe łączniki można zapisać za pomocą koniunkcji, alternatywy i przeczenia:

$\alpha \rightarrow \beta$	jest równoważne z	$(\sim \alpha) \# \beta$
$\alpha \leftrightarrow \beta$	jest równoważne z	$(\alpha \& \beta) \# (\sim \alpha \& \sim \beta)$
$\alpha \leftrightarrow \beta$	jest też równoważne z	$(\alpha \rightarrow \beta) \& (\beta \rightarrow \alpha)$

Nie wyjaśniliśmy jeszcze, jakie znaczenie mają zmienne w twierdzeniach. Znaczenie jest zdefiniowane tylko wtedy, gdy zmienne te są wprowadzone za pomocą kwantyfikatorów. Kwantyfikatory pozwalają zapisywać informacje o zbiorach obiektów. W rachunku predykatów istnieją dwa kwantyfikatory. Jeśli v jest zmienną, zaś P jest twierdzeniem, to

Składnia RP	Nasza składnia	Znaczenie
$\forall v.P$	<code>all(v,P)</code>	„ P jest prawdziwe dla dowolnego v ”
$\exists v.P$	<code>exists(v,P)</code>	„Istnieje taka wartość v , dla której prawdziwe jest P ”

Pierwszy z powyższych kwantyfikatorów to *kwantyfikator ogólny*, bo mówi on o dowolnych wartościach. Drugi kwantyfikator jest *kwantyfikatorem szczególnym*, gdyż mówi o istnieniu pewnego obiektu. Oto przykłady użycia kwantyfikatorów:

```
all(X, mezczyzna(X) -> czlowiek(X))
```

oznacza, że dla dowolnej wartości X , jeśli X jest mężczyzną, to X jest też człowiekiem. Możemy to odczytać „dla wszystkich X , jeśli X jest mężczyzną, to X jest człowiekiem”. Najprostsza postać tego twierdzenia to „każdy mężczyzna jest człowiekiem”. Podobnie

```
exists(Z, ojciec(jan,Z) & kobieta(Z))
```


mówi, że istnieje takie Z, że dla tego Z Jan będzie ojcem, i że to Z będzie kobietą. Zatem „istnieje takie Z, że Jan jest ojcem Z i Z jest kobietą” czyli po prostu „Jan ma córkę”. Oto bardziej skomplikowane wyrażenia rachunku predykatów:

```
all(X, zwierze(X) -> exists(Y, matka(X,Y)) )
all(X, formatRK(X) <=> (atomic(X) # compound(X)))
```

Postać klauzulowa

Jak widzieliśmy przed chwilą, wyrażenia rachunku predykatów zapisane za pomocą \rightarrow (implikacja) i \leftrightarrow (równoważność) można zapisać przy użyciu $\&$ (konjunkcji), $\#$ (alternatywy) oraz \sim (negacji). Tak naprawdę istnieje znacznie więcej tego typu tożsamości, więc z niczego nie rezygnując możemy nie korzystać na przykład z $\#$, \rightarrow , \leftrightarrow i \exists (\exists (X,P)). W wyniku takiej redundancji to samo twierdzenie możemy zapisać na wiele sposobów. Jeśli chcemy wykonywać formalne przekształcenia wyrażeń rachunku predykatów, okazuje się, że jest to zadanie bardzo trudne. Sytuacja znacznie się upraszcza, jeśli możemy wszystko wyrazić w jeden sposób. Wobec tego teraz przyjrzymy się, jak można przekształcić wszystkie twierdzenia rachunku predykatów na jedną ich postać, a mianowicie na *postać klauzulową*, w której te same twierdzenia można wyrazić już na mniej sposobów. Okazuje się, że twierdzenie rachunku predykatów zapisane w formie klauzulowej jest bardzo podobne do zestawu reguł Prologu. Wobec tego badanie postaci klauzulowej jest kluczem do zrozumienia związku Prologu z logiką.

W dodatku B podajemy program Prologu, który automatycznie przekształca formuły rachunku predykatów na klauzule. Między omawianymi teraz zagadnieniami a programem z dodatku B istnieje jedna różnica. Aby ułatwić niektóre przekształcenia, zmienne rachunku predykatów zapisuje się jako atomy. Wobec tego, gdy chcemy przekształcić za pomocą programu z dodatku B wyrażenie

```
(osoba(X) # ~matka(X,Y)) # ~osoba(Y)
```

konieczne będzie zapisanie go jako

```
(osoba(x) # ~matka(x,y)) # ~osoba(y)
```

Przekształcanie wyrażenia rachunku predykatów na postać normalną składa się z sześciu zasadniczych kroków.

Etap 1. Usunięcie implikacji

Zaczynamy od usunięcia wszystkich wystąpień \rightarrow i \leftrightarrow zgodnie z tożsamościami podanymi wcześniej w tym rozdziale. Możemy się spodziewać, że na przykład wyrażenie

```
all(X, mezczyzna(X) -> czlowiek(X))
```

zostanie przekształcone na

```
all(X, mezczyzna(X) # czlowiek(X))
```

Etap 2. Przesuwanie negacji do wewnątrz

Krok ten jest związany z zastosowaniem negacji do wyrażenia nie będącego wyrażeniem atomowym. W takich wypadkach wykonywane jest następujące przekształcenie:

```
~ (czlowiek(cezar) & zyje(cezar))
```

przekształcane jest na

```
~ czlowiek(cezar) # ~ zyje(cezar)
```

natomiast

```
~ all(Y, osoba(Y))
```

jest przekształcane na

```
exists(Y, ~ osoba(Y))
```

Poprawność tego kroku wynika z następujących tożsamości:

$\sim(\alpha \& \beta)$	jest tożsame z	$\sim\alpha \# \sim\beta$
$\sim\exists(v, P)$	jest tożsame z	$\text{all}(v, \sim P)$
$\sim\text{all}(v, P)$	jest tożsame z	$\exists(v, \sim P)$

Po realizacji kroku 2. negacja występuje już jedynie w wyrażeniach obejmujących wyrażenia atomowe. Twierdzenie atomowe i twierdzenie atomowe poprzedzone negacją nazywamy *literalami*. Następne kroki będą dotyczyły literalów jako całości, zaś to, które literały są zanegowane, będzie miało znaczenie jedynie na końcu.

Etap 3. Skolemizacja

Następny krok polega na wyeliminowaniu kwantyfikatorów istnienia. Robi się to przez wprowadzanie nowych symboli stałych, *stałych Skolema*, w miejsce zmiennych wprowadzanych za pomocą tych kwantyfikatorów. Zamiast mówić, że istnieje obiekt o pewnych właściwościach, można taki obiekt nazwać i po prostu podać jego właściwości. Do tego właśnie służą stałe Skolema. Proces ten nadweręża właściwości logiczne wyrażenia bardziej niż którekolwiek inne przekształcenie, ale z kolei ma on ważną cechę. Istnieje interpretacja symboli z wyrażenia, taka, że wyrażenie jest prawdziwe wtedy i tylko wtedy, gdy istnieje interpretacja przekształconej wersji wyrażenia. Taka równoważność na nasze potrzeby jest wystarczająca. Tak więc przykładowo

```
exists(X, kobieta(X) & matka(X, ewa) )
```

zostanie zamienione na

```
kobieta(g197) & matka(g197, ewa)
```

gdzie g197 to niepowtarzalna stała. Stała ta odpowiada kobiecie, której matką jest Ewa. Ważne jest użycie niepowtarzalnego symbolu, gdyż

```
exists(X, kobieta(X) & matka(X, ewa) )
```

nie oznacza, że pewna konkretna osoba jest córką Ewy, ale że taka osoba w ogóle istnieje. W końcu może się okazać, że g197 odpowiada tej samej osobie, co inny symbol stałej, ale to nie wynika z powyższego twierdzenia.

Jeśli w wyrażeniu występują kwantyfikatory ogólne, skolemizacja nie jest taka prosta. Przykładowo, skolemizacja

```
all(X, czlowiek(X) -> exists(Y, matka(X,Y)))
```

(„każdy człowiek ma matkę”) daje

```
all(X, czlowiek(X) -> matka(X,g2))
```

mówiące, że wszyscy ludzie mają tę *samą* matkę — obiekt oznakowany przez g2. Jeśli istnieją zmienne wprowadzone przy użyciu kwantyfikatorów ogólnych, skolemizacja wymaga wprowadzenia symboli funkcyjnych, które opiszą, jak to, co istnieje *zależy* od wybranych zmiennych. Tak więc powyższy przykład powinien zostać przekształcony na

```
all(X, czlowiek(X) -> matka(X,g2(X)))
```

W takim wypadku symbol funkcyjny g2 odpowiada funkcji, takiej, że jeśli przekazuje jej osobę, zwróci ona matkę tej osoby.

Etap 4. Przesunięcie kwantyfikatorów ogólnych na zewnątrz

Ten etap jest już prosty — wystarczy kwantyfikatory ogólne przenieść poza wyrażenie. Nie ma to wpływu na interpretację całego wyrażenia. Przykładowo

```
all(X, mezczyzna(X) -> all(Y, kobieta(Y) -> lubi(X,Y)))
```

jest przekształcane na

```
all(X, all(Y, mezczyzna(X) -> (kobieta(Y) -> lubi(X,Y))))
```

Zmienna występująca w tym wyrażeniu teraz jest wprowadzana przy użyciu kwantyfikatora ogólnego znajdującego się przed wyrażeniem, kwantyfikatory same w sobie nie zawierają już żadnych nowych informacji. Możemy więc całe wyrażenie uprościć, usuwając kwantyfikatory. Trzeba tylko pamiętać, że wszystkie zmienne są wprowadzane przez niejawnie kwantyfikatory, które pominęliśmy. Wobec tego

```
all(X, zywy(X) # martwy(X))
  & all(Y, lubi(maria,Y) # nieczysty(Y))
```

możemy zapisać jako

```
(zywy(X) # martwy(X)) & (lub(maria,Y) # nieczysty(Y))
```

Wyrażenie to oznacza, że niezależnie od tego, jakie X i Y wybierzemy, X jest albo żywy, albo martwy oraz albo Maria lubi Y, albo Y jest nieczysty.

Etap 5. Wyprowadzenie & poza

W tej chwili nasze pierwotne wyrażenie rachunku predykatów bardzo się zmieniło. Już nie mamy jawnie podawanych kwantyfikatorów, jedyne łączniki to & i # (oraz ewentualnie negacje literalów). Teraz nasze wyrażenie zapiszemy w specjalnej postaci normalnej, *koniunkcyjnej postaci normalnej*, w której żadne koniunkcje nie pojawiają się już wewnątrz alternatyw. Możemy więc przekształcić całe wyrażenie w zbiór koniunkcji &, w którym elementy łączone są albo literalami, albo literalami połączonymi #. Założmy, że A, B i C są literalami. Możemy skorzystać z następujących tożsamości:

$(A \& B) \# C$ jest tożsame z $(A \# C) \& (B \# C)$

$(A \# B) \& C$ jest tożsame z $(A \& C) \# (B \& C)$

Przykładowe wyrażenie

```
swieto(X) #
  pracuje(krzs, X) & (zly(krzs) # smutny(krzs))
```

(dla każdego X, jeśli X jest świętem lub Krzys pracuje w X i jest zły lub smutny) jest równoważne z następującym:

```
(swieto(X) # pracuje(krzs, X)) &
  (swieto(X) # (zly(krzs) # smutny(krzs)))
```

(Dla każdego X, po pierwsze X jest świętem lub Krzys w X pracuje, a po drugie X jest świętem lub Krzys jest zły lub smutny).

Etap 6. Przekształcanie w klauzule

Wyrażenie, z którym mamy teraz do czynienia, jest zestawem koniunkcji & obejmujących literały lub literały połączone #. Przyjrzyjmy się temu z ogólnego punktu widzenia, nie zagłębiając się w alternatywy. Wyrażenie może mieć postać:

$(A \& B) \& (C \& (D \& E))$

gdzie litery odpowiadają złożonym twierdzeniom, ale niezawierającym już żadnych &. Wszystkie nawiasy i zagnieżdżenia są już zbędne, gdyż twierdzenia

$(A \& B) \& (C \& (D \& E))$

$A \& ((B \& C) \& (D \& E))$

$(A \& B) \& ((C \& D) \& E)$

oznaczają to samo. Wprawdzie wyrażenia te różnią się budową, ale ich znaczenie jest takie samo. Wynika to stąd, że jeśli przyjmiemy, iż pewien zestaw twierdzeń jest prawdziwy, nie ma znaczenia, jak poszczególne twierdzenia są pogrupowane. Nie ma na przykład znaczenia, czy powiemy, że „A jest prawdziwe, tak samo B i C”, czy też powiemy, że „A i B są prawdziwe, a C także”. Tak więc nawiasy nie zmieniają interpretacji wyrażenia, więc możemy zapisać:

$A \& B \& C \& D \& E$

Po drugie, kolejność zapisania poszczególnych twierdzeń też nie ma znaczenia. Wszystko jedno, czy powiemy, że „zarówno A, jak i B są prawdziwe”, czy powiemy, że „B jest prawdziwe oraz A jest prawdziwe”. Na koniec nie musimy wcale podawać & między wyrażeniami, gdyż z góry wiemy, że na najwyższym poziomie będą same koniunkcje. Tak więc możemy całe wyrażenie zapisać w znacznie bardziej zwartej formie, podając jedynie *zbiór* {A,B,C,D,E}. Skoro mówimy, że jest to zbiór, to nieistotna jest w nim kolejność. Zbiór {A,B,C,D,E} jest tym samym zbiorem, co {B,A,C,E,D}, {E,D,B,C,A} i tak dalej. Wyrażenia, które umieszczane są w tym zbiorze po przekształceniu całego wyrażenia do postaci klauzulowej, nazywamy *klauzulami*. Tak więc rachunek predykatów jest równoważny (w pewnym sensie) ze zbiorem klauzul.

Zajmijmy się teraz dokładniej zagadnieniem, czym są te klauzule. Powiedzieliśmy, że składają się one z literałów połączonych alternatywami, więc jeśli litery od V do Z oznaczają literały, klauzula będzie miała postać:

$$((V \# W) \# X) \# (Y \# Z)$$

Możemy znowu zastosować to samo przekształcenie, które zastosowaliśmy w przypadku wyrażenia najwyższego poziomu. Znow nawiasy są zbędne i nieistotna jest kolejność składników. Możemy zatem zapisać wyrażenia jako klauzulę będącą zbiorem literałów $\{V, W, X, Y, Z\}$ domyślnie połączonych alternatywami.

Teraz już nasze wyrażenie jest w postaci klauzulowej. Co więcej, użyte reguły nie zmieniły się niezależnie od tego, jaka jest ich interpretacja. W postaci klauzulowej wyrażeniu odpowiada zestaw klauzul, z których każda jest zestawem literałów. Literał to albo formuła atomowa, albo zanegowana formuła atomowa. Taka postać jest zwarta, gdyż pominęliśmy wszelkie niejawne koniunkcje, alternatywy i kwantyfikatory ogólne. Musimy oczywiście pamiętać o użytych konwencjach, aby zrozumieć, co zapis w postaci klauzulowej oznacza.

Przyjrzyjmy się pewnemu wyrażeniu (powstałemu na etapie 5.), aby sprawdzić, jak wygląda postać klauzulowa. Najpierw jednak przyjrzyjmy się poprzedniemu przykładowi:

$$(\text{swieto}(X) \# \text{pracuje}(\text{krzys}, X)) \& \\ (\text{swieto}(X) \# (\text{zly}(\text{krzys}) \# \text{smutny}(\text{krzys})))$$

Tak oto otrzymujemy dwie klauzule. Pierwsza zawiera literały

$$\text{swieta}(X), \text{pracuje}(\text{krzys}, X)$$

a druga zawiera literały:

$$\text{wakacje}(X), \text{zly}(\text{krzys}), \text{smutny}(\text{krzys})$$

Oto kolejny przykład. Wyrażenie

$$(\text{osoba}(\text{adam}) \& \text{osoba}(\text{ewa})) \& \\ ((\text{osoba}(X) \# \sim \text{matka}(X, Y) \# \sim \text{osoba}(Y))$$

daje początek trzem regułom. Dwie z nich zawierają po jednym literale:

$$\text{osoba}(\text{adam})$$

oraz

$$\text{osoba}(\text{ewa})$$

W drugim wypadku mamy trzy literały:

$$\text{osoba}(X), \sim \text{matka}(X, Y), \sim \text{osoba}(Y)$$

Aby już ten temat zakończyć, przyjrzyjmy się jeszcze jednemu przykładowi oraz różnym etapom jego przekształcania na postać klauzulową. Zaczniemy od wyrażenia

$$\text{all}(X, \text{all}(Y, \text{osoba}(Y) \rightarrow \text{szanuje}(Y, X)) \rightarrow \text{krol}(X))$$

które mówi, że jeśli wszyscy szanują kogoś, to ten ktoś jest królem (dla każdego X, jeśli każdy Y odpowiada osobie szanującej X, to X jest królem). Kiedy usuniemy implikację (etap 1.), otrzymujemy

$$\text{all}(X, \sim (\text{all}(Y, \sim \text{osoba}(Y) \# \text{szanuje}(Y, X)) \# \text{krol}(X)))$$

Teraz możemy przenieść negację do środka (etap 2.) i otrzymujemy:

$$\text{all}(X, \text{exists}(Y, \text{osoba}(Y) \& \sim \text{szanuje}(Y, X)) \# \text{krol}(X))$$

Następnie w wyniku skolemizacji otrzymujemy:

$$\text{all}(X, (\text{osoba}(\text{fl}(X)) \& \sim \text{szanuje}(\text{fl}(X), X)) \# \text{krol}(X))$$

gdzie fl to funkcja Skolema. Obecnie przechodzimy do usuwania kwantyfikatorów ogólnych (etap 4.), co daje nam:

$$(\text{osoba}(\text{fl}(X)) \& \sim \text{szanunek}(\text{fl}(X), X)) \# \text{krol}(X)$$

Teraz całość przekształcimy w zwykłą postać koniunkcyjną (etap 5.), w której koniunkcje nie występują w alternatywach:

$$(\text{osoba}(\text{fl}(X)) \# \text{krol}(X)) \& \\ (\sim \text{szanuje}(\text{fl}(X), X) \# \text{krol}(X))$$

Oznacza to, że w etapie 6. będziemy mieli dwie klauzule. Pierwsza z nich zawiera dwa literały:

$$\text{osoba}(\text{fl}(X)) \quad \text{krol}(X)$$

a druga ma literały:

$$\text{szanuje}(\text{fl}(X), X) \quad \text{krol}(X)$$

Zapis klauzul

Musimy mieć możliwość zapisania postaci klauzulowej, więc teraz właśnie zajmijmy się zapisem. Przede wszystkim, postać klauzulowa to zbiór klauzul. Dobra konwencja pozwoli nam zapisać klauzule jedną po drugiej, bez zwracania uwagi na kolejność. W klauzulach mamy zbiory literałów, niektórych zanegowanych. Najpierw będziemy zapisywać literały niezanegowane, potem zanegowane. Te dwie grupy rozdzielimy symbolem $:-$. Literały bez negacji będą rozdzielane średnikami (ich kolejność znów nie ma znaczenia), zaś literały zanegowane będą zapisywane bez tyld i będą rozdzielane przecinkami. Całą klauzulę zakończymy kropką. Przy takim zapisie, jeśli mamy n zanegowanych literałów $\sim Q_1, \sim Q_2, \dots, \sim Q_n$ oraz m niezanegowanych literałów P_1, P_2, \dots, P_m , zapiszemy je jako:

$$P_1; P_2; \dots; P_m :- Q_1, Q_2, \dots, Q_n.$$

Wprawdzie sposób zapisu klauzul wprowadziliśmy dość dowolnie, jednak zapis ten ma swoje uzasadnienie. Jeśli zapiszemy klauzulę z alternatywami, z literałami zanegowanymi oddzielnymi od niezanegowanych, otrzymamy:

$$(P_1 \# P_2 \# \dots \# P_m) \# (\sim Q_1 \# \sim Q_2 \# \dots \# \sim Q_n)$$

co jest równoważne z

$$(P_1 \# P_2 \# \dots \# P_m) \# \sim (Q_1 \& Q_2 \& \dots \& Q_n)$$

co jest równoważne z

$$(Q_1 \& Q_2 \& \dots \& Q_n) \rightarrow (P_1 \# P_2 \# \dots \# P_m).$$

Jeśli zamiast „oraz” użyjemy przecinka, zamiast „lub” użyjemy średnika i skorzystamy z symbolu „~” jako zapisu „wynika z tego” (jest to konwencja Prologu), to ostatecznie klauzula przybierze postać:

$$P_1; P_2; \dots; P_m :- Q_1, Q_2, \dots, Q_n.$$

Teraz możemy zapisać twierdzenie o Adamie i Ewie:

```
((osoba(adam) & osoba(ewa)) &
((osoba(X) # ~matka(X,Y)) # ~osoba(Y))
```

staje się

```
osoba(adam) :- .
osoba(ewa) :- .
osoba(X) :- matka(X,Y), osoba(Y).
```

Rzecz zaczyna już wyglądać znajomo — całkiem jak zapisana w Prologu definicja oznaczająca bycie osobą. Jednak inne wyrażenia są bardziej kłopotliwe:

```
swieto(X); pracuje(krzys,X) :-
swieto(X); zly(krzys); smutny(krzys) :- .
```

nie wydaje się być zaledwie podobne do Prologu. Zajmiemy się tym zagadnieniem później.

W dodatku B prezentujemy program wyświetlający klauzule w takiej właśnie notacji. Klauzule pisane zgodnie z naszą konwencją przybierają postać:

```
osoba(fl(X)); krol(X) :- .
krol(X) :- szanuje(fl(X),X).
```

Rezolucja i dowodzenie twierdzeń

Teraz wiemy już, jak wyrażeniom rachunku predykatów nadać elegancką formę, więc możemy zastanowić się, co dalej. Oczywiście interesujące jest stwierdzenie, czy jeśli mamy zestaw twierdzeń (przesłanek), to wynika z tego zestawu coś ciekawego: warto badać wnioski płynące z przyjętych przesłanek. Te dane nam na początku przesłanki nazywać będziemy *aksjomatami* lub *hipotezami*, zaś przesłanki z nich wynikające to będą twierdzenia. Jest to zgodne z praktyką matematyczną, w której na podstawie ściśle ustalonego zestawu aksjomatów wyprowadzamy różne ciekawe twierdzenia. W tym podrozdziale krótko zajmiemy się wyprowadzaniem ciekawych wniosków z danych przesłanek, czyli zajmiemy się *dowodzeniem twierdzeń*.

W latach sześćdziesiątych wiele osób zaczęło badać możliwość oprogramowania automatycznego dowodzenia twierdzeń. Dziedzina ta przez długi czas dynamicznie się rozwijała i to właśnie dało impuls do stworzenia Prologu. Jednym z przełomowych odkryć było odkrycie przez J. Alana Robinsona *zasady rezolucji* oraz jej zastosowanie do automatycznego dowodzenia twierdzeń. Rezolucja jest *metodą wnioskowania*, czyli na podstawie zestawu twierdzeń można wyprowadzać dalsze twierdzenia. Korzystając z zasady rezolucji, możemy automatycznie dowodzić twierdzeń na podstawie przyjętych aksjomatów. Musimy jedynie zdecydować, których twierdzeń należy użyć, a odpowiednie wnioski zostaną wygenerowane automatycznie.

Rezolucję stworzono w celu przekształcania wyrażeń w formie klauzulowej. Kiedy mamy dwie odpowiednio powiązane klauzule, wygenerowana zostanie nowa klauzula, która jest wnioskiem z tych dwóch pierwszych. Zasada jest taka, że jeśli po lewej stronie jednej klauzuli i po prawej stronie drugiej jest takie samo wyrażenie atomowe, to jako wniosek można wygenerować klauzulę powstałą przez skolejenie dwóch klauzul wejściowych bez powtarzającego się wyrażenia. Na przykład

Z:

```
smutny(krzys); zly(krzys) :-
    dzienroboczy(dzisiaj), pada(dzisiaj).
```

oraz:

```
nieprzyjemny(krzys) :- zly(krzys), zmeczony(krzys).
```

wnioskujemy, że:

```
smutny(krzys); nieprzyjemny(krzys) :-
    dzienroboczy(dzisiaj), pada(dzisiaj), zmeczony(krzys).
```

Inaczej mówiąc, jeśli dzisiaj jest dzień roboczy i pada, to Krzys jest smutny lub zły. Poza tym, jeśli Krzys jest zły i zmęczony, jest nieprzyjemny. Wobec tego, jeśli dzisiaj jest dzień roboczy, pada i Krzys jest zmęczony, to Krzys jest smutny lub nieprzyjemny.

Tak naprawdę w tym przykładzie dokonaliśmy dwóch zbyt daleko idących uproszczeń. Po pierwsze, wszystko się komplikuje, jeśli klauzula zawiera zmienne. W takim wypadku dwa wyrażenia atomowe nie muszą być identyczne, ale muszą do siebie „pasować”. Poza tym klauzula wynikowa jest otrzymywana przez dopasowanie dwóch innych reguł w wyniku dodatkowej operacji. Operacja ta obejmuje ukonkretnianie zmiennych na tyle, aby obie klauzule jako struktury zostały dopasowane do struktury nowej klauzuli. Drugie uproszczenie polega na tym, że w przypadku ogólnym w rezolucji można dopasowywać wiele literałów z prawej strony do wielu literałów ze strony lewej. Tutaj będziemy zajmować się jedynie przykładami, w których z każdej klauzuli wybieramy jeden literał.

Przyjrzyjmy się następującemu przykładowi użycia rezolucji ze zmiennymi:

- (1) osoba(fl(X)); krol(X) :- .
- (2) krol(Y) :- szanuje(fl(Y),Y).
- (3) szanuje(Z,artur) :- osoba(Z).

Pierwsze dwie klauzule otrzymaliśmy z zapisu reguły mówiącej, że „jeśli każda osoba szanuje kogoś, to ten ktoś jest królem”. Zmieniliśmy nazwy zmiennych, aby ułatwić sobie objaśnianie. Trzecie wyrażenie to przesłanka mówiąca, że wszyscy szanują Artura. Stosując rezolucję do klauzul (2) i (3) (dopasowujemy dwa literały szanuje), otrzymujemy:

(5) $\text{krol}(\text{artur}) :- \text{osoba}(\text{fl}(\text{artur})).$

W regule (2) dopasowaliśmy Y do artur z (3), zaś w (3) Z dopasowaliśmy do $\text{fl}(Y)$ z (2). Teraz możemy zastosować rezolucję do (1) i (4) i otrzymamy:

(6) $\text{krol}(\text{artur}); \text{krol}(\text{artur}) :- .$

Jest to równoważne z faktem, że Artur jest królem.

Zgodnie z formalną definicją rezolucji jest to proces „dopasowywania”, który nieformalnie nazywa się *unifikacją*. Intuicyjnie można powiedzieć, że wyrażenia atomowe można *zunifikować*, jeśli można je — jako struktury prologowe — dopasować do siebie. W dalszych rozważaniach zobaczymy, że dopasowywanie w większości implementacji Prologu nie jest dokładnie równoważne z unifikacją.

Jak można za pomocą rezolucji coś sprawdzić i udowodnić? Jedną możliwością polega na tym, że można powtarzać rezolucję i sprawdzać, czy pojawia się w końcu oczekiwany przez nas wniosek. Niestety, nie możemy zagwarantować, że będzie to w ogóle miało miejsce, nawet jeśli szukane twierdzenie wynika z przesłanek. W powyższym przykładzie nie można na przykład wyprowadzić z podanych klauzul prostej klauzuli $\text{krol}(\text{artur})$, choć oczywiście jest ona wypływającym z nich wnioskiem. Tak więc być może wypada stwierdzić, że rezolucja nie jest metodą dostatecznie silną na nasze potrzeby? Na szczęście odpowiedź znów jest przecząca, gdyż możemy przekształcić nasz cel tak, aby rezolucja gwarantowała możliwość znalezienia rozwiązania, o ile tylko jest to możliwe.

Ważną formalną właściwością rezolucji jest to, że jest ona *całkowicie falsyfikowalna*, czyli jeśli zbiór klauzul jest *niespójny*, to metodą rezolucji da się z niego wyprowadzić *klauzulę pustą*:

$:- .$

W związku z tym, że rezolucja jest *poprawna*, klauzula pusta nie zostanie wygenerowana w żadnym innym wypadku. Zestaw reguł jest sprzeczny, jeśli nie istnieje możliwość uzgodnienia wszystkich predykatów, symboli stałych i symboli funkcyjnych, aby uzyskać jednocześnie same prawdziwe twierdzenia. Klauzula pusta logicznie jest *falszem* — odpowiada twierdzeniu, które nie może być nigdy prawdziwe. Wobec tego rezolucja może udowodnić, że teza (zbiór przesłanek) jest niesprzeczna, doprowadzając do sprzeczności.

Czy opisane właściwości rezolucji mogą być nam pomocne? Otóż...

Jeśli teza $\{A_1, A_2, \dots, A_n\}$ jest niesprzeczna, formuła B jest wnioskiem z $\{A_1, A_2, \dots, A_n\}$ wtedy i tylko wtedy, gdy teza $\{A_1, A_2, \dots, A_n, \neg B\}$ jest sprzeczna.

Jeśli zatem наша hipoteza jest niesprzeczna, wystarczy dodać *zanegowane* klauzule, które chcemy udowodnić. Rezolucja da regułę pustą wtedy, gdy twierdzenie wynika z przesłanek. Dodawane do hipotez klauzule nazywamy *instrukcjami celu*. Instrukcje celu nie różnią się od hipotez, gdyż jedno i drugie są po prostu klauzulami. Jeśli zatem mamy zestaw klauzul A_1, A_2, \dots, A_n oraz mamy pokazać, że są one ze sobą sprzeczne, nie możemy powiedzieć, czy wynika to stąd, że

$\neg A_1$ wynika z A_2, \dots, A_n , czy też

$\neg A_2$ wynika z A_1, A_3, \dots, A_n , czy też

$\neg A_3$ wynika z $A_1, A_2, A_4, \dots, A_n$,

... i tak dalej.

Kwestią wyboru jest to, które instrukcje uznane zostaną za cel, gdyż z punktu widzenia samej rezolucji, wszystkie są nierozróżnialne.

W naszym przykładzie dotyczącym króla Artura łatwo można sprawdzić, jak otrzymać pustą klauzulę po dodaniu celu:

(6) $:- \text{krol}(\text{artur}).$

(jest to klauzula odpowiadająca $\neg \text{krol}(\text{artur})$). Widzieliśmy już, jak z hipotezy można wywieść klauzulę

(5) $\text{krol}(\text{artur}); \text{krol}(\text{artur}) :- .$

Rezolucja zastosowana do (5) i (6) (z (5) dopasowanie dowolnego wyrażenia atomowego) da nam

(7) $\text{krol}(\text{artur}) :- .$

Na koniec, stosując rezolucję do (6) i (7), otrzymujemy

$:- .$

Tak więc szukane twierdzenie jest wnioskiem z przesłanek i Artur jest królem.

Zupełność rezolucji jest bardzo miłą jej cechą matematyczną. Oznacza to, że jeśli jakiś fakt jest wnioskiem z hipotezy, możliwe powinno być udowodnienie tego przez wykazanie fałszywości zbioru przesłanek uzupełnionych negacją faktu. Kiedy jednak mówimy, że rezolucja pozwoli uzyskać pustą klauzulę, mamy na myśli to, że istnieje etap rezolucji, w których korzystamy z aksjomatów i klauzul otrzymanych w poprzednich krokach, i że po tych etapach otrzymamy klauzulę bez literałów. Jedyny problem polega na znalezieniu odpowiednich etapów rezolucji. Bowiem choć rezolucja mówi, jak wyprowadzić z dwóch klauzul wniosek, nie wskazuje nam ona metody doboru następnej klauzuli w celu dopasowania. Zwykle, jeśli mamy wiele hipotez, każdą można dopasować na wiele sposobów. Co więcej, zawsze kiedy wyprowadzamy nową klauzulę, staje się ona kandydatką do następnego kroku rezolucji. Większość dostępnych opcji nie dotyczy rozwiązywanego zadania i jeśli nie zadamy o to, mnóstwo czasu poświęcimy na badanie bezsensownych rozwiązań.

Przedstawiono już szereg poprawek pierwotnej zasady rezolucji; niektórymi z nich zajmiemy się w następnym rozdziale.

Klauzule Horna

Przyjrzyjmy się teraz udoskonaleniom rezolucji związanym z zastosowaniem pewnego szczególnego rodzaju klauzul — *klauzul Horna*. Klauzula Horna to klauzula zawierająca co najwyżej jeden niezanegowany literał. Okazuje się, że jeśli używamy klauzulowego automatu dowodzącego twierdzeń do wyznaczania wartości funkcji, trzeba używać właśnie klauzul Horna. Jako że rezolucja klauzul Horna jest dość prosta, w praktyce właśnie takiej postaci rezolucji się używa. Zastanówmy się, jak wygląda rezolucja, kiedy ograniczymy się do tego jednego rodzaju klauzul.

Po pierwsze, istnieją dwa rodzaje klauzul Horna: zawierające jeden niezanegowany literał i niezawierające żadnego takiego literału. Typy te nazwiemy klauzulami Horna z *głową* i *bez głowy*. Oba te rodzaje klauzul przybliżymy czytelnikowi na przykładzie (pamiętajmy, że niezanegowane literały znajdują się po lewej od :-):

```
kawaler(X) :- mezczyzna(X), niezonaty(X).
:- kawaler(X).
```

Kiedy rozważamy zbiory klauzul Horna (wraz z instrukcjami celu), wystarczy nam rozważyć zbiory klauzul, takie, aby wszystkie klauzule poza jedną były z *głową*. Wszystkie rozwiązywalne problemy, które można zapisać w formie klauzul Horna, są zapisane tak, że:

- ♦ Istnieje jedna klauzula bez głowy.
- ♦ Wszystkie pozostałe klauzule mają głowę.

Ponieważ możemy dowolnie decydować, które klauzule są celami, możemy zdecydować, że jako cel potraktujemy klauzulę bez głowy, zaś pozostałe będziemy uważać za hipotezy. Jest to rozwiązanie w pewien sposób naturalne.

Czemu mamy rozważać tylko klauzule Horna spełniające podane warunki? Po pierwsze, łatwo zauważyć, że aby problem był rozwiązywany, musi istnieć przynajmniej jedna klauzula bez głowy. Wynika to stąd, że w wyniku rezolucji dwie klauzule Horna z głową dają nam też klauzulę Horna z głową. Zatem, jeśli wszystkie klauzule mają głowy, będziemy mogli wyprowadzać jedynie klauzule z głową. Klauzula pusta nie ma głowy, więc nie będziemy mogli jej wyprowadzić. Drugi wymóg, istnienie tylko jednej klauzuli bez głowy, jest nieco trudniejszy do uzasadnienia. Okazuje się jednak, że jeśli między aksjomatami istnieje więcej klauzul bez głowy, każdy dowód przez rezolucję można przekształcić na dowód używający co najwyżej jednej reguły bez głowy. Wobec tego, jeśli z aksjomatów wynika klauzula pusta, wyniknie ona z klauzul z głową i co najwyżej jednej bez głowy.

Prolog

Teraz przyjrzyjmy się, jak Prolog pasuje do opisanych tu zagadnień. Jak już widzieliśmy, niektóre nasze wyrażenia przekształcone zostały na klauzule bardzo podobne do klauzul Prologu, ale inne nadal wyglądały obco. Te podobne do klauzul Prologu zostały

przekształcone na klauzule Horna. Kiedy klauzule Horna zapiszemy zgodnie z przyjętymi normami, po lewej stronie :- pojawi się co najwyżej jedno wyrażenie atomowe. W przypadku ogólnym klauzule mogą mieć szereg wyrażen (odpowiadających niezanegowanym literałam w wyrażeniach atomowych). W Prologu bezpośrednio możemy zapisać jedynie klauzule Horna. Klauzule programu prologowego odpowiadają klauzulom Horna z głową. Jaka część programu prologowego odpowiada instrukcjom celu? Po prostu zapytanie Prologu

```
?- A1, A2, ..., An.
```

odpowiada dokładnie klauzuli Horna bez głowy:

```
:- A1, A2, ..., An.
```

Przed chwilą pokazywaliśmy, że każdy problem zapisany za pomocą klauzul Horna można przedstawić korzystając tylko z jednej klauzuli bez głowy. Dokładnie odpowiada to sytuacji typowej dla Prologu, w której wszystkie klauzule Prologu mają głowy i tylko jedna jej nie ma — jest to udowodniany cel.

System Prologu oparty jest na rezolucyjnym dowodzeniu twierdzeń przy użyciu klauzul Horna, w konkretnej postaci *rezolucji wejścia liniowego*. W takim wypadku możliwość wyboru obiektu rezolucji w danym kroku jest ograniczona następująco: zaczynamy od instrukcji celu i robimy rezolucję z jedną z hipotez, otrzymujemy nową klauzulę. Następnie stosujemy rezolucję jednej z hipotez z nową klauzulą. Potem rozwiązujemy następną rezolucję korzystając z kolejnej hipotezy, i tak dalej. Za każdym razem robimy rezolucję klauzuli najnowszej z jedną z klauzul spośród przesłanek. Nigdy nie używamy klauzul, które zostały wyprowadzone wcześniej lub rozwiązane z dwiema hipotezami. W języku Prologu jako ostatnią wyprowadzoną klauzulę traktujemy złączenie celów, które należy spełnić. Zaczynamy od zapytania i mamy nadzieję, że na koniec otrzymamy klauzulę pustą. Na każdym etapie znajdujemy klauzulę, której głowa pasuje do jednego z celów, w miarę potrzeb ukonkretniamy zmienne, usuwamy dopasowane cele i na koniec dodajemy treść wywoływanej klauzuli do celów, które należy spełnić. Tak więc możemy zaczynać przekształcając:

```
:- matka(jan,X), matka(X,Y).
```

oraz

```
matka(U,V) :- rodzic(U,V), kobieta(V).
```

na:

```
:- rodzic(jan,X), kobieta(X), matka(X,Y).
```

Tak naprawdę strategia dowodzenia w Prologu jest nawet nieco bardziej ograniczona niż ogólna rezolucja wejściowa. W tym przykładzie dopasowujemy pierwsze literały klauzuli celu, ale równie dobrze moglibyśmy użyć klauzuli drugiej. W Prologu dopasowywany literał wybierany jest w ten sam sposób: jest to pierwszy cel w klauzuli celu. Poza tym nowe cele wyprowadzane z klauzul umieszczane są z *przodu* klauzul celu. To oznacza, że Prolog spełnia podcel, a dopiero potem może przejść dalej.

I to tyle na temat tego, co się dzieje, kiedy Prolog zdecyduje, którą klauzulę dopasować do pierwszego celu. Jak jednak organizowane jest badanie alternatywnych klauzul spełniających ten sam cel? Przede wszystkim Prolog realizuje strategię badania

w *głęb*, a nie *wszerz*. Wobec tego rozważana jednorazowo jest tylko jedna możliwość, dalej pojawiają się wnioski uzyskiwane przy założeniu, że dokonaliśmy dobrego wyboru. Dla każdego celu wybierana jest klauzula w określonej kolejności, zaś dalsze badane są dopiero wtedy, gdy poprzednie cele zawiodły w trakcie prób uzgadniania. Inna strategia mogłaby polegać na śledzeniu przez system równoległe alternatywnych rozwiązań. Potem następowałoby przechodzenie między możliwymi ścieżkami, krótkie ich badanie i potem przechodzenie dalej. To drugie rozwiązanie, przeszukiwanie *wszerz*, ma tę zaletę, że jeśli rozwiązanie istnieje, zostanie znalezione. Przeszukiwanie w *głęb* może wpadać w pętlę i w ten sposób śledzić niektóre możliwe rozwiązania. Jednak z drugiej strony, przeszukiwanie w *głęb* jest znacznie prostsze i mniej wymagające co do pamięci w przypadku typowego komputera.

Na koniec jeszcze uwaga na temat tego, czym dopasowywanie w Prologu może się różnić od unifikacji znanej z metody rezolucji. Większość systemów prologowych pozwala uzgodnić cele typu:

```
rowne(X,X).
?- rowne(cos(Y),Y).
```

czyli możliwe jest dopasowanie termu do jego części. W tym przykładzie $\cos(Y)$ dopasowywane jest do Y , które w $\cos(Y)$ występuje. W wyniku Y będzie oznaczało $\cos(Y)$, zaś to ostatnie będzie oznaczało $\cos(\cos(Y))$, (z uwagi na wartość Y), a to będzie $\cos(\cos(\cos(Y)))$ i tak dalej. Tak więc ostatecznie Y będzie oznaczało pewną strukturę nieskończoną. Zauważmy, że choć Prolog często pozwala tworzyć tego typu potworki, zwykle nie można już ich wypisać. Zgodnie z formalną definicją unifikacji, tego rodzaju „terminy nieskończone” nie mają prawa bytu. Prolog wobec tego nie w pełni jest zgodny z teoretyczną metodą rezolucji pozwalającą dowodzić twierdzenia; aby stał się zgodny, konieczne byłoby dodanie warunku, że zmiennej nie można ukonkretnić strukturą zawierającą tę zmienną. Takie sprawdzenie jest proste do zaimplementowania, ale spowodowałoby znaczne spowolnienie działania programów. Jako że problem ten dotyczy niewielu programów, zwykle nic się tutaj nie poprawia.¹

Prolog i programowanie w logice

W poprzednich podrozdziałach pokazywaliśmy, w jakim sensie Prolog jest oparty na automatycznym dowodzeniu twierdzeń. Na nasze programy możemy spojrzeć teraz jak na hipotezy opisujące świat, zaś na nasze zapytania jako twierdzenia, których należy dowieść. Wobec tego programowanie w Prologu nie polega na wskazywaniu komputerowi, co ma kiedy zrobić, ale na wskazywaniu znanych faktów i żądaniu wyciągnięcia wniosków. Idea takiego programowania jest bardzo kusząca i wielu badaczy poświęciło dużo pracy *programowaniu w logice* jako praktycznej metodzie rozwiązywania

¹ Standard Prologu przewiduje, że wynik jest *niezdefiniowany*, jeśli system Prologu próbuje dopasować term do nieukonkretnionej części samego siebie, lub programy powodujące taką sytuację będą nieprzenośne. Program przenośny powinien gwarantować, że zawsze, kiedy występuje sprawdzanie, jawnie użyty zostanie predykat wbudowany `unify_with_occurs_check/2` zamiast normalnej operacji unifikacji. Zgodnie ze swoją nazwą, predykat ten działa jak `=/2`, ale zawodzi, jeśli wykryje niedopuszczalną próbę ukonkretnienia zmiennej.

problemów. Pozostaje to w opozycji do tradycyjnego programowania w takich językach jak C czy Pascal, w których musimy dokładnie wskazać kolejne kroki korzystając z konstrukcji rozumianych bezpośrednio przez komputer.

Zaletą programowania w logice jest to, że programy są łatwiejsze do zrozumienia. Nie trzeba w nich umieszczać mnóstwa szczegółowych opisów, *jak* coś zrobić; programy te przypominają raczej opis *postaci* rozwiązania. Co więcej, jeśli programy bliższe są opisowi tego, co należy zrobić, powinno być dość prosto sprawdzić, czy program robi to, co powinien (wystarczy przeczytanie programu, a być może nawet automat). Podsumowując, zaletami programowania w języku logiki jest to, że programy są *deklaratywne* i jednocześnie *proceduralne* (strukturalne). Wiemy, *co* program wylicza, zaś mniej już nas interesuje, *jak* to jest wyliczane. Nie będziemy mogli zająć się tutaj ogólnymi zagadnieniami programowania w logice. Czytelnikom zainteresowanym tym tematem możemy polecić książkę Roberta Kowalskiego „Logic for Problem Solving” wydaną w 1979 roku przez North Holland oraz „Introduction to Logic Programming” Christophera Hoggera wydaną przez Academic Press w roku 1984.

Przyjrzyjmy się jeszcze krótko Prologowi jako kandydatowi do miana język programowania w logice. Od razu jest oczywiste, że niektóre programy prologowe opisują logicznie prawdy dotyczące świata. Jeśli zapiszemy:

```
matka(X,Y) :- rodzic(X,Y), kobieta(Y).
```

możemy to potraktować jako stwierdzenie, że bycie matką oznacza bycie rodzicem i kobietą. Tak więc klauzula ta wyraża twierdzenie, które naszym zdaniem jest prawdziwe, a także pokazuje, jak *wykazać*, że ktoś jest matką. Analogicznie, klauzule

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).
```

mówią, co oznacza przyłączenie jednej listy do początku innej. Jeśli przed listę X wstawiana jest lista pusta, wynikiem jest po prostu X . Z drugiej strony, jeśli lista nie-pusta jest wstawiana przed inną listę, to głowa wyniku jest taka sama, jak głowa listy wstawianej, zaś ogon jest wyznaczany przez dołączenie ogona pierwszej listy przed listę drugą. Klauzule te można traktować jako opis dołączania oraz jako opis, jak należy praktycznie łączyć ze sobą dwie listy.

No dobrze, to były ciekawe przykłady programów prologowych, ale jaką interpretację logiczną można przypisać następującym klauzulom:

```
member1(X,Lista) :- var(Lista), !, fail.
member1(X,[X|_]).
member1(X,_[Lista]) :- member1(X,Lista).
print(0) :- !.
print(N) :- write(N), N1 is N-1, print(N1).
rzeczownik(N) :-
    name(N,Nazwa1), append(Nazwa2,[115],Nazwa1),
    name(RdzenN,Nazwa2), rzeczownik(RdzenN).
implikuje(Zalozenie,Wniosek) :-
    asserta(Zalozenie),
    call(Wniosek),
    retract(Wniosek).
```

Pojawia się też problem związany z wszystkimi predykatami wbudowanymi. Cele typu `var(Lista)` nie mówią nic o listach ani byciu elementem, ale odnoszą się do aktualnego

etapu wykonania programu (pewna zmienna jest nieukonkretniona); warunek taki może być prawdziwy tylko przez pewien czas tworzenia dowodu. Analogicznie, odcięcie mówi coś o dowodzie twierdzenia (wskazuje, które możliwości można odrzucić). Te dwa cele można traktować jako sposób zapisania informacji kontrolnych o sposobie dowodzenia. Analogicznie, predykaty typu `write(N)` nie przedstawiają sobą nic ciekawego z punktu widzenia logiki, ale wskazują, na jakim etapie jest dowód (kiedy `N` jest ukonkretnione) i inicjują komunikację z użytkownikiem. Cel `name(N,Nazwa1)` mówi coś o strukturze wewnętrznej tego, co w rachunku predykatów jest niepodzielnym symbolem. W Prologu można przekształcać symbole na łańcuchy znaków, struktury na listy i struktury na klauzule. Operacje te naruszają prostą, jednorodną naturę twierdzeń rachunku predykatów. W ostatnim przykładzie użycie `asserta` oznacza, że reguła dodaje coś do zbioru aksjomatów. W logice każdy fakt lub reguła informują o pewnej niezależnej prawdzie, niezależnej od innych faktów i reguł. W tym wypadku mamy naruszenie tej zasady. Poza tym, jeśli użyjemy tej reguły, będziemy mieli inny zestaw aksjomatów w różnych chwilach dowodzenia! Na koniec fakt, że w regule użycie `write` jako celu oznacza, że zmienna logiczna może zawierać twierdzenie będące częścią aksjomatyki. Tego w ogóle nie da się wyrazić w rachunku predykatów, jest to przykład tego, co może zaoferować logika wyższego rzędu.

Teraz, kiedy zapoznaliśmy się z tymi przykładami, widać, że niektóre programy Prologu można zrozumieć jedynie po zastanowieniu się, co się kiedy dzieje i jak wskazuje się systemowi, co ma zrobić. W przypadku skrajnym program `gensym` z rozdziału 7 właściwie nie ma interpretacji deklaratywnej.

Czy zatem jest w ogóle sens traktować Prolog jako język programowania w logice? Czy korzystając z Prologu będziemy mogli w ogóle wykorzystać zalety programowania w logice? Odpowiedź na oba te pytania jest twierdząca, a to dlatego, że korzystając z odpowiedniego stylu programowania nadal możemy osiągnąć pewne korzyści z powiązania Prologu z logiką. Kluczem do tego jest podział programów na części, aby wydzielić w nich niewielkie części operacje inne niż logiczne. Przykładowo, w rozdziale 4. pokazywaliśmy, jak niektóre zastosowania odcięcia można zastąpić predykatem `not`. W wyniku tego typu podstawień program zawierający wiele odcięć można przekształcić na program zawierający tylko jedno odcięcie (w definicji `not`). Użycie predykatu `not`, choć nie jest on dokładnie odpowiednikiem logicznego przeczenia „ \neg ”, umożliwia „uchwycenie” logicznej interpretacji programu. Analogicznie, ograniczenie stosowania predykatów `asserta` i `retract` do definicji nielicznych predykatów (takich jak `gensym` i `findall`) pozwala tworzyć programy czytelniejsze od programów, w których oba predykaty są wielokrotnie używane.

Tak więc nie wszystkie cele programowania w logice zostały osiągnięte w Prologu. Niemniej, Prolog jest praktycznym systemem programowania, którego zaletą jest klarowność i deklaratorywność charakterystyczne dla języków programowania w logice. Trwają jednocześnie prace nad nowymi wersjami języka Prolog, które byłyby bliższe logice niż obecna. Jednym z najważniejszych zadań stojących przed badaczami tej dziedziny jest stworzenie praktycznego systemu, w którym zbędne byłoby odcięcie, zaś predykat `not` odpowiadałby dokładnie zaprzeczeniu logicznemu.

Więcej informacji o teorii programowania w logice można znaleźć w książkach „Logic for Problem Solving” Roberta Kowalskiego (wydana w 1979 roku przez North Holland) oraz „Introduction to Logic Programming” Christophera Hoggera (wydana przez Academic Press w roku 1984).

Rozdział 11.

Projekty w Prologu

Ten rozdział zawiera szereg projektów, które warto zrealizować, aby doskonalić swoje umiejętności programowania. Niektóre projekty są proste, zaś niektóre należy potraktować raczej jako długoterminowe, realizowane równolegle z przerabianiem wcześniejszych rozdziałów niniejszej książki. Projekty prostsze można realizować jako uzupełnienia do ćwiczeń. Projekty nie są ustawione w żadnej szczególnej kolejności, choć te z drugiej części tego rozdziału są bardziej otwarte i ambitne, wymagają pewnej wiedzy z dziedzin sztucznej inteligencji i informatyki. W niektórych projektach zakłada się znajomość pewnej dziedziny, więc osoby nie będące fizykami-teoretykami nie powinny się załamywać, jeśli nie będą potrafiły napisać programu różniczkującego trójwymiarowe pole wektorowe.

Szereg programów w języku Prolog opublikowano w raporcie „How to solve it with Prolog”, zredagowanym przez H. Coelho, J.C. Cotta i L.M. Pereira. Raport ten jest rozpowszechniany przez portugalskie Laboratório Nacional de Engenharia Civil w Lizbonie. Zawiera setki niewielkich przykładów, problemów i ćwiczeń z takich dziedzin jak wnioskowanie w bazach danych, przetwarzanie języka naturalnego, rozwiązywanie równań symbolicznych i tak dalej. Raport ten nie jest podręcznikiem, więc przedstawiane w nim programy zawierają niewiele objaśnień.

Łatwiejsze projekty

1. Zdefiniuj predykat „spłaszczający” listę w listę zawierającą wszystkie atomy listy pierwotnej, ale nie zawierającą żadnych podlist. Na przykład, nie powinno zawieść wywołanie:

```
?- flatten([a,[b,c],[[d],[e]]], [a,b,c,d,e]).
```

Problem ten można rozwiązać na przynajmniej sześć różnych sposobów.

2. Napisz program wyliczający odstęp między dwiema datami w postaci Dni-Miesiące zakładając, że daty należą do tego samego, nieprzestępnego roku. Należy zauważyć, że (-) to infiksowa postać funktora binarnego. Nie powinno zawieść na przykład wywołanie:

Wyższa Szkoła
Zarządzenia i Bankowości
ul. Armii Krajowej 4, 30-150 Kraków
tel. (012) 638-65-77, 66/16x (012) 637-33-47
wpis do Rejestru MEN nr 55 z dnia 11.05.1995r.
Kraj. REGON 149115-12087-27005-00
NIP 677-17-56-169 REGON 350814846

3. W rozdziale 7. przekazaliśmy dość informacji, aby tworzyć programy różniczkujące i upraszczające wyrażenia algebraiczne. Rozwiń te programy tak, aby mogły przekształcać także wyrażenia zawierające funkcje trygonometryczne. Osoby zainteresowane tematem mogą uwzględnić także operatory geometrii różniczkowej, jak *div*, *grad* i *curl*.
4. Napisz program, który wygeneruje zaprzeczenie zdania logicznego. Zdania logiczne zbudowane są z atomów, funktora unarnego *not* i funktorów binarnych *and*, *or* oraz *implies* (implikuje). Należy podać odpowiednie deklaracje operatorów odpowiadających funktorom, na przykład korzystając z operatorów *~, &, # i ->*, jak to opisano w rozdziale 10. Wyrażenie zanegowane należy doprowadzić do najprostszej postaci, tak aby *not* dotyczyło jedynie atomów. Na przykład negacja zdania

```
p implies (q and not (r))
```

powinna mieć postać

```
p and (not(q) or r)
```

5. Przez indeks rozumiemy listę słów występujących w tekście uporządkowaną alfabetycznie wraz z informacją, ile razy dane słowo w tekście wystąpiło. Napisz program generujący indeks do zestawu słów zapisanych jako łańcuchy Prologu składające się ze znaków ASCII.
6. Napisz program „rozumiejący” proste zdania w postaci:

```
— to —
— jest to —
Czy — to —?
```

Program powinien dawać prawidłowe odpowiedzi (tak, nie, ok, nie wiem) na podstawie zdań podanych wcześniej, na przykład:

```
Jan to meczyszna.
ok
Meczyszna to osoba.
ok
Czy Jan to osoba?
tak
Czy Maria to osoba?
nie wiem
```

Każde zdanie powinno zostać przekształcone na klauzulę Prologu, która następnie będzie dodana do bazy wiedzy lub wykonana. Wobec tego powyższe zdania zostaną przekształcone na:

```
meczyszna(jan).
osoba(X) :- meczyszna(X).
?- osoba(jan).
?- osoba(maria).
```

Jeśli uznasz to za stosowne, możesz skorzystać z reguł gramatyki. Klauzula główna decydująca o sposobie prowadzenia dialogu może mieć postać:

```
rozmowa :-
    repeat,
    czytaj(Zdanie),
```

```
analizuj(Zdanie,Klauzula).
odpowiedz_na(Klauzula).
Klauzula = stop.
```

7. Algorytm alfa-beta (α - β) to metoda przeszukiwania drzew gier wspominana w wielu książkach poświęconych programowaniu sztucznej inteligencji. Zaimplementuj ten algorytm w Prologu.
8. Innym problemem szeroko omawianym w literaturze poświęconej programowaniu jest problem N-królowych. Zaimplementuj program znajdujący wszystkie sposoby umieszczenia czterech królowych na szachownicy o wymiarach 4 na 4 tak, aby żadna z figur nie była pozostałych. Jednym ze sposobów rozwiązania tego problemu jest napisanie generatora, który będzie sprawdzał kolejne permutacje w celu zapewnienia prawidłowego rozmieszczenia królowych.
9. Napisz program przekształcający zdania logiczne (opisane w punkcie 4.) tak, aby wszystkie *and*, *or*, *implies* i *not* zostały zastąpione pojedynczym spójnikiem *nand*, który definiuje się następująco:

```
( $\alpha$  nand  $\beta$ ) =  $\neg(\alpha \wedge \beta)$ 
```

10. Jednym ze sposobów zapisywania całkowitych liczb dodatnich jest użycie termów Prologu zawierających liczbę 0 i funktor *s* z jednym argumentem. Tak więc 0 zapisujemy jako takie, 1 jako *s(0)*, 2 jako *s(s(0))* i tak dalej; każda liczba jest zapisywana jako funktor *s*, którego argumentem jest liczba o jeden mniejsza. Stwórz definicje standardowych operacji algebraicznych — dodawania, mnożenia i odejmowania na tak zapisywanych liczbach. Przykładowo, predykat *plus* może zachowywać się następująco:

```
?- plus(s(s(0)),s(s(s(0))),X).
X = s(s(s(s(s(0)))))
```

co oznacza $2 + 3 = 5$. W przypadku odejmowania konieczne będzie wprowadzenie sposobu zapisu wyników mniejszych od zera. Zdefiniuj też predykat odpowiadający porównaniu „mniejsze niż”. Jakie argumenty muszą być ukonkretnione, aby takie definicje zadziałały? Co będzie się działo w innych przypadkach? Jak to się ma do standardowych operacji algebraicznych Prologu? Postaraj się zdefiniować operacje bardziej skomplikowane, jak dzielenie liczb całkowitych czy wyznaczanie pierwiastka kwadratowego.

Projekty zaawansowane

Wprowadźcie problemy z tego podrozdziału są otwarte, ale wszystkie były już implementowane w Prologu przez różnych programistów. Niektóre z tych problemów to proste rozszerzenia wcześniej przedstawianych programów, a inne są całkowicie nowe i ich rozwiązanie wymaga znajomości literatury z dziedziny sztucznej inteligencji czy informatyki.

1. Załóżmy, że mamy mapę z drogami łączącymi miasta. Napisz program planujący trasę między dwoma miastami i podający szacunkowy czas podróży. Mapa powinna zawierać dane o odległościach, rodzaju dróg i ich stanie, spodziewanym natężeniu ruchu, pochyłościach i dostępności paliwa.
2. W dostępnych obecnie wersjach Prologu wbudowane są jedynie operacje całkowitoliczbowe. Napisz zestaw programów obsługujących arytmetykę liczb rzeczywistych zapisywanych jako ułamki lub jako mantysy i wykładniki.
3. Napisz procedury odwracające i mnożące macierze.
4. Na kompilowanie języka programowania wysokiego poziomu na język niskiego poziomu można patrzeć jako na kolejne przekształcenia drzew składniowych. Napisz taki kompilator, który najpierw będzie kompilował wyrażenia arytmetyczne. Następnie dodaj struktury sterujące (jak *if...then...else*). Składnia wynikowego kodu nie ma w tym wypadku większego znaczenia. Przykładowo, wyrażenie $x+1$ może zostać „uproszczone” do postaci $\text{inc } x$, gdzie inc będzie zadeklarowane jako operator unarny. Problem alokacji pamięci możemy na razie odłożyć zakładając, że kod będzie wykonywany na maszynie opartej na stosie (bezaadresowej).
5. Zaproponuj opis złożonych gier planszowych, jak szachy i go, oraz postaraj się zrozumieć, jak można wykorzystać możliwości Prologu w dziedzinie dopasowywania się do wzorców do zaimplementowania strategii gry.
6. Zaproponuj formalizm, który pozwoli wyrazić zbiór aksjomatów, takich jak teoria grup, geometria euklidesowa, semantyka znaczeniowa, oraz zastanów się nad programem automatycznego dowodzenia twierdzeń z danej dziedziny.
7. Interpreter klauzul Prologu można napisać w samym Prologu (zobacz: „Przetwarzanie programów” w rozdziale 7.). Napisz interpreter, który zrealizuje różne sposoby wykonywania programu prologowego, na przykład umożliwi zmianę kolejności wykonywania (zamiast standardowego wykonywania programu od lewej do prawej), korzystając z jakiegoś mechanizmu określania kolejności interpretacji klauzul.
8. Zapoznaj się z literaturą z dziedziny sztucznej inteligencji dotyczącą generowania planów rozwiązywania problemów oraz zrealizuj taki generator planów.
9. Opisz w Prologu problem interpretacji linii w kontekście pewnej sceny stanowiącej tło. Elementy obrazu mogą być opisane za pomocą zmiennych w ten sposób, aby obraz stanowił zestaw warunków nałożonych na te zmienne.
10. Korzystając z reguł gramatyki, napisz program, który będzie analizował zdania w postaci:¹
 Fred saw John.
 Mary was seen by John.
 Fred told Mary to see John.
 John was believed to have been seen by Fred.
 Was John believed to have told Mary to see Fred?

¹ W poniższych przykładach zachowano oryginalną, angielską wersję zdań, gdyż deklinacja i koniugacja w języku polskim uczyniłyby to zadanie bardzo trudnym. Oczywiście bardziej ambitni czytelnicy mogą spróbować swoich sił uwzględniając odmianę ograniczonej liczby słów — *przyp. tłum.*

11. System reguł wnioskowania, jakiego używa się w badaniach nad sztuczną inteligencją, ma postać zestawu reguł w postaci „*jeśli sytuacja to akcja*”. W ten sposób zapisuje się systemy ekspertowe; na przykład poniższe zdania są typowymi przykładami wziętymi z używanych w praktyce systemów:

Farmakologia: jeśli czynnik X jest czwartorzędową solą amonową, zaś czynnik Y jest salicylanem, to X i Y oddziałują zwiększając absorpcję przez tworzenie par jonowych.

Gra w szachy: jeśli czarny król może przejść na pole sąsiadujące z czarnym skoczkiem i odległość białego króla od skoczka jest większa od jednego pola, skoczek nie jest zagrożony.

Medycyna: jeśli środowiskiem kultury jest krew, organizm jest gram-ujemny i organizm ma postać pałeczki, to istnieje 60-procentowe prawdopodobieństwo, że organizmem chorobotwórczym jest *Pseudomonas aeruginosa*.

Napisz w Prologu program, który będzie interpretował taki zbiór reguł wnioskowania. Sugerowane dziedziny wiedzy to rozpoznawanie roślin lub zwierząt na podstawie ich cech charakterystycznych. Na przykład reguła botaniczna może wyglądać następująco:

jeśli roślina ma kwadratowy pęd, ma parzyste liście, dwuwargowy okryty kwiat i owoce składające się z małych nasion w kielichu, należy do rodziny *Labiatae*.

Jeśli w takiej sytuacji na pytanie programu „Czy kwiat ten ma kwadratowy pęd?” odpowiemy przecząco, odrzucona zostanie możliwość, że roślina należy do rodziny *Labiatae*.

12. Napisz program, który przekształci angielskie zdania języka prawniczego do postaci rachunku predykatów.
13. Napisz program dowodzący twierdzeń za pomocą rachunku predykatów.
14. Napisz program symulujący psychiatrę, który będzie powtarzał pewne wyrażenia bazując na słowach kluczowych występujących w danych wejściowych. Przypomnijmy, że podobny program z rozdziału 3. zamieniał słowa kluczowe zdania na inne słowa. Typowy dialog może wyglądać następująco (odpowiedzi komputera pogrubiono):

W czym problem?
 Rzecz w tym, że mam zbyt dużo pracy.
 Co mam rozumieć przez zbyt dużo pracy?
 Pisanie listów.
 Rozumiem. Proszę dalej.
 I jeszcze mycie samochodu mojej matki.
 Proszę powiedzieć mi więcej o swojej rodzinie.
 Dlaczego mam to mówić?
 Dlaczego ma pan co robić?

W tym wypadku słowami kluczowymi są rzecz w tym, matka i dlaczego. Brak znanego systemowi słowa kluczowego powoduje wygenerowanie tekstu Rozumiem. Proszę dalej.

15. Napisz program analizujący zdania dotyczące zdarzeń w biurze, na przykład „Kowalski przyjdzie do biura o 15:00 na spotkanie”. Przydatne może być użycie reguł gramatyki. Program powinien na koniec wygenerować podsumowanie z informacjami kto, co, gdzie, kiedy, na przykład:

```
kto: kowalski
gdzie: biuro
kiedy: 15:00
co: spotkanie
```

Podsumowanie może mieć postać wewnętrznej bazy danych, na podstawie której możliwe będzie odpowiadanie na pytania:

```
Gdzie jest Kowalski o 15:00?
gdzie: biuro
co: spotkanie
```

16. Napisz interfejs języka naturalnego do systemu plików, aby móc uzyskiwać odpowiedzi na pytania:

Właścicielem ilu plików jest David?

Czy Chris współdzieli PROG.MAC z David?

Kiedy Bill zmieniał plik VIDEO.C?

Program musi potrafić pytać o różne części systemu plików, na przykład o właścicieli plików czy daty.

Dodatek A

Odpowiedzi do niektórych ćwiczeń

W tym dodatku zamieściliśmy sugerowane rozwiązania części ćwiczeń. Jak w przypadku większości ćwiczeń polegających na programowaniu, zwykle nie ma jednej prawidłowej odpowiedzi i czytelnicy mogą ćwiczenia rozwiązać całkiem inaczej niż podpowiadają autorzy, choć równie dobrze. Tak czy inaczej, trzeba sprawdzić w używanej implementacji Prologu, czy znalezione rozwiązanie działa zgodnie z oczekiwaniami. Nawet osoby, które napisały prawidłowy, choć inny, program, mogą sporo skorzystać zapoznając się z rozwiązaniami przez nas sugerowanymi, gdyż obejrzenie innego rozwiązania tego samego problemu zawsze jest pouczające.

Ćwiczenie 1.3. Oto przykładowe definicje związków rodzinnych.

```
jest_matka(Mama) :- matka(Mama,Dziecko).
jest_ojcem(Tata) :- ojciec(Tata,Dziecko).
jest_synem(Syn) :- rodzic(Rodzic,Syn), mezczyzna(Syn).
siostra(Siostra,Osoba) :-
    rodzic(Rodzic,Siostra), rodzic(Rodzic,Osoba),
    kobieta(Siostra), rozne(Siostra,Osoba).
dziadek(Dziadek,X) :- rodzic(Rodzic,X), ojciec(Dziadek,Rodzic).
rodzenstwo(S1,S2) :-
    rodzic(Rodzic.S1), rodzic(Rodzic.S2), rozne(S1,S2).
```

Zauważmy, że w definicjach siostra i rodzenstwo korzystamy z predykatu rozne. Dzięki temu system nie uważa, że ktoś może być sam sobie siostrą czy rodzeństwem. Na razie przedstawiliśmy zbyt mało informacji o Prologu, aby możliwe było samodzielne zdefiniowanie predykatu rozne.

Ćwiczenie 5.2. Poniższy program bez końca wczytuje znaki (z bieżącego pliku wejściowego) i ponownie je wyświetla, zamieniając jednak litery a na b.

```
go :- repeat, get_char(C), obsluz(C), fail.
obsluz(a) :- !, put(b).
obsluz(X) :- obsluz(X).
```

Ogromnie ważne jest odcięcie w pierwszej klauzuli obsluz (dlaczego?).

Ćwiczenie 7.9. Oto program generujący trójki pitagorejskie:

```
pitagoras(X,Y,Z) :-
    wtrojce(X,Y,Z),
    Sumakw is X*X + Y*Y, Sumakw is Z*Z.
wtrojce(X,Y,Z) :-
    is_integer(Suma),
    minus(Suma,X,Suma1), minus(Suma1,Y,Z).
minus(Suma,Suma,0).
minus(Suma,D1,D2) :-
    Suma>0, Suma1 is Suma-1,
    minus(Suma1,D1,D3), D2 is D3+1.
is_integer(0).
is_integer(N) :- is_integer(N1), N is N1 + 1.
```

W pokazanym programie do generowania możliwych trójek liczb całkowitych X, Y, Z użyto predykatu wtrojce. Następnie sprawdza się, czy dana trójka jest trójką pitagorejską. Definicja wtrojce musi zagwarantować, że wygenerowane zostaną wszystkie możliwe trójki liczb całkowitych. Najpierw generowane są możliwe sumy X, Y i Z. Następnie za pomocą niedeterministycznego predykatu minus generowane są wartości X, Y i Z dające wyznaczoną sumę.

Ćwiczenie 9.1. Oto program przekształcający proste reguły gramatyki w klauzule Prologu. Zakładamy tutaj, że reguła nie zawiera fraz z dodatkowymi argumentami, celów ujętych w nawiasy klamrowe, alternatyw ani odcięć.

```
?- op(1199,xfx,-->).
przekształc((P1-->P2),(G1:-G2)) :-
    lewa_strona(P1,S0,S,G1),
    prawa_strona(P2,S0,S,G2).
lewa_strona(P0,S0,S,G) :-
    nonvar(P0), znacznik(P0,S0,S,G).
prawa_strona((P1,P2),S0,S,G) :-
    !,
    prawa_strona(P1,S0,S1,G1),
    prawa_strona(P2,S1,S,G2),
    and(G1,G2,G).
prawa_strona(P,S0,S,true) :-
    islist(P),
    !,
    append(P,S,S0).
prawa_strona(P,S0,S,G) :- znacznik(P,S0,S,G).
znacznik(P,S0,S,G) :- atom(P), G =..[P,S0,S].
and(true,G,G) :- !.
and(G,true,G) :- !, and(G1,G2,(G1,G2)).
islist([]) :- !, islist([_]).
append([A|B],C,[A|D]) :- append(B,C,D).
append([],X,X).
```

W tym programie zmienne o nazwach zaczynających się od P to opisy fraz w regułach gramatyki (atomy lub listy słów). Zmienne zaczynające się od G to cele Prologu. Zmienne zaczynające się od S to argumenty celów Prologu (reprezentują one ciągi słów). Dla osób zainteresowanych tym zagadnieniem prezentujemy następny program, który obsługuje interpretację bardziej dowolnych reguł gramatyki. Jednym ze

sposobów, w jakie Prolog może obsługiwać reguły gramatyczne, jest użycie zmodyfikowanej wersji consult, w której klauzule w postaci A --> B będą przekształcane przed ich dodaniem do bazy danych. Zdefiniowaliśmy kilka operatorów, które mają pełnić funkcję nawiasów klamrowych; w niektórych implementacjach Prologu istnieją definicje wbudowane, dzięki czemu term {X} jest innym zapisem struktury ' {} '(X).

```
?- op(1101,fx,'{ }').
?- op(1100,xf,'{ }').
?- op(1199,xfx,-->).
przekształc((P0-->Q2),(P:-Q)) :-
    lewa_strona(P0,S0,S,P),
    prawa_strona(Q0,S0,S,Q1),
    splaszcz(Q1,Q).
lewa_strona((NT,Ts),S0,S,P) :- !,
    nonvar(NT),
    islist(Ts),
    znacznik(NT,S0,S1,P),
    append(Ts,S,S1).
lewa_strona(NT,S0,S,P) :-
    nonvar(NT), znacznik(NT,S0,S,P).
prawa_strona((X1,X2),S0,S,P) :- !,
    prawa_strona(X1,S0,S1,P1),
    prawa_strona(X2,S1,S,P2),
    and(P1,P2,P).
prawa_strona((X1;X2),S0,S,(P1;P2)) :-
    !, or(X1,S0,S,P1), or(X2,S0,S,P2).
prawa_strona({P},S,S,P) :- !.
prawa_strona(!S.S!) :- !.
prawa_strona(Ts,S0,S,true) :-
    islist(Ts),
    !,
    append(Ts,S,S0).
prawa_strona(X,S0,S,P) :- znacznik(X,S0,S,P).
or(X,S0,S,P) :-
    prawa_strona(X,S0a,S,Pa),
    ( var(S0a), S0a \== S, !,
      S0=S0a, P=Pa; P=(S0=S0a,Pa) ).
znacznik(X,S0,S,P) :-
    X =.. [F|A], append(A,[S0,S],AX), P =.. [F,AX].
and(true,P,P) :- !.
and(P,true,P) :- !.
and(P,Q,(P,Q)).
splaszcz(A,A) :- var(A), !.
splaszcz((A,B),C) :- !, splaszcz1(A,C,R), splaszcz(B,R),
splaszcz(A,A).
splaszcz1(A,(A,R),R) :- var(A), !.
splaszcz1((A,B),C,R) :-
    !, splaszcz1(A,C,R1), splaszcz1(B,R1,R).
splaszcz1(A,(A,R),R).
islist([]) :- !.
islist([_]).
append([A|B],C,[A|D]) :- append(B,C,D).
append([],X,X).
```

Ćwiczenie 9.2. Definicja ogólnej wersji fraza może mieć postać:

```
faza(Ftyp,Słowa) :-
    Ftyp =.. [Predykat|Argumenty],
    append(Argumenty,[Słowa,[]],Noweargumenty),
    Cel =.. [Predykat|Noweargumenty],
    call(Cel).
```

przy czym korzystamy z definicji `append` z punktu „Łączenie struktur” w rozdziale 3.

Dodatek B

Klauzulowa postać programów

Zgodnie z obietnicą z rozdziału 10., proces przekształcania wyrażeń na klauzule zilustrujemy pokazując fragmenty programu prologowego realizującego to przekształcenie. Program główny ma postać:

```
przekształc(X) :-
    implout(X,X1),           /* Etap 1 */
    negin(X1,X2),            /* Etap 2 */
    skolem(X2,X3,[]),        /* Etap 3 */
    univout(X3,X4),          /* Etap 4 */
    conjn(X4,X5),            /* Etap 5 */
    clausify(X5,Klauzule),    /* Etap 6 */
    pcclauses(Klauzule).     /* Pokaż uzyskane klauzule */
```

Jest to definicja predykatu `przekształc`, który — jeśli wywołamy w Prologu `cel przekształc(X)`, gdzie X jest wyrażeniem rachunku predykatów — pokaże to wyrażenie w formie klauzul. Wyrażenia rachunku predykatów w programie będziemy zapisywać jako struktury, tak jak to sugerowaliśmy wcześniej. *Trzeba pamiętać, że zmienne w rachunku predykatów będą przedstawiane jako atomy*, dzięki czemu pewne przekształcenia będą prostsze. Zmienne rachunku predykatów można odróżnić od stałych dzięki przyjęciu określonej konwencji nazewnicznej. Przykładowo, nazwy zmiennych mogłyby zawsze zaczynać się jedną z liter: x , y i z . Jednak w programie nie musimy znać przyjętej konwencji, gdyż zmienne zawsze najpierw pojawiają się w kwantyfikatorach, zatem łatwo je odnaleźć. Jedynie przy czytaniu otrzymanych wyników potrzebna będzie wiedza, które nazwy są nazwami zmiennych rachunku predykatów, a które stałymi.

Przed wszystkim potrzebne są nam deklaracje operatorów:

```
?- op(200,fx,~).
?- op(400,xfy,#).
?- op(400,xfy,&).
?- op(700,xfy,->).
?- op(700,xfy,<=>).
```

Powyższe definicje wymagają kilku słów objaśnienia. Operator (\sim) ma niższy priorytet niż $(\#)$ i $(\&)$. Na początek musimy przyjąć pewne ważne założenie. Chodzi o to, że nazwy zmiennych są w miarę potrzeb zmieniane, dzięki czemu ta sama zmienna nigdy nie będzie zdefiniowana przez więcej niż jeden kwantyfikator w regule. Dzięki temu unika się ewentualnych konfliktów nazw.

Konwersja na klauzulę oparta jest na omawianej w rozdziale 7. technice przekształcania drzewa. Zapisując łączniki logiczne jako funktory, umożliwiamy zapis reguł rachunku predykatów jako struktur, które można zapisać w formie drzew. Każdemu z sześciu podstawowych kroków przekształcania na klauzulę odpowiada jedno przekształcenie drzewa, które zamienia drzewo wejściowe na drzewo wyjściowe.

Etap 1. — usuwanie implikacji

Definiujemy predykat `implout`, taki, że `implout(X,Y)` oznacza, że Y jest wyrażeniem powstałym z wyrażenia X po usunięciu implikacji.

```
implout((P <-> Q),((P1 & Q1) # (~P1 & ~Q1))) :-
    !, implout(P,P1), implout(Q,Q1).
implout((P -> Q),(~P1 # Q1)) :-
    !, implout(P,P1), implout(Q,Q1).
implout(all(X,P),all(X,P1)) :- !, implout(P,P1).
implout(exists(X,P),exists(X,P1)) :- !, implout(P,P1).
implout((P & Q),(P1 & Q1)) :-
    !, implout(P,P1), implout(Q,Q1).
implout((P # Q),(P1 # Q1)) :-
    !, implout(P,P1), implout(Q,Q1).
implout((~P),(~P1)) :- !, implout(P,P1).
implout(P,P).
```

Etap 2. — przeniesienie negacji do wewnątrz

Musimy tym razem zdefiniować dwa predykaty: `negin` i `neg`. Cel `negin(X,Y)` oznacza, że Y jest wyrażeniem wyprowadzonym przez przeniesienie negacji „do wewnątrz” w wyrażeniu X . Jest to nasze główne zadanie. Cel `neg(X,Y)` oznacza, że Y jest wyrażeniem stworzonym przez zastosowanie przekształcenia do wyrażenia $\sim X$. W obu wypadkach zakładamy, że przeprowadzono już etap 1, więc nie musimy troszczyć się o operatory \rightarrow i \leftrightarrow .

```
negin(~P,P1) :- !, neg(P,P1).
negin(all(X,P),all(X,P1)) :- !, negin(P,P1).
negin(exists(X,P),exists(X,P1)) :- !, negin(P,P1).
negin((P & Q),(P1 & Q1)) :-
    !, negin(P,P1), negin(Q,Q1).
negin((P # Q),(P1 # Q1)) :-
    !, negin(P,P1), negin(Q,Q1).
negin(P,P).
neg(~P,P1) :- !, negin(P,P1).
neg(all(X,P),exists(X,P1)) :- !, neg(P,P1).
neg(exists(X,P),all(X,P1)) :- !, neg(P,P1).
neg((P & Q),(P1 # Q1)) :- !, neg(P,P1), neg(Q,Q1).
neg((P # Q),(P1 & Q1)) :- !, neg(P,P1), neg(Q,Q1).
neg(P,~P).
```

Etap 3. — skolemizacja

Predykat `skolem` ma trzy argumenty: na wyrażenie w pierwotnej postaci, wyrażenie przekształcone oraz liczbę zmiennych zdefiniowanych dotychczas przez kwantyfikatory ogólne.

```
skolem(all(X,P),all(X,P1),Zmienne) :-
    !, skolem(P,P1,[X|Zmienne]).
skolem(exists(X,P),P2,Zmienne) :-
    !,
    gensym(f,F),
    Sk =.. [F|Zmienne],
    subst(X,Sk,P,P1),
    skolem(P1,P2,Zmienne).
skolem((P # Q),(P1 # Q1),Zmienne) :-
    !, skolem(P,P1,Zmienne), skolem(Q,Q1,Zmienne).
skolem((P & Q),(P1 & Q1),Zmienne) :-
    !, skolem(P,P1,Zmienne), skolem(Q,Q1,Zmienne).
skolem(P,P,_).
```

W tej definicji korzystamy z dwóch nowych predykatów. `gensym` musi być zdefiniowany tak, aby cel `gensym(X,Y)` powodował ukonkretnienie Y nowym atomem stworzonym na podstawie atomu X i liczby. W ten sposób generujemy stałe skolemowskie. Predykat `gensym` zdefiniowaliśmy w podrozdziale „Użycie bazy danych; `random`, `gensym`, `findall`” w rozdziale 7. Drugi nowy predykat to `subst`. Żądamy, aby `subst(V1,V2,F1,F2)` było prawdziwe po podstawieniu $V2$ za $V1$ w wyrażeniu $F1$ (uzyskujemy $F2$). Zdefiniowanie tego predykatu zostawiamy jako ćwiczenie dla czytelnika, choć podobne predykaty definiowaliśmy w podrozdziałach „Przetwarzanie list” (rozdział 7.) i „Tworzenie składników struktur i sięganie do nich” (rozdział 6.).

Etap 4. — przesunięcie kwantyfikatorów ogólnych na zewnątrz

Po realizacji tego punktu konieczne będzie, oczywiście, wskazanie, które atomy Prologu odpowiadają zmiennym rachunku predykatów, a które stałym. Już nie będziemy mogli skorzystać z wygodnej zasady, że zmienne są wskazywane przez kwantyfikatory. Oto program przesuwający i usuwający kwantyfikatory ogólne:

```
univout(all(X,P),P1) :- !, univout(P,P1).
univout((P & Q),(P1 & Q1)) :-
    !, univout(P,P1), univout(Q,Q1).
univout((P # Q),(P1 # Q1)) :-
    !, univout(P,P1), univout(Q,Q1).
univout(P,P).
```

Powyższe reguły stanowią definicję predykatu `univout`, takiego, że cel `univout(X,Y)` jest równoważny ze stwierdzeniem, że Y jest takie samo jak X , ale po usunięciu kwantyfikatorów ogólnych.

Trzeba zauważyć, że w definicji `univout` zakładamy, że operacja może być stosowana dopiero po wykonaniu trzech etapów poprzednich. Wobec tego niedopuszczalne jest użycie w wyrażeniu implikacji czy kwantyfikatorów szczególnych.

Etap 5. — zamiana & i

Program przekształcający wyrażenie na koniunkcyjną postać normalną jest zwykle bardziej skomplikowany niż pokazano. Kiedy pojawia się wyrażenie w postaci $(P \# Q)$, gdzie P i Q to dowolne formuły, najpierw trzeba przekształcić P i Q (powiedzmy, w $P1$ i $Q1$), a potem dopiero sprawdzić, czy wyrażenie jako całość nadaje się do przekształcenia. W tym procesie zakładamy zachowanie odpowiedniej kolejności, gdyż może się zdarzyć, że ani P , ani Q nie będą zawierały $\&$ na najwyższym poziomie, zaś $P1$ i $Q1$ będą. Oto program:

```
conjn((P # Q),R) :-
    !,
    conjn(P,P1), conjn(Q,Q1),
    conjn1((P1 # Q1),R).
conjn((P & Q),(P1 & Q1)) :-
    !, conjn(P,P1), conjn(Q,Q1).
conjn(P,P).
conjn1(((P & Q) # R),(P1 & Q1)) :-
    !, conjn((P # R),P1), conjn((Q # R),Q1).
conjn1((P # (Q & R)),(P1 & Q1)) :-
    !, conjn((P # Q),P1), conjn((P # R),Q1).
conjn1(P,P).
```

Etap 6. — zamiana na klauzule

Teraz, w naszej ostatniej części programu, musimy jeszcze zapisać wyrażenia jako klauzule. Najpierw definiujemy predykat `clausify`, który zajmuje się tworzeniem i wewnętrznym zapisem zbiorów klauzul. Zbiór to zestaw zapisany jako lista, przy czym każdej klauzuli odpowiada struktura `cl(A,B)`. W takiej strukturze A jest listą niezanegowanych literalów, zaś B to lista literalów zanegowanych. Predykat `clausify` ma trzy argumenty. Pierwszy z nich to wyrażenie w formie uzyskanej z etapu 5. Drugi i trzeci argument służą do definiowania listy klauzul. Predykat `clausify` tworzy listę kończącą się zmienną, a nie `[]`, następnie zwraca tę zmienną w trzecim argumentcie. Dzięki temu inne reguły mogą potem dodawać coś na koniec listy ukonkretniając tę zmienną. Jedną z cech programu jest sprawdzanie, czy te same wyrażenia atomowe nie pojawiają się w tej samej klauzuli jako zaprzeczone i niezaprzeczone. Jeśli tak się stanie, klauzula nie jest dodawana do listy, gdyż jest ona zawsze prawdziwa i nic nie daje. Poza tym sprawdza się, czy te same literały nie występują w klauzuli dwukrotnie.

```
clausify((P & Q),C1,C2) :-
    !, clausify(P,C1,C3), clausify(Q,C3,C2).
clausify(P,[cl(A,B)|Cs]) :-
    inclause(P,A,[],B,[], !).
clausify(_,_C,C).
inclause((P # Q),A,A1,B,B1) :-
    !,
    inclause(P,A2,A1,B2,B1), inclause(Q,A,A2,B,B2).
inclause((~P),A,A,B1,B) :-
    !, notin(P,A), putin(P,B,B1).
inclause(P,A1,A,B,B) :- notin(P,B), putin(P,A,A1).
notin(X,[X|_]) :- !, fail.
notin(X,[_|L]) :- !, notin(X,L).
notin(X,[]).
```

```
putin(X,[],X) :- !.
putin(X,[X|L],[X|L1]) :- !.
putin(X,[Y|L],[Y|L1]) :- putin(X,L,L1).
```

Prezentacja klauzul

Teraz zdefiniujemy predykat `pclauses`, który spowoduje pokazanie wyrażenia w sposób zgodny z naszym zapisem.

```
pclauses([]) :- !, nl, nl.
pclauses([cl(A,B)|Cs]) :-
    pclause(A,B), nl, pclause(Cs).
pclosure(L,[]) :-
    !, pdisj(L), write(' ').
pclosure([],L) :-
    !, write(':- '), pconj(L), write(' ').
pclosure(L1,L2) :-
    pdisj(L1),
    write(':- '), pconj(L2), write(' ').
pdisj([L]) :- !, write(L).
pdisj([L|Ls]) :- write(L), write(' '), pdisj(Ls).
pconj([L]) :- !, write(L).
pconj([L|Ls]) :- write(L), write(' '), pconj(Ls).
```

Dodatek C

Przenośne programy w standardowym Prologu

W tym dodatku poruszymy pewne kwestie związane z pisaniem w Prologu programów, które będą mogły być używane przez innych, na innych komputerach i w innych systemach. Pokażemy pewne rozwiązania ułatwiające tworzenie takich właśnie programów.

Przenośność standardu Prologu

Trzeba pogodzić się z tym, w miarę jak modyfikowane są poszczególne elementy komputera oraz systemu operacyjnego, konieczne jest dostosowywanie systemów Prologu do tych zmian (dotyczy to zresztą także wszystkich innych języków programowania wysokiego poziomu). Program obsługujący Prolog (czyli pozwalający użytkownikom ładować i uruchamiać programy, korzystać z predykatów wbudowanych i pozwalający ogólnie korzystać z możliwości Prologu) na jakimś komputerze i w środowisku programowym nazywamy *implementacją* Prologu. Różni ludzie tworzą różne implementacje Prologu, często starają się przy tym wykorzystać specyficzne cechy używanego systemu. Wobec tego poszczególne implementacje z punktu widzenia programisty Prologu różnią się między sobą. Jedna implementacja zawierać może predykat wbudowany `foo/1`, zaś w innej ten sam predykat nazywa się `baz/1`; w trzeciej jest predykat `foo/1`, ale robi on coś całkiem innego. Program w Prologu, który w jednym systemie działa bez zarzutu, w innym systemie nagle działać przestaje. Są to kiepskie wiadomości dla osób chcących pisać programy *przenośne*, których będą mogli używać inni (a także dla osób chcących cudze programy wykorzystywać).

W celu uniknięcia opisanych problemów wiele pracy osób tworzących Prolog i programistów Prologu poświęcono na stworzenie standardu Prologu (ISO/IEC 13211-1), który pojawił się ostatecznie w 1995 roku. Standard dokładnie mówi, czym jest program w Prologu i jak jest wykonywany. Zawiera specyfikację predykatów wbudowanych z opisem ich działania. Standard ten został udokumentowany w książce Pierre'a

Deransarta, Abdela Ali Ed-Dbali i Laurenta Cervoni'ego „Prolog: The Standard” wydanej przez Springer Verlag w 1996 roku. Gdyby wszystkie implementacje Prologu były z tym standardem zgodne, problemy przenośności w ogóle by nie istniały.

Różne implementacje Prologu

W chwili pisania tego istnieje niewiele lub zgoła nie ma implementacji Prologu, które byłyby całkowicie zgodne ze standardem Prologu. Na szczęście w książce tej ograniczyliśmy się do niewielkiego podzbioru języka i omawiane przez nas zagadnienia szybko są uwzględniane w istniejących implementacjach języka. Standard języka stanowi najlepszy wzorzec tego, jak powinien wyglądać system Prologu, jego ustalenia są powszechnie akceptowane. Obecnie najlepszą metodą pisania przenośnych programów w Prologu jest zachowywanie zgodności ze standardem, dlatego tak właśnie postępowaliśmy w tej książce.

Fakt, że dana implementacja nie jest zgodna ze standardem, nie musi oznaczać, że nie można w niej pisać programów zgodnych ze standardem Prologu. Osoby, dla których przenośność programów jest ważna, powinny programy pisać na tyle zgodnie ze standardem, na ile tylko jest to możliwe. W szczególności nieprzenośne mogą być programy korzystające z predykatów wbudowanych nieopisanych w niniejszej książce (chyba że są ujęte w standardzie i opisane w książce Deransarta i innych). Problemy pojawiają się, kiedy w danej implementacji standardowe narzędzia Prologu dostępne są w innej postaci, na przykład pod innymi nazwami.

Na szczęście, nawet jeśli dana implementacja Prologu nie zawiera predykatów opisanych w standardzie, często zawiera narzędzia, które pozwolą brakujące predykaty stworzyć.

- ♦ Jeśli wykorzystywana implementacja nie zawiera potrzebnego predykatu standardowego, być może uda się taki predykat napisać za pomocą predykatów w tej implementacji istniejących. Dobrą strategią jest trzymanie takich (i tylko takich) predykatów w osobnym pliku (jest to swojego rodzaju „plik zgodności”), plik ten będzie używany przez osoby korzystające z tej samej implementacji Prologu. Osoby korzystające z implementacji zgodnej ze standardem co do potrzebnych cech, mogą załadować program pomijając plik zgodności. Osoby używające innej implementacji, niezgodnej ze standardem, będą prawdopodobnie musiały stworzyć własny plik zgodności (być może korzystając z Twojego jako punktu wyjścia). Praca ta będzie przydatna także później, kiedy osoby takie będą chciały korzystać z innych programów. Kiedy plik zgodności zostanie już raz utworzony, wszystkie inne programy będą mogły używać jego predykatów tak, jakby używany Prolog był zgodny ze standardem.
- ♦ Jeśli dana implementacja zawiera potrzebny predykat, ale ma on definicję inną niż przewiduje to standard, trzeba spróbować umieścić w pliku kompatybilności definicję predykatu zapewniającą jego zgodność ze standardem. Konieczna będzie jednak zmiana nazwy predykatu używanego w programie oraz w pliku zgodności, aby uniknąć konfliktu z predykatem

zdefiniowanym w danej implementacji. W takim wypadku program nie będzie całkowicie zgodny ze standardem, ale łatwo będzie udokumentować modyfikacje wymagane do tego, aby program stał się zgodny ze standardem Prologu.

W praktyce sprawdzenie, które predykaty standardu muszą być dodane, można zrobić tylko raz dla danej implementacji, ewentualnie raz w danej firmie używającej tej implementacji.

Czego się wystrzegać

Poniżej wymienimy wybrane zagadnienia, co do których poszczególne implementacje Prologu różnią się od standardu lub w przypadku których programista z uwagi na przenośność musi zachować szczególną ostrożność. Lista ta nie jest oczywiście pełna, gdyż nowe implementacje mogą być niezgodne ze standardem na różne nieprzewidziane sposoby.

Nazki. W standardowym Prologu znaki to atomy o długości 1 (rozdział 2.) Niektóre predykaty wbudowane służą do obsługi znaków, inne do obsługi liczbowych kodów znaków (na przykład kodów ASCII). Implementacje mogą różnić się co do tego, co obsługują: znaki czy ich kody. Kody znaków zależne są od konkretnej implementacji.

Łańcuchy. Standard pozwala programiście wybrać między różnymi znaczeniami terminu prologowego składającego się ze znaków zamkniętych w podwójnym cudzysłowie (jak "abc"). W niektórych implementacjach część tych znaczeń jest niedostępna, dlatego metody tej w ogóle nie wykorzystywaliśmy w książce.

Nieznane predykaty. Standard pozwala programiście wybierać jedną z wielu akcji, które mają być wykonywane w przypadku, kiedy wywołany zostanie predykat nie mający definicji. W niektórych implementacjach część tych akcji jest niedostępna, więc lepiej na tym zachowaniu nie polegać.

Aktualizacja bazy danych. Dziwne rzeczy mogą się dziać, jeśli podczas spełniania celu zawierającego jakiś predykat usuwane są klauzule tegoż predykatu. Wprawdzie standard mówi, co powinno dziać się w takim wypadku, ale nie wszystkie implementacje zachowują się zgodnie z zaleceniami, więc lepiej unikać tego typu rozwiązań.

Nazwy predykatów. Predykaty wbudowane mogą w różnych implementacjach różnie się nazywać. Przykładowo \+ może nazywać się not. Najlepiej trzymać się nazw standardowych i w razie potrzeby tworzyć plik kompatybilności zawierający nazwy niestandardowe.

Priorytety operatorów. Priorytety te mogą być różne w różnych implementacjach. Najlepiej polegać na priorytetach standardowych w miarę potrzeb, o ile to możliwe, jawnie wywoływać op/3 dla używanych operatorów przed załadowaniem programu.

Porównywanie termów. Niektóre implementacje mogą tej funkcjonalności nie zawierać. Wyniki porównywania termów zwykle zależą od kodów znaków, wobec czego są w pewnej mierze zależne od implementacji (choć standard nakłada pewne rygory na to, co może zrobić porównywanie termów).

Wejście, wyjście. Systemy Prologu dawniej zachowywały się znacznie inaczej niż obecnie jeśli chodzi o operacje wejścia i wyjścia z plików; nie zawsze używane były do tego strumienie. Wewnętrzna struktura nazw plików jest zawsze zależna od implementacji.

Dyrektywy. Poszczególne implementacje mają różne dyrektywy wpływające na sposób ładowania programów.

Ładowanie programów. Standard nie mówi nic na temat, jakie predykaty powinny być dostępne do ładowania (konsultowania) programów, więc tutaj mogą występować różnice między implementacjami.

Arytmetyka. W książce niniejszej nie używaliśmy zbyt często arytmetyki. Standard Prologu przewiduje zbiór funktorów, których używa się w wyrażeniach arytmetycznych (używaliśmy jedynie niewielkiego podzbioru). Implementacje mogą różnie traktować liczby całkowite i zmiennoprzecinkowe, różnie obsługiwać błędy i tak dalej.

Kontrola wystąpień. Standard mówi, że programy Prologu nie powinny móc tworzyć termów cyklicznych (rozdział 10.) W niektórych implementacjach do termów cyklicznych stworzono specjalne narzędzia, ale w celu zachowania przenośności należy z tych możliwości zrezygnować.

Definicje wybranych predykatów standardowych

Aby ułatwić Czytelnikowi tworzenie własnego pliku zgodności, w tym podrozdziale podamy proste definicje predykatów wbudowanych standardowego Prologu, których używaliśmy w niniejszej książce. Wiele implementacji Prologu obsługuje przynajmniej podstawowe predykaty wbudowane, dzięki czemu można ich użyć do definiowania innych potrzebnych predykatów.

W poniższych definicjach ograniczyliśmy się do podzbioru predykatów Clocksina i Mellisha, więc nie możemy pozostać wierni do końca standardowi. W szczególności nie możemy zapisać obsługi błędów, wobec czego niektóre definicje mogą zawieść, jeśli według standardu w danej sytuacji powinien wystąpić błąd. Poniższe definicje powinny wystarczyć w większości poprawnych zastosowań, ale mogą zachowywać się w nieprzewidziany sposób, jeśli zostaną użyte nieprawidłowo.

W przypadku definiowania predykatów pomocniczych, ich nazwy zaczynamy od \$\$, aby zminimalizować ryzyko konfliktu z predykatami już istniejącymi.

Oto zestawienie podanych dalej definicji. Programy te można znaleźć w formie czytelnej dla komputera pod adresem <http://www.dai.ed.ac.uk/homes/chrism/pinsp>.

Predykat standardowy	Zdefiniowany za pomocą predykatów niestandardowych
atom_chars/2	name/2
number_chars/2	name/2
get_char/1	get0/1, name/2
put_char/1	put/1, name/2
dynamic/1	(brak)
close/1	seeing/1, see/1, seen/0, telling/1, tell/1, told/0
current_input/1	seeing/1
current_output/1	telling/1
open/1	seeing/1, see/1, telling/1, tell/1
set_input/1	see/1
set_output/1	tell/1
write_canonical/1	display/1
\+/1	not/1
number/1	integer/1
@=</2, @>/2, @==/2, @</2	name/2

Przetwarzanie znaków

```
% atom_chars/2
%
% Zamiana atomów na listy znaków i odwrotnie
%
% Znane problemy:
%   zamiast generować błąd, zawodzi
%   przekształca ciąg znaków zawierających jedynie cyfry
%   na liczbę zamiast na atom

atom_chars(Atom,Chars) :-
    var(Atom), nonvar(Chars), !,
    '$$collect_codes'(Chars,Codes),
    name(Atom,Codes).
atom_chars(Atom,Chars) :-
    name(Atom,Codes),
    '$$collect_codes'(Chars,Codes).

'$'collect_codes'([_Ch|_Chs],[_Co|_Cos]) :-
    (nonvar(Chs): nonvar(Cos)), !,
    name(Ch,[Co]),
    '$$collect_codes'(Chs,Cos).
'$'collect_codes'([],[]).
```

```
% number_chars/2
%
% Przekształca liczbę na listy znaków i odwrotnie
```

```
%
% Znane problemy:
%   zamiast generować błąd, zawodzi
%   z nie-liczb generuje znaki

number_chars(Num,Chars) :-
    var(Num), nonvar(Chars), !,
    '$$collect_codes'(Chars,Codes),
    name(Num,Codes).
number_chars(Num,Chars) :-
    nonvar(Num),
    name(Num,Codes),
    '$$collect_codes'(Chars,Codes).

% get_char/1
%
% Pobranie pojedynczego znaku
%
% Znane problemy:
%
%   Konieczna jest aktualizacja definicji '$$end_of_file_code' tak,
%   aby zapisać znak oznaczający koniec pliku.

get_char(Char) :- get0(Code), '$$code_to_char'(Code,Char).

'$code_to_char'(Code,Char) :-
    '$$end_of_file_code'(Code), !, Char=end_of_file.
'$code_to_char'(Code,Char1) :- name(Char,[Code]),
    '$$name_to_atom'(Char,Char1).

% niektóre wersje 'name' tworzą liczby, zaś trzeba zapewnić, że
% tworzony będzie atom.
'$name_to_atom'(0,'0') :- !.
'$name_to_atom'(1,'1') :- !.
'$name_to_atom'(2,'2') :- !.
'$name_to_atom'(3,'3') :- !.
'$name_to_atom'(4,'4') :- !.
'$name_to_atom'(5,'5') :- !.
'$name_to_atom'(6,'6') :- !.
'$name_to_atom'(7,'7') :- !.
'$name_to_atom'(8,'8') :- !.
'$name_to_atom'(9,'9') :- !.
'$name_to_atom'(X,X).

'$end_of_file_code'(-1).

% put_char/1
%
% Wyprowadzenie pojedynczego znaku
%
% Znane problemy:
%
%   Zawodzi, jeśli podany zostanie atom nie będący znakiem (powinien
%   zostać wygenerowany błąd).

put_char(X) :- name(X,[C]), put(C).
```

Dyrektywy

```
% dynamic/1
%
% Deklaruje zbiór predykatów jako dynamiczne
%
% Znane problemy:
%
%   Niec nie robi. To oznacza przynajmniej, że dyrektywy dynamic
%   nie będą powodowały błędów, aczkolwiek dyrektywa w obecnej
%   postaci nie pozwala czynić wybranych predykatów dynamicznymi.

?- op(1200,fx,':-').
?- op(1100,fx,dynamic).

dynamic(_).
```

Wejście i wyjście strumieniowe

```
% Poniższe predykaty muszą być używane razem.
% Predykaty te zakładają, że prdykat
%
% '$$open'(Filename,Mode)
%
% (Filename to atom z nazwą pliku, Mode to 'read' (odczyt)
% lub 'write' (zapis)) jest aktualizowany dynamicznie, aby
% uwzględnić, które pliki są obecnie otwarte.
%
% Strumień jest zapisywany jako term '$$stream'(F), gdzie F jest
% atomem z nazwą pliku. Ta sama struktura używana jest do strumieni
% user_input oraz user_output, gdzie F = user_input lub user_output.
%
% Znane problemy związane z poniższą grupą definicji:
%
%   W nazwie strumienia używana jest nazwa pliku - konwencja ta
%   nie może być w pełni stosowana w standardowym Prologu.
%   Dla danego pliku może otworzyć tylko jeden strumień wejścia
%   i wyjścia.
%   Nie można otworzyć plików o nazwach called user_input
%   i user_output

:- dynamic('$$open'/2).

% close/1
%
% Zamknij aktualnie otwarty strumień.
%
% Znane problemy:
%
%   Zawodzi, jeśli strumień nie jest otwarty (powinien
%   zgłaszać błąd).
%   Zamyka plik do czytania i pisania (niezależnie od tego,
%   do czego plik był otwarty)
```

```

close(user_input) :- !.
close(user_output) :- !.
close('$$stream'(user_input)) :- !.
close('$$stream'(user_output)) :- !.
close('$$stream'(File)) :-
    '$$open'(File,_,!).
    '$$closefiles'(File).

'$$closefiles'(File) :-
    retract('$$open'(File,Mode)),
    '$$closefile'(File,Mode),
    fail.
'$$closefiles'(_).

'$$closefile'(File,read) :- !.
    seeing(Current),
    see(File), seen,
    see(Current).
'$$closefile'(File,write) :-
    telling(Current),
    tell(File), told,
    tell(Current).

% current_input/1
%
% Sprawdza, jaki jest aktualny strumień wejściowy.

current_input('$$stream'(F)) :- seeing(F).

% current_output/1
%
% Sprawdza, jaki jest aktualny strumień wyjściowy.

current_output('$$stream'(F)) :- telling(F).

% open/3
%
% Otwiera plik do czytania lub pisania.
%
% Znane problemy:
%

open(File,read,'$$stream'(File)) :- !.
    '$$close_if_open'(File,read), % plik już otwarty - zamknij
    seeing(Old), % zapamiętaj aktualne wejście
    see(File), % ponownie otwórz
    assert('$$open'(File,read)),
    see(Old). % ale nie zmieniaj aktualnego wejścia
open(File,write,'$$stream'(File)) :-
    '$$close_if_open'(File,write), % plik już otwarty - zamknij
    telling(Old), % zapamiętaj aktualne wyjście
    tell(File), % otwórz plik ponownie
    assert('$$open'(File,write)),
    tell(Old). % ale nie zmieniaj aktualnego wyjścia

'$$close_if_open'(File,Mode) :-
    retract('$$open'(File,Mode)). !.

```

```

    '$$closefile'(File,Mode).
    '$$close_if_open'(_,!).

% set_input/1
%
% Zmienia bieżące wejście

set_input(user_input) :- !.
    see(user).
set_input('$$stream'(user_input)) :- !.
    see(user).
set_input('$$stream'(F)) :-
    '$$open'(F,read), !.
    see(F).

% set_output/1
%
% Zmienia bieżące wyjście

set_output(user_output) :- !.
    tell(user).
set_output('$$stream'(user_output)) :- !.
    tell(user).
set_output('$$stream'(F)) :-
    '$$open'(F,write), !.
    tell(F).

```

Różne

```

% write_canonical/1
%
% Wypisuje term pomijając deklaracje operatorów

write_canonical(X) :- display(X).

% \+/1
%
% Negacja czyli zawodzi

?- op(900,fy,\+).

\+ X :- not(X).

% number/1
%
% Sprawdzenie, czy coś jest liczbą.
% Ograniczenia: nie zawodzi tylko dla liczb całkowitych

number(X) :- integer(X).

% @<, @>, @=<, @=>
%
% Porównywanie termów
%
% Ograniczenia: nie zawodzi tylko dla atomów, zakłada użycie
% zestawu znaków ASCII

```

```
?- op(700,xfx,@<).
?- op(700,xfx,@>).
?- op(700,xfx,@=<).
?- op(700,xfx,@>=).
```

```
X @=< Y :- atom(X), atom(Y), X=Y, !.
X @=< Y :- X @< Y.
```

```
X @> Y :- Y @< X.
```

```
X @>= Y :- Y @=< X.
```

```
X @< Y :- atom(X), atom(Y), name(X,XC), name(Y,YC), '$$aless'(XC,YC).
```

```
'$$aless'([],[_],_) :- !.
'$aless'([C],[_],[_]) :- C<C1, !.
'$aless'([C|Cs],[C|Cs1]) :- '$aless'(Cs,Cs1).
```

Dodatek D

Różne wersje Prologu

Istnieje wiele różnych wersji Prologu przeznaczonych dla różnych środowisk i do różnych celów. Różnice między tymi wersjami częściowo wynikają z różnorodności stosowanych komputerów i systemów operacyjnych. Nie ma dwóch różnych komputerów, na które równie łatwo byłoby pisać wszystkie rodzaje programów, więc twórcy różnych systemów Prologu implementują w nich różne zestawy funkcji pomocniczych. Podobnie ma się sprawa z różnymi systemami operacyjnymi, nawet uruchamianymi na tym samym komputerze. System operacyjny to program, który steruje pracą całego komputera, między innymi zapewnia odpowiedni podział zasobów pomiędzy różnych użytkowników komputera. Niektóre systemy operacyjne pozwalają programiście używać większości możliwości komputera, inne są bardziej restrykcyjne, i to jest kolejne źródło różnic poszczególnych wersji Prologu. Na koniec należy zauważyć, że twórcy różnych wersji Prologu mają różne poglądy na istotność poszczególnych funkcji, konieczny do zrealizowania zakres funkcjonalny i różne wymagania natury estetycznej. W wyniku tego nie ma dwóch takich samych wersji Prologu. Sytuacja ta zapewne nie ulegnie zbyt szybko zmianie, gdyż Prolog stale się rozwija i jest stale rozszerzany o nowe funkcje.

W niniejszej książce omawialiśmy wersję Prologu nie całkiem zgodną z jakąkolwiek istniejącą implementacją czy konkretnym systemem operacyjnym. Staraliśmy się zaprezentować „jądro” Prologu, które jest podobnie zrealizowane we wszystkich systemach, z którymi użytkownicy mogą się spotkać. Czytelnicy, którzy opanują zawarty w tej książce materiał, nie będą mieć problemów z zastosowaniem dowolnej wersji Prologu. W większości wypadków przykłady z książki można od razu uruchamiać. Czasami nieco inna może być składnia i inne niektóre predykaty wbudowane, ale podstawy samego języka pozostają dokładnie takie, jakie tu opisaliśmy.

Najlepszą metodą poznania używanej wersji Prologu jest przeczytanie załączonej dokumentacji. Wystarczy krótkie wprowadzenie dla użytkownika, gdyż osoby znające Prolog jako taki nie powinny mieć wtedy problemów z dostosowaniem swojej wiedzy do konkretnego rozwiązania. W niniejszym dodatku podamy kilka faktów, na które warto zwrócić uwagę, poza tym bardziej szczegółowo omówimy dwa konkretne, dość rozpowszechnione Prologi. Trzeba jednak podkreślić, że większość istniejących wersji Prologu cały czas się zmienia i konieczne jest sięgnięcie do aktualnej dokumentacji. Oto kilka najważniejszych, najbardziej typowych różnic między wersjami Prologu.

ul. Armii Krajowej 4, 50-150 Kraków
tel. (012) 638-65-77 tel.fax (012) 637-33-47
wpis do Rejestru MEN nr 55 z dnia 11.05.1999r
Krajowy Rejestr Sądowy 15401115-12007-27006-06
NIP 677-17-58-100 REGON 350614846

Składnia

Każdy ma własne zdanie na temat tego, jaka składnia byłaby najbardziej naturalna i najwygodniejsza w użyciu. Na szczęście Prolog ma składnię bardzo prostą, więc nie ma tu miejsca na zbyt wielkie różnice. Jednym z diskutowanych zagadnień jest sposób rozróżniania zmiennych od atomów. Używaliśmy tutaj nazw zmiennych zaczynających się wielkimi literami oraz atomów zaczynających się literami małymi. Poza tym pozwoliliśmy, aby atomy były ciągami symboli, jak „*”, „.” czy „=”. W niektórych wersjach Prologu przyjęto odwrotną konwencję użycia wielkich i małych liter. Czasami nazwy zmiennych zaczynają się wyróżnionym znakiem, na przykład `_OSOBA` czy `*OSOBA`. Było to przydatne rozwiązanie w starszych systemach, w których nie były rozróżniane małe i wielkie litery. Inne różnice mogą występować w sposobie zapisu klauzul: sposobie oddzielenia głowy od treści i sposobie rozdzielania celów w treści. Poza tym inny może być sposób wyróżniania zapytań. Zamiast `:-`, `.` i `?` mogą być używane inne atomy. Czasami zapis może być nieco bardziej skomplikowany. W pierwszych systemach głowa i cele klauzuli były umieszczane kolejno po sobie, przy czym głowa była poprzedzona symbolem plusa, a poszczególne cele były poprzedzane minusem. Tak więc można zetknąć się na przykład z takimi zapisami klauzul:

```
wuj(X,Z) :- rodzic(X,Y), brat(Y,Z).
Wuj(x,z) <- Rodzic(x,y) & Brat(y,z).
WUJ(_X,_Z) :- RODZIC(_X,_Y), BRAT(_Y,_Z).
+WUJ(*X,*Z) -RODZIC(*X,*Y) -BRAT(*Y,*Z).
{(WUJ X1 X3) (RODZIC X1 X2) (BRAT X2 X3)}
```

Ograniczenia implementacji

Z uwagi na to, że różne komputery różnie obsługują pamięć, twórcy poszczególnych wersji Prologu mogą mieć kłopoty z zapewnieniem możliwości nieograniczonego wzrostu pewnych struktur. Przykładowe cechy zwykle różniące komputery to rozmiar liczb całkowitych, dostępność liczb zmiennoprzecinkowych, maksymalna liczba argumentów funktora, maksymalna liczba znaków w atomie, maksymalna liczba klauzul predykatu i tak dalej.

Cechy środowiska

Z uwagi na różnice między systemami operacyjnymi, niektóre wersje Prologu mogą pozwalać przerywać działanie programu, edytować pliki bez utraty stanu środowiska Prologu, jednocześnie uruchamiać szereg programów prologowych, pobierać dane z urządzeń specjalnych i do takich urządzeń je wysyłać. Jednak żadna z tych cech nie jest na tyle uniwersalna, aby dostępna była we wszystkich implementacjach. Tak jak w przypadku dodatkowych możliwości Prologu, używany system operacyjny zwykle przypisuje specjalne znaczenie pewnym kombinacjom klawiszy — na przykład różne mogą być znaki „końca pliku”, które należy podać kończąc dane wprowadzane przez `consult(user)`. Inne klawisze mogą zostać zinterpretowane jako żądanie podania stanu systemu czy żądanie modyfikacji wprowadzonych wcześniej danych. Tego typu cechy nie należą do samego Prologu, ale są ważne, gdyż od nich zależy sposób korzystania z systemu.

Kompilacja

Większość wersji Prologu zapamiętuje klauzule w postaci zbliżonej do tej, w jakiej je wprowadzono, czyli jako tekst. Kiedy używana jest jakaś klauzula, system musi ją przejrzeć i określić, co należy z nią zrobić. Tego typu systemy nazywamy *interpreterami*. Inna możliwość to przekształcenie w systemie klauzul na instrukcje, które mogą być wykonywane przez komputer bezpośrednio — wtedy mamy do czynienia z *kompilatorem*. Korzystając z kompilatora, możemy uruchamiać programy bezpośrednio, bez konieczności każdorazowej ich interpretacji. Wobec tego programy takie zwykle działają szybciej. Z drugiej jednak strony tekstowa forma programu nie jest po kompilacji dostępna, więc zwykle brak pewnych informacji przydatnych przy usuwaniu błędów — przykładowo, nie zawsze system pozwala wyświetlić klauzule danego predykatu. W niektórych systemach można wybrać między kompilacją a interpretacją. Wtedy trzeba starannie rozważyć zalety i wady obu rozwiązań.

Predykaty wbudowane

Zasadnicza część Prologu jest taka sama niemalże w każdym systemie, jednak różne mogą być predykaty wbudowane. Czasami po prostu dołączane są dodatkowe predykaty, które pozwalają lepiej wykorzystać możliwości używanego komputera. Czasami predykaty realizujące te same funkcje mogą nieznacznie różnić się w sposobie działania. W systemach Prologu wystarcza istnienie albo predykatów `functor` i `arg`, albo `=..`. Wynika to stąd, że pierwsze dwa predykaty można zdefiniować za pomocą trzeciego i odwrotnie. Zwykle trzeba rozróżnić, jak typowe możliwości Prologu można uzyskać w wykorzystywanej jego implementacji. Niektóre systemy zawierają *biblioteki*, w których znajduje się szereg przydatnych programów oraz dodatkowe możliwości, rozszerzające funkcje oferowane przez predykaty wbudowane. Przykładowo, reguły gramatyki mogą być wbudowane w jądro systemu, ale mogą być też obsługiwane przez specjalnie dostosowane do tego biblioteki.

Moduły

Wiele wersji Prologu zawiera własne narzędzia pozwalające organizować duże programy i biblioteki tak, aby uniknąć konfliktów nazw między różnymi częściami takiego dużego programu. Narzędzia te pozwalają dzielić programy na moduły. Każdy moduł ma własną przestrzeń nazw predykatów, dzięki czemu można decydować, które predykaty będą widoczne. Predykat domyślnie widoczny jest jedynie w tym module, w którym go zdefiniowano. Można jednak predykat zadeklarować jako globalny (*public*), co oznacza, że będzie on widoczny we wszystkich modułach, które go zaimportują. Zwykle istnieją dwa moduły predefiniowane, `prolog` i `user`. Wszystkie predykaty wbudowane należą do modułu `prolog` i są automatycznie importowane do wszystkich nowych modułów. Moduł `user` to moduł, w którym znajdują się wszystkie predykaty definiowane przez użytkownika, dla których nie wskazano innego modułu, więc, jak widać, modułów możemy używać nie mając nawet pojęcia o ich istnieniu.

Usuwanie błędów

Nadal prowadzone są badania mające na celu sprawdzenie, jakie narzędzia do usuwania błędów byłyby najbardziej przydatne w Prologu. Obecnie w różnych wersjach Prologu dostępne są różne narzędzia. Jednak zagadnienia omówione w rozdziale 8. będą dla czytelników wystarczające, aby rozpoznać wszelkie inne mechanizmy stosowane w różnych wersjach systemu.

Dodatek E

Dialekt edynburski

W tym dodatku krótko opisujemy Prolog zrealizowany przez Davida Warrena, Fernanda Pereirę i Luisa Byrda na Uniwersytecie w Edynburgu. Implementacja ta stała się *de facto* standardem, oparte na niej inne wersje Prologu dostępne są na szereg innych komputerów, od najmniejszych mikrokomputerów po największe maszyny klasy *mainframe*. Opisane w tej książce „jądro” Prologu jest zrealizowane także w dialekcie edynburskim, ale w tym dodatku uwzględnimy jedynie znaczące odstępstwa od wzorca. Najpierw pokażemy przykładową sesję, a potem bardziej szczegółowo omówimy różnice. Na koniec wskażemy dialekty pochodzące od Prologu z Edynburga.

Przykładowa sesja

Oto jak wygląda przykładowa sesja z opisywanym dialektem Prologu. W tym przykładzie pokażemy dokładną postać informacji wyświetlanych na terminalu. Zamieścimy dodatkowe komentarze, by było jasne, co się w danej chwili dzieje.

Uruchamiamy monitor systemu operacyjnego TOPS-10 i uruchamiamy Prolog.

```
r prolog
Prolog-10 version 3.3 Copyright (C) 1981 by D. Warren, F. Pereira and L. Byrd
| ?- lubi(X,Y).
no
```

Nagłówek może być oczywiście nieco inny w przypadku użycia innej wersji. Znaki | ?- to symbol zachęty — Prolog oczekuje na zapytania. Zadaliliśmy zapytanie i uzyskaliśmy odpowiedź *no*. Nic dziwnego, jako że w bazie danych nie ma żadnych faktów. Załóżmy, że mamy plik *test.pl* i chcemy go wczytać:

```
| ?- ['test.pl'].
test.pl consulted 58 words 0.01 sec.
yes
```

Zadaliliśmy zapytanie składające się z nazwy pliku (jako atomu) w nawiasie kwadratowym. W tym pliku mamy informacje o tym, kto kogo lubi.

```
| ?- lubi(jan,bertrand).
no
| ?- lubi(jan,albert).
no
```

Zadaliśmy kilka zapytań o to, kto kogo lubi. Prolog nie mógł uzgodnić żadnego z tych celów.

```
| ?- listing(lubi).
lubi(jan,alfred).
lubi(alfred,jan).
lubi(bertrand,jan).
lubi(dawid,bertrand).
lubi(jan,_1) :- lubi(_1,bertrand).
yes
```

Aby sprawdzić, jakie mamy klauzule predykatu `lubi`, zadaliśmy zapytanie zawierające predykat wbudowany `listing`. W tym wypadku zaprezentowane zostały wszystkie klauzule `lubi`. Zauważmy, że wszystkie nieukonkretnione zmienne Prolog wyświetlił jako liczbę poprzedzoną podkreśleniem. Klauzula z końca bazy danych została zapisana w pliku jako:

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

Teraz dalej:

```
| ?- lubi(jan,X).
X = alfred ;
X = dawid ;
no
```

Zażądaliśmy alternatywnych rozwiązań zapytania, gdyż wprowadziliśmy średnik (;) i *ENTER*. Istnieją dwa możliwe rozwiązania.

```
| ?- lubi(X,Y).
X = jan, Y = alfred ;
X = alfred, Y = jan ;
X = bertrand, Y = jan ;
X = dawid, Y = bertrand ;
X = jan, Y = dawid ;
no
```

Tym razem rozwiązanie wymagało podania wartości dwóch zmiennych. Tym razem także żądaliśmy podawania kolejno wszystkich możliwych rozwiązań.

```
| ?- [user].
```

Teraz zażądaliśmy od Prologu wczytywania klauzul z pliku `user`, czyli klauzule będą odczytywane z terminala. Klauzule te będą dodawane na koniec bazy danych. Podczas odczytywania klauzul (a nie zapytań) Prolog jako znak zachęty wyświetla (i) zamiast zwykłego `| ?-`.

```
| lubi(tymoteusz,bertrand).
| user consulted 10 words 0.03 sec.
yes
```

Już po pierwszej klauzuli wcisnęliśmy *Ctrl+Z*, czyli zażądaliśmy zakończenia wczytywania. Moglibyśmy oczywiście wprowadzić więcej faktów i reguł. Tymczasem Prolog zakończył wykonywanie poprzedniego zapytania i ponownie zgłosił swoją gotowość do odbioru dalszych zapytań.

```
{ ?- lubi(jan,X).
X = alfred ;
X = dawid ;
X = tymoteusz ;
no
```

Znów zażądaliśmy podania wszystkich rozwiązań. Nowa klauzula spowodowała podanie dodatkowej odpowiedzi, której nie było, kiedy poprzednio zadaliśmy to samo zapytanie.

```
| ?- lubi(bertrand,Y).
Y = jan
yes
```

Teraz po otrzymaniu pierwszej odpowiedzi po prostu wcisnęliśmy *Enter*. Wykonanie zapytania zostało zakończone sukcesem i Prolog oczekuje na następne zapytanie.

```
| ?- core 36864 (7680 lo-seq + 29184 hi-seq)
heap 2560 = 1573 in use + 987 free
global 1177 = 16 in use + 1161 free
local 1024 = 16 in use + 1008 free
trail 511 = 0 in use + 511 free
0.36 sec. runtime
```

W odpowiedzi na ostatnie `| ?-` wcisnęliśmy *Ctrl+Z*, co oznacza, że chcemy zakończyć sesję. Prolog wyświetlił nieco informacji statystycznych i powrócił do monitora TOPS-10. Kompletny przebieg sesji został zarejestrowany w pliku *prolog.log* w katalogu głównym.

Składnia

Składnia edynburskiego dialektu Prologu jest w zasadzie taka sama, jak opisana w niniejszej książce. Składnia ta jest nieco bardziej pobłażliwa, jeśli chodzi o to, co rozumie się przez poprawne atomy i zmienne. Wszystkie przykłady pokazane w tej książce powinny zadziałać bez żadnych zmian. Trzeba pamiętać o pewnej różnicy związanej z operatorami o wysokim priorytecie. Jako że przecinek (,) sam jest operatorem, aby uniknąć niejednoznaczności zapisu, wszystkie terminy z operatorami o takim samym lub wyższym priorytecie trzeba zapisywać w nawiasach. Dzięki temu mamy gwarancję, że na przykład

```
cos(a,b,c)
```

zawsze będzie zinterpretowane jako struktura z trzyargumentowym funktorem `cos`, a nie — na przykład —

```
cos(a,'.(b,c))
```

czy

```
cos('.(a,','(b,c)))
```

Gdybyśmy chcieli zapisać ostatni termin, moglibyśmy użyć zapisu

```
cos((a,b,c))
```


Reguła dotycząca operatorów o wysokich priorytetach dotyczy jedynie kilku operatorów, jak `:-` czy `;`. Oznacza to, że zapis

```
?- retract(rodzic(A,B) :- ojciec(A,B)).
```

jest w dialekcie edynburskim niepoprawny składniowo; konieczne jest użycie dodatkowego nawiasu.

W dialekcie edynburskim istnieje składnia alternatywna, używana wtedy, gdy terminal lub system operacyjny nie rozróżnia wielkich i małych liter. Zmienne wówczas zaczynają się podkreśleniem. Prolog przechodzi na tę alternatywną formę zapisu za sprawą predykatu wbudowanego `NOLC`, natomiast predykat `LC` pozwala powrócić do tradycyjnego zapisu.

Na koniec jeszcze jedna uwaga. W dialekcie edynburskim Prologu kropka (funktor dwuargumentowy) nie jest predefiniowana jako operator. Można ją zdefiniować samemu, ale nie jest to konieczne, gdy listy i tak zawsze zapisuje się przy użyciu właściwej im notacji.

Ograniczenia systemu

Prolog edynburski pierwotnie był zaimplementowany na maszynie DECsystem-10, więc nie zawiera zbyt wielu ograniczeń, które byłyby odczuwalne podczas normalnej pracy. Priorytety operatorów muszą mieścić się między 1 a 1200 i są one zbliżone do podanych w niniejszej książce. Liczby całkowite muszą należeć do zakresu od -131 072 do 131 071, choć w obliczeniach pośrednich mogą wystąpić liczby większe. Liczby rzeczywiste (zmiennoprzecinkowe) nie są obsługiwane.

Środowisko

Prolog działający w środowisku DECsystem-10 oferuje bardzo przydatną możliwość rejestrowania przebiegu sesji. Normalnie system kopiuje większość danych z monitora do pliku *prolog.log*. Po zakończeniu sesji można w nim sprawdzić, jak sesja przebiegała. Plik *prolog.log* zawiera informacje o tym, co i kiedy robił program oraz jak został zmodyfikowany. Istnieją predykaty wbudowane pozwalające włączać i wyłączać rejestrowanie sesji.

Działanie programu możemy przerwać wciskając kombinację *Ctrl+C*. System prosi wtedy o podanie żądanej akcji: mogą to być *break*, *continue* (dalsze wykonywanie programu), *exit* (koniec działania Prologu), *trace* i *notrace*. W przypadku dwóch ostatnich opcji wykonanie programu jest kontynuowane, tylko zmieniany jest zakres śledzenia wykonania programu. Opcja *break* zawiesza wykonanie programu i uruchamia nową instancję Prologu, z której można korzystać. Po zakończeniu korzystania z tej nowej instancji, wykonanie programu jest kontynuowane.

W DECsystem-10 znakiem końca pliku jest *Ctrl+Z*. Jego wpisanie powoduje zakończenie działania Prologu, zakończenie *break* lub zakończenie *consult* — zależnie od tego, w jakim stanie jest system. Po natknięciu się na koniec pliku, predykat wbudowany *read* dopasowuje swój argument do atomu *end_of_file*.

W Prologu na system DECsystem-10 istnieje szereg ułatwień pozwalających skrócić czas wielokrotnego wczytywania programów. Możliwe jest zapisanie „stanu” Prologu, w tym aktualnej zawartości bazy danych, w pliku, tak że stan można odtworzyć znacznie szybciej niż byłoby to możliwe przy ponownym wczytaniu programu. Poza tym Prolog ten automatycznie odczyta wszelkie dane z pliku *prolog.ini* na początku sesji, przed pobraniem jakichkolwiek informacji z terminala.

W przypadku wystąpienia błędu, Prolog DECsystem-10 wyświetla komunikat o błędzie (z informacją o charakterze błędu). Większość błędów powoduje, że cel, w którym wystąpiły, zawodzi, poza tym program jest wykonywany dalej. Jednak niektóre, najpoważniejsze błędy spowodują zamknięcie wykonywanych programów i poproszenie użytkownika o następne zapytanie.

Kompilacja

Prolog DECsystem-10 umożliwia kompilowanie wybranych klauzul. Znacząco zmniejsza to zużycie pamięci i zwiększa szybkość działania programu. Predykaty wbudowane zawierają funkcję analogiczną do *consult*, ale klauzule ze skompilowanego pliku nie są już interpretowane. Wydajność skompilowanych klauzul możemy poprawić stosując deklaracje trybu, które pozwalają wskazać, jak dane klauzule będą używane (tj. które argumenty kiedy będą ukonkretniane). Istnieją pewne ograniczenia na to, które klauzule można kompilować. Poza tym konieczne jest zapewnienie takiej kolejności deklaracji, aby prawidłowo zadziałała mieszanka klauzul kompilowanych i interpretowanych.

Predykaty wbudowane

Prolog DECsystem-10 zawiera wszystkie omawiane dotąd predykaty wbudowane, poza tym prawidłowo zadziałają reguły gramatyki interpretowane przez *consult*. W tym punkcie opiszemy niektóre odstępstwa od opisu.

Efekt działania predykatu *display* zawsze jest wypisanie argumentu na monitorze, a nie do aktualnego strumienia wyjściowego.

Opisując wyrażenia algebraiczne powiedzieliśmy, że wyrażenia te są wyliczane wtedy, gdy są drugim argumentem *is*. We wszystkich innych wypadkach struktury typu 2+3 pozostają po prostu sobą.

W Prologu DECsystem-10 jest inaczej, gdyż różne inne predykaty także ewaluują wyrażenia liczbowe pojawiające się jako argumenty. Przykładami są operatory porównania (*<*, *=* itd.) i predykat *put*. Wobec tego poniższy zapis zadziała w Prologu DECsystem-10, ale w standardowym Prologu spowoduje błąd:

```
?- 2+4 < 12*(2+8).
yes
```

Dodatkowy smaczek polega na tym, że lista zawierająca pojedynczą liczbę jest traktowana jako wyrażenie arytmetyczne o wartości jednego elementu tej listy. Wobec tego

```
?- X is [25].
X = 25
yes
```

Z uwagi na opisane tu zasady działania, pojedyncze znaki można podawać mnemonicie:

```
?- put("a"), put("b").
ab
yes
```

(pamiętajmy, że "a" to jednoelementowa lista zawierająca kod znaku a).

Składnia zaprzeczenia. Nie istnieje predykat `not`, zamiast niego używany jest operator przedrostkowy `\+`. Nie istnieje predykat nierówności, `\=`.

Zmienne jako cele. Jest to kwestia tylko i wyłącznie składni. Widzieliśmy już, jak można wywołać cel odpowiadający treści zmiennej Prologu za pomocą `call`. W Prologu DECsystem-10 to samo można zrobić inaczej: zamiast używać celu

```
..., call(X), ...
```

można po prostu podać samą zmienną jako cel:

```
..., X, ...
```

Można też normalnie korzystać z `call`, zresztą podanie samej zmiennej `X` jako celu jest przekształcane na `call(X)` przy użyciu `asserta` czy `assertz`.

Argumenty retract. Z uwagi na trudności związane z użyciem zmiennych jako celów, nieco inaczej podawane są treści klauzul w celach `retract`. Problem polega na tym, że jeśli zadamy zapytanie:

```
?- retract((matka(A,B) :- C)).
```

możemy zażądać od Prologu usunięcia klauzuli dokładnie w postaci:

```
matka(A,B) :- C.
```

ze zmienną jako celem w treści tej klauzuli, ale mogłoby to oznaczać też żądanie usunięcia klauzuli `matka` z dowolną treścią, na przykład:

```
matka(X,Y) :- rodzic(X,Y), kobieta(Y).
```

Aby zapewnić jednoznaczność w tego typu sytuacjach, Prolog DECsystem-10 zawsze zaczyna od zastąpienia zmiennych nieukonkretnionych odpowiadających pojedynczemu lub wielu celom z argumentów `retract` strukturami z `call`. Tak więc zapytanie

```
?- retract((matka(A,B):-C)).
```

zostanie zinterpretowane jako:

```
?- retract((matka(A,B):-call(C))).
```

Gdybyśmy chcieli usunąć pierwszą klauzulę `matka` niezależnie od jej treści, moglibyśmy użyć zapisu

```
?- clause(matka(A,B),C), retract((matka(A,B):-C)).
```

W takim wypadku cel `clause` spowoduje ukonkretnienie zmiennej `C`, dostateczne, aby uniknąć opisanego przekształcenia.

Dodatkowe predykaty wbudowane

Niezależnie od opisanych dotąd predykatów wbudowanych, Prolog DECsystem-10 zawiera szereg innych dodatków.

Formy warunkowe pozwalają tworzyć cele w postaci:

```
..., (lub!(jan,X) -> drewniany(X); plastikowy(X)), ...
```

W przypadku tych skomplikowanych celów chodzi o to, że jeśli „warunek” przed `->` zostanie uzgodniony, to wywołany zostanie cel zza `->`. W przeciwnym razie wywołany zostanie trzeci cel. Wszystkie podane cele mogą być celami Prologu. Warunkowość zachowuje się tak, jakby zdefiniowano ją następująco:

```
?- op(1050,xfy,->)..
?- op(1100,xfy,';').
(X -> Y; Z) :- call(X), !, call(Y).
(X -> Y; Z) :- call(Z).
```

Indeksowana baza danych pozwala powiązać informacje z bazy danych z konkretnymi wartościami, bez konieczności korzystania ze standardowego mechanizmu pobierania danych. Jeśli na przykład chcemy zapisywać informacje o wieku setek ludzi, normalnie zapisywalibyśmy setki klauzul predykatu `wiek`. Jeśli potem chcielibyśmy określić wiek jakiejś osoby, Prolog musiałby przeszukiwać kolejno wszystkie dane. Informacje te normalnie są powiązane z predykatami, jeśli więc predykat ma dużo klauzul, przeszukiwanie może długo trwać. Poindeksowana baza danych pozwala powiązać informacje z nazwami bardziej bezpośrednio.

Możliwość sięgania do przodków. Notację celów przodków omawialiśmy w rozdziale poświęconym usuwaniu błędów. Prolog DECsystem-10 zawiera predykaty wbudowane, które umożliwiają sięganie do przodków z samego programu.

Informacje statystyczne. Prolog DECsystem-10 zawiera predykaty wbudowane, które umożliwiają łatwe sprawdzenie, jak szybko wykonuje się program i ile pamięci potrzebuje.

Usuwanie błędów

Prolog DECsystem-10 zawiera takie funkcje usuwania błędów, jak opisane w niniejszej książce. Poza wspomnianymi predykatami wbudowanymi istnieją predykaty umożliwiające wskazanie, które porty mają być kontrolowane podczas śledzenia.

Literatura

„DECsystem-10 Prolog User's Manual”, Wydział Sztucznej Inteligencji Uniwersytetu w Edynburgu, Edynburg, Szkocja.

„C-Prolog User's Manual”, CAAD Studio, Wydział Architektury Uniwersytetu w Edynburgu, Edynburg, Szkocja. Jest to dokumentacja systemu działającego w systemie operacyjnym Unix.

„Prolog-1 User's Manual”, Expert Systems Ltd, 9 West Way, Oxford, Anglia. Jest to dokumentacja systemu działającego na różnych komputerach, od Z-80 z systemem CP/M po VAX z systemem operacyjnym VMS.

Współczesne systemy Prologu

Wraz z nastaniem epoki stacji roboczych o dużej mocy obliczeniowej, działających pod kontrolą systemów operacyjnych opartych na Uniksie, szczególnie popularne stały się dwie implementacje Prologu. Quintus Prolog nawiązuje do tradycji z Edynburga, zawiera szereg nowych możliwości, jak liczby zmiennoprzecinkowe, kompilacja programów, podział programu na moduły, konsolidacja programów prologowych z programami napisanymi w innych językach oraz szereg dodatkowych predykatów wbudowanych. SICStus Prolog oparty jest na Quintus Prolog, zawiera dodatkowe możliwości przeznaczone do programowania sterowanego danymi i obiektowego. Więcej informacji:

„Quintus Prolog Reference Manual”, Quintus Computer Systems Inc., Mountain View, Kalifornia, USA.

„SICStus Prolog User's Manual”, Szwedzki Instytut Informatyki, PO Box 1263, S-164 28 KISTA, Szwecja.

Dodatek F micro-Prolog

Opiszemy jeszcze możliwości micro-Prologu działającego na komputerach Z-80 pod kontrolą systemu operacyjnego CP/M. micro-Prolog działa także na innych mikrokomputerach, większość poniższych informacji dotyczy także tych innych wersji.

Przykładowa sesja

We wszystkich przykładach micro-Prologu korzystać będziemy ze składni standardowej. Inne rodzaje składni, w tym składnia zgodna z dialektem edynburskim, mogą być w miarę potrzeb ładowane do pamięci.

Oto przykładowa sesja micro-Prologu. Najpierw nakazujemy CP/M uruchomienie Prologu:

```
A>PROLOG
Micro Prolog 3.00 S/N
© 1982 Logic Programming Associates Ltd.
9999 Bytes Free
&?((lub1 x y))
Clause error at (lub1 x y)
```

Znaki & używane są przez micro-Prolog jako symbol zachęty — Prolog informuje, że oczekuje na polecenia. Znak ? oznacza, że chcemy zadać zapytanie. Dalej znajduje się ciąg celów (domyślnie koniunkcja) w nawiasach. Każdy cel podawany jest w nawiasach, najpierw predykat, potem jego argumenty. Zmienne oznaczają się słowami zaczynającymi się od x, y, z, X, Y, Z i dalej mogą występować cyfry. W przypadku powyższego celu zapytaliśmy, czy ktokolwiek lubi kogokolwiek. Jako że nie mamy jeszcze klauzuli lub1, micro-Prolog zgłosił błąd.

```
&.LOAD TEST
```

Ważnym poleceniem jest LOAD; za poleceniem podajemy nazwę pliku. Podanie pokazanego polecenia powoduje załadowanie klauzuli z pliku TEST.LOG i uzupełnienie bazy danych, podobnie jak ma to miejsce w przypadku consult. Teraz możemy zadawać zapytania:

```
&?((lub1 jan bertrand))
?
&?((lub1 jan alfred))
&.
```

Zwróćmy uwagę, że micro-Prolog zgłasza, że zapytanie zawiodło przez wypisanie pytania, zaś jeśli zapytanie nie zawiedzie, wyświetlany jest następny symbol zachęty. Aby sprawdzić, jakie klauzule `lubi` istnieją, używamy polecenia `LIST`. Możemy poprosić o wypisanie całego programu lub tylko klauzul pewnego zbioru predykatów. Sprawdźmy, jakie mamy klauzule `lubi`:

```
&.LIST (lubi)
((lubi jan alfred))
((lubi alfred jan))
((lubi bertrand jan))
((lubi(dawid bertrand))
(lubi jan x)
(lubi x bertrand))
```

Aby uzyskać odpowiedzi na jakieś zapytania, korzystamy z predykatów wbudowanych. `PP` powoduje wyświetlenie przekazanego argumentu, predykat `FAIL` zawodzi.

```
&.?((lubi jan x) (PP x) (FAIL))
alfred
dawid
?
```

Tak oto otrzymaliśmy dwa rozwiązania zapytania „Kogo lubi Jan?”

Jeśli chcielibyśmy bezpośrednio z klawiatury dodać do bazy danych nowe klauzule, nie musimy używać żadnych specjalnych poleceń — po prostu wpisujemy bezpośrednio te klauzule. Składnię klauzul micro-Prologu opiszemy dalej.

```
&.(lubi tymoteusz bertrand))
&.?((lubi jan x) (PP x) (FAIL))
alfred
dawid
tymoteusz
?
```

Aby zakończyć działanie micro-Prologu i wrócić do CP/M, używamy polecenia `QT` z kropką:

```
& QT.
A>
```

Składnia

Składnia micro-Prologu jest całkiem inna niż składnia przedstawiona wcześniej w tej książce, ale łatwo można do niej przywyknąć. Założenie jest takie, że istnieje tylko jeden rodzaj termów: lista. Jeśli chcemy stworzyć term składający się z funktora `f` i czterech argumentów, używamy pięcioelementowej listy: głową jest `f`, pozostałe cztery elementy listy to argumenty. Tak więc term

```
f(a,g(2,3).c)
```

w micro-Prologu zapisujemy jako

```
(f (g 2 3) c)
```

Mamy tutaj też inną składnię list: listy zamykane są w nawiasy, a elementy rozdzielane są spacjami.

Klauzule zapisuje się jako listy termów, w których pierwszym termem jest głowa klauzuli, pozostałe termy to cele, które są połączone koniunkcją. Oto bardziej skomplikowana klauzula:

```
((podmien (z1|z2) (x|y)) (zmien z1 x) (podmien z2 y) )
```

Jest to druga klauzula podmien z rozdziału 3., podrozdział „Odwzorowania”. Pionowa kreska ma takie samo znaczenie, jak w składni standardowej.

Ograniczenia systemu

W micro-Prologu obsługiwane są liczby zmiennoprzecinkowe o dokładności do ośmiu cyfr i wykładniku o zakresie od -127 do 127 (podstawa to 10). Nazwy atomów (w micro-Prologu nazywane „stałymi”) mogą mieć co najwyżej 60 znaków, term nie może zawierać więcej niż 64 zmienne. Ograniczenia te praktycznie nie mają większego znaczenia.

Środowisko

micro-Prolog zawiera szereg ułatwień dla programistów. Predykaty wbudowane `LOAD` i `SAVE` umożliwiają wczytywanie programów z plików i zapisywanie ich do plików. Dopracowany edytor wierszowy i edytor struktury programu umożliwiają modyfikowanie istniejących programów bez konieczności opuszczania środowiska Prolog. Program napisany w micro-Prologu może zostać przerwany przez wciśnięcie `Ctrl+C`.

Predykaty wbudowane

Nazwy i zastosowanie predykatów wbudowanych micro-Prologu znacznie się różnią od tych omawianych w niniejszej książce. Oto krótkie ich omówienie.

Rodzaj termu można sprawdzić za pomocą `NUM`, `CON` i `VAR`, które nie zawiodą odpowiednio dla liczb, stałych (atomów) i zmiennych. Poza tym `SYS` sprawdza, czy stała jest nazwą predykatu wbudowanego, zaś `INT` sprawdza, czy argument jest liczbą całkowitą.

Predykaty do obsługi bazy danych to `ADDCL` (podobny do `assert`), `CL` (podobny do `clause`) i `DELCL` (podobny do `retract`). Predykaty te mogą zawierać dodatkowy argument całkowitoliczbowy, który pozwala łatwo sięgnąć do N-tej klauzuli procedury.

Jedynym rodzajem termów micro-Prologu jest lista, więc predykat typu `=..` jest zbędny. Predykat `STRING` pozwala programiście tworzyć owe atomy i sięgać do wewnętrznej struktury nazw atomów, podobnie jak `name`.

Cele złożone można tworzyć za pomocą predykatów wbudowanych `OR`, `NOT` i `IF`. Predykat `FAIL` zawsze zawodzi, zaś odciecie zapisywane jest jako ukośnik (`/`).

Szereg predykatów to predykaty algebraiczne stosowane zamiast pojedynczego `is`. Predykaty te zaprojektowano tak, aby były możliwie wszechstronne, dzięki czemu cel

```
(SUM x y z)
```

powoduje dopasowanie z do sumy y i x , jeśli x i y są ukonkretnione. Jeśli jednak początkowo ukonkretnione są y i z , a x nie, x otrzyma wartość różnicy $z - y$.

Operacje plikowe w micro-Prologu podobne są w zasadzie do standardowych, ale nie istnieje metoda zapisu bieżącego strumienia wejściowego ani wyjściowego. Aby dane zapisać do pliku lub je z pliku odczytać, korzysta się ze specjalnych predykatów `WRITE` i `READ`, którym podaje się za każdym razem nazwę pliku. Istnieją specjalne predykaty wejścia i wyjścia do obsługi terminala.

micro-Prolog pozwala programiście tworzyć programy składające się z fragmentów nazywanych modułami. Dzięki temu możliwa jest minimalizacja ryzyka konfliktu nazw, poza tym łatwiej jest programistom wymieniać się programami.

Usuwanie błędów

W micro-Prologu istnieje śledzenie, ale trzeba je jawnie załadować (`LOAD`) przed użyciem. Podawane są informacje o stanie realizacji celów, z wyjątkiem predykatów wbudowanych. Podawane są komunikaty z portów `CALL`, `EXIT` i `FAIL` (nazywanych odpowiednio `ENTER`, `FINISH` i `FAIL`). Użytkownik może kontrolować zachowanie portów `CALL`, może wtedy podać `CONTINUE` (dalsze wykonywanie ze śledzeniem), `SKIP` (koniec śledzenie aż do skończenia bieżącego celu), `FINISH` (aktualny cel natychmiast jest uzgadniany) oraz `FAIL` (aktualny cel natychmiast zawodzi).

Literatura

„micro-Prolog: Programming in Logic”, K.L. Clark i F.G. McCabe, Prentice-Hall, 1984.

Skorowidz

A

abort, 183
 aksjomaty, 216
 akumulator, 66, 67
 algorytm, 11
 alfa-beta, 227
 alternatywa celów, 120
 analiza języka naturalnego, 187
 DCG, 202
 drzewo parsowania, 197
 gramatyka, 187, 193
 logika, 202
 notacja reguł gramatycznych, 194
 parser, 199
 parsowanie, 190
 reguły gramatyczne, 190, 196
 aplikacje konsolowe, 92
 append, 89, 178, 186
 arg, 115, 116
 argumenty, 15, 18, 208
 struktury, 115
 arytmetyka, 38, 40
 ASCII, 226
 asserta, 112, 113, 148, 224
 assertz, 112, 113
 atom, 110
 atom_chars, 114, 117
 atomic, 111
 atomy, 32, 37
 end_of_file, 101
 generowanie, 146
 user_input, 103
 user_output, 103

B

badanie w głąb, 222
 baza danych, 15, 17, 145
 dodawanie klauzul, 113
 przeszukiwanie, 18, 130
 znaczniki, 21
 bezpieczne programowanie, 165
 biblioteki, 194
 bieżący strumień
 wejściowy, 102
 wyjściowy, 102
 błędy
 poprawianie, 184
 składniowe, 168
 sterowania, 168
 typograficzne, 96
 usuwanie, 165
 zapełnione definicje, 170
 break, 183
 budowanie struktur, 115

C

call, 85, 120
 CALL, 171, 172, 177
 cele, 21, 42
 alternatywa, 120
 generatory, 86
 koniunkcja, 42, 119
 nawracanie, 45
 przodkowie, 180
 rodzicielskie, 79
 spełnianie, 42
 sprawdzanie, 179
 testery, 86
 złożone, 119
 zmiana sposobu spełnienia, 182