Programowanie w logice Prolog

Piotr Fulmański

Piotr Fulmański[1]
e-mail: fulmanp@math.uni.lodz.pl

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki
Banacha 22, 90-238, Łódź
Polska

Data ostaniej modyfikacji: **18 maja 2009**

Spis treści

Wstęp

Niniejszy podręcznik stanowi sformalizowaną wersję notatek jakie sporządzałem dla siebie prowadząc zajęcia „Programowanie w logice". Cel jaki przyświecał mi, to chęć pokazania, że świat języków programowania nie kończy się na programowaniu obiektowym. Wykorzystanie języka logiki do opisu otaczającej nas rzeczywistości odwzorowywanej w środowisku komputerowym, daje nam zupełnie inną jakość i uczy całkowicie odmiennego spojrzenia na problem „programu" i jego „kodu". I choć przydatność takich języków może być dyskusyjna[1] to jednak wydaje mi się, że ich elementarna znajomość jest konieczna dla osób, które chcą zostać informatykiem.

W prezentowanym materiale główny nacisk położony jest na zdobycie praktycznej umiejętności posługiwania się takim językiem i jego „czucia", niż teorii programowania w logice. Niemniej pewna część materiału poświęcona jest także zagadnieniom natury teoretycznej. Wybranym językiem jest Prolog a wszystkie przykłady zostały przetestowane przy użyciu jednej z jego najbardziej popularnych wersji – SWI-Prolog[2] w wersji 5.6.59.

Ze względu na formę zajęć, całość materiału podzielona została na dwie zasadnicze część: wykładową i ćwiczeniową. Materiał każdej z części powinien wystarczyć do przeprowadzenia semstralnych zajęć. Część wykładowa obejmuje rozdziały **??**–**??** i poświęcona jest następującym zagadnieniom

- a

Część ćwiczeniowa zawiera materiał czy to do samodzielnej pracy, czy też nadający się do przeprowadzenia zajęć na pracowni komputerowej. Zbiór zadań sukcesywnie jest poszerzany, ale nie jest to wcale takie łatwe. Wbrew pozorom trudno jest znaleźć zadania różne (na tyle, aby ich rozwiązanie wymagało przynajmniej w pewnej ich części innego podejścia niż zadania poprzednie) i zarazem na tyle łatwe aby dało się je w prosty sposób wytłumaczyć. Dobrze przy tym aby zadania nie były całkiem abstrakcyjnym manipulowaniem symbolami w stylu „sztuka dla sztuki". Daltego jeśli ktokolwiek czytając ten podręcznik wpadnie na jakieś nowe zadanie albo alternatywne rozwiązanie dla tych już podanych, to z wielką chęcią je zamieszcze.

Na zakończenie pragnę podziękować (nielicznym niestety) studentom, którzy z jednej strony wykazali zainteresowanie przedmiotem, z drugiej zaś wzbogacili ten podręcznik a przede wszystkim:

- Pani Magdalenie Zych za opracownie bardzo wyczerpujących odpowiedzi do pytań jakie stawiam na koniec każdego rozdziału (wykładu).

[1] Wynika to zwykle z problemów związanych z wykorzystaniem języka logicznego w realnej aplikacji.
[2] `www.swi-prolog.org`

- Panu Krzysztofowi Jastrzębskiemu za mozolne i niestrudzone rozwiązywanie zadań jakie stawiałem na ćwiczeniach.

Piotr Fulmański
Łódź, 2008

Rozdział 1

Podstawy

1.1 Zdumiewający początek

Już sama nazwa języka – Prolog – niesie w sobie informację o jego przeznaczeniu. Słowo „prolog" pochodzi bowiem od sformułowania *programmation en logique* co w języku francuskim oznacza właśnie *programowanie w logice*. Prolog został stworzony w 1971 roku przez Alaina Colmeraurera i Phillipe'a Roussela. Choć jego teoretyczne podstawy stanowi rachunek predykatów pierwszego rzędu, to jednak ogranicza się tylko do klauzul Horna (o czym więcej powiemy w rozdziale **??**). Jeszcze w pierwszych latach XXI wieku był bardzo chętnie używany w wielu programach związanych z

- logiką matematyczną (automatyczne dowodzenie twierdzeń);
- przetwarzaniem języka naturalnego;
- symbolicznym rozwiązywaniem równań;
- sztuczną inteligencją;
- przechowywaniem i przetwarzaniem danych.

I choć powoli jego miejsce zajmują wygodniejsze narzędzia jak na przykład silniki regułowe (o czym powiemy w rozdziale **??**), to wciąż stanowi wspaniały model dydaktyczny.

Najważniejszą i zarazem często najbardziej zaskakującą i zdumiewającą rzeczą związaną z Prologiem jest to, że

Pisanie programu w Prologu nie polega na opisywaniu algorytmu!

Jak to? Przecież od lat, z mozołem i w wielkim trudzie wpajano nam do głowy, że zanim zaczniemy pisać program to musimy ułożyć odpowiedni algorytm. Gdy się już tego nauczyliśmy i przyjęliśmy za pewnik, nagle okazuje się, że wcale tak nie musi być. Niestety bardzo trudno jest przestać myśleć algorytmicznie o problemie. Jest to silniejsze od nas, bo tak nas nauczono. Tym czasem w Prologu istotne jest coś zupełnie innego. Oto bowiem zamiast opisywać algorytmu, opisujemy **obiekty** związane z problemem i **relacje** pomiędzy tymi obiektami. Stąd właśnie Prolog często bywa określany jako język **opisowy** i **deklaratywny**. Oznacza to, że implementując rozwiązanie jakiegoś problemu nie podajemy **jak** go rozwiązać (jak to ma miejsce w imperatywnych językach programowania tj. C lub Java) ale określamy **czego on dotyczy** używajac do tego **faktów** i **reguł**. Rolą Prologu jest **wywnioskowanie** rozwiązania na podstawie podanych przez nas informacji.

1.2 Obiekty i relacje

Programowanie w Prologu polega na „definiowaniu" obiektów i określaniu wiążących ich relacji. Odmienne jednakże od tradycyjnego (tzn. występującego w klasycznym programowaniu obiektowym) jest pojmowanie obiektu.

1.2.1 Obiekt „klasyczny"

Pomimo, iż zajmujemy się Prologiem to aby uświadomić sobie co jest w nim tak odmiennego od innych języków, poświęćmy trochę miejsca na przypomnienie, czym jest „klasyczny" obiekt, znany z takich języków jak np. C++ czy Java. W tym ujęciu **obiekt** to podstawowe pojęcie wchodzące w skład paradygmatu obiektowości w analizie i projektowaniu oprogramowania oraz w programowaniu. Jego koncepcja ma ułatwić cyfrową reprezentację realnych obiektów. Czym charakteryzują się rzeczywiste obiekty?

Obiekty jakie otaczją nas w rzeczywistym świecie posiadają dwie istotne cechy: **stan** w jakim w danej chwili się znajdują[1] oraz **zachowanie** jakie dla nich jest typowe. I tak psy mają swój stan (kolor, wagę, są głodne lub najedzone...) oraz zachowanie (szczekanie, bieg, leżenie, merdanie ogonem...). Także telewizory mają swój stan (włączony lub wyłączony, głośność, numer programu...) oraz zachowanie (zmiana głośności lub programu, włączenie, wyłączenie...). Prawidłowe określenie stanu i zachowania ma bardzo duże znaczenie dla dalszego sposobu i komfortu „obsługi" obiektu w programie.

Obiekt w ujęciu języków obiektowych jest bardzo podobny do obiektów świata rzeczywistego: także posiada stany (cechy) i przypisane jemu „zachowanie". Taki obiekt przechowuje swoje stany w zmiennych zwanych **polami** a wpływ na jego zachowanie mamy za pośrednictwem funkcji, nazywanych też **metodami**. Metody operując na polach dostarczają jednocześnie podstawowych mechanizmów komunikacji obiekt–obiekt. Zasada ukrywania stanów wewnętrznych obiektu i interakcja z nim tylko poprzez dobrze zdefiniowany zbiór metod znana jest pod nazwą **enkapsulacji danych** i stanowi fundamentalną zasadę programowania zorientowanego obiektowo.

Każdy utworzony przez programistę obiekt jest instancją pewnej **klasy**. W tym ujęciu, klasa zdefiniowana przez programistę, staje się nowym typem, który może być używany na równi z typami wbudowanymi. Jako przykład takiej klasy rozważmy klasę `Punkt` umożliwiającą utworzenie dowolnej ilości obiektów opisujących (różne) punkty.

```
class Punkt
{
private:
  double x;
  double y;

public:
  Punkt(double x, double y);
  void Przesun(Wektor w);
};
```

Jeśli ponad to założymy, że mamy dostępną klasę `Wektor`, wówczas staje się możliwe napisanie kodu jak poniżej

```
Punkt p(1.0,2.0);
```

[1] Lub **cechy** jakie posiadają.

```
Wektor w(2.5,-1.5);

p.Przesun(w);
```

Posumowując:

- Programowanie zorientowane obiektowo (*ang. OOP, object-oriented programming*) to paradygmat (sposób) programowania posługujący się pojęciem **obiektu** jako metody reprezentacji danych w programie.
- Każdy obiekt, będący instancją pewnej klasy, posiada zbiór cech (będących zmiennymi pewnych typów) go opisujących oraz zbiór metod (funkcji) których wywołanie na rzecz tego obiektu ma sens.
- W tym ujęciu, klasa zdefiniowana przez programistę, staje się nowym typem, który może być używany na równi z typami wbudowanymi.
- Z OOP nierozerwalnie związane są pojęcia enkapsulacji danych, dziedziczenia, polimorfizmu.

Wiedząc czym są, najlepiej chyba wszystkim znane, obiekty z języków OOP wystarczy teraz powiedzieć, że obiekt Prologowy **nie** jest takim właśnie obiektem nawet w najmniejszej części. Jaki więc jest?

1.2.2 Obiekt w Prologu

Obiekt w sensie Prologu jest czymś, co możemy nazwać bytem. Nie definiujemy **z czego** się on składa i co można z nim zrobić (co ma miejsce w OOP), ale **jaki jest**. Dodatkowo, dla każdego obiektu definiujemy **relacje** jakim obiekt ten podlega. Przy pomocy obiektów opisujemy interesujący nas wycinek świata. Działanie programu prologowego objawia się możliwością stawiania pytań związanych z uprzednio opisanym światem.

Najprostszym sposobem opisu świata (problemu), jest podanie faktów z nim związanych, jak na przykład:

```
ciezszy(pomarancz,jablko).
ciezszy(jablko,mandarynka).
ciezszy(arbuz,pomarancz).
ciezszy(jablko,winogrono).
```

Powyższe fakty stwierdzają, że

- `ciezszy(pomarancz,jablko).` – pomarańcz jest cięższa od jabłka,
- `ciezszy(jablko,mandarynka).` – jabłko jest cięższe od mandarynki,
- itd.

Tak więc powyżej określiliśmy kilka obiektów (`pomarancz`, `jablko`, itd) i powiązaliśmy je między sobą relacją `ciezszy` wyrażającą, który z obiektów jest cięższy od innych. Istotne jest to, że nadal nie jest nam znana masa żadnego z obiektów – one po prostu nie posiadają cech.

Okazuje się zdumiewające, że już nawet jeden fakt jest poprawnym (przynajmniej składniowo) programem Prologu. Po „uruchomieniu" takiego programu, możemy zadawać pytania związane z rzeczywistością jaką opisuje

```
?- ciezszy(pomarancz,jablko).
Yes
```

W ten oto sposób otrzymujemy twierdzącą odpowiedz na pytanie: *Czy pomarańcz jest cięższa od jabłka?*. Będąc precyzyjniejszym, to pytanie brzmi: *Czy wiadomo coś na temat tego, że pomarańcz jest cięższa od jabłka?*. Jest to istotne rozróżnienie, gdyż wówczas reakcję

```
?- ciezszy(winogrono,arbuz).
No
```

należy odczytywać jako: *Nic nie wiadomo na temat tego, że winogrono jest cięższe od arbuza. Nie oznacza to jednak, że tak nie jest.* Taka interpretacja jest właściwsza, co pokazuje kolejny przykład

```
?- ciezszy(arbuz,winogrono).
No
```

Z podanych faktów, przez przechodniość i znajomość pojęcia *ciężaru* możemy wywnioskować, że odpowiedź powinna być twierdząca, według rozumowania

```
ponieważ prawdą jest, że

ciezszy(arbuz,pomarancz)

i prawdą jest, że

ciezszy(pomarancz,jablko)

i prawdą jest, że

ciezszy(jablko,winogrono).

czyli

arbuz > pomarancz > jablko > winogrono

inaczej

arbuz > ... > winogrono

więc prawdą jest, że

ciezszy(arbuz,winogrono)
```

Jednak Prolog nie wie, że w stosunku do relacji `ciezszy` może przechodniość stosować, w związku z czym, **w świetle znanych faktów i związków**, udziela odpowiedzi negatywnej. W ten oto sposób dochodzimy do sytuacji, gdy musimy poinformować Prolog o pewnych relacjach, czyli określić reguły.

Zajmijmy się zatem relacją przechodniości i utworzeniem dla niej odpowiednich reguł. W matematyce relacja (dwuargumentowa) R na zbiorze A, co zapisujemy $R \subseteq A^2$ jest

przechodnia, gdy dla wszystkich elementów $a, b, c \in A$, jeżeli elementy (a, b) są w relacji R i elementy (b, c) są w relacji R, to także elementy (a, c) są w relacji R. Jako przykłady takich relacji można podać np. relacje większości, relacja zawierania zbiorów czy relację *być rodzeństwem*. Przechodnia nie jest natomiast relacja różności, relacja *być rodzicem* czy *być przyjacielem*. W przypadku rozważanej przez nas relacji wystarczy dodać taką **regułę**[2]

```
ciezszy(X,Y) :- ciezszy(X,Z),ciezszy(Z,Y).
```

W powyższej regule symbol `:-` oznacza *jeśli* (*jeśli zachodzi prawa strona to zachodzi lewa*) a symbol przecinka (`,`) pełni rolę operatora logicznego *i* (*AND*). Symbole `X`, `Y` oraz `Z` są nazwami zmiennych (w Prologu nazwa zmiennej rozpoczyna się od dużej litery).

Umieszczając fakty i regułę w jednym pliku (`owoce.pl`) możemy teraz przetestować działanie programu. Uruchamiamy zatem interpreter

```
fulmanp@fulmanp-laptop-fs12:~$ swipl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.47)
Copyright (c) 1990-2007 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).
```

i wczytujemy program

```
?- [owoce].
% t compiled 0.00 sec, 1,176 bytes

Yes
```

Zróbmy test na znajomość elementarnych faktów:

```
?- ciezszy(pomarancz,jablko).
More? [ENTER]

Yes
```

Chwilowo pomijamy znaczenie komunikatu `More?` i naciskamy `ENTER` gdy się on pokaże. Wszystko się zgadza, zatem pora na test przechodniości:

```
?- ciezszy(arbuz,winogrono).
More? [ENTER]

Yes
```

Tym razem odtrzymaliśmy odpowiedź zgodną z oczekiwaniem. Możemy jednak dowiedzieć się znacznie więcej, zadając np. pytanie *od jakich obiektów jest cięższy arbuz*:

[2]Reguła ta nie do końca jest poprawna i zasadniczo problem powinien zostać rozwiązany w inny sposób, ale na tym etapie poznawania Prologa jest to rozwiązanie akceptowalne.

```
?-  ciezszy(arbuz,X).

X = pomarancz [;]

X = jablko [;]

X = mandarynka [;]

X = winogrono [;]
ERROR: Out of local stack
```

Po wyświetleniu każdej z możliwości, Prolog czeka na naszą decyzję: naciśnięcie `ENTER` oznacza zakończenie poszukiwania alternatywnych odpowiedzi, średnik (`;`) oznacza kontynuowanie poszukiwania. Niemiły komunikat pojawiający się na końcu[3] należy w tym przypadku odczytać jako *nie wiadomo nic o innych możliwościach (obiektach).* Symbol średnika (`;`), zgodnie z intuicją, czytamy jako *lub* (*OR*). Z operatorów logicznych możemy skorzystać także podczas formułowania zapytania, np. *czy istnieją owoce X, Y takie, że arbuz jest cięższy od X i jednocześnie X jest cięższy od Y*:

```
?- ciezszy(arbuz,X),ciezszy(X,Y).

X = pomarancz,
Y = jablko [;]

X = pomarancz,
Y = mandarynka [ENTER]

Yes
```

1.3 Program w Prologu

1.3.1 Struktura i składnia

Wiedząc już jak należy rozumieć pojęcie obiektu i reguł go opisujących, możemy spróbować w bardziej kompletny sposób przedstawić strukturę i składnię programu Prologowego.

Przede wszystkim, powtórzmy to jeszcze raz, programowanie w Prologu polega na „definiowaniu" obiektów i określaniu wiążących ich relacji. Zatem przystępując do rozwiązania jakiegoś problemu musimy bardzo uważnie się zastanowić

1. z jakimi obiektami mamy do czynienia,
2. jakie relacje (związki) łączą wytypowane przez nas obiekty.

Musimy przy tym zdawać sobie jasno sprawę z kilku faktów.

1. Z punktu widzenia języka, obiekty nie są rozróżnialne semantycznie. Oznacza to, że obiekt `slon` i `slonia` w poniższych faktach

[3] Pomijamy na tym etapie powód jego pojawienia się.

```
jestDuzy(slon).
lubi(zosia,slonia).
```

są **różnymi** obiektami, pomimo tego, że my „wiemy" iż są tym samym.

2. Nie wszystkie relacje jawnie określają wszystkie obiekty, których dotyczą.

```
brakuje mi przykladu
```

3. Wybór formy relacji (reguły) powinien być wystarczająco precyzyjny aby mógł być potraktowany jak **definicja** w problemie jaki rozwiązujemy.

Praca z Prologiem składa się zwykle z następujących etapów:

1. Definiowanie obiektów poprzez definiowanie **faktów** dotyczących obiektów i związków między nimi.
2. Definiowanie **reguł** dotyczących obiektów i związków między nimi.
3. Zapytania o obiekty i związki między nimi.

Podczas zapisywania programu[4] stosujemy następującą konwencję.

- Nazwy relacji i obiektów muszą zaczynać się małymi literami.
- Nazwy rozpoczynające się od dużej litery oznaczają zmienne.
- Najpierw zapisujemy relację, a potem, rozdzielone przecinkami i ujęte w nawias okrągły, obiekty których ona dotyczy.
- Nazwy obiektów występujących w nawiasach nazywamy **argumentami**.
- Nazwę relacji znajdującej się przed nawiasem nazywamy **predykatem**.
- Nie można jako predykatu użyć zmiennej. Innymi słowy, nie można się dowiedzieć jaka relacja łączy obiekty `jas` i `malgosia`

```
X(jas,malgosia).
```

- Fakt i regułę kończymy znakiem kropki.
- Kolejność obiektów umieszczonych w nawiasie jest dowolna, ale trzeba stosować ją konsekwentnie. O ile bowiem dobrze znanym faktem jest to, że Ala lubi swojego kota, to nie oznacza to, że kot ten lubi Alę.
- Zbiór faktów i reguł nazywamy **bazą danych**.
- Składnia reguły jest następująca

```
<lewaCzesc> :- <prawaCzesc>.
```

[4]Kod programu jest zwykłym plikiem tekstowym z rozszerzeniem `pl`

co możemy czytać jako

lewaCzesc zachodzi (jest prawdą), gdy zachodzi prawaCzesc (jest prawda),

gdzie

- `<lewaCzesc>` to predykat i ewentualne argumenty umieszczone w nawiasach okrągłych, np.

  ```
  lubi(X,Y)
  silnia(0,X)
  ```

- `<prawaCzesc>` to jedno lub więcej **wyrażeń atomowych** połączonych operatorami logicznymi: **i** (`,`) oraz **lub** (`;`) i poprzedzonych ewentualnie operatorem **negacji** (`\+`). Wyrażenie atomowe w tym kontekście to wyrażenie, dla którego można obliczyć wartość logiczną, a które nie może być już rozłożone na wyrażenia prostsze, np.:

  ```
  N>0
  A is B-1
  silnia(N,X)
  \+ lubi(malgosia,X)
  ```

1.3.2 Praca z programem – zapytania

Praca z programem Prologowym także odbywa się inaczej niż w innych językach programowania. Raczej trudno mówić o uruchamianiu programu i jego działaniu jako samodzielnej i niezależnej aplikacji, gdyż programy Prologu z natury są raczej interakcyjne. Bardziej adekwatnym określeniem zamiast *uruchamianie* wydaje się być *formułowanie zapytań* lub też *interakcyjny tryb zapytanie–odpowiedź*[5].

Zapisany program wczytujemy poleceniem (znaki `?-` są tzw. znakiem zachęty)

```
?- [plikBezRozszerzenia].
```

i od tego momentu możemy formułować **zapytania**, np.

```
?- posiada(piotr,ksiazka).
```

Zapytanie to, w języku naturalnym brzmiałoby

Czy Piotr ma książkę?

Na potrzeby przetwarzania przez Prolog należy czytać je jednak trochę inaczej

Czy istnieje fakt mówiący, że Piotr ma książkę?

Prolog przeszuka całą dostępną bazę wiedzy (w postaci faktów i reguł) i jeśli zostanie znalezione coś co pasuje do zapytania i zwraca przy tym wartość logiczną *prawda*, wówczas zostanie zwrócona odpowiedź `yes`; w przeciwnym razie `no`. Raz jeszcze zaznaczamy, że `no` nie oznacza *„nie”*, ale *„nie wiem”*. Sam proces przeszukiwania, o którym

[5] Są to moje propozycje na określenie tego z czym mamy do czynienia. Ewentualne „zamienniki” są mile widziane.

powiemy sobie w dalszej części (patrz rozdział **??** tutu), odbywa się *linia po linii*, czyli fakty i reguły rozpatrywane są w kolejności ich umieszczenia w pliku.

Zamiast szukać odpowiedzi na pytanie

Czy Piotr ma książkę?

możemy chcieć zapytać

Co ma Piotr?

co w języku Prologu bardziej należy czytać jako

Jeśli Piotr ma X, to X jest tym czego szukam.

```
?- posiada(piotr,X).
```

Majac więcej faktów

```
lubi(jas,piernik).
lubi(jas,malgosia).
lubi(malgosia,cukierek).
lubi(malgosia,piernik).
```

możemy konstruować zapytania złożone, np.

```
?- lubi(jas,malgosia), lubi(malgosia,jas).
```

czyli

Czy prawdą jest, że Jaś lubi Małgosię i Małgosia lubi Jasia?

lub

```
?- lubi(jas,X), lubi(malgosia,X).
```

czyli

Szukam tego wszystkiego co lubi zarówno Jas jak i Małgosia.

Odpowiedź na pytanie o to co lubi Jaś lub Małgosia uzyskamy zapytaniem

```
?- lubi(jas,X); lubi(malgosia,X).
```

1.4 Pytania i odpowiedzi

Pytanie 1.1. Co oznacza programowanie w logice? *Programowanie w logice opiera się na rachunku kwantyfikatorów (tzn. rachunku predykatów pierwszego rzędu). Podając zbiór predykatów i podstawiając do nich stałe programista tworzy bazę faktów, następnie określając związki logiczne między nimi otrzymuje zbiór reguł. Jednym z języków tak rozumianego programowania jest Prolog. Praca z Prologiem może zatem polegać na:*

- *uzyskiwaniu odpowiedzi TAK/NIE na pytanie o prawdziwość pewnego zdania*[6]*. Z uwagi na to, że rachunek kwantyfikatorów nie jest rozsztrzygalny odpowiedź negatywna oznacza, że*
  - *podane zdanie jest rozstrzygalne i nieprawdziwe lub*
  - *zdanie jest nierozstrzygalne;*
- *uzyskiwaniu odpowiedzi na pytanie o ekstensję funkcji zdaniowej*[7] *przy czym zwracana jest odp. „NIE" jeśli ekstensja funkcji jest pusta.*

Pytanie 1.2. Jak należy rozumieć pojęcie obiektu występujące w Prologu? Tak samo jak w OOP? *Pojęcie „obiekt" w językach zorientowanych obiektowo oznacza jednoznacznie identyfikowalną (np. poprzez adres) strukturę zawierającą pewne parametry jako dane i zbiór operacji/procedur/funkcji określonych na tych parametrach. Wartości parametrów określają stan obiektu, który może zmieniać się w trakcie pracy programu. W Prologu natomiast definicja obiektu jest definicją operacyjną*[8]*: podaje się zbiór predykatów spełnianych przez definiowane obiekty.*

Uwaga: *Obiekty mogą pozostawać we wzajemnych zależnościach wyrażanych za pomocą implikacji, której poprzednikiem i następnikiem są określone formuły zdaniowe. Możemy zatem definiować jedne obiekty poprzez inne w ten sposób, że z prawdziwości pewnych predykatów określonych na znanych już obiektach i na tym definiowanym wynika prawdziwość jakiegoś predykatu określonego (między innymi) na obiekcie definiowanym.*

Pytanie 1.3. Co nazywamy relacją, faktem, regułą?

Relacja: *związek między obiektami wyrażony za pomocą predykatu przez nie spełnianego. Inaczej, można powiedzieć, że predykat określony na pewnym zbiorze zmiennych jest konkretną, nazwaną relacją między obiektami będącymi wartościami zmiennych z jego dziedziny. Np.: predykat p(A,B,C,D) prawdziwy dla „wektorów" (a,b,c,d); (e,f,g,h) i nieprawdziwy dla (a,f,c,h) wyznacza zarówno relację między obiektami (a,b,c,d); (e,f,g,h) ale także między (a,f,g,h)*[9] *.*

Fakt: *zdanie otrzymane przez wstawienie do określonego w programie predykatu wartości zmiennych (a zatem obiektów), dla których jest ono prawdziwe. Podana tu definicja faktu odnosi go całkowicie do formlizmu stosowanego w Prologu, w istocie fakt ma odzwierciedlać pewien elementarny fragment rzeczywistości modelowanej w programie.*

Reguła: *zdanie prawdziwe wiążące implikacją obiekty i relacje wchodzące w jego skład.*

[6] Zdanie powinno być konsystentne (konsystentny – wewnętrznie spójny lub zgodny z czymś) w formie zapsiu z bazą a jego dziedzina musi się zawierać w zbiorze stałych występujących w bazie.

[7] Ekstensja funkcji zdaniowej – zakres funkcji zdaniowej wyznaczony przez zbiór przedmiotów, których nazwy wstawione w miejsce zmiennych wolnych zmieniają tę funkcje w zdanie prawdziwe.

[8] Definicja operacyjna to taka definicja, w której znaczenie definiowanej nazwy określane jest drogą podania czynności (operacji) niezbędnych do określenia znaczenia tej nazwy.

[9] Ważna jest także przynależności stałych do dziedziny predykatu, gdyż jak to zostało wspomniane, odpowiedź negatywna na postawione pytanie zostanie udzielona przez Prolog także wtedy, gdy dla reprezentowanej przez wstawione do relacji stałe rzeczywistości utworzone zdanie jest nie tyle nieprawdziwe co pozbawione sensu.

Rozdział 2

Składnia Prologu

2.1 Termy

Program Prologu składa się z **termów**. Wyróżniamy cztery rodzaje termów: **atomy** (ang. *atoms*), **liczby** (ang. *numbers*), **zmienne** (ang. *variables*) i **termy złożone** (ang. *compound terms*). Atomy i liczby wspólnie określane są jako **stałe** (ang. *constants*). Zbiór złożony z atomów i termów złożonych nazywany jest też zbiorem **predykatów**[1]. Każdy term zapisywany jest jako ciąg znaków pochodzących z następujących czterech kategorii:

- **duże litery**: `A-Z`
- **małe litery**: `a-z`
- **cyfry**: `0-9`
- **znaki specjalne**: `% + - * / \ ~ ^ < > : . ? @ # $ &`

Zbiór ten uzupełnia **znak podkreślenia** (`_`), który zwykle traktowany jest jak litera.

Atomy

Atom jest ciągiem znaków utworzonym z

- małych i dużych liter, cyfr i znaku podkreślenia z zastrzeżeniem, że pierwszym znakiem musi być mała litera, np.

  ```
  jas, a, aLA, x_y_z, abc
  ```

- dowolnego ciągu znaków ujętego w apostrofy, np.

  ```
  'To też jest atom'
  ```

- symboli, np. `?-` lub `:-` .

[1] Tak przyjęta terminologia odbiega trochę od pojęć i terminów używanych w rachunku predykatów (nazywanym też rachunkiem predykatów pierwszego rzędu (ang. *first order predicate calculus*), logiką pierwszego rzędu (ang. *first-order logic*), rachunkiem kwantyfikatorów). Trochę więcej na ten temat powiemy w rozdziale **??**.

Liczby

W SWI-Prologu dostępne są zarówno liczby całkowite jak i rzeczywiste

```
-17, 23, 99.9, 123e-3
```

Zmienne

Zmienna jest ciągiem znaków utworzonym z małych i dużych liter, cyfr i znaku podkreślenia z zastrzeżeniem, że pierwszym znakiem musi być duża litera lub znak podkreślenia, np.

```
X, Kto, _123, X_1_2, _
```

Ostatni z wymienionych przykładów, pojedynczy znak podkreślenia, to tak zwana **zmienna anonimowa**. Korzystamy z niej zawsze wtedy, gdy interesuje nas tylko czy coś jest prawdą, ale zupełnie nie interesuje nas co, np.

Czy ktoś lubi Jasia?

```
?- lubi(_,jas).
```

Należy pamiętać, że wielokrotnym wystąpieniom zmiennej anonimowej w jednym wyrażeniu mogą być przypisane różne wartości.

```
?- a(1,2)=a(X,Y).
X = 1,
Y = 2
?- a(1,2)=a(X,X).
No
?- a(1,2)=a(_,_).
Yes
```

Termy złożone

Term złożony, inaczej **struktura**, to obiekt złożony z innych obiektów, czyli atomów, liczb, zmiennych a także innych termów złożonych. Termy złożone mają postać

```
f(arg_1,...,arg_n)
```

gdzie `arg_1`,...,`arg_n` są termami, natomiast `f` jest atomem (nazwą relacji). Korzystając z możliwości zagnieżdżania innych termów w termach złożonych, możemy lepiej opisać interesujący nas problem. Fakty

```
posiada(piotr,auto).
posiada(marcin,auto).
```

pozwalają jedynie stwierdzić, że obiekty `piotr` i `marcin` związane są relacją `posiada` z obiektem `auto`, czyli mówiąc „normalnie”, Piotr i Marci mają auto. Trudno powiedzieć jakie to jest auto i czy przypadkiem to nie jest to samo auto. Zapisując te fakty inaczej

```
posiada(piotr,auto(nissan,almera)).
posiada(marcin,auto(fiat,punto)).


maAuto(X) :- posiada(X,auto(_,_)).
```

wciąż mamy możliwość dowiedzenie się czy obaj mają auto, ale jeśli będziemy chcieli, to możemy zapytać o coś bardziej szczegółowego, np. marka i model

```
?- maAuto(piotr).
Yes
?- posiada(piotr,auto(X,Y)).
X = nissan,
Y = almera
```

2.2 Klauzule, program i zapytania

W rozdziale 1 zawarto elementarne informacje związane z programami Prologu, ich składnią i strukturą. Jak wiemy, program Prologu składa się zasadniczo z dwóch rodzai „konstrukcji programistycznych", jeśli można takiego terminu użyć. Są to *fakty* i *reguły*, które określane są jednym terminem **klauzule** (ang. *clauses*).

Fakty

Fakty (ang. *fact*) jest to predykat zakończony znakiem kropka ('.'), np.

```
lubi(piotr,ciastko).
toJuzKoniec.
```

Intuicyjnie rozumiany termin *fakt* jest stwierdzeniem o rozpatrywanych obiektach, które bezdyskusyjnie uważamy za prawdziwe.

Reguły

Każda reguła (ang. *rule*) składa się z dwóch części: **głowy** (ang. *head*) i **ciała** (ang. *body*). Głowa to jeden predykat, natomiast ciało to jeden lub więcej predykatów rozdzielonych przecinkami (',') i/lub średnikami (';'). Reguła kończy się znakiem kropki. Przecinek lub średnik pełnią rolę operatorów logicznych, oznaczajacych odpowiednio koniunkcję (co zapisywać będziemy też jako: i, and, `&`) i alternatywę (co zapisywać będziemy też jako: lub, or, `|`). Z tego też powodu dopuszczalne jest użycie nawiasów okrągłych w charakterze elementu grupującego. Głowa od ciała oddzielona jest operatorem `:-` (głowa jest po lewej stronie operatora).

```
a(X,Y) :- b(X,Z), c(Z,Y).
nieWiekszy(X,Y) :- mniejszy(X,Y); rowny(X,Y).
a(X,Y) :- b(X,Z); (c(X,Y), d(X,Y)).
a(X,Y) :- b(X,Z); (c(Y,X); c(X,Z), d(Z,Y)).
```

Intuicyjnie rozumiany termin *reguła* jest zbiorem warunków (ciało) jakie muszą być spełnione aby cel (głowa) został spełniony (spełnienie oznacza w tym przypadku możliwość przypisania/stwierdzenia dla danego elementu wartości logicznej *prawda*).

Program

Program w Prologu to uporządkowany zbiór klauzul. Słowo „uporządkowany" jest w tym przypadku istotne, gdyż kolejność klauzul w pliku źródłowym ma istotne znaczenie – klauzule rozpatrywane są w kolejności występowania o czym można się przekonać porównując wyniki działania dwóch programów

```
Program 1          Program 2

a(b).              a(c).
a(c).              a(d).
a(d).              a(b).


?- a(X).           ?- a(X).
X = b ;            X = c ;
X = c ;            X = d ;
X = d              X = b
```

Zapytania

Zapytanie ma taką samą strukturę jak ciało reguły i tak jak ono kończy się kropką. Zatwierdzenie zapytania, które wpisujemy po znaku zachęty `?-`, następuje po naciśnięciu klawisza `[ENTER]`. Należy rozumieć to jako zlecenie Prologowi poszukiwania, czy można, na podstawie podanych faktów i reguł, wykazać prawdziwość predykatów tworzących zapytanie a w konsekwencji i jego samego. Odpowiedź `Yes` oznacza, że istnieje taki ciąg przekształceń i podstawień, który pozwala wykazać prawdziwość zapytania. Odpowiedź `No` oznacza, że na podstawie wiedzy posiadanej w postaci faktów i reguł, nie można tego wykazać. Nie oznacza to jednak, że tak nie jest.

Zapytanie nie zawierające żadnej zmiennej, np.

```
a(b,c).
```

nazywane jest **zapytaniem elementarnym** (?) (ang. *ground query*). Oczekiwaną odpowiedzią na takie zapytanie jest `yes` lub `no` (tak/nie, prawda/fałsz). Teoretycznie, zapytania tego typu są znacznie łatwiejsze do weryfikacji, gdyż często wystarczy znaleźć dpowiedni fakt. Dla programu jak poniżej

```
a(1,2).
a(2,3).
a(3,4).
a(4,5).
```

przykładowym elementarnym zapytaniem będzie

```
?- a(2,3).
Yes
```

Zapytania zawierające zmienną nazywane są zapytaniami **nieelementarnymi** (?) (ang. *non-ground query*). W tym przypadku, znalezienie odpowiedzi może zająć więcej czasu. Odpowiedzią oczekiwaną na takie zapytanie jest znalezienie właściwego podstawienia dla zmiennych. Przyjrzyjmy się prostemu programowi mnożącemu dwie liczby naturalne

```
mul(0,_,0).
mul(1,X,X).
mul(X,Y,R) :- X > 1, X1 is X-1, mul(X1,Y,R1), R is R1 + Y.
```

Występującą w regule formułę `X1 is X-1` rozumiemy następująco: `X1` *przyjmuje wartość będącą wynikiem wykonania operacji* `X-1`. Powyższy program opiera się na rekurencyjnej definicji mnożenia. Otóż iloczyn dwóch liczb x i y, dla $x > 1$ definiujemy rekurencyjnie jako

$$x \cdot y = y + (x - 1) \cdot y.$$

Dla $x = 0$ albo $x = 1$ mamy

$$x \cdot y = 0, \qquad \text{dla} \quad x = 0,$$

$$x \cdot y = y, \qquad \text{dla} \quad x = 1.$$

Efekty działania zgodne są z oczekiwaniem

```
?- mul(4,3,X).
X = 12 [ENTER]
Yes

?- mul(0,3,X).
X = 0 [ENTER]
Yes

?- mul(1,3,X).
X = 3 [ENTER]
Yes

?- mul(4,3,12).
More? [ENTER]
Yes
```

Pierwsze z zapytań zmusza Prolog do dosyć długiego ciągu poszukiwań celem znalezienia właściwej odpowiedzi.

1. Początek obliczania celu dla zapytania `mul(4,3,X).`, czyli poszukiwanie odpowiedzi na pytanie: *Ile to jest cztery razy trzy?*

2. Zgodnie z definicją rekurencyjną, aby obliczyć wynik dla $4 \cdot 3$, należy do wyniku operacji $r1 = 3 \cdot 3$ dodać 3. Ale w tym momencie nie mamy wyniku $r1$. Trzeba go najpierw obliczyć, czyli przechodzimy do kolejnego kroku obliczeń.

3. Aby obliczyć wynik dla $3 \cdot 3$, należy do wyniku operacji $r2 = 2 \cdot 3$ dodać 3. Ale w tym momencie nie mamy wyniku $r2$. Trzeba go najpierw obliczyć, czyli przechodzimy do kolejnego kroku obliczeń.

4. Aby obliczyć wynik dla $2 \cdot 3$, należy do wyniku operacji $r3 = 1 \cdot 3$ dodać 3. Ale w tym momencie nie mamy wyniku $r3$. Trzeba go najpierw obliczyć, czyli przechodzimy do kolejnego kroku obliczeń.

5. Wynik dla $1 \cdot 3$, podany jest jako fakt. Oznacza to, że możemy określić wynik tego działania. Wynosi on 3. Znając ten wynik, możemy go zwrócić do poprzedniego wywołania (mającego miejsce w punkcie 4).

6. Cofamy się do wywołania z punktu 4. Teraz znamy wartość zmiennej $r3$ (została ona obliczona przez wywołanie z punktu 5) i możemy dzięki temu obliczyć wynik dla $2 \cdot 3 = r3 + 3 = 6$. Wynik ten przekazujemy do wywołania poprzedzającego (punkt 3).

7. Cofamy się do wywołania z punktu 3. Teraz znamy wartość zmiennej $r2$ (została ona obliczona przez wywołanie z punktu 4) i możemy dzięki temu obliczyć wynik dla $3 \cdot 3 = r2 + 6 = 9$. Wynik ten przekazujemy do wywołania poprzedzającego (punkt 2).

8. Cofamy się do wywołania z punktu 2. Teraz znamy wartość zmiennej $r1$ (została ona obliczona przez wywołanie z punktu 3) i możemy dzięki temu obliczyć wynik dla $4 \cdot 3 = r1 + 9 = 12$. Wynik ten przekazujemy do wywołania poprzedzającego (punkt 1).

9. Cofamy się do wywołania z punktu 1. Poszukiwanym wynikiem jest `X=12`.

Opisane drzewo wywołań wygląda następująco

```
mul(4,3,X)
 |
 3 + mul(3,3,X)
 .    |
 .    3 + mul(2,3,X)
 .    .    |
 .    .    3 + mul(1,3,X)
 .    .    .    |
 .    .    .    X=3
 .    .    .    |
 .    .    3 + 3
 .    .    |
 .    .    X=6
 .    .    |
 .    3 + 6
 .    |
 .    X=9
 .    |
 3 + 9
 |
 X=12
```

Jednak nie zawsze musi być tak prosto – zapytanie wyglądające jak zapytanie elementarne, może też pociągać za sobą znaczną ilość operacji i obliczeń, co pokazuje ostatnie z zapytań, tj. `mul(4,3,12).`. W istocie, pociąga ono za sobą identyczną sekwencję wywołań jak dla zapytania pierwszego.

2.3 Pytania i odpowiedzi

Pytanie 2.1. Czym sa w Prologu stałe, zmienne, struktury? Podaj przykład.

stała: *konkretny obiekt (np: a, kowalski, 65748) lub konkretna relacja (litera, nazwisko, liczba, :-). Stałe dzielą się na:*

- *liczby;*
- *atomy (są zbudowane z dowolnych symboli ewentualnie ujętych w pojedynczy cudzysłów przy czym zawsze zaczynają się małą literą);*

zmienna: *relacja może być rozumiana jako 'funkcja' określona na pewnym zbiorze obiektów. Wówczas przez zmienną rozumiemy dowolny ale nie ustalony element z dziedziny jakiejś relacji. Nazwy zmiennych są atomami rozpoczynającymi się zawsze wielką literą. Np.: weźmy relację* `student/2`*, która jest prawdziwa jeśli argumentami są nazwisko studenta i nazwa przedmiotu na egzaminie z którego ściągał. Wstawiając do niej zmienne (Kowalski, Origami) otrzymamy predykat prawdziwy dla par utworzonych przez konkretne nazwisko i odpowiadający mu przedmiot(y). Jeśli bazą programu jest zbiór:*

```
student(a,teoria_pola).
student(b,mechanika_kwantowa).
student(c,wychowanie_fizyczne).
student(d,szkolenie_bhp).
student(d,geometria_różniczkowa).
```

to w wyniku śledztwa przeprowadzonego w Prologu otrzymamy:

```
1 ?- student(Kowalski,Origami).

Kowalski = a,
Origami = teoria_pola ;

Kowalski = b,
Origami = mechanika_kwantowa ;

Kowalski = c,
Origami = wychowanie_fizyczne ;

Kowalski = d,
Origami = szkolenie_bhp ;

Kowalski = d,
Origami = geometria_różniczkowa ;

No
2 ?-
```

Istnieje szczególny typ zmiennej tzw. zmienna anonimowa, oznaczana znakiem podkreślenia (_) której użycie w zapytaniu np.:

```
?- pytanie(_,coś).
```

powoduje uzyskanie odpowiedzi TAK/NIE na pytanie „czy istnieje w bazie stała spełniająca relację `pytanie` *jednocześnie ze stałą* `coś`*?" (Innymi słowy: „czy istnieje w bazie stała będąca w relacji* `pytanie` *z* `coś`*?").*

struktura: *mając zdefiniowane pewne obiekty możemy, traktując je jak stałe, zdefiniować nowe, złożone z nich obiekty nazywane strukturami. Kontynuując przykad kryminalny, określmy relację* `wykroczenie/2` *która przyjmuje za argumenty rok popełnienia wykroczenia i pozycję niegodziwca na społecznej drabinie UŁ; przykładowy obiekt wygląda wówczas następująco:*

```
wykroczenie(2007,student(d,szkolenie_bhp)).
```

Uwaga: *wszystkie opisane tu pojęcia nazywamy wspólnie termami.*

Pytanie 2.2. Co to jest predykat? *Bardzo często nie mamy możliwości opisania zbioru wymieniając jego elementy. Wówczas użyteczny okazuje się sposób definiowania zbioru przez określenie właściwości wspólnych dla wszystkich elementów tego zbioru. Zapis*

$$\{x|P(x)\}$$

oznacza zbiór wszystkich takich elementów x, dla których wyrażenie $P(x)$, gdzie x jest zmienną a P stwierdzeniem, jest prawdziwe. Na przykład

$$\{x|x \quad \text{jest liczbą całkowitą większą niż 3 i niewiększą niż 10}\}$$

oznacza zbiór liczb $\{4, 5, 6, 7, 8, 9, 10\}$.

Tak więc elementem zbioru $\{x|P(x)\}$ jest każdy obiekt t, dla którego wyrażenie $P(t)$ jest prawdziwe. Wyrażenie $P(x)$ nazywane jest **predykatem**, **funkcją zdaniową** *lub* **formą zdaniową**. *W języku angielskim funkcjonuje określenie* propositional function *oddające to, że, każdorazowy wybór konkretnego podstawienia za zmienną x powoduje utworzenie zdania (które można rozumieć jako propozycję określenia czegoś jako coś, twierdzenie, wniosek), które jest albo fałszywe albo prawdziwe.*

W rachunku predykatów pierwszego rzędu (ang. first-order logic*, predykat może pełnić rolę właściwości przypisanej obiektom lub relacji je łączącej. Przypatrzmy się takim zdaniom*

- *Miś jest żółty*[2].
- *Banan jest żółty.*
- *Samochód Ferrari jest żółty*[3].

*Wszystkie one pasują do szablonu „...*x *...*jest żółty". *Wyrażenie „jest żółty" to właśnie predykat opisujący cechę bycia żółtym. Jeśli przyjmiemy oznaczać ten predykat jako* `jest_zolty` *lub krócej* `zolty`*, wówczas każde zdanie* `zolty(x)` *czytamy jako* x jest żółty*, co rozumiemy jako* obiekt x posiada cechę mówiącą o tym, że jest żółty.

Z kolei zdania

[2]Chodzi o Kubusia Puchatka w wizualizacji The Walt Disney Company

[3]Zdaniem miłośników marki, prawdziwe Ferrari powinno być pomalowane na czerwono. Wzięło się to ze zwyczaju założyciela, który malował swoje samochody wyścigowe na tzw. *rosso corsa*. Do dziś czerwony kolor jest najpopularniejszy wśród samochodów Ferrari, mimo że oficjalnym kolorem jest kanarkowy żółty – *giallo modena*, taki jak ten w tle znaczka Ferrari, przejęty przez założyciela z herbu miasta Modena, w którym się urodził.

- *Jaś dał piernika Małgosi.*
- *Cezary dał książkę Gosi.*

powstały przez odpowiednie podstawienia w szablonie ... x ... dał ... y ... z .... *Szablon* ktoś dał coś komuś *jest predykatem opisującym w tym przypadku związek (relację) pomiędzy obiektami. Także i w tym przypadku częściej będzie się używało jego krótszych wersji, np.* `dal(x,y,z)`.

Rozdział 3

Ewaluacja zapytania

3.1 Dopasowywanie wyrażeń

Dwa termy nazwiemy pasującymi do siebie (ang. *match*) jeśli są identyczne lub mogą stać się identyczne w wyniku podstawienia odpowiednich wartości za zmienne (ang. *variable instantiation*). Istotnym jest, aby podstawienie za zmienne było identyczne w całym wyrażeniu. Jedynym wyjątkiem jest zmienna anonimowa, która może mieć inna wartość w różnych miejscach. Na przykład dwa termy

```
jakisTerm(a,b)
jakisTerm(a,X)
```

pasują do siebie, gdyż podstawienie za zmienną `X` atomu `b` czyni te termy identycznymi

```
?- jakisTerm(a,b)=jakisTerm(a,X).
X = b
```

Nie bądą natomiast pasować termy w następujących przykładach

```
?- jakisTerm(a,b)=jakisTerm(X,X).
No
?- jakisTerm(a,X)=jakisTerm(X,b).
No
```

gdyż `X` nie może mieć jednocześnie nadanej wartości `a` i `b`. Zastąpienie `X` zmienną anonimową (`_`) powoduje, że wyrażenia zaczynają pasować do siebie

```
?- jakisTerm(a,b)=jakisTerm(a,_).
Yes
?- jakisTerm(a,_)=jakisTerm(_,b).
Yes
```

Dopasowywanie do siebie wcale nie jest takim trywialnym procesem, o czym możemy się przekonać patrząc na poniższy przykład

```
?- b(X,a)=b(f(Y),Y), d(f(f(a)))=d(U), c(X)=c(f(Z)).
X = f(a),
Y = a,
U = f(f(a)),
Z = a
```

Opisany powyżej proces dopasowywania nazywany jest także **unifikacją** (ang. *unification*).

Algorytm unifikacji

Niech $T1$ i $T2$ będą termami.

- Jeśli algorytm zwróci wartość $FAIL$ oznacza to, że unifikacja nie jest możliwa.
- Jeśli algorytm zwróci wartość $NULL$ oznacza to, że wyrażenia pasują do siebie bez konieczności dokonywania podstawienia.
- Jeśli algorytm zwróci podstawienie typu $a|b$ oznacza to, że nie ma więcej podstawień koniecznych do unifikacji $T1$ i $T2$.
- Jeśli algorytm zwróci listę $SUBST$, to zawiera ona wszystkie podstawienie niezbędne do unifikacji $T1$ i $T2$.

1. Jeśli $T1$ i $T2$ nie są jednocześnie termem złożonym, wówczas
   (a) Jeśli $T1$ i $T2$ są identyczne, wówczas zwróć $NULL$.
   (b) Jeśli $T1$ jest zmienną i jeśli $T1$ występuje w $T2$, wówczas zwróć $FAIL$, w przeciwnym razie zwróć $T2|T1$.
   (c) Jeśli $T2$ jest zmienną i jeśli $T2$ występuje w $T1$, wówczas zwróć $FAIL$, w przeciwnym razie zwróć $T1|T2$.
   (d) Zwróć $FAIL$.
2. Jeśli $T1$ i $T2$ są jednocześnie termem złożonym, wówczas
   (a) Jeśli nazwy termów $T1$ i $T2$ są różne, wówczas zwróć $FAIL$.
   (b) Jeśli termy $T1$ i $T2$ mają różną ilość argumentów, wówczas zwróć $FAIL$.
   (c) Wyczyść listę $SUBST$. Lista ta będzie zawierała wszystkie podstawienia niezbędne do unifikacji $T1$ i $T2$.
   (d) Dla i zmieniającego się od 1 do ilości argumentów termu $T1$ wykonaj
      i. Wywołaj algorytm unifikacji dla i-tego argumentu z $T1$ i i-tego argumentu z $T2$. Wynik zapisz z zmiennej S.
      ii. Jeśli S zawiera $FAIL$, wówczas zwróć $FAIL$.
      iii. Jeśli S jest różne od $NULL$, wówczas
         A. Zastosuj S do pozostałych części termów $T1$ i $T2$.
         B. Dodaj do listy $SUBST$ podstawienia z S.
   (e) Zwróć $SUBST$.

Kroki 1 (b) i 1 (c) są warunkami chroniącymi przed próbą unifikacji zmiennej z wyrażeniem zawierajacym tą zmienną, co mogłoby prowadzić do nieskończonej rekurencji, np.

```
X i a(X)
```

lub

```
a(X,X) i a(b(X),b(X))
```

3.2 Obliczanie celu

Zatwierdzenie zapytania powoduje uruchomienie procesu mającego na celu wykazanie, że istnieje ciąg podstawień i przekształceń pozwalający przypisać zapytaniu wartość logiczna *prawda*. Poszukiwanie takiego dowodu nazywane jest **obliczaniem celu** (ang. *goal execution*). Każdy predykat wchodzący w skład zapytania staje się (pod)celem, który Prolog stara się spełnić jeden po drugim. Jeśli identyczne zmienne wstępują w kilku podcelach, wówczas, jak to było już opisane, związane jest z nimi identyczne podstawienie.

Jeśli cel pasuje do głowy reguły, wówczas mają miejsce odpowiednie podstawienia wewnątrz reguły[1] i tym samym otrzymujemy nowy cel, zastępujący niejako cel początkowy. Jeśli cel ten składa się z kilku predykatów, wówczas zostaje on podzielony na kilka podceli, przy czym każdy z nich traktujemy jak cel pierwotny. Proces zastępowania wyrażenia przez inne wyrażenie nazywamy **rezolucją** i można opisać go następującym algorytmem

1. Dopóki zapytanie nie jest puste, wykonuj:
   (a) Wybierz term z zapytania[2].
   (b) Znajdź fakt lub regułę unifikującą się z termem[3]. Jeśli nie ma żadnego faktu lub reguły, zwróć FAIL, w przeciwnym razie kontynuuj.
      i. Jeśli znaleziono fakt, usuń go z zapytania.
      ii. Jeśli znaleziono regułę, zastąp term ciałem reguły.
2. Zwróć SUCCESS.

Stosowanie unifikacji i rezolucji pozwala na wykazanie prawdziwości lub jej braku, według następujących zasad

1. Jeśli cel jest zbiorem pustym, zwróć *prawdę*.
2. Jeśli nie ma głów reguł lub faktów unifikujących się z rozważanym wyrażeniem, zwróć *fałsz*.
3. W przypadku niepowodzenia (otrzymanie wartości *fałsz*), wróć do takiego miejsca, w którym stosując rezolucję możesz uzyskać inne wyrażenie i ponów cały proces. Zasada ta nazywana jest **nawracaniem** (ang. *backtracking*) (więcej przykładów związanych z nawracaniem podanych zostanie w rozdziale 5).

W opisie tym szczególnie istotny jest krok 3, który niejako powoduje restart całego algorytmu. Oznacza to, że Prolog w punkcie 1 (b) zapamiętuje miejsce występowania unifikatora i w odpowiednim momencie jest w stanie poszukiwać kolejnych unifikatorów występujących za właśnie wybranym. Powyższe zasady stosujemy tak długo, aż wyczerpiemy wszystkie możliwości wybierając na każdym z etapów kolejne dostępne wyrażenia. Dzięki temu mamy możliwość znalezienia kilku różnych dowodów.

Pokażemy teraz prosty przykład, który pozwoli lepiej pokazać kiedy mamy do czynienia z unifikacją a kiedy z rezolucją. Dla programu

[1] Wewnątrz (w ciele) reguły, czyli po prawej stronie operatora `:-`
[2] Zwykle termy wybierane są od lewej do prawej.
[3] Zwykle fakty i reguły przeszukiwane są w kolejności ich występowania w pliku.

```
a(b,c).
a(c,d).
aa(X,Y) :- a(X,Z),a(Z,Y).
```

zapytanie

```
aa(b,A).
```

pociąga za sobą następujący proces.

Krok 1. Rezultatem unifikacji dla `aa(b,A)` oraz `aa(X,Y)` jest podstawienie

```
X = a   A = Y
```

Resolucja: zastępując `aa(b,A)` przez `a(X,Z), a(Z,Y)` i stosując uzyskane podstawienie otrzymujemy nowe zapytanie:

```
a(b,Z),a(Z,Y).
```

Krok 2. Z uzyskanego w poprzednim kroku zapytania wybieramy atom `a(b,Z)` i w wyniku unifikacji z faktem `a(b,c)` otrzymujemy podstawienie

```
Z=c
```

Rezolucja: ponieważ unifikacja dotyczyła faktu więc rozpatrywany atom z zapytania zostaje usunięty (zastąpiony przez element pusty) po czym do otrzymanego w ten sposób wyrażenia stosujemy unifikację w wyniku czego otrzymujemy kolejne zapytanie

```
a(c,Y).
```

Krok 3. W uzyskanym w poprzednim kroku zapytaniu występuje tylko jeden atom `a(c,Y)` i w wyniku unifikacji z faktem `a(c,d)` otrzymujemy podstawienie

```
Y=d
```

Rezolucja: ponieważ unifikacja dotyczyła faktu więc rozpatrywany atom z zapytania zostaje usunięty w wyniku czego otrzymujemy puste zapytanie, co oznacza koniec procesu.

Innymi słowy można powiedzieć, że unifikacja jest, podobnie jak w „tradycyjnym" programowaniu, przypisywaniem wartości do zmiennych, natomiast rezolucja sposobem przekonstruowywania zapytania.

Więcej szczegółów na ten temat podamy w dalszej części wykładu (patrz rozdział **??**) – teraz zależy nam głównie na wyrobieniu właściwej intuicji.

Kolejny przykład pokaże, że proces unifikacji, rezolucji i nawracania prowadzi często do rezultatów sprzecznych z oczekiwaniem. Rozważmy taki program

```
mniej(p1,p2).
mniej(p2,p3).
mniej(p3,p4).

mniej(X,Y) :- mniej(X,Z),mniej(Z,Y).
```

Intencja jest jasna: definiujemy kilka obiektów (tj. `p1`, `p2`, `p3`, `p4`) powiązanych relacją `mniej`. Do tego wprowadzamy regułę przechodniości, która w założeniu ma pozwolić na powiązanie ze sobą np. obiektu `p1` i `p4`. Niestety zapytanie

```
?- mniej(A,B).
```

nie zwraca tego co, naszym zdaniem, powinno zwrócić

```
A = p1,
B = p2 ;

A = p2,
B = p3 ;

A = p3,
B = p4 ;

A = p1,
B = p3 ;

A = p1,
B = p4 ;

ERROR: Out of local stack
```

Jest para `(A=p2, B=p3)`, ale gdzie para `(A=p2, B=p4)`? Wytłumaczenie tego jest następujące (równolegle z opisem proszę śledzić drzewo wywołań, bo choć nie pozbawione wad, może ułatwić kontrolowanie co i kiedy jest wywoływane). Ponumerujmy najpierw linie programu, aby łatwiej nam było się do nich odnosić

```
1 mniej(p1,p2).
2 mniej(p2,p3).
3 mniej(p3,p4).

4 mniej(X,Y) :- mniej(X,Z),mniej(Z,Y).
```

Zapytanie `mniej(A,B).` powoduje, ze Prolog, szuka (od poczatku pliku, po kolei) czegoś, co może z tym zapytaniem zostać zunifikowane. Tak więc po kolei, pasuje fakt 1 (linia 2 w drzewie wywołań – patrz dalej), więc jako wynik mamy

```
A = p1,
B = p2 ;
```

Szukając dalej, pasuje fakt 2 (linia 3 w drzewie wywołań) i mamy

```
A = p2,
B = p3 ;
```

i pasuje też fakt 3 (linia 4 w drzewie wywołań), w związku z czym mamy

```
A = p3,
B = p4 ;
```

Szukamy dalej a dalej mamy regułę (4) (linia 5 w drzewie wywołań). A więc spróbujemy użyć reguły, do tego aby znaleźć podstawienie za `A` i `B`. Aby jednak użyć reguły, należy spełnić dwa warunki jakie w niej występują. Pierwszym warunkiem jest poszukiwanie czegoś, co spełni `mniej(X,Z)` (linia 6 w drzewie wywołań (d.w.)). Ponieważ w tym momencie `mniej(X,Z)` pełni rolę podcelu, więc rozpoczynamy przeszukiwanie pliku od początku. Dla tego podcelu znajdujemy fakt 1, który do niego pasuje (linia 7 w d.w.). To powoduje, że mamy podstawienie `(X=p1, Z=p2)`, dzięki czemu możemy przejść do próby wykazania drugiej części reguły, która teraz przyjmuje postać z linii 8 drzewa wywołań i staje się nowym podcelem. Ponownie rozpoczynamy przeszukiwanie pliku od poczatku w celu znalezienia czegoś co unifikuje się z tym podcelem. Jako pierwszą pasującą znajdujemy regułę 2 (linia 9 w d.w.). Dopasowanie to powoduje, że `Y` przyjmuje wartość `p3`. W tym momencie, wszystkie podcele reguły z linii 5 d.w. są spełnione (podstawienia to `X=p1`, `Z=p2`, `Y=p3`) i może zostać zwrócony następujący wynik

```
A = p1,
B = p3 ;
```

Kontynuujemy poszukiwania tego co może spełnić podcel `m(p2,Y)` (linia 8 w d.w.). Fakt 3 nie pasuje, ale mamy regułę (4, linia 10 w d.w.). W tym momencie, w wyniku wcześniejszych podstawień, regula ta jest wywołana jako `mniej(p2,Y)` w wyniku czego jej podcele przyjmą postać `mniej(p2,Z)` i `mniej(Z,Y)`. Zauważmy, że spełniając regułę z linii 10 d.w. spełniamy tym samym drugi warunek (linia 8 w d.w.) reguły z linii 5 d.w. a więc i nasz główny cel. Aby ją spełnić należy znaleźć „coś" co spełni jej pierwsza część, która przyjmuje postać `mniej(p2,Z)` (linia 11 w d.w.). Podobnie jak wcześniej (linia 8 w d.w.), widzimy, że pierwszy podcel spełnia fakt 2. Zatem wiemy już, że `Z` ma wartosc `p3` (linia 12 w d.w.). Zatem aby spełnić drugą część reguły, trzeba znaleźć fakt lub regułę spełniającą `mniej(p3,Y)` (linia 13 w d.w.). Spelnia to fakt 3 i w związku z tym mamy podstawienie `Y=p4`. Mając spełnione podcele z linii 11 i 13 d.w. możemy powiedzieć, że spełniony jest podcel z linii 8 d.w. a w konsekwencji reguła z linii 5 d.w. Wszystko to zachodzi przy następujących podstawieniach: `X=p1` `Y=p4` (i choć to mało dla nas istotne, także `Z=p2`). Potwierdzeniem takiego rozumowania jest kolejny wynik zwrócony przez Prolog

```
A = p1,
B = p4 ;
```

Powracamy do podcelu z linii 13 d.w. i kontynuujemy poszukiwania tego co go spełnia. Kolejną, nierozpatrywaną rzeczą (w sensie fakt, reguła), jest reguła 4 (linia 15 w d.w.). W tym momencie, w wyniku wcześniejszych podstawień, reguła ta jest wywołana jako `mniej(p3,Y)` w wyniku czego jej podcele przyjmą postać `mniej(p3,Z)` i `mniej(Z,Y)`. Zauważmy, że reguła z linii 15 spełnia podcel z linii 13, co z koleii powoduje spełnienie reguły z linii 10 d.w.. Spełniamy tym samym drugi warunek (linia 8 w d.w.) reguły z linii 5 d.w. a więc i nasz główny cel. Aby jednak spełnić regułę z linii 15 należy znaleźć fakt lub regułę spełniające jej pierwszą część, która przyjmuje postać `mniej(p3,Z)` (linia 16 w d.w.). Widzimy, że podcel ten spełnia reguła 3 (linia 17 w d.w.). Zatem wiemy już, że `Z` ma wartosc `p4`. Zatem aby spełnić drugą część reguły, trzeba znaleźć fakt lub regułę spełniającą `mniej(p4,Y)` (linia 18 w d.w.). Żaden fakt tego nie spełnia, ale może spełnić reguła (4) (linia 19 w d.w.). Aby jednak reguła mogła być spełniona, należy spełnić jej dwa podcele. Pierwszy podcel, w wyniku podstawień przyjmuje postać `mniej(p4,Z)` (linia 20 w d.w.). Podobnie jak to było przed chwilą, żaden fakt tego nie spełnia, ale

może spełnić reguła (4) (linia 21 w d.w.). Aby jednak reguła mogła być spełniona, należy spełnić jej dwa podcele. Pierwszy podcel, w wyniku podstawień przyjmuje postać `mniej(p4,Z)` (linia 22 w d.w.). Podobnie jak to było przed chwilą jedynym faktem lub regułą, które mogą ewentualnie spełnić ten podcel, jest znowu reguła (4) (linia 23 w d.w.). Jak widzimy reguła będzie wywoływać regułę, która znowu wywoła regułę itd. To dlatego jako ostatni widoczny efekt działania otrzymujemy

```
ERROR: Out of local stack
```

czyli przepełnienie stosu w wyniku zbyt dużej ilości wywołań rekurencyjnych.

```
Drzewo wywołań dla programu

mniej(p1,p2).
mniej(p2,p3).
mniej(p3,p4).
mniej(X,Y) :- mniej(X,Z),mniej(Z,Y).

i zapytania

mniej(A,B).


 1 m(A ,B).
   |
 2 +-m(p1,p2).
   |
 3 +-m(p2,p3).
   |
 4 +-m(p3,p4).
   |
 5 +-m(X,Y) :- m(X,Z),m(Z,Y).
     |
 6   +---m(X ,Z)
     |   |
 7   |   +-m(p1,p2).
     |
 8   +-------------,m(p2,Y)
                   |
 9                 +-m(p2,p3).
                   |
10                 +-m(X,Y) :- m(X,Z),m(Z,Y).
                     |
11                   +---m(p2,Z)
                     |   |
12                   |   +-m(p2,p3).
                     |
13                   +-------------,m(p3,Y)
                                   |
14                                 +-m(p3,p4).
                                   |
15                                 +-m(X,Y) :- m(X,Z),m(Z,Y).
                                     |
16                                   +---m(p3,Z)
```

```
                                    |   |
17                                  |   +-m(p3,p4).
                                    |
18                                  +-------------,m(p4,Y)
                                                  |
19                                                +-m(X,Y) :- m(X,Z),m(Z,Y).
                                                    |
20                                                  +---m(p4,Z)
                                                        |
21                                                      +-m(X,Y) :- m(X,Z),m(Z,Y).
                                                          |
22                                                        +---m(p4,Z)
                                                              |
23                                                            +-m(X,Y) :- ...
                                                                |
...                                                             ...
```

Jak więc widzimy powodem niepowodzenia jest dopuszczenie do sytuacji, w której reguła będzie wywoływać samą siebie. Co gorsza podcel reguły będzie się tylko unifikował z głową reguły (np. linie 18 i 19, 20 i 21, 22 i 23 itd.). W takich (dosyć typowych) sytuacjach problematycznych, rozwiązaniem jest użycie innych nazw dla faktów, głowy reguły i przynajmniej częściowe przekonstruowanie reguły/faktów, np. w następujący sposób

```
mniej(p1,p2).
mniej(p2,p3).
mniej(p3,p4).

jestMniejszy(X,Y) :- mniej(X,Y).
jestMniejszy(X,Y) :- mniej(X,Z),jestMniejszy(Z,Y).
```

Ten program daje już poprawne i zgodne z oczekiwaniami wyniki

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

X = p1,
Y = p2 ;

X = p2,
Y = p3 ;

X = p3,
Y = p4 ;

X = p1,
Y = p3 ;

X = p1,
Y = p4 ;

X = p2,
```

```
Y = p4 ;

No
```

3.3 Pytania i odpowiedzi

Pytanie 3.1. Co to jest unifikacja? *Unifikacja (ang.* unification*) oznacza ujednolicanie, które w konkretnych dyscyplinach naukowych może być różnie rozumiane*[4].

Logika *W logice jest to proces ujednolicania, w wyniku którego zakresy pojęciowe lub znaczenia niezwiązane ze sobą lub w jakiś sposób niezgodne, nabywają zgodności i stają się częścią większej całości.*

Informatyka *W informatyce unifikację określić możemy jako operację na dwóch lub większej ilości drzew, która znajduje takie przyporządkowanie zmiennych, że drzewa te są równe. Stosując do zapisu drzewa* **notację polską**[5]*, drzewa*

```
(+ x 2)
(+ (+ y 3) z)
```

są unifikowalne dla

```
z=2
x=(+ y 3)
```

Nie są unifikowalne

```
(+ x 2) i (+ y 3)
(+ x 2) i (- x x)
(+ 2 3) i (+ 3 2).
```

Otrzymany zbiór przyporządkowań nazywamy **unifikatorem**.

Matematyka *Niech E będzie wyrażeniem składającym się ze zmiennych $x_1, \ldots, x_n$ i stałych, natomiast $t_1, \ldots, t_n$ będą wyrażeniami. Zbiór przyporządkowań $\eta = \{t_1|x_1, ..., t_n|x_n\}$*[6] *nazywamy* **podstawieniem**. *Wyrażenie E_η nazywane jest* **instancją** *wyrażenia E jeśli otrzymane jest z E przez zastąpienie wszystkich wystąpień zmiennych x_i przez odpowiadające im wyrażenia t_i, $i = 1, \ldots, n$. Podstawienie η nazywamy* **unifikatorem** *dla zbioru wyrażeń $\{E_1, \ldots, E_m\}$ jeśli $E_{1_\eta} = \cdots = E_{m_\eta}$. Dla wyrażeń*

$$E_1 = x^2,$$

$$E_2 = y^3$$

unifikatorem jest

$$\eta = \{z^3|x, z^2|y\}.$$

[4] Różnie nie w sensie „odmiennie", ale w sensie „specyficznie".
[5] Czyli najpierw wartość węzła a potem jego dzieci.
[6] Zapis $t_i|x_i$ należy czytać: wyrażenie t_i podstawione za zmienną x_i.

Prolog *W Prologu unifikacja oznacza proces, w którym dla dwóch atomów (jednego z zapytania, drugiego będącego faktem lub głową reguły[7]) poszukiwane jest takie podstawienie, dzięki któremu staną się one identyczne.*

Pytanie 3.2. Co to jest rezolucja? *Rezolucja to metoda automatycznego dowodzenia twierdzeń oparta na generowaniu nowych klauzul (wyrażeń) aż dojdzie się do sprzeczności. W ten sposób można udowodnić, że dane twierdzenie nie jest spełnialne, lub też, co jest równoważne, że jego zaprzeczenie jest tautologią. Metoda ta pozwala w oparciu o dwa wyrażenia zawierające dopełniające się postaci literału, wygenerować nowe wyrażenie zawierające wszystkie literały z wyjątkiem dopełniających się, zgodnie z poniższą regułą wnioskowania*

```
a|b, ~a|c
---------
   b|c
```

Według tej reguły, jeśli przyjmiemy, że a lub b jest prawdą *i jednocześnie, że a jest* fałszem *lub c jest* prawdą*, wówczas b lub c jest* prawdą*. Istotnie, jeśli a jest* prawdą*, wówczas aby drugie wyrażenie było prawdziwe (a takie jest założenie), c musi być* prawdą*. Jeśli a jest* fałszem*, wówczas aby pierwsze wyrażenie było prawdziwe (a takie jest założenie), b musi być* prawdą*. Tak więc niezależnie od wartości logicznej a, jeśli przyjmiemy prawdziwość założeń, wówczas b lub c musi być* prawdą*.*

Rozważmy następujący przykład. Z założenia prawdziwości reguł

Jeśli zachodzi b to zachodzi a.

oraz

Jeśli zachodzi c to zachodzi b.

wynika, że zachodzi a jeśli tylko zachodzi c*. Dokładnie taki sam wynik uzyskamy stosując rezolucje. Zapiszmy podane reguły*

```
a :- b
b :- c
```

które zgodnie z prawem

$$x => y \iff \bar{x}|y,$$

gdzie symbol$^-$ *oznacza negację, przekształcamy do postaci*

```
~b | a
~c | b
```

gdzie symbol ~ *oznacza negację. Stosując rezolucję otrzymujemy*

```
~c | a
```

co w postaci reguły można zapisać jako

```
a :- c
```

[7] Głową, czyli częścią występującą po lewej stronie operatora `:-`.

Pytanie 3.3. Co to jest nawracanie? *Wielokrotnie w Prologu zdarza się, że cel może zostać spełniony na wiele alternatywnych sposobów. Za każdym razem, gdy zachodzi konieczność wybrania jednej z wielu możliwości, Prolog wybiera pierwszą z nich (w kolejności występowania w pliku) zapamiętując przy okazji miejsce w którym wybór ten został dokonany. Jeśli w jakimś momencie nie powiedzie się próba obliczenia celu, system ma możliwość powrócenia do miejsca otatnio dokonanego wyboru i zastąpienia go wyborem alternatywnym. Zachowanie takie nazywamy nawracaniem (więcej przykładów związanych z nawracaniem podanych jest w rozdziale 5).*

Prawdę mówiąc, dokładnie ten sam mechanizm działa także, gdy powiedzie się obliczanie celu. O ile jedak w poprzednim przypadku pozwala on na znalezienie choć jednego rozwiązania, to w tym przypadku pozwala znaleźć rozwiązania alternatywne.

Rozdział 4

Listy

4.1 Składnia

Lista jest uporządkowanym ciągiem elementów o dowolnej długości. Jako element listy może być użyty każdy prawidłowy term Prologu, tzn. atom, liczba, zmienna, term złożony w tym także inna lista. Umieszczamy je pomiędzy nawiasami kwadratowymi (`[` i `]`) i rozdzielamy przecinkiem (`,`), np.

```
[a, X, [], b(c, Y), [a, d, e], 123]
```

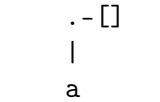
Lista pusta zapisywana jest jako para pustych nawiasów

```
[]
```

Wewnętrzna reprezentacja listy opiera się o dwuargumentowy funktor kropka (`.`). Z reprezentacji tej można korzystać tak samo jak z notacji z nawiasami kwadratowymi. Choć jest ona mniej wygodna, to czasem ułatwia zrozumienie, dlaczego nasz program zachowuje się tak a nie inaczej, gdy przetwarza listę.
Korzystajac z tego funktora, listę zawierającą jeden element `a` możemy zapisać jako

```
.(a,[])
```

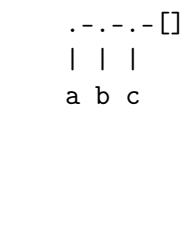
co na rysunku można przedstawić w następujący sposób



Listę zawierającą 3 elementy: `a`,`b`,`c` możemy zapisać jako

```
.(a,.(b,.(c,[])))
```

co na rysunku można przedstawić jako



W notacji z nawiasami kwadratowymi powyższe przykłady zapisujemy, odpowiednio, jako

```
[a]
[a,b,c]
```

4.2 Głowa i ogon

Listy zawsze przetwarza się dzieląc je na dwie (logiczne) części: **głowę** (ang. *head*), którą stanowi pierwszy element listy (i zarazem pierwszy argument funktora `.`) i **ogon** (ang. *tail*) (stanowiącego drugi argument funktora `.`), który jest wszystkim co pozostało z listy po „odjęciu" od niej głowy. Lista pusta nie ma głowy ani tym bardziej ogona. Głową listy zawierającej tylko jeden element jest ten właśnie element, natomiast ogon jest listą pustą.

Do rozdzielenia (rozłożenia) listy[1] na głowę i ogon służy symbol `|`. Elementy po lewej stronie symbolu odnoszą się do głowy lub do kilku pierwszych elementów listy, natomiast po prawej oznaczają głowę

```
?- []=[H|T].
No

?- [1,2]=[H|T].
H = 1,
T = [2]

?- [1]=[H|T].
H = 1,
T = []

?- [1,[2,3]]=[H|T].
H = 1,
T = [[2, 3]]

?- [[1,2],3]=[H|T].
H = [1, 2],
T = [3]

?- [1,2,3,4]=[Ha,Hb|T].
Ha = 1,
Hb = 2,
T = [3, 4]

?- [[1,2,3],4]=[[H1|T1]|T2].
H1 = 1,
T1 = [2, 3],
T2 = [4]
```

Przyglądając się powyższym przykładom zauważamy, że

[1] Używany także do konstruowania listy.

- Ogon listy, jeśli tylko istnieje, jest zawsze listą (pustą lub nie, ale listą).
- Głowa (a w ogólności: wszystkie elementy występujące przed `|`) jest elementem listy i jako element listy może być dowolnym termem (a zatem może, ale nie musi być listą).

Zaskakujące, że to już wszystko co o listach, rozpatrywanych czysto teoretycznie, można powiedzieć. Zaskakujące dlatego, że lista jest główną (w sensie „siły" czy możliwości) strukturą danych Prologu. Prostota ta sprawia, że czasem aż trudno uwierzyć, że coś bedzie działać, a gdy już działa to trudno zrozumieć jak to się dzieje. Dlatego, celem lepszego zapoznania ze sposobami postępowania z listami, spróbujemy zdefiniować kilka predykatów, pozwalających wykonać pewne elementarne operacje na listach.

Predykat sprawdzający czy coś jest listą.

Właściwie każdy sposób w jaki staramy się rozwiązać problem w Prologu to rekurencja. Zaczynamy od najprostszego przypadku, po czym uogólniamy go na dowolnie złożony. Nie inaczej jest w przypadku predykatu sprawdzającego, czy coś jest listą. Najprostszy przykład listy, to lista pusta. Wszystkie inne listy dają się natomiast rozłożyć na głowę i ogon, przy czym ogon musi być listą. Stąd ostatecznie otrzymujemy

```
czyLista([]).
czyLista([H|T]) :- czyLista(T).
```

Predykat sprawdzający czy coś należy do listy

W tym przypadku najprostszy warunek jest następujący: element `X` należy do listy, jeśli `X` jest głową listy

```
isMember(X,[X|_]).
```

lub to samo w inny sposób

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

Jeśli natomiast nie jest głową, to musi neleżeć do ogona listy

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

Jak to jednak często w Prologu bywa, zwykle każdy predykat można użyć w celu zupełnie innym niż ten, do którego został przewidziany. W tym przypadku predykat `isMemeber/2` może zostać użyty po to aby **wygererować** wszystkie elementy należące do listy

```
?- isMember(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
No

?- isMember(X,[a,[b,c],d]).
X = a ;
X = [b, c] ;
X = d ;
No
```

Predykat łączący dwie listy

Przypadek elementarny: lista pusta połączona z listą `List` daje w wyniku niezmienioną listę

```
polacz([],List,List).
```

Przypadek ogólny, który można wyrazić opisowo: *aby połączyć coś (*`[H|T]`*) z listą (*`List`*), trzeba najpierw dołączyć ogon tego czegoś (*`T`*) do listy (*`List`*) a następnie do wyniku tego połączenia (*`Res`*) dopisać głowę (*`H`*)*

```
polacz([H|T],List,[H|Res]) :- polacz(T,List,Res).
```

Działanie zgodne jest z oczekiwaniem

```
?- polacz([1,2,3],[a,b,c],Res).
Res = [1, 2, 3, a, b, c]
```

a ciąg wywołań wygląda jak poniżej

```
?- polacz([1,2,3],[a,b,c],Res).

   głowa 1, łącze [2, 3] z [a, b, c]
      głowa 2 łącze [3] z [a, b, c]
         głowa 3 łącze [] z [a, b, c] wynik to [a, b, c]
      wynik to [3, a, b, c]
   wynik to [2, 3, a, b, c]

Res = [1, 2, 3, a, b, c]
```

Podobnie jak poprzednio, także i tym razem możemy predykat `polacz/3` użyć, niezgodnie z jego pierwotnym przeznaczeniem, do znalezienia odpowiedzi na pytanie, jaką listę należy połączyć z listą `[1,2,3]` aby otrzymać listę `[1,2,3,a,b,c]`

```
?- polacz([1,2,3],X,[1,2,3,a,b,c]).
X = [a, b, c]
```

Co więcej, możemy poszukiwać wszystkich par list, które połączone dają nam `[1,2,3,a,b,c]`

```
?- polacz(X,Y,[1,2,3,a,b,c]).
X = [],
Y = [1, 2, 3, a, b, c] ;

X = [1],
Y = [2, 3, a, b, c] ;

X = [1, 2],
Y = [3, a, b, c] ;

X = [1, 2, 3],
Y = [a, b, c] ;

X = [1, 2, 3, a],
```

```
Y = [b, c] ;

X = [1, 2, 3, a, b],
Y = [c] ;

X = [1, 2, 3, a, b, c],
Y = [] ;
```

4.3 Pytania i odpowiedzi

Pytanie 4.1. Czym jest lista w prologu? *Mając niepusty zbiór obiektów K możemy utworzyć zbiór jego wszystkich podzbiorów oznaczany formalnie 2^K. Listę elementów z K możemy zatem utożsamić z pewnym elmentem należącym do 2^K. Generalnie lista ma służyć reprezentacji jakichś „danych", a te są z kolei reprezentowane w Prologu przez termy. Listy są zatem także* **reprezentowane** *przez pewne termy i co ważne, może się to odbywać na różne sposoby. W każdej liście możemy wyodrębnić* **głowę** *czyli jej pierwszy element, (który sam może być jakąś listą!) oraz* **ogon**, *czyli pozostałą jej część.*

Pytanie 4.2. Wymień możliwe sposoby zapisu listy.

- `.(X,Y)` *lista o nieokreślonej liczbie elementów. Jej głową jest* `X`, *ogonem —* `Y`.
- `[X|Y]` *j.w.*
- `[X,Y]` *dokładnie dwuelementowa lista o głowie* `X` *i ogonie* `Y`, *przy czym zarówno* `X` *jak i* `Y` *mogą być listami.*

Oto kilka przykładów mogących ułatwić zrozumienie róznic między powyższymi reprezentacjami. Bazują one na zmuszeniu Prologu do rozwiązania za nas problemu: które z tych schematów i kiedy są sobie równoważne?

```
?- .(X,X)=X.
X = [**, **, **, **, **, **, **, **, **|...] ;
No

?- [X,X]=X.
X = [**, **] ;
No

?- [X|X]=X.
X = [**, **, **, **, **, **, **, **, **|...] ;
No

?- [1,X] = X.
X = [1, **] ;
No

?- [1|X] = X.
X = [1, 1, 1, 1, 1, 1, 1, 1, 1|...] ;
No
```

```
?- .(1,X) = X.
X = [1, 1, 1, 1, 1, 1, 1, 1, 1|...] ;
No

?- [X,Y] = [X|Y].
Y = [**] ;
No

?- [X,Y] = [X|[Y]].
Yes

?- .(X,Y) = [X,Y].
Y = [**] ;
No

?- .(X,.(Y,[])) = [X,Y].
Yes

?- .(X,Y) = [X|Y].
Yes
```

Pytanie 4.3. Podaj przykład wykorzystania listy. *Weźmy program, który liczy ilość elementów listy:*

```
card([],0).
card([H|T], X) :- card(T, Y), X is Y + 1.
```

Można go wykorzystać do dalszego sprawdzania jak działają reprezentacje list:

```
?- card([a|[b]],Y).
Y = 2 ;
No

?- card([a|[b,c,d]],Y).
Y = 4 ;
No

?- card([a,[b,c,d]],Y).
Y = 2 ;
No

?- card([a,b,c,d],Y).
Y = 4 ;
No
```

Można się w ten sposób bardzo długo bawić... warto jednak uważnie spojrzeć na poniższe wyniki:

```
?- card([a,Y],H).
```

```
H = 2 ;
No

?- card([a|Y],H).

Y = [],
H = 1 ;

Y = [_G266],
H = 2 ;

Y = [_G266, _G269],
H = 3 ;

Y = [_G266, _G269, _G272],
H = 4 ;

Y = [_G266, _G269, _G272, _G275],
H = 5

[Ctrl+C]
Action (h for help) ? [b] break

?- card(.(a,Y),H).

Y = [],
H = 1 ;

Y = [_G813],
H = 2 ;

Y = [_G813, _G816],
H = 3 ;

Y = [_G813, _G816, _G819],
H = 4

[Ctrl+C]
Action (h for help) ? [b] break
```

Powyższy eksperyment potwierdza tezy dotyczące ilości elementów na poszczególnych listach.

Rozdział 5

Odcięcie

5.1 Wiele rozwiązań

Mechanizm odcięcia pozwala zaniechać nawracania przez Prolog do wcześniej dokonanych wyborów. Zajmijmy się na początek wieloma rozwiązaniami. Załóżmy, iż mamy w bazie następujące fakty

```
a(b,c).
a(b,d).
a(e,f).
a(d,g).
```

Wydając zapytanie

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

otrzymamy następujacy ciąg odpowiedzi

```
?- a(X,Y).
X = b  Y = c ;
X = b  Y = d ;
X = e  Y = f ;
X = d  Y = g ;
No
```

Dzięki nawracaniu znajdowane są wszystkie rozwiązania z bazy danych. Prolog w żadnym przypadku nie analizuje czy też nie zapamiętuje zwracanych wyników. Stąd też wynikiem zapytania

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

jest

```
?- a(X,_).
X = b ;
X = b ;
X = e ;
X = d ;
No
```

Jak widać odpowiedź `X = b` pojawiła się dwukrotnie, ale z punktu widzenia języka są to dwa różne rozwiązania.

Pamiętajmy, że w Prologu istotna jest kolejność występowania faktów i reguł. Dla programu

```
a(c).
a(X) :- b(X).
b(d).
b(e).
a(f).
a(g).
b(h).
a(i).
```

efektem zapytania `a(X)` jest

```
?- a(X).
X = c ;
X = d ;
X = e ;
X = h ;
X = f ;
X = g ;
X = i ;
No
```

Kolejność otrzymanych wyników na pierwszy rzut oka może być zaskoczeniem, ale po uważniejszym przyjrzeniu się im, wszystko powinno stać się jasne.

Krok 1 Prolog przeszukuje swoją bazę wiedzy w poszukiwaniu faktu/reguły, które pozwolą jemu ukonkretnić zmienną `X` w zapytaniu `a(X).` Jako pierwszy z zapytaniem unifikuje się fakt `a(c)` i stąd pierwsza odpowiedź `X = c`.

Krok 2 Następnie z zapytaniem unifikuje się reguła `a(X) :- b(X).`, w związku z czym Prolog będzie starał się obliczyć wszystkie cele z niej wynikające, czyli w tym przypadku `b(X)`.

Krok 2.1 Z podcelem `b(X)` jako pierwszy unifikuje się fakt `b(d)`. To powoduje spełnienie całej reguły `a(X) :- b(X).` a tym samym i zapytania `a(X)` i stąd druga odpowiedź `X = d`.

Krok 2.2 Następnie z podcelem `b(X)` unifikuje się fakt `b(e)`. To powoduje spełnienie całej reguły `a(X) :- b(X).` a tym samym i zapytania `a(X)` i stąd trzecia odpowiedź `X = e`.

Krok 2.3 Następnie z podcelem `b(X)` unifikuje się fakt `b(h)`. To powoduje spełnienie całej reguły `a(X) :- b(X).` a tym samym i zapytania `a(X)` i stąd czwarta odpowiedź `X = h`.

Krok 2.4 Ponieważ nie ma już niczego co pozwoli obliczyć podcel `b(X)`, więc tym samym Prolog powraca do ostatniego punktu decyzyjnego, czyli poszukuje faktu/reguły występujących po regule `a(X) :- b(X).` i spełniających zapytanie `a(X).`.

Krok 3 Jako kolejny, z zapytaniem `a(X)` unifikuje się fakt `a(f).` i stąd piąta odpowiedź `X = f`.

Krok 4 Jako kolejny, z zapytaniem `a(X)` unifikuje się fakt `a(f).` i stąd szósta odpowiedź `X = g`.

Krok 5 Jako kolejny, z zapytaniem `a(X)` unifikuje się fakt `a(f).` i stąd siódma odpowiedź `X = i`.

Krok 6 Brak unifikatorów dla `a(X)`. Zakończenie obliczeń dla zapytania.

Rozważmy teraz przykład z dwoma celami mającymi po kilka rozwiązań.

```
a(X,Y) :- b(X), c(Y).

b(d).
b(e).
b(f).

c(g).
c(h).
c(i).
```

A oto uzyskany wynik

```
?- a(X,Y).
X = d  Y = g ;
X = d  Y = h ;
X = d  Y = i ;
X = e  Y = g ;
X = e  Y = h ;
X = e  Y = i ;
X = f  Y = g ;
X = f  Y = h ;
X = f  Y = i ;
No
```

Ponownie przeanalizujmy dlaczego otrzymaliśmy taki właśnie wynik.

Krok 1 Prolog przeszukuje swoją bazę wiedzy w poszukiwaniu faktu/reguły, które pozwolą jemu ukonkretnić zmiene `X` i `Y` w zapytaniu `a(X,Y).` Jako pierwsza z zapytaniem unifikuje się reguła `a(X,Y) :- b(X), c(Y).`, w związku z czym Prolog będzie starał się obliczyć wszystkie cele z niej wynikające, czyli w tym przypadku najpierw `b(X)`.

Krok 2.1 Z podcelem `b(X)` jako pierwszy unifikuje się fakt `b(d)`. To powoduje spełnienie pierwszego podcelu i ukonkretyzowanie zmiennej `X` przez `d`.

Krok 2.1.1 Prolog próbuje obliczyć drugi z podceli reguły `a(X,Y) :- b(X), c(Y).`, czyli `c(Y)`. Jako pierwszy z takim podcelem unifikuje się fakt `c(g)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `g`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca pierwszą odpowiedź `X = d  Y = g`.

Krok 2.1.2 Następnie z podcelem `c(Y)` unifikuje się fakt `c(h)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `h`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca drugą odpowiedź `X = d  Y = h`.

Krok 2.1.3 Następnie z podcelem `c(Y)` unifikuje się fakt `c(i)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `i`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca trzecią odpowiedź `X = d  Y = i`.

Krok 2.1.4 Ponieważ nie ma już niczego co pozwoli obliczyć podcel `c(Y)`, więc tym samym Prolog powraca do ostatniego punktu decyzyjnego, czyli poszukuje faktu/reguły występujących po fakcie `b(d)` i pozwalających obliczyć podcel `b(X)`.

Krok 2.2 Jako kolejny z podcelem `b(X)` unifikuje się fakt `b(e)`. To powoduje spełnienie pierwszego podcelu i ukonkretyzowanie zmiennej `X` przez `e`.

Krok 2.2.1 Prolog próbuje obliczyć drugi z podceli reguły `a(X,Y) :- b(X), c(Y).`, czyli `c(Y)`. Jako pierwszy z takim podcelem unifikuje się fakt `c(g)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `g`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca czwartą odpowiedź `X = e  Y = g`.

Krok 2.2.2 Następnie z podcelem `c(Y)` unifikuje się fakt `c(h)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `h`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca piątą odpowiedź `X = e  Y = h`.

Krok 2.2.3 Następnie z podcelem `c(Y)` unifikuje się fakt `c(i)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `i`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca szóstą odpowiedź `X = e  Y = i`.

Krok 2.2.4 Ponieważ nie ma już niczego co pozwoli obliczyć podcel `c(Y)`, więc tym samym Prolog powraca do ostatniego punktu decyzyjnego, czyli poszukuje faktu/reguły występujących po fakcie `b(e)` i pozwalających obliczyć podcel `b(X)`.

Krok 2.3 Jako kolejny z podcelem `b(X)` unifikuje się fakt `b(f)`. To powoduje spełnienie pierwszego podcelu i ukonkretyzowanie zmiennej `X` przez `f`.

Krok 2.3.1 Prolog próbuje obliczyć drugi z podceli reguły `a(X,Y) :- b(X), c(Y).`, czyli `c(Y)`. Jako pierwszy z takim podcelem unifikuje się fakt `c(g)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `g`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca czwartą odpowiedź `X = f  Y = g`.

Krok 2.3.2 Następnie z podcelem `c(Y)` unifikuje się fakt `c(h)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `h`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca piątą odpowiedź `X = f  Y = h`.

Krok 2.3.3 Następnie z podcelem `c(Y)` unifikuje się fakt `c(i)`. To powoduje spełnienie drugiego podcelu i ukonkretyzowanie zmiennej `Y` przez `i`. Ponieważ spełnione zostały wszystkie podcele reguły, więc prolog zwraca szóstą odpowiedź `X = f  Y = i`.

Krok 2.3.4 Ponieważ nie ma już niczego co pozwoli obliczyć podcel `c(Y)`, więc tym samym Prolog powraca do ostatniego punktu decyzyjnego, czyli poszukuje faktu/reguły występujących po fakcie `b(f)` i pozwalających obliczyć podcel `b(X)`.

Krok 3 Brak unifikatorów dla `a(X,Y)`. Zakończenie obliczeń dla zapytania.

Czasem nawet, celowo lub przez przypadek, możemy wygenerować nieskończoną ilość rozwiązań

```
a(0).
a(X) :- a(Y), X is Y+1.
```

Wydając teraz zapytanie `a(X)`, otrzymywać będziemy wszystkie kolejne liczby całkowite począwszy od 0.

```
?- a(X).
X = 0 ;
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5
...
itd.
```

Wyjaśnienie takiego zachowania Prologu jest bardzo proste (proszę porównać z drzewem wywołań).

Krok 1 Prolog przeszukuje swoją bazę wiedzy w poszukiwaniu faktu/reguły, które pozwolą jemu ukonkretnić zmienną `X` w zapytaniu `a(X)`. Jako pierwszy z zapytaniem unifikuje się fakt `a(0)` i stąd pierwsza odpowiedź `X = 0`.

Krok 2 Następnie z zapytaniem unifikuje się reguła `a(X) :- a(Y), X is Y+1.`, w związku z czym Prolog będzie starał się obliczyć wszystkie cele z niej wynikające, czyli w tym przypadku `a(Y)` oraz `X is Y+1.`.

Krok 2.1 Z podcelem `a(Y)` jako pierwszy unifikuje się fakt `a(0)`. To powoduje spełnienie pierwszego podcelu i ukonkretyzowanie zmiennej `Y` przez wartość `0`. Prolog zwraca pierwszą odpowiedź `X = 0`.

Krok 2.1.1 Prolog próbuje obliczyć drugi z podceli reguły, czyli `X is Y+1`. Obliczenie tego podcelu powoduje ukonkretyzowanie zmiennej `X` przez wartość `1`. Prolog zwraca drugą odpowiedź `X = 1`.

Krok 2.2 Następnie z podcelem `a(Y)` unifikuje się reguła `a(X) :- a(Y), X is Y+1.`, w związku z czym Prolog będzie starał się obliczyć wszystkie cele z niej wynikające, czyli w tym przypadku `a(Y)` oraz `X is Y+1.`.

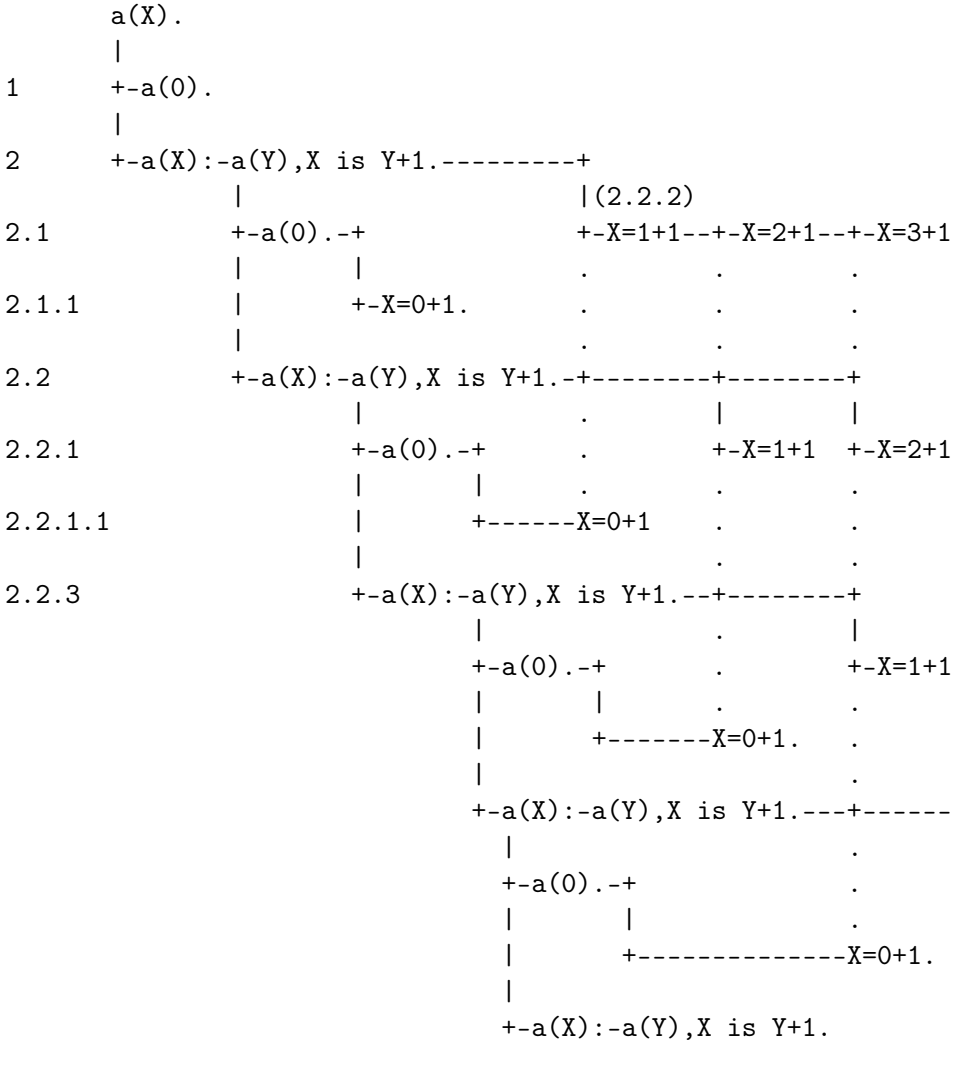
Krok 2.2.1 Z podcelem `a(Y)` jako pierwszy unifikuje się fakt `a(0)`. To powoduje spełnienie pierwszego podcelu i ukonkretyzowanie zmiennej `Y` przez wartość `0`.

Krok 2.2.1.1 Prolog próbuje obliczyć drugi z podceli reguły, czyli `X is Y+1`. Obliczenie tego podcelu powoduje ukonkretyzowanie zmiennej `X` przez wartość `1`. Tak więc w tym momencie, spełnione są wszystkie podcele podcelu `a(Y)` z kroku 2.2, co pozwala na ukonkretyzowanie zmiennej `Y` z kroku 2 przez wartość `1`.

Krok 2.2.2 Prolog próbuje obliczyć drugi z podceli reguły z kroku 2, czyli `X is Y+1`. Obliczenie tego podcelu powoduje ukonkretyzowanie zmiennej `X` przez wartość `2`. Prolog zwraca trzecią odpowiedź `X = 2`. Następnie Prolog kontynuuje próby unifikacji za krokiem 2.2.1.

Krok 2.2.3 Kontynuacja próby unifikacji za krokiem 2.2.1 a więc poszukiwanie unifikatora dla `a(Y)` co pozwoli spełnić pierwszy podcel z kroku 2.2. Kolejnym unifikatorem (pierwszym był fakt `a(0)` z kroku 2.2.1) jest reguła `a(X) :- a(Y), X is Y+1.`

. . . itd . . .



5.2 Odcięcie

Wszystkie przykłady jakie mieliśmy okazję do tej pory zobaczyć w tym rozdziale, ilustrują mechanizm nawracania. Aby choć trochę kontrolować sposób realizacj programu w Prologu możemy używać operatora odcięcia: `!`. Jeśli Prolog natrafi na operator odcięcia w regule, nie będzie nawracał z wyborem do wcześniejszych możliwości. Przyjrzyjmy się następującemu programowi (rozważanemu już w tym rozdziale, ale bez symbolu odcięcia)

```
a(X, Y) :- b(X), !, c(Y).
b(d).
b(e).
b(f).

c(g).
c(h).
c(i).
```

Po wydaniu zapytania `a(X,Y).` Prolog udzieli następujących odpowiedzi:

```
?- a(X,Y).
X = d  Y = g ;
X = d  Y = h ;
X = d  Y = i ;
No
```

Jak widać Prolog jako jedyne odpowiedzi wybrał te, dla których `X` skonkretyzowane zostało przez `d`.

Kiedy Prolog próbuje udzielić odpowiedzi na zapytanie `a(X,Y)` korzysta najpierw z pierwszej linii programu. Aby jednak udzielić odpowidzi, musi najpierw zbadać `b(X)`, co udaje się skonkretyzować dla `X = d`. Następnie Prolog natrafia na odcięcie, co powoduje, że ustala `X = d` jako jedyne rozwiązanie dla `X`. Kontynuując, badane jest wyrażenie `c(Y)`. Pierwszym możliwym ukonkretyzowaniem zmiennej `Y` jes `d`. Naciśnięcie `;` powoduje, iż Prolog poszukuje alternatywnych rozwiązań dla `c(Y)`. Zauważmy, że symbol odcięcia wystąpił **przed** ukonkretyzowaniem zmiennej `Y`, przez co wciąż można poszukiwać alternatyw dla `Y`. Nie może jednak szukać alternatyw dla `X`.

Przyjrzyjmy się poniższemu przykładowi, pokazującemu jak symbol odcięcia może „zmienić" program. Teoretycznie z podanych niżej faktów widać, że gdy `X = e`, wówczas spełnione jest `b` oraz `c`.

```
a(X) :- b(X), !, c(X).
b(d).
b(e).
b(f).

c(e).
```

Zastosowanie symbolu odcięcia daje jednak, być może nieoczekiwaną, odpowiedź:

```
?- a(X).
No
```

Gdyby pominąć symbol odcięcia, wówczas odpowiedzią byłoby oczywiście

```
?- a(X).
X = e ;
No
```

Odcięcie będzie miało także wpływ na stosowanie innych reguł. Załóżmy, że pewien program ma dwie reguły dla celu `a`

```
a(X) :- b(X), !, c(X).
a(X) :- d(X).

b(e).
b(f).

c(g).

d(f).
```

Jeśli teraz Prolog natrafi na odcięcie w pierwszej regule, wówczas nie tylko będzie to odcięcie nawracania przy konkretyzowaniu zmiennej `X`, ale zostaną także odcięte pozostałe reguły dla `a(X)`. Prolog nie będzie brał pod uwagę drugiej reguły. W tym przypadku zapytanie `a(X).` zwróci

```
?- a(X).
No
```

Widzimy, iż Prolog teoretycznie może spełnić cel podstawiając `X = f`. Najpierw jednak Prolog stara się wykorzystać pierwszą regułę, ale po natrafieniu na odcięcie musi on porzucić wszystkie alternatywy dla `a(X)`.

Zauważmy, że program zwróci `No` także dla zapytania `a(f)`. Kiedy bowiem Prolog stara się spełnić `a(f)` rozpoczyna od pierwszej reguły, gdzie początkowo „odnosi sukces" spełniając regułę `b(f)`. Następnie natrafia na symbol odcięcia. Próbuje reguł dla `c(f)`, lecz takich nie znajduje. Symbol odcięcia powoduje, że nie może on skorzystać z żadnej innej reguły poza tymi wykorzystanymi przed odcięciem.

5.3 Pytania i odpowiedzi.

Pytanie 5.1. Jak działa mechanizm nawracania w Prologu? Podaj przykład. *Przypuśćmy, że mamy pewien predykat* `a` *określony w programie przez zdania i reguły. Zapiszmy ogólnie regułę dla* `a` *w postaci* `a() :- b(), c(), d().` *przy czym funkcje* `b` , `c` , `d` *są także definiowane zarówno przez fakty, jak i reguły oraz mogą być relacjami między jakimiś zmiennymi. Mechanizm nawracania (czyli schemat postępowania Prologu podczas „przeszukiwania" bazy wiedzy w celu udzielenia odpowiedzi na pytanie* `?- a()`*) można przedstawić następująco:*

- *Przeszukiwanie linia po linii kodu programu w poszukiwaniu predykatu lub reguły o nazwie* `a`*. Jeśli jest to zdanie Prolog sprawdza, czy dla podanych zmiennych jest ono prawdziwe. Wówczas zwraca* `Yes` *i sprawdza kolejną linię kodu. W pozostałych przypadkach (zdanie jest fałszywe lub nie da się określić jego wartości logicznej) Prolog przechodzi od razu do kolejnej linii kodu.*

- *Jeśli odnaleziona będzie reguła, Prolog zacznie przeszukiwanie bazy w celu znalezienia odpowiedzi na zapytanie* `?-b()`. **Uwaga:** *Baza wiedzy jest przeszukiwana znów od pierwszej linii a nie od miejsca w którym wystąpiła reguła dla* `a`. *Po znalezieniu piewszego zdania lub reguły dla* `b` *powtarza się schemat z poprzedniego punktu i tak aż do skonkretyzowania wszystkich argumentów* `b`.

- *Następnie powtarzane są kroki z poprzednich punktów dla* `c` *oraz* `d` *i udzielana jest odpwiedź* `Yes` */*`No` *lub zwracane są wartości zmiennych które spełniają zadaną regułę i da się je znaleźć w pierwszym kroku (np. jakiejś iteracji).*

- *Prolog kontynuuje poszukiwania zdań/reguł określających* `d` *(przy ustalonych uprzednio zmiennych występujących w* `a`, `b`, `c`*) i po znalezieniu odpowiedzi lub kolejnej wartości zmienej wypisuje odpowiedź. Dopiero wtedy, gdy wszystkie możliwości skonkretyzowania wolnych zmiennych w predykacie* `d` *zostaną wyczerpane, Prolog zaczyna poszukiwać kolejnych zdań/reguł dla* `c` *nie zmieniając zmiennych związanych przy określaniu* `b`. *Powtarzane są kroki z podpunktów 2 i dalszych aż do ponownego wypisania odpowiedzi lub wartości zmiennych.*

- *Po wyczerpaniu możliwośći określenia* `b` *Prolog kontynuuje szukanie zdań/reguł dla* `a`. *Po znalezieniu każdej kolejnej powtarzane są znów wszystkie opisane już kroki aż do wypisania odpowiedzi.*

Można przekonać się o słuszności powyższej interpretacji na przykładzie programu znajdującego permutacje elemntów listy (z sortowania naiwnego). Kod programu:

```
usun(X,[X|Xs],Xs).
usun(X,[Y|Ys],[Y|Zs]) :- usun(X,Ys,Zs).
permutacja([],[]).
permutacja(Xs,[Z|Zs]) :- usun(Z,Xs,Ys), permutacja(Ys,Zs).
```

... i jego działanie:

```
 6 ?- permutacja([6,3,4,7,9],X).
X = [6, 3, 4, 7, 9] ;
X = [6, 3, 4, 9, 7] ;
X = [6, 3, 7, 4, 9] ;
X = [6, 3, 7, 9, 4] ;
X = [6, 3, 9, 4, 7] ;
X = [6, 3, 9, 7, 4] ;
X = [6, 4, 3, 7, 9] ;
X = [6, 4, 3, 9, 7] ;
X = [6, 4, 7, 3, 9] ;
X = [6, 4, 7, 9, 3] ;
X = [6, 4, 9, 3, 7] ;
X = [6, 4, 9, 7, 3] ;
X = [6, 7, 3, 4, 9] ;
X = [6, 7, 3, 9, 4] ;
X = [6, 7, 4, 3, 9] ;
X = [6, 7, 4, 9, 3] ;
X = [6, 7, 9, 3, 4] ;
X = [6, 7, 9, 4, 3] ;
```

```
X = [6, 9, 3, 4, 7] ;
X = [6, 9, 3, 7, 4] ;
X = [6, 9, 4, 3, 7] ;
X = [6, 9, 4, 7, 3] ;
X = [6, 9, 7, 3, 4] ;
X = [6, 9, 7, 4, 3] ;
X = [3, 6, 4, 7, 9] ;
X = [3, 6, 4, 9, 7] ;
X = [3, 6, 7, 4, 9] ;
X = [3, 6, 7, 9, 4] ;
X = [3, 6, 9, 4, 7] ;
X = [3, 6, 9, 7, 4]
```

Tak jak się tego spodziewaliśmy - najpierw poszukiwane są odpowiedzi dla pierwszych ustalonych przez program argumentów, czyli znjdowana jest permutacja `[6,3,4,7,9]`*, następnie cyfra 7 jest zastępowana kolejną możliwą (9) i wyznaczana jest ostatnia cyfra ciągu (7), itd.*

Pytanie 5.2. Na ile istotna dla Prologu jest kolejność faktów i reguł? Podaj przykład. *Jeśli program działa deterministycznie i nie ma w nim odcięć, to kolejeność faktów i reguł ma wpływ jedynie na kolejność znajdowania (a zatem i wypisywania na ekranie) odpowiedzi. Dokładniej jest o tym mowa w poprzednim pytaniu. Jeśli nasze zapytanie nie jest deterministyczne tzn. może zostać wygenerowany nieskończony ciąg odpowiedzi, to kolejność predykatów w regułach i faktów może być istotna. Np:*

```
a(X,Y) :- c(X), b(Y).
c(0).
c(X) :- c(Y), X is Y-1.
b(0).
b(X) :- b(Y), X is Y+1.
```

zadziała inaczej niż:

```
a(X,Y) :- b(X), c(Y).
c(0).
c(X) :- c(Y), X is Y-1.
b(0).
b(X) :- b(Y), X is Y+1.
```

o czym przekonują nas wyniki, odpowiednio

```
?- a(C,D).
C = 0,
D = 0 ;

C = 0,
D = 1 ;

C = 0,
D = 2 ;
```

oraz

```
?- a(C,D).
C = 0,
D = 0 ;

C = 0,
D = -1 ;

C = 0,
D = -2 ;
```

Formalnie pełne zbiory odpowiedzi na powyższe zapytanie dla obu reguł są identyczne, lecz biorąc pod uwagę fakt, że dysponujemy skończonym czasem na znalezienie rozwiązań, wygenerowane zbiory nie będą się pokrywać poza punktem `C=0, D=0`.

Pytanie 5.3. Co nazywamy „odcieciem"? Zilustruj działanie tego mechanizmu na przykładzie. *Operator odcięcia (*`!`*) pojawiający się w regule spełnia rolę trwałego podstawienia stałych w miejsce zmiennych poprzedzających ten operator w obrębie reguły. Jego działanie rozciąga się jednak na wszystkie kolejne pojawienia się tych zmienych w programie. Można powiedzieć, że* `!` *przesuwa początek reguły do miejsca pojawienia się w niej i redukuje ilość zmiennych w dalszym kodzie programu. Ponieważ działanie tego operatora zostało już poparte przykładami na wykładzie proponuję, „zbadać" jak zachowa się następujący program*

```
eksperyment(Y) :- !.
```

spójrzmy:

```
?- eksperyment(X).
Yes
?- eksperyment(a).
Yes
?- eksperyment(1).
Yes
?- eksperyment(*).
Yes
?- eksperyment(_).
Yes
```

Okazuje się zatem, że powyższy zapis oznacza po prostu zdanie prawdziwe a zmienna `X` *nie ma określonego typu.*

Rozdział 6

Wejście i wyjście. Operatory

6.1 Czytanie i pisanie termów

6.1.1 Czytanie termów

Najprostszym sposobem pobierania danych od użytkownika jest odczytywanie informacji wprowadzonych za pomocą urządzenia standardowego wejścia jakim zwykle jest klawiatura. Zadanie to realizuje predykat `read`. Niech dany będzie następujący program

```
a(1,b).
a(1,c).
a(2,d).
a(2,e).

e(X) :- read(Y),a(Y,X).
```

Efektem jego działania jest

```
?- e(X).
|: [1.][ENTER]
X = b ;
X = c ;
No
```

Zauważmy, że

- Wprowadzony term musi kończyć się znakiem kropki (`.`).
- Predykat `read` można uzgodnić tylko raz. Jak widać w powyższym przykładzie, Prolog podczas nawracania nie pytał już o kolejne liczby. W tym przypadku nawracanie to można wymusić predykatem `repeat` modyfikując regułę do postaci

  ```
  e(X) :- repeat,read(Y),a(Y,X).
  ```

  w efekcie czego otrzymujemy

  ```
  ?- e(X).
  |: [1.][ENTER]
  ```

```
X = b ;

X = c ;
|: [2.][ENTER]

X = d ;

X = e ;

...
```

Ogólnie mówiąc predykat `repeat/0` kończy się zawsze sukcesem przy pierwszym jego wywołaniu a także we wszystkich wywołaniach będących wynikiem nawracania. Predykaty występujące za `repeat` są wywoływane dopóki wszystkie nie zakończą się sukcesem. Dlatego najprostszym przykładem „pętli" nieskończonej w Prologu jest schemat `repeat, ..., fail.` Poniżej przedstawiamy inny prosty program wykorzystujący predykat `repeat`

```
command_loop:-
  write('repeat example'),nl,
  repeat,
  write('Enter command (end to exit): '),
  read(X),
  write(X), nl,
  X = end.
```

Efektem działania programu jest

```
?- command_loop.
Enter command (end to exit): [ala.][ENTER]
ala
Enter command (end to exit): [ma.][ENTER]
ma
Enter command (end to exit): [kota.][ENTER]
kota
Enter command (end to exit): [end.][ENTER]
end
More? [n] ;
Enter command (end to exit): [end.][ENTER]
end
More? [ENTER]

Yes
```

Jak widzimy SWI-Prolog zachowuje się w tym przypadku trochę inaczej niż tego się spodziewaliśmy. Mianowicie po wpisaniu `end.` wcale nie kończy działania, ale daj nam możliwość poszukiwania rozwiązań alternatywnych. Jest to kontrolowana „anomalia" której wyjaśnienie jest następujące[1].

[1] Źródło: `http://gollem.science.uva.nl/twiki/pl/bin/view/FAQ/MorePrompt`, stan z dnia 24 marca 2009 roku.

SWI-Prolog prompts using "more?"

SWI-Prolog has two modes for prompting after proving a toplevel goal that is controlled by the Prolog flag prompt_alternatives_on. Classical Prolog systems prompt for alternatives if the toplevel goal contains variable, regardless whether or not Prolog knows there are no alternatives. This mode is supported by the value groundness for the prompt_alternatives_on flag.

By default however, prompt_alternatives_on is set to determinism, which means Prolog prompts for alternatives if the goal succeeded with at least one choicepoint. So, a goal as below indeed prompts for the proper alternatives.

```
test :-
    member(X, [hello, world]), writeln(X).

?- test.
hello
More? ;
world
More? ;

No
```

Especially for the more serious developer, it is important that predicates that are designed to leave no choicepoints actually behave like this. This current prompt mode often spots such problems during the debugging phase, where developers often run test queries from the toplevel. You can use the graphical debugger (gtrace/0) to find the choicepoint.

Inny prosty przykład programu pobierającego dane, to program obliczający pole powierzchni kwadratu

```
kwadrat :- read(X), licz(X).

licz(stop) :- !.
licz(X) :- C is X * X, write(C),kwadrat.
```

i efekt jego działania

```
?- kwadrat.
|: 4. [ENTER]
16 [ENTER]
|: stop. [ENTER]

Yes
```

6.1.2 Pisanie termów

Do wprowadzania informacji za pomocą urządzenia standardowego wyjścia jakim zwykle jest ekran służy predykat `write`. Predykat `nl` powoduje przejście do nowej linii. Podobnie jak `read` oba predykaty można uzgodnić tylko raz.

```
a(0) :- !.
a(N) :- write('ala '), nl, N1 is N-1, a(N1).
```

i efekt jego działania

```
?- a(3).
ala
ala
ala

Yes
```

6.2 Czytanie i pisanie znaków

6.2.1 Czytanie znaków

Do pobierania pojedyńczych znaków służy predykat `get_char`

```
?- get_char(Znak).
|: ala [ENTER]


Znak = a
```

Poniżej przedstawiamy inny prosty przykład programu pobierającego pojedyńcze znaki tak długo, jak wprowadzony znak różny jest od znaku `q`

```
a :- get_char(Z),write('wprowadzony znak: '), write(Z), nl, \+ (Z=q), a.
```

```
?- a.
|: ala ma kota [ENTER]
wprowadzony znak: a
wprowadzony znak: l
wprowadzony znak: a
wprowadzony znak:
wprowadzony znak: m
wprowadzony znak: a
wprowadzony znak:
wprowadzony znak: k
wprowadzony znak: o
wprowadzony znak: t
wprowadzony znak: a
wprowadzony znak:

|: q
wprowadzony znak: q

No
```

6.2.2 Pisanie znaków

Wywołanie celu `put_char(X)`, w przypadku gdy zmienna `X` ukonkretniona jest znakiem, powoduje jego wypisanie.

```
?- put_char(a).
a

Yes
?- put_char('a').
a

Yes
```

6.3 Czytanie z pliku i pisanie do pliku

6.3.1 Czytanie z pliku

W Prologu, jak większości języków programowania, pojcie standardowego wejścia/wyjścia uogólnia się na pojęcie strumienia wejściowego/wyjściowego, które związane mogą być z dowolnym urządzeniem wprowadzającym/wyprowadzającym dane jak np. klawiatura, monitor, plik itd.

Isniejące w Prologu wbudowane strumienie `user_input` oraz `user_output` odpowiadają klawiaturze i monitorowi. Praca z plikami możliwa jest dzięki predykatom

- `open` jest to predykat służący do powiązania strumienia z plikiem.
- `close` jest to predykat pozwalający zakończyć pracę z plikiem.

Typowy program czytający dane z pliku ma następujący schemat

```
czytajPlik :-
   open('dane.txt',read,X),
   kodOdczytujacy(X),
   close(X).
```

Argumenty predykatu `open` mają następujące znaczenie

- Pierwszy argument (`'dane.txt'`) to nazwa pliku, który chcemy otworzyć.
- Drugi argument (`read`) to jeden z trybów:
  - `read` – otwarcie do odczytu istniejącego pliku;
  - `write` – utworzenie nowego pliku do zapisu (jeśli plik istnieje, zostanie usunięty);
  - `readwrite` – tryb zapisu i odczytu (jeśli plik istnieje to zostanie otworzony plik istniejący w przeciwnym razie plik zostanie utworzony);
  - `append` – istniejący plik zostanie otworzony do dopisywania.
- Trzeci argument (`X`) – to zmienna, która zostanie związana ze strumieniem.

Aby w predykacie `kodOdczytujacy` móc odczytać (zapisać) dane z (do) pliku, jak ze standardowego wejścia (wyjścia), należy najpierw zmienić strumień wejściowy (wyjściowy). Zmiana bieżącego strumienia wejściowego i wyjściowego odbywa się za pomocą predykatów `set_input` oraz `set_output`. Predykaty `current_input` oraz `current_output` pozwalają sprawdzić jak aktualnie ustawione są strumienie.

Uwzględniając te nowe predykaty, typowy program czytający plik powinien wyglądać tak

```
czytajPlik :-
   open('dane.txt',read,X),
   current_input(CI),
   set_input(X),
   kodOdczytujacy,
   close(X),
   set_input(CI).
```

Uzupełnijmy teraz kod o brakujący predykat `kodOdczytujący`

```
czytajPlik :-
   open('dane.txt',read,X),
   current_input(CI),
   set_input(X),
   kodOdczytujacy,
   close(X),
   set_input(CI).

kodOdczytujacy :- read(Term), obsluz(Term).

obsluz( end_of_file ) :- !.
obsluz(Term) :- write(Term),nl,kodOdczytujacy.
```

Efekt działania tego programu na pliku `dane.txt`

```
linia. 1.
linia. 2.
linia. 3.
i. linia. 4.
a. to. jest. ostatnia. linia. 5.
```

wygląda następująco

```
 ?- czytajPlik.
linia
1
linia
2
linia
3
i
linia
4
```

```
a
to
jest
ostatnia
linia
5

Yes
```

Zwróćmy uwagę na konieczność używania znaku kropki (.) po każdym „wyrazie". Jego pominięcie w pliku z danymi, np.

```
linia. 1.
linia 2.
linia. 3.
```

powoduje wypisanie komunikatu

```
?- czytajPlik.
linia
1
ERROR: (dane.txt:2):
Unhandled exception: dane.txt:2:0: Syntax error: Operator expected
```

Znak kropki można pominąć jeśli użyjemy w odpowiednim miejscu znaku apostrof (') lub cudzysłów ("). Używając znaku ', np.

```
linia. 1.
'linia 2'.
linia. 3.
```

efektem działania programu jest

```
?- czytajPlik.
linia
1
linia 2
linia
3

Yes
```

Używając znaku ", np.

```
linia. 1.
"linia 2".
linia. 3.
```

efektem działania programu jest

```
?- czytajPlik.
linia
1
```

```
[108, 105, 110, 105, 97, 32, 50]
linia
3

Yes
```

6.3.2 Pisanie do pliku

Na podobnych zasadach opiera się pisanie do pliku. Uruchamiając program

```
zapisz :-
   open('out.txt',write,X),
   current_output(CO),
   set_output(X),
   kodZapisujacy,
   close(X),
   set_output(CO).

kodZapisujacy :- read(X),\+ (X='quit'),write(X),nl,flush,kodZapisujacy.
```

i wprowadzając dane jak poniżej

```
?- zapisz.
|: 'ala ma kota'.
|: 'w jasne ciapki'.
|: 'quit'.

No
```

w pliku `out.txt` powinny znaleźć się następujące dane

```
ala ma kota
w jasne ciapki
```

6.4 Operatory

6.4.1 Priorytet

Kolejność „znaków" ($\cdot$, $/$, $+$, $-$) to jedne z pierwszych „praw" matematycznych jakie poznajemy na samym początku naszej edukacji. Potem dowiadujemy się, że te znaki nazywa się **operatorami** a kolejność ich rozpatrywania w wyrażeniu to właśnie **priorytet** (ang. *precedence*). Dzięki temu wiemy, że wynikiem działania

$$5 + 3 \cdot 4$$

nie jest 32 lecz 17. Mocniejsze wiązanie operatora $\cdot$ sprawia, że najpierw musimy wykonać mnożenie 3 prze 4 a dopiero potem wykonać dodawanie. Gdyby oba operatory miały identyczny priorytet, wówczas aby zachować taką kolejność należało by użyć nawiasów

$$5 + (3 \cdot 4).$$

W Prologu z każdym operatorem związana jest liczba całkowita nieujemna[2] określająca jego priorytet. Im mniejsza liczba tym wyższy priorytet operatora. Na przykład operator `*` ma priorytet 400, natomiast `+` ma priorytet 500. Priorytet termów określony jest przez liczbę 0.

6.4.2 -fixowość operatora

Każdy operator może występować w jednej lub kilku wersjach, które nazywać będziemy **fixowością** (?), określających jego położenie względem operandów. Wyróżniamy operatory

- **infixowe** a więc takie, które występują pomiędzy operandami, np. operator `+` traktowany jako operator dodawania dwóch liczb;
- **prefixowe** a więc takie, które występują przed operandem, np. operator `+` traktowany jako operator określający znak liczby;
- **postfixowe** a więc takie, które występują za operandem, np. operator ! oznaczający silnie.

6.4.3 Łączność

Ostatnią rzeczą jaka pozostaje do prawidłowego określenia operatora, jest jego **łączność** (ang. *associativity*). Bez tego trudno jest powiedzieć, czy wynikiem działania

$$9 - 6 - 2$$

będzie 1 czy 5. W przypadku arytmetyki obowiązuje zasada wykonywania działań o identycznym priorytecie od lewej strony do prawej. Inaczej mówiąc, operatory te są **lewostronnie łączne** (ang. *left-associative*), czyli wymuszają następującą kolejność wykonywania działań

$$(9 - 6) - 2$$

W Prologu łączność określana jest za pomocą atomów (wzorców) postaci `afb`, gdzie `a` i `b` mogą przyjąć wartość `x` lub `y`, natomiast `f` określa położenie operatora. Znak `y` powinno się rozumieć jako *na tej pozycji występuje term o priorytecie nie większym od priorytetu operatora*, natomiast `x` rozumiemy jako *na tej pozycji występuje term o priorytecie mniejszym od priorytetu operatora*. Wszystkie możliwe wzorce przedstawiono w tabeli 6.1. Zauważmy, że nie jest możliwe zagnieżdżanie operatorów nie posiadających łączności (np. `is`, wzorzec `xfx`). Podobnie zachowuje się wzorzec `fx`, nie pozwalając napisać np. `--3`.

Sprawdźmy jak określone są operatory wbudowane. Najpierw konkretny wybrany

```
 ?- current_op(Priorytet, Lacznosc, +).

Priorytet = 500,
Lacznosc = fx ;

Priorytet = 500,
```

[2] W będącym przedmiotem naszego zainteresowania SWI-Prolog-u mieści się ona pomiędzy 0 a 1200.

Wzorzec	Łączność		Przykłady
fx	prefix	non-associative	-
fy	prefix	łaczny (prawostronny)	
xf	postfix	non-associative	
yf	postfix	łaczny (lewostronny)	
xfx	infix	non-associative	=, is
xfy	infix	prawostronnie łączny	,
yfy	infix	**nie ma sensu**	
yfx	infix	lewostronnie łączny	+, *

Tablica 6.1: Wzorce określające łączność operatorów w Prologu.

```
Lacznosc = yfx ;

No
```

Jak widać operator + występuje zarówno w wersji prefixowej jak i infixowej. Teraz spróbujmy wyświetlić informacje o wszystkich wbudowanych operatorach

```
?- current_op(Priorytet, Lacznosc, Operator).

Priorytet = 1150,
Lacznosc = fx,
Operator = (volatile) ;

Priorytet = 400,
Lacznosc = yfx,
Operator = << ;

Priorytet = 1200,
Lacznosc = fx,
Operator = (:-) ;

Priorytet = 1200,
Lacznosc = xfx,
Operator = (:-) ;

Priorytet = 700,
Lacznosc = xfx,
Operator = (@=<) ;

Priorytet = 700,
Lacznosc = xfx,
Operator = (is) ;

Priorytet = 500,
Lacznosc = yfx,
Operator = \/ ;
```

```
Priorytet = 1200,
Lacznosc = xfx,
Operator = (-->) ;
```

Jak widać operator `:-` określania reguły także występuje zarówno w wersji prefixowej jak i infixowej. Sens wersji prefixowej poznamy niebawem. Poza tym na „liście" znaleźć można wiele egzotycznych operatorów, których jeszcze nie poznaliśmy, jak np. `@=<` czy `-->`.

6.4.4 Definiowanie własnych operatorów

Zanim przejdziemy do definiowania własnych operatorów zauważmy, że zapis

```
a-b*c
```

to inna forma zapisu

```
-(a,*(b,c))
```

Stąd wynika fakt, że operator arytmetyczny nie powoduje wykonania jakichkolwiek obliczeń. Zapis `1+2` to **nie jest** to samo co 3. Jest to po prostu inny sposób zapisania wyrażenia `+(1,2)`. I właśnie według tej zasady możemy chcieć zamiast

```
lubi(jas,malgosie).
```

pisać

```
jas lubi malgosie.
```

gdzie `lubi` to nasz operator. Próba jego użycia bez wcześniejszego zdefiniowania skazana jest na porażkę

```
?- jas lubi malgosie.
ERROR: Syntax error: Operator expected
ERROR: jas
ERROR: ** here **
ERROR: lubi malgosie .
```

Zdefiniujmy zatem nasz operator

```
?- op(500, xfx, lubi).

Yes
```

i spróbujmy go użyć. Najpierw elementarny test, np.

```
?- lubi(jas,malgosie) = jas lubi malgosie.

Yes
```

Natomiast program

```
lubi(zosia,karola).
kaska lubi antka.
pojdaDoKawiarni(X,Y) :- X lubi Y.
```

zwróci następujące wyniki

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

X = zosia,
Y = karola ;

X = kaska,
Y = antka
```

Konieczność definiowania operatorów za każdym razem gdy uruchamiamy Prolog jest bardzo mało praktyczna, ale na szczęście można definiować obiekty na etapie kompilacji programu. Pozwala na to możliwość tworzenia reguł bez głowy. Napiszmy np. taki program

```
:- write('Witaj swiecie').
:- write('Witaj jeszcze raz').
```

Uruchomienie programu spowoduje wykonanie wszystkich reguł bez głowy

```
?- [t].
Witaj swiecieWitaj jeszcze raz
% t compiled 0.00 sec, -120 bytes

Yes
```

lub

```
?- consult(t).
Witaj swiecieWitaj jeszcze raz
% t compiled 0.00 sec, 0 bytes

Yes
```

Zatem wystarczy do programu dodać regułę

```
:- op(500, xfx, lubi).
```

i po uruchomieniu możemy korzystać z operatora `lubi`

```
?- [t].
% t compiled 0.00 sec, 1,140 bytes

Yes
?- zosia lubi karola.

Yes
```

Ciekawym, choć raczej mało użytecznym, rozwiązaniem jest możliwość „przeciążania" operatorów wbudowanych

```
?- X is 2+3*5.
X = 17

Yes
?- op(100,yfx,+).

Yes
?- X is 2+3*5.
X = 25

Yes
```

6.5 Pytania i odpowiedzi

Pytanie 6.1. Co to jest operator?

Pytanie 6.2. Co określa operator?

Pytanie 6.3. Podaj przykład własnego operatora i jego użycia

Pytanie 6.4. Co to jest priorytet operatora?

Pytanie 6.5. Co to jest łączność operatora?

Rozdział 7

Predefiniowane predykaty

7.1 Sprawdzanie typu termów

Prolog umożliwia manipulowanie termami (stałymi, zmiennymi, liczbami, atomami) w zależności od ich rodzaju dzięki specjalnym procedurom systemowym pozwalającym określić ich typ.

- Predykat `var(X)` jest spełniony, jeżeli `X` jest zmienną wolną (nie związaną).
- Predykat `nonvar(X)` jest spełniony, jeżeli `X` jest termem innym niż zmienna lub jest zmienną związaną.
- Predykat `integer(X)` jest prawdziwy, jeżeli `X` jest stałą lub zmienną związaną całkowitoliczbową.
- Predykat `real(X)` jest prawdziwy, jeżeli `X` jest stałą lub zmienną związaną rzeczywistą.
- Predykat `atom(X)` jest spełniony, jeżeli `X` jest stałą lub zmienną związaną atomową.
- Predykat `atomic(X)` jest prawdziwy, jeżeli `X` jest stałą lub zmienną związaną liczbową lub atomową.

```
?- var(X),X=2.
X = 2
Yes

?- X=2,var(X).
No

?- integer(X),X=2.
No

?- X=2,integer(X).
X = 2

?- X=2,integer(X),nonvar(X).
X = 2
```

```
Yes

?- X = 1+1,integer(X).
No

?- X is 1+1,integer(X).
X = 2

?- X is 0.5+0.5,integer(X).
No

?- X is 0.5+0.5,float(X).
X = 1.0

?- atom(2).
No

?- atom(a).
Yes

?- atom(>).
Yes

?- atomic(2).
Yes

?- X=2,atomic(X).
X = 2

?- X=b,atomic(X).
X = b

?- atomic(p(1)).
No
?-
```

Rozważmy następujący progrm Prologu

```
count(_,[],0).
count(A,[A|T],N) :- !,count(A,T,N1), N is N1 + 1.
count(A,[_|T],N) :- count(A,T,N).
```

Intencją naszą było, aby predykat `count(A,B,C)` zliczał ilość `C` elementów `A` zadanej listy `B`.

```
Yes
?- count(a,[a,b,a,a],N).
N = 3 ;
No
```

```
?- count(a,[a,b,X,Y],N).
X = a
Y = a
N = 3 ;
No

?- Lista=[a,b,X,Y],count(a,Lista,Na),count(b,Lista,Nb).
Lista = [a, b, a, a]
X = a
Y = a
Na = 3
Nb = 1 ;
No
```

Jak jednak widać działanie nie do końca zgodne jest z naszymi oczekiwaniami. Zgodnie bowiem z definicją predykatu `count` sprawdzane są nie faktyczne wystąpienia szukanego termu, lecz jego możliwe dopasowania.

Modyfikując nieznacznie program i wykorzystaujac predykat `atom` uzyskamy pierwotnie planowane działanie.

```
count(_,[],0).
count(A,[B|T],N) :- atom(B),A=B,!,count(A,T,N1), N is N1 + 1.
count(A,[_|T],N) :- count(A,T,N).
```

Efekt działania zmodyfikowanego programu

```
?- Lista=[a,b,X,Y],count(a,Lista,Na),count(b,Lista,Nb).
Lista = [a, b, _G189, _G192]
X = _G189
Y = _G192
Na = 1
Nb = 1 ;
No
```

7.2 Konstruowanie i dekompozycja termów

Jak wiemy, w Prologu nie ma możliwości zapytania o predykat, a więc nie jest możliwe np. takie zapytanie

```
X(jas,malgosia).
```

które intuicyjnie rozumiemy jako pytanie o relację laczącą obiekt `jas` z obiektem `malgosia`. Na szczęście w Prolog mamy do dyspozycji predykaty systemowe przeznaczone do konstruowania i dekompozycji termów. Predykat `=..` służy do konstruowania termu z listy atomów. Cel `Term =.. Lista` jest spełniony, jeżeli lista `Lista` zawiera nazwę funktora termu `Term` i wszystkie jego kolejne argumenty.

```
?- f(a,b)=..L.
L = [f, a, b]
```

```
?- T=..[rectangle,3,5].
T = rectangle(3, 5)

?- Z=..[p,X,f(X,Y)].
Z = p(X, f(X, Y))
```

Predykat `=..` wykorzystać można do manipulowania termami opisującymi różne klasy obiektów w celu wykonania na nich jednej (tej samej) operacji, ale mającej inny przebieg dla każdej klasy.

Załóżmy, że mamy pewien zestaw termów reprezentujących figury geometryczne, np. `kwadrat(X)`,`prostokat(X,Y)`, itd. Jako przykład rozważmy teraz term służący do powiększania figur o zadany czynnik. Chcieli byśmy móc napisać

```
powieksz(FigOld,Czynnik,FigNew)
```

co powinno spowodować powiększenie figury `FigOld` o czynnik `Czynnik` w wyniku czego otrzymamy `FigNew`. Zadanie to możemy rozwiązać w następujący sposób.

```
powieksz(kwadrat(X),Y,kwadrat(X1)) :- X1 is F * X.
powieksz(prostokat(X,Y),Z,prostokat(X1,Y1)) :-
             X1 is F * X, Y1 is F * Y.
```

Efektem działania będzie

```
?- powieksz(kwadrat(3),3,kwadrat(X)).
X = 9

?- powieksz(kwadrat(3),3,A).
A = kwadrat(9)
```

Rozwiązanie takie jest jak najbardziej poprawne, choć ma jedną wadę: musimy z góry uwzględnić wszystkie figury geometryczne, ewentualnie w razie konieczności, rozszerzyć definicję `powieksz/3`. Uogólniona definicja może wyglądać jak poniżej.

```
powieksz(FO,C,FN) :- FO=..[Typ|Arg],
                     mnozliste(Arg,C,ArgNew),
                     FN=..[Typ|ArgNew].
mnozliste([],_,[]).
mnozliste([X|T],C,[X1|T1]) :-
               X1 is X * C, mnozliste(T,C,T1).
```

Tym razem predykat `powieksz` bedzie działał dla dowolnych predykatów, bez potrzeby ich wcześniejszego definiowania

```
?- powieksz(kwadrat(3),3,kwadrat(X)).
X = 9

?- powieksz(kwadrat(3),3,A).
A = kwadrat(9)

?- powieksz(szczupak(3,4,5),3,A).
A = szczupak(9, 12, 15)
```

7.3 Podstawienia, funktor, argument

Do podstawiania za term innego termu służy predykat `subst(SubT,T,Sub,Res)`. Jest on spełniony, jeżeli wszystkie wystąpienia termu `SubT` w termie `T` zostaną zastąpione przez term `Sub` i otrzymamy w efekcie term `Res`.

```
?- subst(sin(x),2*sin(x)*f(sin(x)),t,F).
F = 2*t*f(t)
?- subst(a+b,f(a,A+B),v,T).
T = f(a,v)
A = a
B = b
```

Dla osób dociekliwych pokazujemy jak może wyglądać definicja tego predykatu

```
subst(Term,Term,Term1,Term1) :- !.
subst(_,Term,_,Term) :- atomic(Term),!.
subst(SubT,T,SubT1,T1):-
   T=..[F|Arg],
   substlist(SubT,Arg,SUbT1,Arg1),
   T1=..[F|Arg1].
substlist(_,[],_,[]).
substlist(Sub,[T|Ts],Sub1,[T1|Ts1]):-
   subst(Sub,T,Sub1,T1),
   substlist(Sub,Ts,Sub1,Ts1).
```

Predykat `functor(Term,F,N)` jest spełniony, jeżeli `F` jest głównym funktorem termu `Term`, którego aruność wynosi `N`.

Predykat `arg(N,Term,A)` jest spełniony, jeżeli `A` jest `N`-tym argumentem termu `Term`, przy założeniu, że numerowanie zaczyna się od 1.

```
?- functor(t(f(X),X,t),Func,Arity).
Func = t
Arity = 3

?- arg(2,f(X,t(a),t(b)),Y).
X = _G184
Y = t(a)

?- functor(D,data,3),
   arg(1,D,30),
   arg(2,D,czerwca),
   arg(3,D,2007).
D = data(30,czerwca,2007).
```

7.4 Różne różności, różne równości

Rodzaje relacji równości/różności w Prologu:

- `X=Y`, prawdziwy, gdy term `X` i `Y` unifikują się;

- `X is E`, prawdziwy, gdy `X` unifikuje się z wartością wyrażenia `E`;
- `E1 =:= E2`, prawdziwy, gdy wartości wyrażeń `E1` i `E2` są równe;
- `E1 =\= E2`, prawdziwy, gdy wartości wyrażeń `E1` i `E2` są różne;
- `T1 == T2`, prawdziwy, gdy termy `T1` i `T2` są identyczne (identyfikują się leksyklanie, łącznie z nazwami zmiennych);
- `T1 \== T2`, prawdziwy, gdy termy `T1` i `T2` nie są identyczne.

```
?- X=5,X=5.
X = 5

?- X=5,X=2+3.
No

?- 5=X.
X = 5

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

?- X=5,X is 2+3.
X = 5

?- 2+3 = 2+3.
Yes

?- 2+3 is 2+3.
No

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

?- X=2+3, X is 2+3.
No

?- X=2+3, X =:= 2+3.
X = 2+3

?- 2+3 =:= 2+3.
Yes

?- X=2+3, 2+3 =:= X.
X = 2+3

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

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

?- f(a,b)==f(a,b).
Yes

?- f(a,b)==f(a,X).
No

?- f(a,X)==f(a,Y).
No

?- X\==Y.
Yes

?- t(X,f(a,Y))==f(X,f(a,Y)).
Yes
```

7.5 Manipulowanie bazą danych

Traktując program Prologu jako bazę danych wyróżnić możemy

- klauzule bezwarunkowe – fakty reprezentujące jawne relacje;
- klauzule warunkowe – fakty reprezentujące niejawne relacje.

Do manipulowania bazą klauzul służą:

- `assert(C)` zawsze spełniony cel dodający klauzulę `C`;
- `asserta(C)` zawsze spełniony cel dodający klauzulę `C` na początku bazy;
- `assertz(C)` zawsze spełniony cel dodający klauzulę `C` na końcu bazy;
- `retract(C)` zawsze spełniony cel usuwający klauzulę `C`;

```
?- assert(p(a)),assert(p(b)),assertz(p(c)),assert(p(d)),asserta(p(e)).
Yes

?- p(X).
X = e ;
X = a ;
X = b ;
X = c ;
X = d
```

Dodawane klauzule funkcjonują dokładnie tak samo jak klauzule zawarte w pierwotnym programie. Zastosowanie powyższych predykatów umożliwia adaptowanie programu do zmieniających się warunków działania.

Niech dana będzie następująca baza

```
jestFajnie :- jestSlonce,not(padaDeszcz).
mozeByc :- jestSlonce,padaDeszcz.
jestBeznadziejnie :- padaDeszcz,jestMgla.
padaDeszcz.
jestMgla.
```

Operacje na bazie

```
?- jestFajnie.
No

?- jestBeznadziejnie.
Yes

?- retract(jestMgla).
Yes

?- jestBeznadziejnie.
No

?- assert(jestSlonce).
Yes

?- mozeByc.
Yes

?- retract(padaDeszcz).
Yes

?- jestFajnie.
Yes
```

7.6 Pytania i odpowiedzi

Pytanie 7.1. Kiedy i jak można wykorzystać możliwosć sprawdzania typu termów?

Pytanie 7.2. W jaki sposób możemy uzyskać informaję o funktorze lub jego argumentach?

Pytanie 7.3. W jaki sposób możemy wpływać na strukturę programu (np. zmieniać jego fakty)?

Pytanie 7.4. Jaka jest różnica pomiędzy operatorem = a `is`?.

Rozdział 8

Powtarzamy wiadomości

8.1 O rekurencji raz jeszcze

W ćwiczeniach 13.1 pisaliśmy między innymi funkcje określające pokrewieństwo. Załóżmy, że dopiszemy teraz następującą relację

```
potomek(Nastepca,Przodek) :- rodzic(Przodek,Nastepca).
potomek(Nastepca,Przodek) :- rodzic(Ktos,Nastepca),potomek(Ktos,Przodek).
```

Pierwsza definicja ma zastosowanie „w prostej linii" czyli w bezpośredniej relacji *Przodek–Nastepca*. Druga definicja stosowana jest wówczas gdy *Nastepca* nie jest bezposrednio potomkiem *Przodka*, ale za to ma rodzica (*Ktos*) który jest potomkiem *Przodka* – i tutaj oczywiście trafiamy na rekurencję.

Może się to wydawać zaskakujące, ale samą rekurencję możemy napisać na wiele sposobów.

8.1.1 Sposób 1

```
potomek(Nastepca,Przodek) :- rodzic(Ktos,Nastepca),potomek(Ktos,Przodek).
```

Od tego sposobu zaczęliśmy ten podrozdział – jest to chyba najbardziej naturalne podejście. Rozpoczynamy od poszukiwania rodzica *Nastepcy* a następnie pytamy, czy ten rodzic (*Ktos*) jest potomkiem *Przodka*.

Poszukiwanie takie jest charakterystyczne wówczas gdy znamy Nastepce a poszukujemy *Przodka*.

8.1.2 Sposób 2

```
potomek(Nastepca,Przodek) :- potomek(Ktos,Przodek),rodzic(Ktos,Nastepca).
```

Sposób bardzo podobny do poprzedniego, ale z zamienioną kolejnościa celów w regule. Rozpoczynamy od poszukiwania jakiegokolwiek potomka *Przodka* a następnie sprawdzamy czy znaleziony potomek (*Ktos*) jest rodzicem *Nastepcy*.

Zauważmy, że w tej sytuacji zanim natrafimy na właściwego następcę (*Ktos*) *Przodka* który okaże się rodzicem *Nastepcy* może mieć miejsce wiele wywołań rekurencyjnych i możemy otrzymać wielu potomków, którzy jednak nie będą rodzicem *Następcy*. Z tego też powodu, nie zaleca się stosowania rekurencji jako pierwszej z reguł, choć często jest to najbardziej intuicyjne podejście.

8.1.3 Sposób 3

```
potomek(Nastepca,Przodek) :- rodzic(Przodek,Ktos),potomek(Nastepca,Ktos).
```

Porównywalne do sposóbu 1 jest następujące podejście: szukamy kogo (*Ktos*) rodzicem był *Przodek* a nastepnie sprawdzamy czy istnieje relacja potomek pomiędzy *Nastepca* a *Ktos*.

Poszukiwanie takie jest charakterystyczne wówczas gdy znamy *Przodka* a poszukujemy *Nastepcy*. Zauważmy, że sposób ten okaże się mniej wydajny od sposobu 1 jeśli ilość dzieci będzie większa od 2 (ilość rodziców wynosi 2).

8.1.4 Sposób 4

```
potomek(Nastepca,Przodek) :- potomek(Nastepca,Ktos),rodzic(Przodek,Ktos).
```

Ponownie, jak to miało miejsce dla sposobu 2, możemy opierając się na sposobie 3 skonstruować sposób 4, ale z zamienioną kolejnością celów w regule. Z przyczyn opisanych powyżej, taki sposób zapisu nie jest zalecany ze względu na możliwe obniżenie wydajności.

8.2 Akumulator

Z pewnością już nie raz zetknęliśmy się z akumulatorem, choć wcale nie musieliśmy zdawać sobie z tego sprawy. Czym więc jest ów *akumulator*?

8.2.1 Przykład z listą

Przyjrzyjmy się następującym dwóm definicjom funkcji obliczającej długość listy

1. ```
   len([],0).
   len([_|Tail],Len) :- len(Tail,LenTail),
                        Len is LenTail + 1.
   ```

2. ```
   lenA(List,Len) :- lenA(List,0,Len).
   lenA([],Len,Len).
   lenA([_|Tail],Acc,Len) :- NewAcc is Acc + 1,
                             lenA(Tail,NewAcc,Len).
   ```

Zasadniczą różnicą jest miejsce obliczania wyniku: `len` robi to powracając z rekurencji, natomiast `lenA` schodząc w rekurencji.

Przyjrzyjmy się co dokładniej dziej się w obu funkcjach.
Funkcja `len`

```
?-len([a,b,c],X).
         (11) X = 3
      len([_|Tail],Len) <--- (1) pasuje do tego             2    + 1
  (1) len([a,b,c] ,X) :- len([a,b],LenTail), (10) Len is LenTail + 1
                             (9) LenTail = 2
      len([_|Tail],Len)                                        1    + 1
  (2) len([a,b]   ,LenTail) :- len([a],LenTail), (8) Len is LenTail + 1
```

```
                                  (7) LenTail = 1
     len([_|Tail],Len)                                        0    + 1
  (3) len([a]     ,LenTail) :- len([],LenTail), (6) Len is LenTail + 1
                                  (5) LenTail = 0
     len([],0)
  (4) len([],0)
```

Funkcja `lenA`

```
?-lenA([a,b,c],X).
          (11) X = 3
     lenA(List   ,Len)
  (1) lenA([a,b,c],X) :- lenA([a,b,c],0,Len).
                                  (10) Len = 3
     lenA([_|Tail],Acc,Len)
  (2) lenA([a,b,c] ,0  ,Len) :- NewAcc is 0 + 1, lenA([b,c],1,Len)
                                                      (9) Len = 3
     lenA([_|Tail],Acc,Len)
  (3) lenA([b,c]   ,1  ,Len) :- NewAcc is 1 + 1, lenA([c],2,Len)
                                                    (8) Len = 3
     lenA([_|Tail],Acc,Len)
  (4) lenA([c     ],2  ,Len) :- NewAcc is 2 + 1, lenA([],3,Len)
                                                  (7) Len = 3
     lenA([],Len,Len).
  (5) lenA([],3  ,Len)
             (6) Len = 3
```

Użycie akumulatora wymusza użycie dodatkowego argumentu w predykacie – w przykładzie jest to drugi argument: `Acc`. Dlatego zwykle używa się funkcji opakowującej np. postaci

```
lenA(List,Len) :- lenA(List,0,Len).
```

co zwalnia użytkownika z konieczności podawania w zapytaniu wartości inicjującej (w tym przypadku zera).

8.2.2 Przykład z liczbami

W ćwiczeniach 13.2 posługiwaliśmy się dosyć specyficzną definicją liczby naturalnej. Mając funkcję następnika f mogliśmy dowolną liczbę zapisać jako następnik następnika następnika ... następnika zera, np. $3 = f(f(f(0)))$. Określiliśmy tam m.in. dodawanie takich liczb jako

```
add(0,X,X).
add(f(X),Y,f(Z)) :- add(X,Y,Z).
```

co powoduje następujący ciąg wywołań

```
?-add(f(f(0)),f(f(f(0))),X).
                    (7) X = f(f(f(f(f(0)))))
```

```
      add(f(X)   ,Y         ,f(Z))                          Z
  (1) add(f(f(0)),f(f(f(0))),X   ) :- add(f(0),f(f(f(0))),X)
                                                    (6) Z = f(f(f(f(0))))

      add(f(X),Y         ,f(Z))                      Z
  (2) len(f(0),f(f(f(0))),Z   ) :- add(0,f(f(f(0))),X)
                                                (5) Z = f(f(f(0)))

      add(0,X         ,X)
  (3) len(0,f(f(f(0))),Z)
                 (4) Z = f(f(f(0)))
```

Wersja akumulatorowa dodawania przedstawia się następująco

```
addA(0,X,X).
addA(f(X),Y,Z) :- addA(X,f(Y),Z).
```

co powoduje następujący ciąg wywołań

```
?-addA(f(f(0)),f(f(f(0))),X).
                     (7) X = f(f(f(f(f(0)))))

      addA(f(X)   ,Y         ,Z)              f(Y         ) Z
  (1) addA(f(f(0)),f(f(f(0))),X) :- addA(f(0),f(f(f(f(0)))),X)
                                                  (6) Z = f(f(f(f(f(0)))))

      addA(f(X),Y            ,Z)           f(Y            ) Z
  (2) lenA(f(0),f(f(f(f(0)))),X) :- addA(0,f(f(f(f(f(0))))),X)
                                                  (5) Z = f(f(f(f(f(0)))))

      addA(0,X               ,X)
  (3) lenA(0,f(f(f(f(f(0))))),Z)
                         (4) Z = f(f(f(f(f(0)))))
```

8.3 Z góry na dół czy od dołu do góry?

8.3.1 Z góry na dół

Metoda obliczeniowa „z góry na dół" (ang. *top down computation*), typowa dla Prologu, rozpoczyna od problemu wyjściowego a następnie rozkład go na podproblemy prostsze a te z kolei na jeszcze prostsze itd. aż dojdzie do przykładu trywialnego. W ten oto sposób rozwiązanie dużego zagadnienia stanowi rozwiązanie wielu podproblemów prostszych. Jednym z najprostszych przykładów ilustrujących taką metodologię obliczeniową i programistyczną jest obliczanie wyrazów ciągu Fibonacciego.

```
fibTD(0,0).
fibTD(1,1).
fibTD(N,X) :- N>1,
              N1 is N-1,
              N2 is N-2,
```

```
        fibTD(N1,X1),
        fibTD(N2,X2),
        X is X1 + X2.
```

Jeśli szukamy liczby Fibonacciego dla $N > 1$ obliczamy najpierw liczby Fibonacciego dla $N - 1$ oraz $N - 2$ i używając tych rezultatów pośrednich obliczamy ostatecznie wartość dla N.

8.3.2 Z dołu do góry

Metoda obliczeniowa „z dołu do góry" (ang. *bottom up computation*) rozpoczyna od znanych faktów a następnie rozszerza je w oparciu o posiadane reguły i fakty tak długo aż nie zostanie rozwiązany problem wyjściowy.

W ogólności metoda ta jest mniej efektywna w porównaniu z metodą „z góry na dół" co spowodowane jest generowaniem dużej ilości faktów nie mających nic wspólnego z poszukiwanym rozwiązaniem. Z tego powodu w Prologu używana jest metoda z „góry na dół" choć możliwe jest „symulowanie" metody „z dołu do góry" przy wykorzystaniu dodatkowych zmiennych – właśnie poznanych akumulatorów. Działanie takie ma o tyle sens, że często funkcje wykorzystujące akumulator (czyli działające „z dołu do góry") okazują się wydajniejsze od funkcji „z góry na dół".

```
fibBU(N,X) :- fibBU(0,0,1,N,X).

fibBU(N,X,_,N,X)
fibBU(N1,X1,X2,N,X) :- N1<N,
                       N2 is N1 + 1,
                       X3 is X1 + X2,
                       fibBU(N2,X2,X3,N,X).
```

Zauważmy, że złożoność obliczeniowa dla tego przypadku jest liniowa w przeciwnieństwie do metody poprzedniej, gdzie była wykładnicza.

8.4 Pytania i odpowiedzi

Pytanie 8.1. Co to jest akumulator? Podaj przykład.

Pytanie 8.2. Obliczenia „z góry na dół" czy „z dołu do góry"? Dlaczego?

Rozdział 9

Sortowanie

W tym rozdziale nie poznamy raczej niczego nowego, ale wzorem rozdziału poprzedniego zawarte tutaj treści potraktujemy jako przypomnienie tego wszystkiego co było omawiane do tej pory. Jako przypomnienie, ale zarazem jako praktyczne wykorzystanie do rozwiązania konkretnych problemów. Przedmiotem naszych rozważań będą różne algorytmy sortujące. Zapewne doskonale znane są nam ich implementacje w językach np. zorientowanych obiektowo, więc łatwo będziemy mogli porównać i ocenić stopień złożoności kodu.

9.1 Sortowanie naiwne

Sortowanie naiwne to chyba najprostszy przykład sortowania. Algorytm jest niezwykle prosty. Niech $\{c_i\}_{i=1,\ldots,n}$ będzie ciągiem, który ma zostać posortowany.

1. Wykonaj podstawienie $\{t_i\}_{i=1,\ldots,n} = \{c_i\}_{i=1,\ldots,n}$.
2. Sprawdź czy ciąg $\{t_i\}_{i=1,\ldots,n}$ jest posortowany. Jeśli tak, przejdź do 5, jeśli nie kontynuuj.
3. Wygeneruj nową permutację ciągu $\{c_i\}_{i=1,\ldots,n}$ i oznacz ją t

$$\{t_i\}_{i=1,\ldots,n} = perm(\{c_i\}_{i=1,\ldots,n}).$$

4. Idź do 2.
5. Ciąg $\{t_i\}_{i=1,\ldots,n}$ jest posortowanym ciągiem $\{c_i\}_{i=1,\ldots,n}$. Koniec algorytmu.

Jak więc widać, generujemy kolejne permutację zadanego ciągu, sprawdzając za każdym razem, czy wygenerowana permutacja nie zawiera elementów w dobrej kolejności. Implementacja tego algorytmu w Prologu jest niezwykle prosta.

```
nSort(X,Y) :- permutacja(X,Y), uporzadkowana(Y).

usun(X,[X|Xs],Xs).
usun(X,[Y|Ys],[Y|Zs]) :- usun(X,Ys,Zs).

permutacja([],[]).
permutacja(Xs,[Z|Zs]) :- usun(Z,Xs,Ys), permutacja(Ys,Zs).
```

```
uporzadkowana([_]).
uporzadkowana([X,Y|R]) :- X =< Y, uporzadkowana([Y|R]).
```

Schemat czynności wykonywanych podczas wykonywania przez programy wszelakich algorytmów najłatwiej zrozumieć na przykładzie. Zacznijmy od sortowania listy jednoelementowej, czyli zapytania, np.: `?- nSort([1],Y).`

- zgodnie z powyższym algorytmem „wywoływany" jest term

  ```
  nSort([1],Y) :- permutacja([1],Y), uporzadkowana(Y).
  ```

  Prolog natrafia na term `permutacja([1],Y)`, więc sprawdza swoją bazę wiedzy w poszukiwaniu odpowiedzi na pytanie `?- permutacja([1],Y).`

- Po dojściu do odpowiedniego termu Prolog buduje zdanie

  ```
  permutacja([1],[Z|Zs]) :- usun(Z,[1],Ys), permutacja(Ys,Zs).
  ```

- Teraz odszukiwana jest funkcja `usun` w celu otrzymania odpowiedzi na pytanie `?- usun(Z,[1],Ys).` Zobaczmy, co Prolog znajduje

  ```
  ?- usun(Z,[1],Ys).
  Z = 1,
  Ys = [] ;
  No
  ```

- Mając powyższe wartości `Z` oraz `Ys` program może je podstawić w odpowiednim miejscu, co daje

  ```
  permutacja([1],[1|Zs]) :- usun(1,[1],[]), permutacja([],Zs).
  ```

- w bazie wiedzy odnajdywane jest zdanie `permutacja([],[]).`, co prowadzi do podstawienia `Zs=[]` i w konsekwencji określenia wartości zmiennej `Y` w sposób: `Y=[1|[]]=[1]`.

- Prolog kontynuuje „obliczenia" dla wyjściowego termu poszukując w bazie wiedzy odpowiedzi na pytanie `?- uporzadkowana([1]).` Ponieważ znajduje się ono explicite w bazie, udzielana jest odpowiedź `Yes`, zatem cały poprzednik wyjściowej implikacji jest prawdziwy i wszystkie wolne zmienne (a zwłaszcza `Y`) zostały podstawione przez stałe, co prowdzi do okeślenia przez Prolog zdania `nSort([1],[1]).` jako prawdziwego i w konsekwencji do wypisania na ekranie odpowiedzi

  ```
  ?- nSort([1],Y).
  Y = [1] ;
  No
  ```

Algorytm sortowania listy dwuelementowej będzie różnił się od powyższego głównie na etapie korzystania z funkcji `permutacja(X,Y).` Zatem prześledźmy ten fragment dla zapytania `?- permutacja([1,2],Y).`:

- wywoływany jest term

```
permutacja([1,2],[Z|Zs]) :- usun(Z,[1,2],Ys), permutacja(Ys,Zs).
```

zatem (podobnie jak poprzednio) znajdowany jest zakres:

```
?- usun(Z,[1,2],Ys).
Z = 1,
Ys = [2] ;

Z = 2,
Ys = [1] ;
No
```

Zmienne są ukonkretniane najpierw przez pierwszą parę `Z=1,Ys=[2]`, zatem...

- ...Prolog stawia kolejne pytanie `?- permutacja([2],Zs).` Zgodnie z analizą zachowania się Prologu w takiej sytuacji (przeprowadzoną dla sortowania listy jednoelementowej) zostanie znaleziona wartość `Zs=[2]`.

- Mając wszystkie zmienne określone i wartość logiczną poprzednika implikacji „prawda" program może wypisać pierwszą permutację, gdyż `[Z|Zs]=[1,2]`.

- Następnym krokiem Prologu jest ukonkretnienie zmiennych `Z` i `Ys` przez drugą znalezioną przez funkcję `usun(Z,[1,2],Ys).` parę, mianowicie `Z=2,Ys=[1]`. Dalsze postępowanie jest oczywiście identczne jak dla poprzedniej pary, doprowadzi więc do znalezienia szukanej permutacji `Y=[Z|Zs]=[2,1]`.

Zastanówmy się teraz jak zostanie rozwiązany problem permutacji listy trójelementowej `[1,2,3]`:

- najpierw uruchomiona zostanie funkcja

```
?- usun(Z,[1,2,3],Ys).
Z = 1,
Ys = [2, 3] ;

Z = 2,
Ys = [1, 3] ;

Z = 3,
Ys = [1, 2] ;
No
```

a za pierwszą parę rozwiązań podstawione zostaną `Z=1,Ys=[2,3]`.

- wywołanie funkcji `permutacja([2,3],Zs).` uruchomi algorytm znany z poprzedniego przykładu, czyli doprowadzi do znalezienia dwóch pierwszych rozwiązań w postaci list `[1,2,3], [1,3,2]`.

- podstawianie za zmienne `Z` oraz `Ys` kolejnych rozwiązań da następne permutacje. Łatwo domyślić się, że rozwiązania pojawią się w kolejności

  `[2,1,3], [2,3,1], [3,1,2], [3,2,1]`

Oczywiście każdorazowo po znalezieniu jakiejś permutacji wyjściowej listy (tzn. tej, która jest argumentem f-cji `nSort`) wykonywana jest rekurencja `uporządkowana` i dopiero, gdy jest ona dla konkrtnej permutacji zdaniem prawidziwym, wypisywane jest rozwiązanie w postaci posortowanej listy lub... list! Niestety, gdy sortujemy tą metodą listy nieróżnowartościowe, to ilość (identycznych) rozwiązań, które pojawią się na ekranie wynosi: $\prod_{i=1}^{k} N_i!$ gdzie k oznacza ilość różnych liczb w liście natomiast N_i jest krotnoscią wystąpienia $i-$tej wartości. Np.: wyniki sortowania listy `[1,1,1,2,2]` pojawi się 12 razy. Można temu zapobiec, używając symbolu odcięcia

```
nSort(X,Y) :- permutacja(X,Y), uporzadkowana(Y),!.
```

Symbol ten jest tym bardziej wskazany, gdyż doskonale wiemy, że po znalezieniu posortowanej sekwencji nie ma sensu kontynuować poszukiwań, bo innych sekwencji i tak się nie znajdzie. Co najwyżej znajdzie się sekwencje jedynie przez Prolog traktowane jako inne[1], np. wspomnianą sekwencję `[1,1,1,2,2]`, która wyświetlona zostanie aż 12 razy, choć za każdym razem jest to taka sama sekwencja.

9.2 Sortowanie bąbelkowe

Działanie algorytmu sortowania bąbelkowego polega na porównywaniu par elementów znajdujących się obok siebie i, jeśli okazuje się to potrzebne, zmienianiu ich kolejności. Każdy element jest tak długo przesuwany w ciągu, aż napotkany zostanie element większy od niego, wtedy w następnych krokach przesuwany jest ten większy element.

Po pierwszym przebiegu ciąg nie musi być jeszcze uporządkowany, ale na ostatniej pozycji znajduje się maksymalny element ciągu. Zatem w drugim przebiegu można sortować krótszy podciąg (z pominięciem elementu ostatniego). Po drugim przebiegu, dwa największe elementy są na swoich miejscach (na końcu ciągu), a więc wystarczy posortować podciąg o dwa elementy krótszy, itd[2]. Sortowanie kończy się, gdy podczas kolejnego przejścia nie dokonano żadnej zmiany.

Implementacja algorytmu w Prologu wygląda następująco.

```
bSort(List,Sorted) :- swap(List,List1), !, bSort(List1,Sorted).
bSort(Sorted,Sorted).

swap([X,Y|Rest],[Y,X|Rest]) :- X > Y.
swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
```

Zacznijmy tym razem od prześledzenia sortowania listy pustej, czyli od zapytania

```
?- bSort([],Sorted).
```

[1] Traktowane jako inne, gdyż w inny sposób uzyskane.

[2] Algorytm można usprawnić jeszcze bardziej. Jeżeli w pewnej iteracji ostatnia zamiana nastąpiła na pozycji i, to w następnej wystarczy sortować tylko elementy na pozycjach od 1 do $i-1$.

- Rozpatrywana jest pierwsza reguła dotycząca tej funkcji, czyli

  ```
  bSort([],Sorted) :- swap([],List1), !, bSort(List1,Sorted).
  ```

- następnie, zgodnie z powyższym zapisem, poszukiwana jest odpowiedź na pytanie

  ```
  ?- swap([],List1).
  ```

  Lista pusta nie pasuje do żadnego z definiujących `swap` schematów, więc zwracana jest odpowiedź `No` i w konsekwencji reguła nie jest spełniona, zatem...

- ... Prolog przechodzi do próbowania drugiej, co skutkuje podstawieniem za zmienną `Sorted` (w drugiej linii programu) listy pustej, co jest ostatecznym rozwiązaniem problemu, gdyż otrzymujemy prawdziwe zdanie: `bSort([],[])`.

Sortowanie bąbelkowe listy jednoelementowej jest analogiczne – także i w tym przypadku predykat `swap` będzie fałszywy, więc pierwsza reguła `bSort` nie zostanie spełniona, druga zaś zwraca wejściową listę.

Przyjrzyjmy się sortowaniu listy dwuelementowej, np. `[2,1]`.

- Pierwsza reguła, po podstawieniu argumentu, wygląda następująco

  ```
  bSort([2,1],Sorted) :- swap([2,1],List1), !, bSort(List1,Sorted).
  ```

  Prolog zaczyna więc znów od funkcji `swap`.

- Na pytanie

  ```
  ?- swap([2,1],List1).
  ```

  odpowiedzi udzieli już pierwsza reguła określająca ten predykat, gdyż zmienne `X` i `Y` są już ukonkretnione odpowiednio przez `2` i `1`, zatem spełniony jest warunek `X>Y` . Zostanie ukonkretniona zmienna `List1=[1,2]`.

- Ponieważ Prolog zawsze szuka odpowiedzi posuwając się jak najbardziej w głąb, aż do znalezienia pierwszego rozwiązania rozpatrywanego problemu, więc w tym przypadku oznacza to, że nie zostanie wypróbowana druga z reguł określających `swap`. Dodatkowo symbol odcięcia gwarantuje ustalenie zmiennej `List1` aż do końca rozwiązywania zadanego sortowania. Teraz program musi poradzić sobie z zapytaniem

  ```
  ?- bSort([1,2],Sorted).
  ```

  ($\star$), co oznacza, że znów „uruchomi" funkcję `swap` .

- Dokładniej, po ukonkretnieniu zmiennych `X` i `Z` mamy

  ```
  ?- swap([1,2|[]],[2,1|[]]) :- 1>2.
  ```

  zatem brana jest pod uwagę druga reguła

  ```
  swap([1|[2]],[1|Rest1]) :- swap([2],Rest1).
  ```

  Zgodnie z dyskusją sortowania listy jednoelementowej, udzielona zostanie odpowiedź `No`, zatem zastosowana zostanie druga reguła dotycząca `bSort` dla listy `[1,2]` w punkcie ($\star$). Oznacza to, że otrzymamy `Sorted=[1,2]`, co jest jednocześnie szukanym wynikiem.

9.3 Sortowanie przez wstawianie

Sortowanie przez wstawianie przypomina postępowanie osoby układającej książki na półce. Wybieramy jedną książkę i szukamy (począwszy od miejsca w którym ta książka się znajduje aż do pierwszej książki od lewej) takiego miejsca aby tak książka była wyższa lub równa od innej książki i jednocześnie mniejsza od kolejnej. Jeśli znajdziemy takie miejsce to wstawiamy w nie wybraną książkę i bierzemy kolejną, która za nią występowała. Jeśli takiego miejsca nie znajdziemy to pozostawiamy ją na dotychczasowej pozycji i bieżemy książkę następną.

Niech $\{a_i\}_{i=1,\ldots,n}$ będzie sortowanym ciągiem.

1. Niech $j = 2$.
2. Porównywać element a_j, po kolei, ze wszystkimi elementami a_i, $i = j-1, \ldots, 1$ do momentu, gdy zostanie znaleziony taki element a_i, dla którego zachodzić będzie nierówność $a_i \leq a_j$. Wówczas element a_j należy usunąć z jego dotychczasowej pozycji, wstawić pomiędzy elementy a_i oraz a_{i+1} i przejść do kroku następnego.
3. Zwiększyć j o jeden.
4. Jeśli $j > n$ to przejdź do 6.
5. Powrót do 2.
6. Koniec.

Implementacja algorytmu w Prologu wygląda następująco.

```
iSort([],[]).
iSort([X|Tail],Sorted) :- iSort(Tail,SortedTail), insert(X,SortedTail,Sorted).

insert(X,[Y|Sorted],[Y|Sorted1]) :- X>Y, !, insert(X,Sorted,Sorted1).
insert(X,Sorted,[X|Sorted]).
```

Zacznijmy analizę od sortowania listy jednoelementowej:

```
?- iSort([1],Sorted).
```

- Pierwsza w kodzie programu odpowiednia dla tego przypadku reguła, to

  ```
  iSort([1|[]],Sorted) :- iSort([],SortedTail),insert(1,SortedTail,Sorted).
  ```

- Prolog próbuje zatem odpowiedzieć na pytanie

  ```
  ?- iSort([],SortedTail).
  ```

  co udaje się dzięki pierwszej regule w programie. Otrzymujemy w ten sposób podstawienie `SortedTail=[]`.

- Kolejne zapytanie, jakie napotyka program, to term

  ```
  ?- insert(1,[],Sorted).
  ```

Z uwagi na argumenty, jakie już się znajdują wewnątrz tej funkcji, pierwsza reguła dla `insert` nie jest spełniona, druga zwraca zaś zmienną `Sorted=[1|[]]`. Jej ukonkretnienie powoduje spełnienie pierwszej reguły dla `iSort` i jednocześnie jest ona wypisywanym na ekran wynikiem.

Sortowanie listy dwuelementowej, czyli pytamy

```
?- iSort([1,2],Sorted).
```

- Znów, pierwsza reguła, którą Prolog może dopasować do zapytania, to

  ```
  iSort([1|[2]],Sorted) :- iSort([2],SortedTail),insert(1,SortedTail,Sorted).
  ```

- Pierwsze napotkane w tej regule zapytanie `iSort([2],SortedTail).` już rozpatrywaliśmy. Zmienna `SortedTail` zostaje skonkretyzowana listą `[2]`. Prolog kontynuuje regułę, stąd pojawia się pytanie

  ```
  ?- insert(1,[2],Sorted).
  ```

  które prowadzi do skorzystania z reguły

  ```
  insert(1,[2|[]],[2|Sorted1]) :- 1>2,!,insert(1,[],Sorted1).
  ```

  która oczywiście nie jest w tym przypadku spełniona. Prolog przechodzi zatem dalej: `insert(1,[2],[1|[2]])`. Zmienna `Sorted` z reguły `iSort` zostaje podstawiona przez `[1|[2]]=[1,2]`, co jest ostatecznym wynikiem sortowania.

9.4 Sortowanie przez łączenie

Sortowanie przez scalanie (ang. merge sort), to rekurencyjny algorytm sortowania danych realizujący metodę rozwiązywania problemów typu *dziel i zwyciężaj* (ang. *divide and conquer*). Ideą działania tego typu algorytmów jest podział problemu na mniejsze części, których rozwiązanie jest już łatwiejsze a następnie w oparciu o rozwiązane podproblemy konstruowanie rozwiązania problemu wyjściowego.

W algorytmie wyróżniamy trzy zasadnicze kroki

1. Podział sortowanego ciągu na dwa podciągi o równej długości[3].
2. Zastosowane sortowania przez scalanie dla każdego z podciągów niezależnie (o ile tylko rozważany podciąg ma więcej niż jeden element).
3. Połączenie posortowanych podciągów w jeden ciąg posortowany.

Procedura scalania dwóch podciągów $\{a_i\}_{i=1,\ldots,n}$ i $\{b_j\}_{j=1,\ldots,m}$ w ciąg $\{c_k\}_{k=1,\ldots,n+m}$ jest prosta i szybka, gdyż każdy z podciągów jest już posortowany.

[3]Równość ta nie musi być dosłowna. Jeden z ciągów może być trochę dłuższy. Termin *trochę* jest względny i zależy od długości sortowanego ciągu, ale jego intuicyjne znaczenie jest takie, że długość obu ciągów powinna być, na ile to tylko możliwe, taka sama (zbliżona).

1. Utworzenie wskaźników i i j wskazujących na początki ciągów $\{a_i\}_{i=1,\ldots,n}$ i $\{b_j\}_{j=1,\ldots,m}$: $i = 1$, $j = 1$.

2. Jeżeli $a_i \leq b_j$ to należy element a_i wstawić jako kolejny element ciągu $\{c_k\}_{k=1,\ldots,n+m}$ i zwiększyć i o jeden. W przeciwnym przypadku należy element b_j wstawić jako kolejny element ciągu $\{c_k\}_{k=1,\ldots,n+m}$ i zwiększyć j o jeden.

3. Jeżeli $i = n + 1$ (przepisano ostatni element z ciągu $\{a_i\}_{i=1,\ldots,n}$), wówczas należy do ciągu $\{c_k\}_{k=1,\ldots,n+m}$ przepisać wszystkie nieprzepisane jeszcze elementy z ciągu $\{b_j\}_{j=1,\ldots,m}$ i przejść do 6.

4. Jeżeli $j = m + 1$ (przepisano ostatni element z ciągu $\{b_j\}_{j=1,\ldots,m}$), wówczas należy do ciągu $\{c_k\}_{k=1,\ldots,n+m}$ przepisać wszystkie nieprzepisane jeszcze elementy z ciągu $\{a_i\}_{i=1,\ldots,n}$ i przejść do 6.

5. Wróć do 2.

6. Koniec.

Implementacja algorytmu w Prologu wygląda następująco[4].

```
mSort([],[]).
mSort([X],[X]).
mSort(List,Sorted):- List=[_,_|_],polowa(List,L1,L2),
                     mSort(L1,Sorted1),mSort(L2,Sorted2),
                     myMerge(Sorted1,Sorted2,Sorted).

myMerge([],L,L).
myMerge(L,[],L) :- L\=[].
myMerge([X|T1],[Y|T2],[X|T]) :- X=<Y,myMerge(T1,[Y|T2],T).
myMerge([X|T1],[Y|T2],[Y|T]) :- X>Y,myMerge([X|T1],T2,T).

polowa(L,A,B):-polowa2(L,L,A,B).
polowa2([],R,[],R).
polowa2([_],R,[],R).
polowa2([_,_|T],[X|L],[X|L1],R) :- polowa2(T,L,L1,R).
```

W tekście programu pojawiają się jawnie predykaty określające sortowanie listy pustej i jednoelementowej zacznijmy więc od `?- mSort([1,2],Sorted).`

- Reguła pozwalająca zacząć rozwiązywanie problemu przyjmuje postać

  ```
  mSort([1,2],Sorted):- [1,2]=[_,_|_],polowa([1,2|[]],L1,L2),
                        mSort(L1,Sorted1),mSort(L2,Sorted2),
                        myMerge(Sorted1,Sorted2,Sorted).
  ```

- po przypisaniu do wejściowej listy schematu w pierwszym kroku uruchamiana jest funkcja `?- polowa([1,2|[]],L1,L2).` Reguły dla tej funkcji dają

  ```
  polowa([1,2|[]],A,B):-polowa2([1,2|[]],[1,2|[]],A,B).
  ```

[4] W ćwiczeniu 9.2 zamieszczono inną wersję tego programu.

a w konsekwencji

```
polowa2([1,2|[]],[1|[2]],[1|L1],R) :- polowa2([],[2],L1,R).
```

Teraz Prolog może określić zmienne `L1=[]` i `R=[2]` co daje z kolei skonkretyzować `L1=A=[1]` i `L2=B=R=[2]` z wyjściowego wywołania funkcji `polowa`.

- Kolejny krok to udzielenie odpowiedzi na pytanie `?- mSort([1],Sorted1).` Otrzymujemy je z drugiej reguły określającej ten predykat, tzn. `Sorted1=[1]`. Podobnie, w kolejnym kroku znajdowana jest wartość zmiennej `Sorted2=[2]`.
- Następny ruch Prologu to wywołanie

  ```
  ?- myMerge([1],[2],Sorted).
  ```

  Pierwsza reguła, której schemat danych odpowiada temu zapytaniu, to

  ```
  myMerge([1|[]],[2|[]],[1|T]) :- 1=<2,myMerge([],[2|[]],T).
  ```

  Korzystając następnie z pierwszej reguły dla `myMerge` określana jest zmienna `T=[2|[]]=[2]`, stąd zaś wynika, że zmienna `Sorted` ma wartość `[1|[2]]`. Określone są zatem wszystkie zmienne w całej regule sortowania dla wejściowej listy. Zwrócona zostanie wartość `Sorted=[1,2]`

Sortowanie listy trójelementowej, np. `[3,2,1]` przebiega oczywiście zgodnie ze schematem narzuconym przez trzecią regułę dla funkcji `mSort`. Algorytm zaczyna się jak zwykle...

- Określona zostaje zmienna `List=[3,2|[1]]`.
- Wywoływana jest funkcja `?- polowa([3,2|[1]],L1,L2).` Oczywiście dopiero trzecia reguła dla funkcji `polowa2` jest spełniona przez nasz schemat zmiennej `List`, mamy zatem

  ```
  polowa2([3,2|[1]],[3|[2,1]],[3|L1],R) :- polowa2([1],[2,1],L1,R).
  ```

- w wyniku zadziałania drugiej reguły dla `polowa2` określone zostają zmienne `L1=[]` oraz `R=[2,1]` , co daje wartości `L1=[3]` oraz `L2=[2,1]` (dla funkcji `polowa`).
- uruchamiana jest funkcja `mSort` kolejno dla list `L1=[3]` oraz `L2=[2,1]`. W konsekwencji zwracane są wartości zmienych `Sorted1=[3]` oraz `Sorted2=[1,2]`.
- Kolejny krok to znalezienie wartości zmiennej `Sorted` dzięki regule

  ```
  myMerge([3|[]],[1|[2]],[1|T]) :- 3>1,myMerge([3|[]],[2],T).
  ```

  Działanie funkcji `myMerge` dla list jednoelementowych jest już znane. Otrzymamy `T=[2,3]`. Stąd, `Sorted=[1|T]=[1,2,3]`., co jest wynikem wypisanym przez Prolog na ekranie.

Warto zauważyć, że funkcja `myMerge(Lista1,Lista2,Wynik)` wstawia po kolei elementy z `Lista1` do listy `Lista2` tworząc listę posortowaną dzięki temu że obie z list podanych jako jej argumenty już są posortowane.

9.5 Pytania i odpowiedzi

Pytanie 9.1. Przeprowadź analizę działania algorytmu sortowania szybkiego.
Implementacja algorytmu sortowania szybkiego w Prologu

```
qSort([],[]).

qSort([X|Tail], Sorted) :-
     split(X, Tail, Small, Big),!,
     qSort(Small, SortedSmall),
     qSort(Big, SortedBig),
     lacz(SortedSmall, [X|SortedBig], Sorted).

split(H,[],[],[]).
split(H,[X|T],[X|L],G) :- X=<H,split(H,T,L,G).
split(H,[X|T],L,[X|G]) :- X>H,split(H,T,L,G).

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

Pytanie 9.2. Przeprowadź analizę działania predykatów `polowa` oraz `polacz` z programu implementującego sortowanie przez łączenie.
Implementacja algorytmu sortowania przez łączenie

```
mSort([X],[X]).
mSort(List,Sorted):- List=[_,_|_],polowa(List,L1,L2),
                     mSort(L1,Sorted1),mSort(L2,Sorted2),
                     polacz(Sorted1,Sorted2,Sorted).

polowa([],[],[]).
polowa([X],[X],[]).
polowa([X,Y|T],[X|L1],[Y|R1]) :- polowa(T,L1,R1).

polacz([],L,L).
polacz(L,[],L).
polacz([LH|LT],[RH|RT],[LH|T]) :- LH < RH, polacz(LT,[RH|RT],T).
polacz([LH|LT],[RH|RT],[RH|T]) :- LH > RH, polacz(RT,[LH|LT],T).
```

Rozdział 10

Od problemu do jego (efektywnego) rozwiązania

10.1 Rebus i jego pierwsze rozwiązanie

W rozdziale tym spróbujemy pokazać jak rozwiązywać konkretne problemy przy pomocy Prologa. Cała sztuka polega bowiem na tym, aby z jednej strony wykorzystać potencjał drzemiący w języku a z drugiej umieć „dopomóc jemu" ten potencjał ujawnić.

Rozwiążemy następującą zagadkę logiczną.

Należy znaleźć takie przyporządkowanie cyfr do liter aby rebus

```
   DONALD
 + GERALD
=========
 = ROBERT
```

był prawdziwym działaniem matematycznym. Przyporządkowanie jest różnowartościowe, tzn. każdej literze odpowiada inna cyfra.

Najbardziej uniwersalne rozwiązanie polega na generowaniu wszystkich możliwych przyporządkowań cyfr za litery i sprawdzaniu czy otrzymujemy poprawne rozwiązanie. Generowanie permutacji zostało opisane w rozdziale poświęconym sortowaniu przy okazji omawiania sortowania naiwnego. Dlatego tutaj jedynie przypomnimy niezbędny fragment kodu

```
permutacja([],[]).
permutacja(Xs,[Z|Zs]) :- usun(Z,Xs,Ys), permutacja(Ys,Zs).

usun(X,[X|Xs],Xs).
usun(X,[Y|Ys],[Y|Zs]) :- usun(X,Ys,Zs).
```

Teraz możemy napisać właściwy program

```
imiona:- permutacja([0,1,2,3,4,5,6,7,8,9],X),
         [A,B,D,E,G,L,N,O,R,T]=X,
         X1 is D*100000+O*10000+N*1000+A*100+L*10+D,
```

```
        X2 is G*100000+E*10000+R*1000+A*100+L*10+D,
        X3 is R*100000+O*10000+B*1000+E*100+R*10+T,
        suma(X1,X2,S),wypisz(X3,S,X).
```

i uzupełnić go o brakujące predykaty pomocnicze

```
suma(X1,X2,X3):- X3 is X1+X2.
wypisz(X1,X2,X):- X1=X2,write(X).
```

To już wszystko. Program składający się z tych trzech części, na komputerze autora, znajduje rozwiązanie rebusu w 22.32s

```
A, B, D, E, G, L, N, O, R, T
4, 3, 5, 9, 1, 8, 6, 2, 7, 0

   526485
 + 197485
=========
 = 723970
```

Jest to rozwiązanie najbardziej uniwersalne, gdyż pozwala rozwiązać każdy tego typu rebus. Niestety ceną jest czas, który bez dodatkowych zabiegów pozostaje bardzo długi. Dlatego w kolejnych podrozdziałach będziemy próbowali, w oparciu o wiedzę jaką dodatkowo potrafimy uzyskać przyglądając się bliżej problemowi, przyspieszyć działanie programu. Jak wiadomo, czas obliczeń jest ściśle zależny od posiadanego systemu. Dlatego będziemy posługiwać się pewnymi abstrakcyjnymi jednostkami pozwalającymi oddać zależności procentowe co pozwoli ocenić zysk stosowanych rozwiązań niezależny od sprzętu. Dlatego przyjmujemy, że czas wykonania pierwszej wersji programu wynosi 100%.

10.2 Rozwiązanie drugie

23 sekundy to dużo czasu. Przyjrzyjmy się więc uważniej rebusowi...

Zauważmy, że w ostatniej kolumnie mamy następujące działanie `D + D = T`. Zatem mamy zależność jaką muszą spełniać litery `D` oraz `T`

$$\begin{array}{ll} T = 2D & \text{jeżeli} \quad D < 5, \\ T = 2D - 10 & \text{jeżeli} \quad D > 4. \end{array} \tag{10.1}$$

Uwzględnienie tak prostego warunku, poprzez wprowadzenie niewielkiej modyfikacji do programu pozwala na zmniejszenie czasu poszukiwania rozwiązania do wartości 50.98% (11.38s).

```
warunek(D,T):-D<5,T is 2*D.
warunek(D,T):-D>4,T is 2*D-10.

imiona:- permutacja([0,1,2,3,4,5,6,7,8,9],X),[A,B,D,E,G,L,N,O,R,T]=X,
        warunek(D,T),
        X1 is D*100000+O*10000+N*1000+A*100+L*10+D,
        X2 is G*100000+E*10000+R*1000+A*100+L*10+D,
        X3 is R*100000+O*10000+B*1000+E*100+R*10+T,
        suma(X1,X2,S),wypisz(X3,S,X).
```

Co jest interesujące, zyskaliśmy tylko i wyłącznie na obliczeniach arytmetycznych, gdyż to właśnie one nie są wykonywane gdy predykat `warunek1` nie jest spełniony. Dalej natomiast generowane są wszystkie możliwe permutacje zbioru cyfr.

10.3 Rozwiązanie trzecie

Warunek (10.1) wyrażony jest bardzo ogólnie, ale my doskonale wiemy, że ilość par `(D,T)`, która go spełnia jest skończona i wcale nie taka duża. Wykorzystajmy więc i ten fakt i zmodyfikujmy program do następującej postaci

```
warunek(1,2).
warunek(2,4).
warunek(3,6).
warunek(4,8).
warunek(5,0).
warunek(6,2).
warunek(7,4).
warunek(8,6).
warunek(9,8).

imiona:- permutacja([0,1,2,3,4,5,6,7,8,9],X),[A,B,D,E,G,L,N,O,R,T]=X,
         warunek(D,T),
         X1 is D*100000+O*10000+N*1000+A*100+L*10+D,
         X2 is G*100000+E*10000+R*1000+A*100+L*10+D,
         X3 is R*100000+O*10000+B*1000+E*100+R*10+T,
         suma(X1,X2,S),wypisz(X3,S,X).
```

Niby jest to niewielka zmiana, ale wyposażając Prolog w wiedzę w postaci gotowych faktów redukujemy ilość wykonywanych operacji. Skutkuje to zmniejszeniem czasu wykonania programu do 40.00% (9.15s).

10.4 Rozwiązanie czwarte

Przyglądając się zadaniu raz jeszcze zauważamy, że przedostatnia kolumna to podobny związek jak w kolumnie ostatniej, który dotyczy liter `L` oraz `R`. Tym razem jednak musimy pamiętać o uwzględnieniu ewentualnego przeniesienia z kolumny ostatniej. Tak więc rozpatrywać będziemy dwa przypadki.

- Jeśli z kolumny ostatniej nie będzie przeniesienia (co nastąpi wtedy gdy $D < 5$) wówczas kolumna ostatnia nie wpływa w żaden sposób na przedostatnią i para (L, R) spełnia warunki (10.1) wyrażone z pomocą faktów `warunek(1,2)`, ..., `warunek(9,8)`. W Prologu zapiszemy to jako (zamiast `warunek` piszemy `w1`, gdyż zamiast jednego warunku jak poprzednio będziemy mieć dwa warunki)

  ```
  w2(L,R,D):-D<5,w1(L,R).
  ```

- Jeśli nastąpi przeniesienie z ostatniej kolumny (co nastąpi wtedy gdy $D > 4$) wówczas do kolumny przedostatniej dodawana jest wartość jeden. Oznacza to, że

dla $D > 4$ odpowiedni warunek przyjmuje postać

$$\underbrace{L + L + 1}_{\text{suma powiększona o 1}} \quad = R$$

co w Prologu zapiszemy

```
w2(L,R,D):-D>4,RR is R-1,w1(L,RR).
```

W ten oto sposób znowu odrzucamy część obliczeń, które i tak nie mogły by okazać się prawidłowym rozwiązaniem (ze względu na niespełnianie powyższych warunków) i zmniejszamy czas do 35.66% (7.96s).

Zanim przejdziemy do kolejnej modyfikacji zauważmy jeszcze tylko, że warunek dla $D > 4$ możemy także zapisać w następujący sposób

```
w2(L,R,D):-D>4,Tmp is 2*L-10+1,R=Tmp.
```

co jednak nie wpływa na czas wykonania.

10.5 Rozwiązanie piąte

Ostatnie zaprezentowane rozwiązanie właściwie wyczerpało możliwości skrócenia wykonania programu przez eliminację obliczeń. W tym momencie zasadniczym źródłem spowolnienia jest generowanie wszystkich permutacji. Co więcej, generowane są permutacje, które i tak nie mogą być rozwiązaniem. Wiemy, że para (D, T) spełniać musi odpowiednie warunki wyrażone za pomocą faktów `warunek(1,2)`,...,`warunek(9,8)`. Tak więc np. ma sens generowanie permutacji zbioru $\{0, 3, 4, 5, 6, 7, 8, 9\}$ gdy przyjmiemy, że $D = 1$, $T = 2$, ale zupełnie nie ma to sensu, gdy $D = 2$, $T = 1$.

Napiszmy zatem predykat `p(Lista)` generujący takie permutacje, że przyporządkowanie dla D i T ma sens. Najprościej jest to zrealizować w oparciu o fakty `warunek(1,2)`, ...,`warunek(9,8)`. Rozpoczniemy więc od

```
p([D,T|List]) :- warunek(D,T)
```

gdzie `List` to permutacja zbioru $\{0, \ldots, 9\}$ pomniejszonego o elementy przypisane do D oraz T. Permutacja zbioru to w Prologu permutacja listy, a jej pomniejszenie to usunięcie elementu. Zatem otrzymujemy kolejne fragmenty predykatu. Najpierw usunięcie elementów

```
LL=[0,1,2,3,4,5,6,7,8,9],usun(D,LL,L1),usun(T,L1,L2)
```

a następnie permutacja tak zmienionej listy

```
permutacja(L2,List)
```

Wszystko razem daje poszukiwany predykat

```
p([D,T|List]) :- warunek(D,T),
                 LL=[0,1,2,3,4,5,6,7,8,9],
                 usun(D,LL,L1),
                 usun(T,L1,L2),
                 permutacja(L2,List).
```

Na zakończenie musimy umieścić wywołanie celu `p(X)` w ciele celu `imiona` i zmienić kolejność elementów na liście, tak aby `D` oraz `T` były pierwszymi elementami

```
imiona:- p(X),[D,T,A,B,E,G,L,N,O,R]=X,
         X1 is D*100000+O*10000+N*1000+A*100+L*10+D,
         X2 is G*100000+E*10000+R*1000+A*100+L*10+D,
         X3 is R*100000+O*10000+B*1000+E*100+R*10+T,
         suma(X1,X2,S),wypisz(X3,S,X).
```

Decyzja o takim kierunku modyfikacji okazuje się słuszna co potwierdza osiągnięty rezultat (2.5s) równy 11.2% początkowego czasu (dla pierwszego programu).

Cały program w tej wersji wygląda następująco

```
usun(X,[X|Xs],Xs).
usun(X,[Y|Ys],[Y|Zs]) :- usun(X,Ys,Zs).
permutacja([],[]).
permutacja(Xs,[Z|Zs]) :- usun(Z,Xs,Ys), permutacja(Ys,Zs).

suma(X1,X2,X3):- X3 is X1+X2.
wypisz(X1,X2,X):- X1=X2,write(X).

warunek(1,2).
warunek(2,4).
warunek(3,6).
warunek(4,8).
warunek(5,0).
warunek(6,2).
warunek(7,4).
warunek(8,6).
warunek(9,8).

p([D,T|List]) :- warunek(D,T),
                 LL=[0,1,2,3,4,5,6,7,8,9],
                 usun(D,LL,L1),
                 usun(T,L1,L2),
                 permutacja(L2,List).

imiona:- p(X),[D,T,A,B,E,G,L,N,O,R]=X,
X1 is D*100000+O*10000+N*1000+A*100+L*10+D,
X2 is G*100000+E*10000+R*1000+A*100+L*10+D,
X3 is R*100000+O*10000+B*1000+E*100+R*10+T,
suma(X1,X2,S),wypisz(X3,S,X).
```

10.6 Rozwiązanie szóste

Powodzenie modyfikacji wprowadzonej w poprzedniej wersji programu pozwala mieć nadzieję, że analogiczna poprawka dotycząca liter `L` oraz `R` także przyniesie nam oszczędności czasowe. Wymaga to zmodyfikowania predykatu `p(Lista)` do postaci

```
p([D,T,L,R|List]) :- warunek(D,T),
                     n(D,T,L,R),
                     LL=[0,1,2,3,4,5,6,7,8,9],
                     usun(D,LL,L1),
                     usun(T,L1,L2),
                     usun(L,L2,L3),
                     usun(R,L3,L4),
                     permutacja(L4,List).
```

oraz programu głównego

```
imiona:- p(X),[D,T,L,R,A,B,E,G,N,O]=X,
         X1 is D*100000+O*10000+N*1000+A*100+L*10+D,
         X2 is G*100000+E*10000+R*1000+A*100+L*10+D,
         X3 is R*100000+O*10000+B*1000+E*100+R*10+T,
         suma(X1,X2,S),wypisz(X3,S,X).
```

Pozostaje teraz napisać predykat `n(D,T,L,R)`, który dla zadanej pary (D, T) zwróci odpowiednią parę (L, R). Pamiętać należy, że para (L, R)

- nie może zawierać cyfr przyporządkowanych dla D i T;
- musi spełniać warunki z rozwiązania czwartego dotyczące zależności pomiędzy D, L oraz R.

Zdefiniujmy ten predykat jako

```
n(D,T,L,RR) :- D>4, l(D,T,L), R is L*2+1,r(R,RR).
n(D,T,L,RR) :- D<5, l(D,T,L), R is L*2,r(R,RR).
```

gdzie `l(D,T,L)` to peredykat, który przy ustalonych wartościach dla D i T daje wszystkie mozliwosci dla L, różne od D i T, natomiast `r(R,RR)` to predykat obliczający wartość dla R zależnie od wartości L.

Predykat `r(R,RR)` jest bardzo prosty do napisania

```
r(R,RR):-R>9,RR is R-10.
r(R,R):-R<10.
```

Predykat `l(D,T,L)` wcale nie jest dużo trudniejszy

```
l(D,T,L) :- LL=[1,2,3,4,5,6,7,8,9,0],usun(D,LL,L1),usun(T,L1,L2),jest(L,L2).

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

Zauważmy, że predykat `jest/2` to nic innego jak predykat `member/2` sprawdzajacy czy zadany element należy do listy. Jest to kolejny przykład kiedy użycie w Prologu predykatu w sposób odmienny od jego pierwotnego przeznaczenia pozwala otrzymać pożądane zachowanie (w tym przypadku chodzi nam o możliwość wymienienia wszystkich elementów listy).

Tym oto sposobem udaje się osiagnąć wynik równy 1.12% (0.25s).

10.7 Rozwiązanie siódme

Ostatni otrzymany wynik jest już akceptowalny, ale zauważmy, że możemy go poprawić. Oto bowiem w trzeciej kolumnie od końca mamy zależność analogiczną do zależności w kolumnie ostatniej i przedostatniej. Wykorzystując zatem zdobyte do tej pory doświadczenie, możemy zmodyfikować program tak aby zostały uwzględnione odpowiednie warunki dla liter A oraz E. Finalna wersja programu przedstawia się następująco

```
usun(X,[X|Xs],Xs).
usun(X,[Y|Ys],[Y|Zs]) :- usun(X,Ys,Zs).
permutacja([],[]).
permutacja(Xs,[Z|Zs]) :- usun(Z,Xs,Ys), permutacja(Ys,Zs).

suma(X1,X2,X3):- X3 is X1+X2.
wypisz(X1,X2,X):- X1=X2,write(X).

warunek(1,2).
warunek(2,4).
warunek(3,6).
warunek(4,8).
warunek(5,0).
warunek(6,2).
warunek(7,4).
warunek(8,6).
warunek(9,8).

n(D,T,L,RR) :- D>4, l(D,T,L), R is L*2+1,r(R,RR).
n(D,T,L,RR) :- D<5, l(D,T,L), R is L*2,r(R,RR).

r(R,RR):-R>9,RR is R-10.
r(R,R):-R<10.

l(D,T,L) :- LL=[1,2,3,4,5,6,7,8,9,0],usun(D,LL,L1),usun(T,L1,L2),jest(L,L2).

jest(X,[X|_]).
jest(X,[Y|Ys]) :- jest(X,Ys).

p([D,T,L,R,A,E|List]) :- warunek(D,T),
                         n(D,T,L,R),
                         n(L,R,A,E),
                         LL=[0,1,2,3,4,5,6,7,8,9],
                         usun(D,LL,L1),
                         usun(T,L1,L2),
                         usun(L,L2,L3),
                         usun(R,L3,L4),
                         usun(A,L4,L5),
                         usun(E,L5,L6),
                         permutacja(L6,List).
```

```
imiona:- p(X),
         [D,T,L,R,A,E,B,G,N,O]=X,
         X1 is D*100000+O*10000+N*1000+A*100+L*10+D,
         X2 is G*100000+E*10000+R*1000+A*100+L*10+D,
         X3 is R*100000+O*10000+B*1000+E*100+R*10+T,
         suma(X1,X2,S),wypisz(X3,S,X).
```

W tej wersji czas działania wynosi zaledwie 0.13% (0.03s) pierwotnego czasu działania dla pierwszej wersji programu.

Rozdział 11

Efektywny Prolog

W niniejszym rozdziale spróbujemy zebrać w jedną całość wszystkie „porady” jakie do tej pory się pojawiły a jakie związane były ze sposobem pisania programów w Prologu. Jak już wielokrotnie mieliśmy okazję się przekonać, wszybkość działania programu zależy od tego w jaki sposób go napiszemy a że napisać można go na mnóstwo różnych sposobów więc i dylematów jest wiele.

11.1 Czy tylko deklaratywność?

Deklaratywność rozumiemy jako

W tym kontekście Prolog uważany jest za język deklaratywny lub mówiąc inaczej, język który nie jest proceduralny. Teoretycznie jest to prawdą, ale w praktyce na kod programu (tj. reguły) trzeba patrzeć zarówno jak na wyrażenia deklaratywne definiujące pewne informacje jak i sposoby (procedury) użycia tych informacji. Stąd na regułę postaci

```
a(X,b) :- a(X,c)
```

możemy spojrzeć dwojako.

- Jak na definicję: `X` *jest w relacji* `a` *z* `b` *jeśli* `X` *jest w relacji* `a` *z* `c`.
- Jak na procedurę obliczeniową (algorytm): *aby wykazać, że* `X` *jest w relacji* `a` *z* `b` *należy wykazać, że* `X` *jest w relacji* `a` *z* `c`.

Dla jasności powiedzmy, że taka „dwoistość” występuje też w przypadku proceduralnych języków programowania. Na instrukcję języka C

```
x = y + z;
```

także możemy spojrzeć dwojako.

- Jak na definicję (wzór matematyczny): `x` *jest równe sumie elemntu* `y` *i* `z`.
- Jak na procedurę obliczeniową (algorytm):
  1. Pobierz wartość `y`.
  2. Pobierz wartość `z`.
  3. Dodaj do siebie dwie ostatnio pobrane wartości.

4. Wynik ostatniej operacji arytmetycznej zapisz w zmiennej `x`.

To, który ze sposobów interpretacji wybierzemy, często zależy od kontekstu. Doskonale wiadomo, że deklaratywne spojrzenie na instrukcje języka C podobną do poprzedniej, tj.

```
x = x + 1;
```

prowadzi do sprzeczności a jednak z doświadczenia wiemy, że ta instrukcja jest jak najbardziej poprawna. Na marginesie zaznaczmy tylko, że bardzo często właśnie początkujący programiści mają duży kłopot właśnie z takimi instrukcjami. Jako, że początkujący, więc nieprzyzwyczajeni do myślenia proceduralnego, patrzą na kod jak na definicje co wprowadza w ich głowach zament, bo *jak niby* `x` *może być równy* `x` *plus 1 i co to właściwie znaczy?!*

A teraz wróćmy do przykładu, który pojawił się już w rozdziale 3, strona 24. Występuje tam reguła postaci

```
mniej(X,Y) :- mniej(X,Z),mniej(Z,Y).
```

która z deklaratywnego punktu widzenia jest częścią poprawnej logicznie definicji pojęcia `mniej`. Oto dwa obiekty `X` i `Y` są w relacji `mniej` (mniejszości) gdy istnieje taki trzeci obiekt `Z` dla którego `X` jest mniejszy od `Z` i jednocześnie `Z` jest mniejszy od `Y`. Teoretycznie definicji tej nic nie można zarzucić a jednak wiemy, że nie jest ona poprawna i prowadzi do „zapętlenia" programu. Przyczyna „zapętlenia" staje się jasna, gdy spojrzymy na zapis z proceduralnego (obliczeniowego) punktu widzenia. Oto gdy zarówno `Z` jak i `Y` nie są znane cel `mniej(X,Z)` niczym nie różni się od `mniej(X,Y)`. Można powiedzieć, że mamy wówczas regułę postaci

```
mniej(X,Y) :- mniej(X,Y).
```

To jednak dostrzec możemy wtedy, gdy spojrzymy na całą regułę nie jak na definicję, ale jak na algorytm. Rozwiązaniem w tym przypadku jest niedopuszczenie do „samowywołania" co osiągneliśmy przez wprowadzenie następującej zmiany

```
jestMniejszy(X,Y) :- mniej(X,Z),jestMniejszy(Z,Y).
```

Wniosek jest więc taki: w Prologu musimy myśleć zarówno deklaratywnie jak i proceduralnie.

11.2 Zmniejszać przestrzeń rozważań

Jedna z najważniejszych rad prologowych jest następująca

Jeśli jakiś test może nie być prawdziwy to niech to się stanie tak szybko jak tylko jest to możliwe.

Z problemem tym spotkaliśmy się już w podrozdziale 8.1 a także przy okazji optymalizacji rozwiązania problemu z rozdziału 10.

W pierwszym przypadku mieliśmy do czynienia z sytuacją, w której istotną relę odgrywała kolejność wywołania podceli. Zredukowanie przestrzeni rozważań osiąga się tutaj dzięki spostrzeżeniu, że jeśli spełnienie jednego warunku jest mniej prawdopodobne (zdarza się rzadziej) niż innego, to warunek ten korzystnie jest umieścić jako pierwszy, według schematu

```
test(X) :- warunekRzadki(X),warunekCzesty(X).
```

a nie według schematu

```
test(X) :- warunekCzesty(X),warunekRzadki(X).
```

W drugim przypadku wymagane było przekonstruowanie programu a czasem jest to wręcz zmiana logiki jego działania. Dla zilustrowania tego problemu załóżmy, że mamy zestaw predykatów pozwalających manipulować zbiorami. Zbiory w tym przypadku reprezentowane oczywiście będą za pomocą list. Załóżmy dalej, że należy dopisać predykat sprawdzający czy dwa zbiory są identyczne. Bardzo naiwne rozwiązanie tego problemu jest następujące

```
takieSame(Set1,Set2) :- permutacja(Set1,Perm),Perm=Set2.
```

Taki predykat będzie działał tyle tylko, że czas jego działania będzie niakceptowalnie długi. Dużo rozsądniej jest napisać

```
takieSame(Set1,Set2) :- sortuj(Set1,Sort1),sortuj(Set2,Sort2),Sort1=Sort2.
```

Podejście to jednak wymaga zmiany koncepcji rozwiązania problemu co nie zawsze jest takie proste jak w tym przypadku. Często się jednak opłaca.

11.3 Niech pracują za nas inni

Przyzwyczajenia jakie zostały u nas wyrobione przez lata stykania się z pewnymi (utartymi) metodami postępowania bywają tak silne, że często mając do dyspozycji nawet znacznie silniejsze narzędzia, korzystamy z tych które dobrze znamy (a właściwie to zapamiętaliśmy) a nie tych, które są efektywne. Większość z nas wie co należy zrobić aby w danym języku programowania, np. C, pomnożyć skalarnie przez siebie dwa wektory. Piszemy pętle, mnożymy element po elemencie i wszystko sumujemy. Może to wyglądać np. tak

```
iloczyn=0;
for(i=0;i<10;i++)
{
  iloczyn+=wektor1[i]*wektor2[i];
}
```

W środowisku obliczeniowym Scilab (lub jego komercyjnym odpowiedniku: Matlab) wektory i macierze funkcjonują jako typ wbudowany. Wystarczy w nich więc napisać

```
iloczyn=wektor1*wektor2;
```

Problem polega na tym, że wiele osób na początku, niejako „odruchowo”, wybiera pierwszy sposób mimo, że środowisko dostarcza znacznie efektywniejszego narzędzia.

Nie inaczej sytuacja wygląda w Prologu, w którym narzędziem takim jest unifikacja. Unifikacja a więc dopasowywanie jest mechanizmem, który czy tego chcemy czy nie i tak uruchamiany jest przy każdej, nazwijmy to, próbie wywołania predykatu. Skoro i tak musi ten mechanizm zadziałać, to można pozwolić jemu, aby część pracy wykonał za nas. Spotkaliśmy się już z tym w rozdziale 10. W podrozdziale 10.2 udało nam się ustalić warunki jakie muszą spełniać pewne dane `D` i `T`, przy założeniu, że są różne i przyjmują wartości co najwyżej ze bioru wszystkich cyfr dziesiętnych

```
warunek(D,T):-D<5,T is 2*D.
warunek(D,T):-D>4,T is 2*D-10.
```

W dalszej części korzystaliśmy z tego predykatu. Potem (podrozdział 10.3) udaje nam się jednak dostrzec, że przy warunkach zadania zbiór par (D, T) jest mały i równie dobrze można zastąpić go zbiorem faktów

```
warunek(1,2).
warunek(2,4).
warunek(3,6).
warunek(4,8).
warunek(5,0).
warunek(6,2).
warunek(7,4).
warunek(8,6).
warunek(9,8).
```

i pozwolić wkroczyć do akcji unifikacji.

Problem ten ilustruje także bardzo dobrze następujące zadanie: napisać predykat sprawczający czy na liście są dokładnie dwa elementy. Najbardziej uniwersalny sposób jest następujący

```
dokladnieDwa(Lista) :- dlugosc(Lista,Dlugosc),Dlugosc=2.
```

Jest to sposób najbardziej uniwersalny (najbardziej ogólny) przez co najwolniejszy. Wykorzystując specyfikę problemu, możemy napisać znacznie szybszą wersję w postaci

```
dokladnieDwa(Lista) :- dlugosc(Lista,3).
```

lub wręcz taką

```
dokladnieDwa([_,_]).
```

Nie ma też nic prostszego jak napisanie programu zamieniającego elementy listy dwuelementowej o ile tylko porzucimy utarte schematy myślenia. „Normalnie" zamianę elemntów realizuje się przy pomocy trzeciej zmiennej pomocniczej, ale w Prologu można to przerzucić na mechanizm unifikacji pisząc

```
zamien([A,B],[B,A]).
```

Na koniec zauważmy, że bazę faktów

```
f(z,a).
f(z,b).
f(z,c).
f(z,d).
f(z,e).
```

możemy też przedstawić jako

```
f(a,z).
f(b,z).
f(c,z).
f(d,z).
f(e,z).
```

Druga wersja jest szybsza, gdyż umożliwia odrzucenie niepasujących faktów już na etapie sprawdzania pierwszego argumentu.

11.4 Pracuj na głowie...

Najlepiej działać na głowie a precyzyjniej rzecz ujmując na kilku pierwszych elementach listy. Chodzi o to aby nie trzeba było często przebiegać przez całą listę celem wykonania pewnej pracy na jej ostatnich lub wręcz ostatnim elemencie. W zadaniu 13.4.4 z ćwiczenia 13.4 rozpatrujemy problem odwracania listy. Rozwiązanie jakie podajemy wygląda następująco

```
odwroc([],[]).
odwroc([X|Xs],Zs) :- odwroc(Xs,Ys), dodajNaKoncu(Ys,[X],Zs).
```

Nie jest to rozwiązanie najlepsze, ale ze względu na to że pewne pojęcia zostaną wprowadzone dopiero w dalszej częsci, na tamtym etapie wystarczające (akceptowalne). Zauważmy jednak, że predykat `dodajNaKoncu` musi przejść po wszystkich elementach listy zanim znajdzie się na samym jej końcu i dopiero wtedy dodaje zadany element. Można to rozwiązać znacznie szybciej przy wykorzystaniu zmiennej akumulatorowej

```
odwroc(L1,L2) :- odwroc(L1,[],L2).

odwroc([H|T],Tmp,Res) :- odwroc(T,[H|Tmp],Res).
odwroc([],Res,Res).
```

Rozwiązanie to jest znacznie szybsze, gdyż operujemy tylko na pierwszych elementach listy, zarówno „zdejmując" elementy jak i „wkładając" je do niej.

11.5 ...albo odcinaj to co zbędne

```
sgn(X) :- X=0, write('zero').
sgn(X) :- X>0, write('dodatnie').
sgn(X) :- X<0, write('ujemne').

sgn(0) :- write(zero).
sgn(X) :- X>0, write(dodatnie).
sgn(X) :- X<0, write(ujemne).

sgn(0) :- write(zero),!.
sgn(X) :- X>0,write(dodatnie),!.
sgn(X) :- X<0, write(ujemne).

sgn(0) :- write(zero),!.
sgn(X) :- X>0,write(dodatnie),!.
sgn(_) :- write(ujemne).
```

Rozdział 12

Rachunek zdań

Rachunek zdań (ang. *propositional calculus*, *propositional logic*, *sentential calculus*) to dział logiki matematycznej badający, jak prawdziwość zdań złożonych tworzonych przy pomocy różnych spójników zależy od prawdziwości wchodzących w ich skład **zdań prostych**. Zdania proste są najmniejszą, niepodzielną „jednostką" składową dowolnego zdania (złożonego) i dlatego nazywa się je także **formułami atomowymi** (ang. *atomic sentences*). W klasycznym rachunku zdań przyjmuje się założenie, że każdemu zdaniu można przypisać jedną z dwu wartości logicznych – **prawdę** albo **fałsz**, które umownie przyjęto oznaczać 1 i 0. Przedmiotem badań rachunku zdań są formalne reguły pozwalające określić jak prawdziwość zdań atomowych wpływa na prawdziwość zdania złożonego. Na przykład o zdaniach

$p =$ *Łódź jest stolicą Polski.*
$q =$ *Warszawa jest stolicą Polski.*

zgodnie z posiadaną przez nas wiedzą powiemy odpowiednio *fałsz* i *prawda*. Co ciekawe, w rachunku zdań treść rozpatrywanych zdań nie ma znaczenia. Istotna jest jedynie ich **wartość logiczna**. Wartość logiczną zdań złożonych powstałych przez zastosowanie **spójników zdaniowych** wyznaczona jest jednoznacznie przez własności tychże spójników i zależy wyłącznie od prawdziwości lub fałszywości zdań składowych, nie zależy natomiast od ich treści. Tak więc jeśli z dwóch zdań atomowych

$p' =$ *Łódź jest stolicą Polski.* (*fałsz*)
$q' =$ *Agent 007 to James Bond.* (*prawda*)

utworzymy zdanie złożone za pomocą spójnika „*i*" to choć będzie ono bez sensu z punktu widzenia niesionej treści, to jego wartość logiczna będzie taka sama jak wartość logiczna zdania utworzonego ze zdań p i q.

Zdefiniujemy teraz formalnie syntaktykę rachunku zdań.

Definicja 12.1 (Syntaktyka rachunku zdań)**. Zdanie atomowe (formuła atomowa)** *jest najmniejszą, niepodzielną jednostkę zdaniową, której można przypisać jedną z dwóch wartości logicznych* – **prawdę** *lub* **fałsz**. **Zdanie (złożone)**, *nazywane dalej ogólnie* **formułą**, *definiujemy rekurencyjnie w następujący sposób*

1. *Każda formuła atomowa jest formułą.*
2. *Dla każdej formuły f, $\neg f$ też jest formułą. Formułę $\neg f$ nazywamy* **negacją** *(ang.* negation*) (formuły) f.*

3. *Dla wszystkich formuł f i g, $f \vee g$ też jest formułą. Formułę $f \vee g$ nazywamy* **dysjunkcją** *(ang.* disjunction*) (formuł) f i g.*

4. *Dla wszystkich formuł f i g, $f \wedge g$ też jest formułą. Formułę $f \wedge g$ nazywamy* **koniunkcją** *(ang.* conjunction*) (formuł) f i g.*

5. *Każda formuła f będąca częścią innej formuły g nazywa się jej* **podformułą**.

Zwróćmy uwagę na fakt, że według powyższej definicji formuła to nic innego jak pewien *napis* zbudowany według określonych reguł. Wszak o tym mówi właśnie syntaktyka. Abyśmy mogli czytać np. symbol $\neg$ jako *nie* musimy określić teraz znaczenie wprowadzonych symboli a więc ich **semantykę**.

Definicja 12.2 (Semantyka rachunku zdań)**.** *Niech L będzie zbiorem wartości $\{0, 1\}$ nazywanym zbiorem wartości logicznych. Element 0 nazywamy* **fałszem**, *element 1 nazywamy* **prawdą**. *Dalej, niech A będzie zbiorem formuł atomowych a $\mathcal{A} : A \to L$ funkcją przyporządkowującą każdej formule atomowej element* prawda *albo* fałsz. *Rozszerzmy teraz funkcję $\mathcal{A}$ do funkcji $\mathcal{F} : F \to L$, gdzie $F \supseteq A$ jest zbiorem formuł jakie można zbudować z formuł atomowych należących do zbioru A. Funkcję $\mathcal{F}$ definiujemy następująco*

1. *Dla każdej formuły atomowej $a \in A$, $\mathcal{F}(a) = \mathcal{A}(a)$.*

2. *Dla każdych dwóch formuł f, g*

$$\mathcal{F}((f \wedge g)) = \begin{cases} 1, & \text{jeśli} \quad \mathcal{F}(f) = 1 \quad \text{i} \quad \mathcal{F}(g) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

3. *Dla każdych dwóch formuł f, g*

$$\mathcal{F}((f \vee g)) = \begin{cases} 1, & \text{jeśli} \quad \mathcal{F}(f) = 1 \quad \text{lub} \quad \mathcal{F}(g) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

4. *Dla każdej formuły f*

$$\mathcal{F}((\neg f)) = \begin{cases} 1, & \text{jeśli} \quad \mathcal{F}(f) = 0 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

W dalszej części o $\mathcal{A}$ lub $\mathcal{F}$ mówić będziemy po prostu **przyporządkowanie** *(ang.* assignment*).*

W oparciu o powyższą definicję możemy teraz nadać znaczenie poszczególnym symbolom czytając je jako *i* ($\wedge$), *lub* ($\vee$), *nie* ($\neg$). Ponad to możemy każdy z nich opisać za pomocą tzw. **tablicy prawdy**

f	g	$f \wedge g$	$f \vee g$	$\neg f$
0	0	0	0	1
0	1	0	1	
1	0	0	1	0
1	1	1	1	

Zauważmy, że zgodnie z powyższą definicją możemy powiedzieć, że symbole $\wedge$, $\vee$ oraz $\neg$ są symbolami operatorów. Skoro zaś mówimy o operatorach to zwykle wymagane jest

określenie ich priorytetowości. Teoretycznie w naszym przypadku nie musimy tego robić, ponieważ priorytety wymuszone są przez nawiasy i nie może zdarzyć się na przykład taka formuła

$$f \wedge g \vee f$$

Jeśli już to albo taka

$$(f \wedge g) \vee f$$

albo taka

$$f \wedge (g \vee f).$$

Ze względów czysto praktycznych (bardziej czytelny i zwarty zapis) wprowadzimy jednak priorytety (zachowujac przy tym możlwiość użycia nawiasów do grupowania jeśli zajdzie taka potrzeba) określając, że symbol $\wedge$ rozpatrywany jest przed $\vee$ zaś $\neg$ przed $\wedge$.

Definicja 12.3. *Niech f będzie formułą a $\mathcal{F}$ przyporządkowaniem. Jeśli $\mathcal{F}$ określone jest dla każdej formuły atomowej występującej w f, wówczas o $\mathcal{F}$ powiemy, że jest odpowiednie (ang.* suitable*) dla f lub że jest odpowiednim przyporządkowaniem (ang.* suitable assignment*).*

Definicja 12.4. *Jeśli $\mathcal{F}$ jest odpowiednie dla f i $\mathcal{F}(f) = 1$ wówczas powiemy, że $\mathcal{F}$ jest* **modelem** *(ang.* model*) dla f lub, że f zachodzi (ang.* hold*) dla przypisania $\mathcal{F}$. Zapisywać będziemy to w następujący sposób*

$$\mathcal{F} \models f.$$

W przeciwnym razie piszemy

$$\mathcal{F} \not\models f.$$

Definicja 12.5. *Formuła f jest* **spełnialna** *(ang.* satisfiable*) jeśli istnieje dla f co najmniej jedne model. W przeciwnym przypadku formułę f nazywamy* **niespełnialną** *(ang.* unsatisfiable*) lub* **sprzeczną** *(ang.* contradictory*). Zbiór formuł S nazywamy spełnialnym jeśli istnieje model $\mathcal{F}$ taki, że $\mathcal{F} \models f$, gdzie f jest dowolną formułą ze zbioru S.*

Definicja 12.6. *Formułę f nazywamy* **prawdziwą** *(ang.* valid*) lub* **tautologią** *(ang.* tautology*) jeśli każde odpowiednie przyporządkowanie jest modelem dla f. Zapisywać będziemy to jako*

$$\models f.$$

W przeciwnym razie piszemy

$$\not\models f.$$

Z powyższych definicji wynika, że aby (skończoną) formułę f nazwać spełnialną (tautologią), wystarczy sprawdzić skończoną ilość przyporządkowań dla formuł atomowych z f. Jeśli bowiem formuła f składa się z n formuł atomowych $a_1, \ldots, a_n$ wówczas istnieje dokładnie 2^n różnych przyporządkowań. Sprawdzenia takiego można więc dokonać w usystematyzowany sposób za pomocą **tablicy prawdy** (ang. *truth-table*)

	a_1	a_2	$\cdots$	a_n	f
$\mathcal{F}_1$	0	0	$\cdots$	0	$\mathcal{F}_1(f)$
$\mathcal{F}_2$	0	0	$\cdots$	1	$\mathcal{F}_2(f)$
$\vdots$			$\ddots$		$\vdots$
$\mathcal{F}_{2^n}$	1	1	$\cdots$	1	$\mathcal{F}_{2^n}(f)$

Jeśli formuła f jest spełnialna to w ostatniej kolumnie przynajmniej w jednym wierszu powinniśmy otrzymać wartość 1. Jeśli formuła f jest tautologią to ostatnia kolumna powinna zawierać tylko i wyłącznie wartości 1.

Twierdzenie 12.1. *Formuła f jest tautologią wtedy i tylko wtedy, gdy $\neg f$ jest niespełnialna.*

Dowód. Dowód pozostawiamy jako ćwiczenie. □

Twierdzenie 12.2. *Dla dowolnych formuł f, g i h zachodzą następujące równoważności*

1. *przemienność (ang.* commutativty*)*

$$f \wedge g \equiv g \wedge f$$

$$f \vee g \equiv g \vee f$$

2. *łączność (ang.* associativity*)*

$$(f \wedge g) \wedge h \equiv f \wedge (g \wedge h)$$

$$(f \vee g) \vee h \equiv f \vee (g \vee h)$$

3. *rozdzielność (ang.* distributivity*)*

$$f \wedge (g \vee h) \equiv (f \wedge g) \vee (f \wedge h)$$

$$f \vee (g \wedge h) \equiv (f \vee g) \wedge (f \vee h)$$

4. *pochłanianie (absorpcja) (ang.* absorption*)*

$$f \wedge (f \vee g) \equiv f$$

$$f \vee (f \wedge g) \equiv f$$

5. *prawo podwójnego przeczenia (ang.* double negation*)*

$$\neg\neg f \equiv f$$

6. *prawo de'Morgana (ang.* de'Morgans laws*)*

$$\neg(f \wedge g) \equiv \neg f \vee \neg g$$

$$\neg(f \vee g) \equiv \neg f \wedge \neg g$$

7. *idempotentność (ang.* idempotency*)*

$$f \wedge f \equiv f$$

$$f \vee f \equiv f$$

8. *(???) identyczności(ang.* identity*) w uogólnionej wersji (ang.* tautology laws*)*

$$f \vee g \equiv f, \quad \text{jeśli } f \text{ jest tautologią}$$

$$f \wedge g \equiv g, \quad \text{jeśli } f \text{ jest tautologią}$$

9. spełnialność (ang. unsatisfiability laws*)*

$$f \vee g \equiv g, \quad \text{jeśli } f \text{ nie jest spełnialne}$$

$$f \wedge g \equiv f, \quad \text{jeśli } f \text{ nie jest spełnialne}$$

Dowód. Dowód z wykorzystaniem definicji semantycznej lub tablic prawdy pozostawiamy jako ćwiczenie. □

W dalszej części przyjmujemy następujące skrócone formy zapisu

- zamiast $\neg f \vee g$ piszemy $f \rightarrow g$,
- zamiast $(f \wedge g) \vee (\neg f \wedge \neg g)$ piszemy $f \leftrightarrow g$,
- zamiast $(f \rightarrow g) \wedge (g \rightarrow f)$ piszemy $f \leftrightarrow g$,
- zamiast $a_1 \wedge \cdots \wedge a_n$ piszemy $\bigwedge_{i=1}^{n} a_i$,
- zamiast $a_1 \vee \cdots \vee a_n$ piszemy $\bigvee_{i=1}^{n} a_i$.

Twierdzenie 12.3. *Formuła g jest nazywana* **wnioskiem** *(ang.* consequence*) ze zbioru formuł $\{f_1, \ldots, f_n\}$ jeśli dla każdego przypisania $\mathcal{F}$ odpowiadającego wszystkim formułom $\{f_1, \ldots, f_n\}$ oraz formule g zachodzi, że jeśli $\mathcal{F}$ jest modelem dla $\{f_1, \ldots, f_n\}$ wówczas jest także modelem dla g. Następujące stwierdzenia są równoważne*

1. *g jest wnioskiem z formuł $\{f_1, \ldots, f_n\}$.*
2. *$(\bigwedge_{i=1}^{n} f_i) \rightarrow g$ jest tautologią.*
3. *$(\bigwedge_{i=1}^{n} f_i) \wedge g$ jest niespełnialne.*

Dowód. Dowód tutu. □

12.1 Postać normalna

Okazuje się, że każda formuła, bez względu na to jak by skomplikowaną się wydawała, daje się zapisać w pewien uniwersalny (znormalizowany) sposób.

Definicja 12.7. *Formułę atomową lub jej negację ogólnie nazywać będziemy* **literałem** *(ang.* literal*). W szczególności o literale odpowiadającym niezanegowanej formule atomowej powiemy, że występuje w* **afirmacji**, *a o literale odpowiadającym zanegowanej formule atomowej powiemy, że występuje w* **negacji**. *Ten ostatni nazywać będziemy także* **literałem zanegowanym**. *Ponad to przyjmujemy następującą umowę. Jeśli l jest literałem oraz a formułą atomową, wówczas $\bar{l}$ definiujemy jako*

$$\bar{l} = \begin{cases} \neg a & \text{jeśli} \quad l = a, \\ a & \text{jeśli} \quad l = \neg a, \end{cases}$$

Definicja 12.8 (Postać normalna)**.** *Powiemy, że formuła f jest w* **koniunkcyjnej postaci normalnej** *(ang.* conjunctive normal form*), (***CNF***) jeśli jest koniunkcją dysjunkcji literałów, czyli jeśli jest postaci*

$$f = \bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m} a_{i,j}),$$

gdzie $a_{i,j}$ są literałami. Formuła f jest w **dysjunkcyjnej postaci normalnej** *(ang.* disjunctive normal form*), (***DNF***) jeśli jest dysjunkcją koniunkcji literałów, czyli jeśli jest postaci*

$$f = \bigvee_{i=1}^{n} (\bigwedge_{j=1}^{m} a_{i,j}),$$

gdzie $a_{i,j}$ są literałami.

Twierdzenie 12.4. *Dla każdej formuły f istnieje równoważna jej formuła f_C w postaci CNF i równoważna jej formuła f_D w postaci DNF.*

Dowód. Dowód //tutu//. □

12.1.1 Przekształcanie do postaci normalnej

Podamy teraz dwa sposoby przekształcenia dowolnej formuły do postaci normalnej.

Sposób I

Chcąc przekształcić dowolną formułę f do postaci CNF należy wykonać następujące kroki

1. Dopóki jest to możliwe, wykonuj następujące podstawienia w formule f
   - formułę $\neg\neg g$ zastąp przez formułę g,
   - formułę $\neg(g \wedge h)$ zastąp przez formułę $\neg g \vee \neg h$,
   - formułę $\neg(g \vee h)$ zastąp przez formułę $\neg g \wedge \neg h$.
2. Dopóki jest to możliwe, wykonuj następujące podstawienie w formule f: formułę $p \vee (q \wedge r)$ zastąp przez formułę $(p \vee q) \wedge (p \vee r)$.

Chcąc przekształcić dowolną formułę f do postaci DNF należy w kroku 2, dopóki jest to możliwe, wykonywać następujące podstawienie w formule f: formułę $p \wedge (q \vee r)$ zastąp przez formułę $(p \wedge q) \vee (p \wedge r)$.

Sposób II

Dla formuły f należy skonstruować tablicę prawdy. Aby otrzymać równoważną postać CNF należy

1. Odrzucić wszystkie wiersze, dla których przyporządkowanie zwraca wartość 1.
2. Dla każdego z pozostałych wierszy budujemy formułę. Jeśli w i-tym wierszu przypisanie $\mathcal{F}_i$ dla formuły atomowej a_j zwraca 1 wówczas do budowanej formuły f_i formuła atomowa wchodzi w negacji, w przeciwnym razie w afirmacji.

3. Elementy formuł f_i łączymy znakiem $\vee$.

4. Formuły f_i łączymy znakiem $\wedge$.

Przeanalizujemy teraz każdy z kroków na przykładzie. Niech dana będzie formuła, dla której tablica prawdy wygląda w następujący sposób

	p	q	r	f
$\mathcal{F}_1$	0	0	0	$\mathcal{F}_1(f) = 1$
$\mathcal{F}_2$	0	0	1	$\mathcal{F}_2(f) = 0$
$\mathcal{F}_3$	0	1	0	$\mathcal{F}_3(f) = 1$
$\mathcal{F}_4$	0	1	1	$\mathcal{F}_4(f) = 0$
$\mathcal{F}_5$	1	0	0	$\mathcal{F}_5(f) = 0$
$\mathcal{F}_6$	1	0	1	$\mathcal{F}_6(f) = 1$
$\mathcal{F}_7$	1	1	0	$\mathcal{F}_7(f) = 1$
$\mathcal{F}_8$	1	1	1	$\mathcal{F}_8(f) = 1$

Po wykonaniu kroku 1 tablica przyjmie postać

	p	q	r	f
$\mathcal{F}_2$	0	0	1	$\mathcal{F}_2(f) = 0$
$\mathcal{F}_4$	0	1	1	$\mathcal{F}_4(f) = 0$
$\mathcal{F}_5$	1	0	0	$\mathcal{F}_5(f) = 0$

Po wykonaniu kroku 2 na wierszu

- $i = 2$ otrzymujemy $f_i = \{p, q, \neg r\}$,
- $i = 4$ otrzymujemy $f_i = \{p, \neg q, \neg r\}$,
- $i = 5$ otrzymujemy $f_i = \{\neg p, q, r\}$.

Po wykonaniu kroku 3 na wierszu

- $i = 2$ otrzymujemy $f_i = p \vee q \vee \neg r$,
- $i = 4$ otrzymujemy $f_i = p \vee \neg q \vee \neg r$,
- $i = 5$ otrzymujemy $f_i = \neg p \vee q \vee r$.

Po wykonaniu kroku 4 otrzymujemy ostateczną postać CNF dla formuły f

$$f_C = (p \vee q \vee \neg r) \wedge (p \vee \neg q \vee \neg r) \wedge (\neg p \vee q \vee r).$$

Aby otrzymać równoważną postać DNF należy

1. Odrzucić wszystkie wiersze, dla których przyporządkowanie zwraca wartość 0.

2. Dla każdego z pozostałych wierszy budujemy formułę. Jeśli w i-tym wierszu przypisanie $\mathcal{F}_i$ dla formuły atomowej a_j zwraca 1 wówczas do budowanej formuły f_i formuła atomowa wchodzi w afirmacji, w przeciwnym razie w negacji.

3. Elementy formuł f_i łączymy znakiem $\wedge$.

4. Formuły f_i łączymy znakiem $\vee$.

12.2 Formuła Horna

Definicja 12.9 (Formuła Horna)**.** *Formułę f zapisaną w postaci CNF nazywamy formułą Horna (ang.* Horn formula*) jeśli każda dysjunkcja zawiera co najwyżej jeden literał, który nie jest zanegowany.*

Formuły Horna mają dosyć ciekawą interpretację. Otóż każda formuła tego typu może być interpretowana jako ciąg „warunków" typu *jeśli ... to.* Na przykład formuła

$$f = (f \vee \neg g) \wedge (\neg f \vee \neg g \vee h) \wedge (\neg f \vee \neg g)$$

równoważna jest formule

$$f = (g \rightarrow f) \wedge (f \wedge g \rightarrow h) \wedge (f \wedge g \rightarrow 0)$$

co można czytać jako *f zachodzi jeśli prawdą jest, że z g wynika f oraz prawdą jest, że ...* lub inaczej *f zachodzi jeśli prawdą jest, że jeśli g to f oraz prawdą jest, że jeśli....*

Formuły Horna mają też bardzo praktyczną własność. Jak wiemy w oparciu o tablicę prawdy możemy sprawdzić spełnialność danej formuły. Niestety wraz ze wzrostem formuły, wykładniczo wzrasta czas na to potrzebny. W przypadku formuł Horna istnieje algorytm dający nam odpowiedź w czasie liniowym (zależnym od ilość formuł atomowych).

//tutu// algorytm
//tutu// twierdzenie

12.3 twierdzenie tutu

Twierdzenie 12.5. *Zbiór formuł S jest spełnialny wtedy i tylko wtedy gdy każdy skończony podzbiór zbioru S jest spełnialny.*

Dowód. Dowód
tutu
. □

12.4 Rezolucja

Rezolucja jest prostą operacją dokonującą zmiany syntaktycznej na kilku formułach – z dwóch formuł generowana jest trzecia formuła. Formuła ta powiększa zbiór formuł i może być użyta w kolejnym kroku rezolucji.

Warunkiem wstępnym dla stosowania rezolucji jest przedstawienie formuły w postaci CNF. Jak wiemy formuła CNF to formuła o ogólnej postaci

$$f = (l_{1,1} \vee \cdots \vee l_{1,n_1}) \wedge \cdots \wedge (l_{k,1} \vee \cdots \vee l_{k,n_k})$$

Zatem upraszczając notację możemy formułę f zapisać jako

$$f = \{l_{1,1}, \ldots, l_{1,n_1}\}, \ldots, \{l_{k,1}, \ldots, l_{k,n_k}\}.$$

Mówimy wówczas, że f jest zbiorem **klauzul** (ang. *clauses*) postaci $\{l_{i,1}, \ldots, l_{k,n_i}\}$. Elementy każdej kaluzuli traktuje się tak jak elementy połączone za pomocą $\vee$, kaluzule natomiast tak jak „obiekty" połączone za pomocą $\wedge$. Taki sposób zapisu, ma jeszcze jedną zaletę. Zauważmy, że wszelkie prawa jak np. łączność czy pochłanianie automatycznie wynika z własności dotyczących zbiorów.

Definicja 12.10. *Niech c_1, c_2 i r będą klauzulami. Wówczas r nazywane jest* **resolwentą** *(ang.* resolvent*) c_1 i c_2 jeśli istnieje literał l taki, że $l \in c_1$, $\bar{l} \in c_2$ i zachodzi*

$$r = (c_1 - \{l\}) \cup (c_2 - \{\bar{l}\}).$$

Inaczej mówiąc, resolwenta jest efektem zastosowania rezolucji.

W ten oto sposób podaliśmy formalną definicję pojęcia rezolucji, które pojawiło się w rozdziale 3. Zauważmy, że rezolucja dotyczy tylko takich klauzul, w których co najmniej jeden literał ma tą własność, że w jednej klauzuli występuje w afirmacji, w drugiej zaś w negacji. W takiej sytuacji w wyniku zastosowania rezolucji otrzymujemy klauzule która jest sumą tych dwóch klauzul minus literał występujący w afirmacji i negacji.

Twierdzenie 12.6. *Jeśli r jest resolwentą dwóch klauzul Horna, wówczas r także jest klauzulą Horna.*

Dowód. Dowód
tutu
. □

Twierdzenie 12.7. *Niech f będzie formułą postaci CNF i niech r będzie resolwentą klauzul $c_1, c_2 \in f$. Wówczas zbiory f i $f \cup \{r\}$ są równoważne.*

Dowód. Dowód
tutu
. □

Definicja 12.11. *Niech f będzie zbiorem klauzul. Wyrażenie $Res(f)$ definiujemy jako*

$$Res(f) = f \cup \{r : r \text{ jest resolwentą dwóch klauzul z } f\}.$$

Ponad to definiujemy

$$Res^0(f) = f,$$
$$Res^{k+1}(f) = Res(Res^k(f))$$

dla $k > 0$ oraz

$$Res^*(f) = \bigcup_{k \geq 0} Res^k(g).$$

Twierdzenie 12.8. *Dla każdego skończonego zbioru klauzul f istnieje $k \geq 0$ takie, że*

$$Res^k(f) = Res^{k+1}(f) = \cdots = Res^*(f).$$

Dowód. Dowód
tutu
. □

Twierdzenie 12.9. *Zbiór klauzul f jest niespełnialny wtedy i tylko wtedy, gdy gdy klauzula pusta należy do $Res^*(f)$.*

Dowód. Dowód
tutu
. □

12.5 Pytania i odpowiedzi.

Pytanie 12.1. Podaj dowód twierdzenia 12.4.

Pytanie 12.2. Stosując tablicę prawdy sprawdź, czy zadana formuła f jest tautologią.

Pytanie 12.3. Podaj dowód twierdzenia 12.2.

Rozdział 13

Ćwiczenia

13.1 Ćwiczenie 1

13.1.1 Zadanie

Utworzyć plik zawierający następujące fakty:

- Piotr lubi góry.
- Arek jest wysoki.
- Piotr dał Hani tulipany.
- Michał podróżuje autobusem do Paryża.
- Trojkąt, kwadrat, okrąg to figury.

13.1.2 Zadanie

Utworzyć plik zawierający reguły opisujące następujące zdania o świecie:

- Każdy człowiek je tylko mięso lub ser lub owoce.
- Jeśli X jest babcią Y, to Y jest wnukiem lub wnuczką.
- Dwie osoby są rodzeństwem, jeśli mają tych samych rodziców.
- X jest figurą jeśli jest trojkatem lub kwadratem lub okręgiem.

13.1.3 Zadanie

Napisać reguły opisujące następujące relacje pokrewieństwa:

- `jest_matka(X,Y)`;
- `jest_synem(X,Y)`;
- `brat(X,Y)`;
- `babcia(X,Y)`.

Jeśli to okaże się potrzebne użyj innych reguł, np. `ojciec(X,Y)`, `kobieta(X)`, `rodzic(X,Y)`.

13.1.4 Zadanie

Napisać program obliczający silnię.

13.1.5 Zadanie

Napisać program obliczający n-ty wyraz ciągu Fibonacciego.

13.2 Ćwiczenie 2

Liczby naturalne możemy zdefiniować w następujacy sposób

- $a_0 = 0$,
- $a_1 = 1 = 0 + 1$,
- $a_2 = 2 = 1 + 1$,
- $a_3 = 3 = 2 + 1$,
- $a_4 = 4 = 3 + 1$,
- $\ldots$,
- $a_n = a_{n-1} + 1$

Przyjmując, że mamy funkcję następnika $f(\cdot)$ dającą element następny, oraz symbol początkowy 0, możemy to samo zapisać jako

$$0, 1 = f(0), 2 = f(1), 3 = f(2), \ldots, n = f(n-1),$$

czyli inaczej

$$0, 1 = f(0), 2 = f(f(0)), 3 = f(f(f(0))), \ldots$$

Stąd możemy zapisać relację stwierdzającą czy coś jest liczbą naturalną w następujacy sposób

```
lN(0).
lN(f(X)) :- lN(X).


?- lN(f(f(f(0)))). % 3 jest liczba naturalna?

Yes
?-
```

Przy tak przyjętej definicji liczby naturalnej możemy wprowadzić relację większe lub równe (ang. gEq – *greater or equal*).

```
gEq(X, 0) :- lN(X).
gEq(f(X), f(Y)) :- gEq(X, Y).
```

13.2.1 Zadanie

Proszę napisać relację większy.

13.2.2 Zadanie

Operację dodawania można traktować jako relację wiążącą dwie (dodawane) liczby z trzecią będącą ich sumą. Rekurencyjnie dodawanie liczb naturalnych możemy opisać jako:

$$0 + X = X$$

$$f(X) + Y = f(X + Y)$$

Zatem dodanie liczb 3 i 2 wymaga następującego rozumowania

$$3 + 2 = f(2) + 2 = f(2 + 2) = f(4) = 5$$

Proszę napisać (relacje) dodawania.

13.2.3 Zadanie

Prosze napisać relację mnożenia (pamiętajmym, że mnożenie to wielokrotne dodawanie).

13.2.4 Zadanie

Prosze zapisać potęgowanie (pamiętajmy, że potęgowanie to wielokrotne mnożenie)

13.2.5 Zadanie

Proszę napisać program przeliczający zapis w postaci ciągu następników na liczbę naturalną.

13.2.6 Zadanie

Proszę napisać program zamieniający liczbę naturalną na ciąg następników.

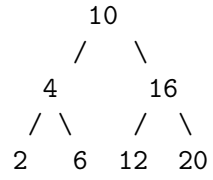
13.2.7 Zadanie

Napisz program mnożący dwie liczby naturalne podane jako ciąg następników, który będzie zwracał konkretną wartość liczbową (czyli nie ciąg następników).

13.3 Ćwiczenie 3

13.3.1 Zadanie

Zapisać drzewo binarne postaci



jako strukturę rekurencyjną, w której przyjmujemy, że węzeł drzewa binarnego zawiera liczbę oraz dwa drzewa binarne. Jeśli któreś z drzew binarnych w węźle nie występuje zapisujemy wówczas w to miejsce `pusty`.

13.3.2 Zadanie

Napisać predykat sprawdzajacy, czy przekazany argument jest drzewem binarnym (w sensie takiego drzewa, w którym każdy węzeł ma maksymalnie dwoje dzieci).

13.3.3 Zadanie

Napisać predykat sprawdzajacy, czy przekazany argument jest drzewem binarnym (w sensie takiego drzewa, w którym każdy węzeł ma maksymalnie dwoje dzieci oraz lewe podrzewo ma zawsze elementy mniejsze równe elementowi w węźle a prawe większe).

13.3.4 Zadanie

Napisać predykat sprawdzajacy, czy dany element należy do drzewa.

13.3.5 Zadanie

Napisać predykat obliczający sumę elementów znajdujących się w węzłach drzewa binarnego.

13.3.6 Zadanie

Napisać predykat zliczający ilość węzłów w drzewie binarnym

13.3.7 Zadanie

Napisać predykat przechodzący po drzewie w porządku preorder (węzeł, lewy podwęzeł, prawy podwęzeł).

13.3.8 Zadanie

Napisać predykat przechodzący po drzewie w porządku inorder (lewy podwęzeł, węzeł, prawy podwęzeł).

13.3.9 Zadanie

Napisać predykat przechodzący po drzewie w porządku postorder (lewy podwęzeł, prawy podwęzeł, węzeł).

13.3.10 Zadanie

Napisać predykat zwracający element maksymalny z drzewa binarnego.

13.3.11 Zadanie

Napisać predykat zliczający ilość liści w drzewie.

13.4 Ćwiczenie 4

13.4.1 Zadanie

Napisać predykat zliczający ilość elementów na liście.

13.4.2 Zadanie

Napisać predykat zliczający ilość wszystkich elementów na liście włącznie z elementami podlist.

13.4.3 Zadanie

Napisać predykat usuwający pojedyńcze wystąpienie elementu na liście.

13.4.4 Zadanie

Napisać predykat odwracający listę.

13.4.5 Zadanie

Napisać predykat zwracający ostatni element listy.

13.4.6 Zadanie

Napisać predykat zastępujący wszystkie wystąpienia elementu innym elementem.

13.5 Ćwiczenie 5

13.5.1 Zadanie

Napisać predykat znajdujący ostatni element na liście.

13.5.2 Zadanie

Napisać predykat sprawdzający czy dwa podane elementy sąsiadują ze sobą na liście.

13.5.3 Zadanie

Napisać predykat usuwający wszystkie wystąpienia zadanego elementu na liście.

13.5.4 Zadanie

Napisać predykat zamieniający wskazany element na inny.

13.5.5 Zadanie

Napisać predykat usuwający z listy wszystkie powtórzenia.

13.6 Ćwiczenie 6

13.6.1 Zadanie

Napisać program symulujący działanie maszyny Turinga. Tablica przejść określana jest w kodzie, natomiast stan taśmy jest argumentem predykatu. Kolejnym argumentem jest ilość iteracji jakie mają być wykonane. Algorytm powinien wypisywać wszystkie stany w jakich znalazła się maszyna i zawartość taśmy.

13.6.2 Zadanie

Tablice prawdy. Napisać program, który oblicza wartość logiczną wyrażenia boolowskiego, dla wszystkich kombinacji zmiennych np. dla wyrażenia

```
and(A,or(not(B),A))
```

zwrócić powinien coś w stylu

```
A = T B = T => T
A = T B = F => T
A = F B = T => F
A = F B = F => F
```

13.7 Ćwiczenie 7

13.7.1 Zadanie

Napisać program rozpoznający wielomiany zmiennej x. Wielomian definiujemy następująco:

- stała jest wielomianem zmiennej x;
- x jest wielomianem zmiennej x;
- suma, różnica, iloczyn są wielomianami zmiennej x;
- wielomian dzielony przez liczbę lub stałą jest wielomianem.

13.7.2 Zadanie

Napisać program obliczający pochodną.

13.8 Ćwiczenie 8

13.8.1 Zadanie

W ogrodzie siedziały cztery pary narzeczonych zajadając się śliwkami. Andzia zjadła 2 śliwki, Beata – 3, Celina – 4, Danusia – 5. Ich narzeczeni również nie próżnowali. Andrzej zjadł tyle, co jego narzeczona, Bogumił 2 razy tyle, co jego narzeczona, Cezary 3 razy tyle co jego narzeczona, wreszcie Damian 4 razy tyle, co jego narzeczona. Wszyscy razem zjedli 44 śliwki. Jakie są imiona par narzeczonych?

13.8.2 Zadanie

Piecioro przyjaciół rywalizowało na biezni. Wincenty ze smutkiem opowiedział, ze mu sie nie udało zajac pierwszego miejsca. Grzegorz przybiegł na mete jako trzeci po Dymitrze. Wincenty zauwazył na marginesie, ze Dymitr nie zajał drugiego miejsca, a Andrzej nie był ani pierwszym ani ostatnim. Borys powiedział, ze przybiegł na mete w slad za Wincentym.

13.8.3 Zadanie

Sześciu mężczyzn o różnych imionach (Andrzej, Dariusz, Mateusz, Michał, Tomasz, Wojciech) jest żonatych z kobietami o różnych imionach (Agnieszka, Ewa, Róża, Ania, Alicja, Ola)[1] oraz posiada sześć różnych samochodów (BMW, Fiat, Ford, Mercedes, Opel, Renault) w sześciu różnych kolorach (biały, czerwony, granat, zielony, żółty, szary). Każdy pracuje w innym miejscu pracy (bank, hotel, politechnika, restauracja, sąd, szpital) i ma inne hobby(czytanie, fotografia, majsterkowanie, pływanie, rowery, żeglarstwo).

Wiedząc, że

- Agnieszka i Andrzej są małżeństwem.
- Ola ma czerwony samochód.
- Fiat jest zielony.
- Właściciel Opla pracuje w restauracji.
- Pracownik politechniki lubi czytanie.
- Wojciech pływa w wolnych chwilach.
- Właściciel granatowego Forda lubi fotografować.
- Tomasz jest właścicielem BMW.
- Mąż Ewy lubi żeglarstwo.
- Dariusz ma biały samochód.
- Mąż Róży posiada Mercedesa.
- Właściciel szarego auta pracuje w sądzie.

[1] Obowiązuje monogamia.

- Po lewej stronie pracownika banku mieszka właściciel fiata, a po prawej pasjonat kolarstwa.
- Po lewej stronie pracownika hotelu mieszka właściciel granatowego samochodu, a z drugiej strony pracownik szpitala.
- Michał mieszka na prawo od Ani, a na lewo od niej mieszka Mateusz.
- Sąsiadem Alicji od lewej strony jest właściciel żółtego auta, a jego sąsiadem od lewej strony jest znany majsterkowicz.

odpowiedz na pytania

- Kto pracuje w szpitalu?
- Jaki Kolor ma Renault?

13.9 Ćwiczenie 9

13.9.1 Zadanie

Opierając się na wiadomościach z rozdziału 12 (a w szczególności podrozdziału 12.1) napisz program który dowolną formułę rachunku zdań sprowadzi do postaci klauzulowej. Zadanie to wykonać można albo przez przekształcanie formuły (sposób I ze strony 12.1.1), albo za pomocą tablicy prawdy (którą najpierw należy utworzyć w oparciu o wartości formuły dla różnych przyporządkowań, sposób II ze strony 12.1.1).

Rozdział 14

Rozwiązania ćwiczeń

14.1 Odpowiedzi do zadania 13.1

14.1.1 Odpowiedzi do zadania 13.1.1

```
lubi(piotr,gory).
wysoki(arek).
dal(piotr,hania,tulipan).
podrozuje(michal,autobus,paryz).
figura(trojkat).
figura(kwadrat).
figura(okrag).

?- [t].
\% t compiled 0.00 sec, 1,988 bytes

Yes
?- wysoki(arek).

Yes
?- wysoki(X).

X = arek ;

No
?- figura(trojkat).

Yes
?- figura(X).

X = trojkat ;

X = kwadrat ;

X = okrag ;

No
?- X(trojkat).
ERROR: Syntax error: Operator expected
ERROR: X(trojkat
ERROR: ** here **
ERROR: ) .
?-
```

14.1.2 Odpowiedzi do zadania 13.1.2

```
/* Każdy człowiek je tylko mięso lub ser lub owoce. */
/*
mieso(schab).
ser(gouda).
```

```
owoce(banan).
je(czlowiek,X) :- mieso(X); ser(X); owoce(X).
*/
jedzenie(mieso).
jedzenie(ser).
jedzenie(owoce).
je(czlowiek,X) :- jedzenie(X).

/* Jeśli X jest babcią Y, to Y jest wnukiem lub wnuczką X. */
/* raczej nie tak:

wnuk(jan,ada).
wnuczka(ania,ada).
babcia(Y,X) :- wnuk(X,Y); wnuczka(X,Y).

ze względu na związek jeśli... to...,
który zapisujemy jako: to :- jeśli
*/
babcia(ala,zosia).
babcia(ala,henio).
wnuk(X,Y) :- babcia(Y,X).
wnuczka(X,Y) :- babcia(Y,X).

/* Dwie osoby są rodzeństwem, jeśli mają tych samych rodziców. */
rodzic(ala,ela).
rodzic(ala,adam).
rodzic(jan,ela).
rodzic(jan,adam).
rodzic(henio,fiona).
rodzic(bilbo,antek).
rodzic(bilbo,beata).

rodzenstwo(X,Y) :- rodzic(A,X), rodzic(A,Y), rodzic(B,X), rodzic(B,Y), A\=B.

/* X jest figurą jeśli jest trojkatem lub kwadratem lub okręgiem. */
figura(trojkat).
figura(kwadrat).
figura(okrag).

jest_figura(X) :- figura(X).
```

14.1.3 Odpowiedzi do zadania 13.1.3

14.1.4 Odpowiedzi do zadania 13.1.4

```
silnia(0,F) :- F is 1.

silnia(N1,F1) :- N1>0,
                 N2 is N1-1,
```

```
                silnia(N2,F2),
                F1 is N1*F2.

/*
po pierwsze N1 musi byc wieksze od 0,
po drugie musimy najpierw znac wartość
  silni dla wyrazu o jeden mniejszego
gdy juz te wartosc znamy, to mozemy
obliczyc właściwa wartosć czyli F1
*/
```

14.1.5 Odpowiedzi do zadania 13.1.5

```
fib(0,F) :- F is 1.
fib(1,F) :- F is 1.

fib(N,F):- N>1,
           N1 is N-1,
           N2 is N-2,
           fib(N1,F1),
           fib(N2,F2),
           F is (F1 + F2).
```

14.2 Odpowiedzi do zadania 13.2

14.2.1 Odpowiedzi do zadania 13.2.1

```
gEq(X, 0) :- lN(X),X\=0.
gEq(f(X), f(Y)) :- gEq(X, Y).
```

lub

```
gEq(f(X), 0) :- lN(X).
gEq(f(X), f(Y)) :- gEq(X, Y).
```

14.2.2 Odpowiedzi do zadania 13.2.2

```
add(0, X, X).
add(f(X), Y, f(Z)) :- add(X, Y, Z).
```

Działanie

```
?- add(f(f(f(0))), f(f(0)), X).
X = f(f(f(f(f(0)))))
Yes
?-
```

14.2.3 Odpowiedzi do zadania 13.2.3

```
mul(0, X, 0).
mul(f(X), Y, Z) :- mul(X, Y, XY), add(XY, Y, Z).
```

Działanie

```
?- mul(f(f(0)),f(f(f(0))),Z).
Z = f(f(f(f(f(f(0))))))
Yes
?-
```

Idea, jak to działa

```
mul(f(f(0)),f(f(f(0))),Z)
  |
  mul(f(0),f(f(f(0))),XY)
    |
    mul(0,f(f(f(0))),XY) ukonkretnienie XY wartoscia 0
                          bo pasuje to do mul(0,X,0)
                         , add(0,f(f(f(0)),XY)
```

14.2.4 Odpowiedzi do zadania 13.2.4

```
pow(f(X), 0, 0).
pow(0, f(X), f(0)).
pow(f(N), X, Y) :- pow(N, X, Z), mul(Z, X, Y).
```

14.2.5 Odpowiedzi do zadania 13.2.5

```
value(0,0).
value(f(X),Wartosc) :- value(X,WartoscTmp),Wartosc is WartoscTmp+1.
```

14.2.6 Odpowiedzi do zadania 13.2.6

```
repr(0,0).
repr(N,f(X)) :- N>0, N1 is N-1, repr(N1,X).
```

14.2.7 Odpowiedzi do zadania 13.2.7

```
mulVal(X,Y,V) :- mul(X,Y,Z), value(Z,V).
```

14.3 Odpowiedzi do zadania 13.3

14.3.1 Odpowiedzi do zadania 13.3.1

```
(10,(4,(2,pusty,pusty),(6,pusty,pusty)),(16,(12,pusty,pusty),(20,pusty,pusty)))
```

14.3.2 Odpowiedzi do zadania 13.3.2

```
drzewoB(pusty).
drzewoB((_,Lewe,Prawe)) :- drzewoB(Lewe), drzewoB(Prawe).
```

14.3.3 Odpowiedzi do zadania 13.3.3

```
zwroc((X,_,_),X).

db(pusty).
db((_,pusty,pusty)).
db((X,pusty,P)):-db(P),zwroc(P,Z),X<Z.
db((X,L,pusty)):-db(L),zwroc(L,Z),X>=Z.
db((X,L,P)):-db(L),db(P),zwroc(L,Z1),zwroc(P,Z2),X < Z2,X >= Z1.
```

14.3.4 Odpowiedzi do zadania 13.3.4

```
elementB(X,(X,_,_)).
elementB(X,(Y,Lewe,Prawe)) :- \+ (X=Y), (elementB(X,Lewe);elementB(X,Prawe)).
```

14.3.5 Odpowiedzi do zadania 13.3.5

```
suma(pusty,0).
suma((Element,Lewe,Prawe),Suma) :-
      suma(Lewe,SumaL),
      suma(Prawe,SumaP),
      Suma is SumaL+SumaP+Element.
```

14.3.6 Odpowiedzi do zadania 13.3.6

```
ileWezlow(pusty,0).
ileWezlow((_,Lewe,Prawe),Ile) :-
      ileWezlow(Lewe,IleL),
      ileWezlow(Prawe,IleP),
      Ile is IleL+IleP+1.
```

14.3.7 Odpowiedzi do zadania 13.3.7

```
preorder((X,L,P),Xs) :- preorder(L,Ls), preorder(P,Ps), append([X|Ls],Ps,Xs).
preorder(pusty,[]).
```

14.3.8 Odpowiedzi do zadania 13.3.8

```
inorder((X,L,P),Xs) :- inorder(L,Ls), inorder(P,Ps), append(Ls,[X|Ps],Xs).
inorder(pusty,[]).
```

14.3.9 Odpowiedzi do zadania 13.3.9

```
postorder((X,L,P),Xs) :- postorder(L,Ls),postorder(P,Ps),
                         append(Ps,[X],Ps1),append(Ls,Ps1,Xs).
postorder(pusty,[]).
```

14.3.10 Odpowiedzi do zadania 13.3.10

```
max(pusty,fail).
max((Element,_,pusty),Element).
max((_,_,Prawe),Max) :- \+ (Prawe=pusty), max(Prawe,Max).
```

14.3.11 Odpowiedzi do zadania 13.3.11

```
ileLisci(pusty,0).
ileLisci((_,pusty,pusty),1).
ileLisci((_,Lewe,Prawe),Ile) :- \+ (Lewe=pusty,Prawe=pusty),
                                ileLisci(Lewe,IleL),
                                ileLisci(Prawe,IleP),
                                Ile is IleL+IleP.
```

14.4 Ćwiczenie 4

14.4.1 Odpowiedzi do zadania 13.4.1

```
co([],0).
co([H|T],X):- co(T,X1), X is X1+1.
```

14.4.2 Odpowiedzi do zadania 13.4.2

```
co([],0).
co(H,1) :- \+lista(H).
co([H|T],X):- co(H,X1),co(T,X2), X is X1+X2.
```

14.4.3 Odpowiedzi do zadania 13.4.3

```
usun(X,[X|Xs],Xs).
usun(X,[Y|Ys],[Y|Zs]) :- usun(X,Ys,Zs).
```

14.4.4 Odpowiedzi do zadania 13.4.4

```
odwroc([],[]).
odwroc([X|Xs],Zs) :- odwroc(Xs,Ys), polacz(Ys,[X],Zs).
```

14.4.5 Odpowiedzi do zadania 13.4.5

```
ostatni([X],X).
ostatni([H|T],X) :- ostatni(T,X).
```

14.4.6 Odpowiedzi do zadania 13.4.6

```
zastap([],Y1,Y2,[]).
zastap([Y1|T1],Y1,Y2,[Y2|T2]) :- zastap(T1,Y1,Y2,T2).
zastap([H|T1],Y1,Y2,[H|T2]) :- \+ (H =Y1), zastap(T1,Y1,Y2,T2).
```

14.5 Ćwiczenie 5

14.5.1 Odpowiedzi do zadania 13.5.1

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

14.5.2 Odpowiedzi do zadania 13.5.2

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

14.5.3 Odpowiedzi do zadania 13.5.3

```
usunAll(_,[],[]).
usunAll(X,[X|T],Z) :- !, usunAll(X,T,Z).
usunAll(X,[Y|T1],[Y|T2]) :- usunAll(X,T1,T2).
```

14.5.4 Odpowiedzi do zadania 13.5.4

```
%elementSzukany,NowyElement,Lista,Wynik
zamien(_,_,[],[]).
zamien(X,Y,[X|T1],[Y|T2]) :- !, zamien(X,Y,T1,T2).
zamien(X,Y,[Z|T1],[Z|T2]) :- !, zamien(X,Y,T1,T2).
```

14.5.5 Odpowiedzi do zadania 13.5.5

```
%zmienna A pelni role akumulatora, gromadzacego wszystkie
%pojedyncze wystapienia elementow
usunPowt(L,W) :- usunPowtHelp(L,[],W).
usunPowtHelp([],A,A).
usunPowtHelp([H|T],A,L) :- nalezy(H,A), !, usunPowtHelp(T,A,L).
usunPowtHelp([H|T],A,L) :- usunPowtHelp(T,[H|A],L).
```

14.6 Ćwiczenie 6

14.6.1 Odpowiedzi do zadania 13.6.1

Kod programu

```
%maszyna turinga - definicja funkcji przejscia wg. formatu
%(stan,odczytanySymbol,nowyStan,symbolDoZapisania,kierunek)
d(s,a,1,b,r).
d(d,b,2,a,r).
d(1,a,1,b,l).
d(1,b,k,a,r).
d(2,a,k,b,r).
d(2,b,2,a,l).

%predykaty pomocnicze
odwroc([],[]).
odwroc([X|Xs],Zs) :- odwroc(Xs,Ys), polacz(Ys,[X],Zs).

polacz([],Lista,Lista).
polacz([H|T],Lista,[H|Res]) :- polacz(T,Lista,Res).
%koniec predykatow pomocniczych

%poruszanie sie po tasmie w prawo
%gdy ,,skonczyla'' sie tasma - dodaje elementy puste
tape([],1,[],[.],[]).
tape([],Pos,[.|L],H,R) :- Pos>1,
                          PosTmp is Pos -1,
                          tape([],PosTmp,L,H,R).

%gdy tasma sie nie ,,skonczyla'' - wykorzystuje elementy, ktore sa
tape([HT|TT],1,[],HT,TT).
tape([HT|TT],Pos,[HT|L],H,R) :- Pos>1,PosTmp is Pos -1,
                                tape(TT,PosTmp,L,H,R).

%poruszanie sie po tasmie w lewo
%odwracam tasme i stosuje reguly dla poruszania sie w prawo a potem odwracam
%i zamieniam wyniki
tape(List,-1,RRev,H,L) :- odwroc(List,RevList),
                          tape(RevList,1,L,H,R),
                          odwroc(R,RRev).
tape(List,Pos,RRev,H,LRev) :- Pos < -1,
                              PosTmp is Pos * (-1),
                              odwroc(List,RevList),
                              tape(RevList,PosTmp,L,H,R),
                              odwroc(L,LRev),
                              odwroc(R,RRev).

turing(_,_,_,0) :- !.
```

```
turing(Tape,Pos,State,Steps) :- tape(Tape,Pos,L,H,R),
                          d(State,H,NewState,NewSymbol,Dir),
                          aktualizujPos(Pos,Dir,NewPos),
                          aktualizujTasme(L,NewSymbol,R,NewTasma),
                          wypiszTasme(NewTasma,NewPos),
                          wypiszStan(NewState),
                          NewSteps is Steps-1,
                          turing(NewTasma,NewPos,NewState,NewSteps).

aktualizujPos(P,r,NP) :- NP is P + 1.
aktualizujPos(P,l,NP) :- NP is P - 1.

aktualizujTasme(L,NewSymbol,R,NewTasma) :- polacz(L,[NewSymbol],T),
                                           polacz(T,R,NewTasma).

wypiszTasme(T,P) :- tape(T,P,L,H,R),wyp(L),wypS(H),wyp(R).
wyp([]).
wyp([H|T]) :- write(H),write(','),wyp(T).
wypS(X) :- write('|'),write(X),write('|').

wypiszStan(S) :- write(' nowy stan: '),write(S),nl.
```

Wywołanie

```
?- turing([a,b,a,b,a,a,a,b,a],5,s,3).
a,b,a,b,b,|a|a,b,a, nowy stan: 1
a,b,a,b,|b|b,a,b,a, nowy stan: 1
a,b,a,b,a,|b|a,b,a, nowy stan: k
More?
```

14.6.2 Odpowiedzi do zadania 13.6.2

Wersja działajaca dla określonej ilości zmiennych, wymagająca ingerencji w kod przy zmianie ich ilości.

```
v(true).
v(fail).

a(A,B):-A,B.
o(A,B):-(A;B),!.
n(A):- \+A.

sprawdz(A,B,C,Expr) :- v(A),v(B),v(C),
                       write(' '),write(A),
                       write(' '),write(B),
                       write(' '),write(C),
                       write(Expr),calc(Expr),fail.

calc(Expr):- Expr,write(' -> T'),nl.
calc(Expr):- \+Expr,write(' -> F'),nl.
```

Wersja działajaca dla dowolnej ilości zmiennych, które muszą być przekazane w liście jako pierwszy argument wywołania, np.

```
sprawdz([A,B,C],a(A,o(n(B),C))).
```

Program.

```
v(true).
v(fail).

a(A,B):-A,B.
o(A,B):-(A;B),!.
n(A):- \+A.

sprawdz([],_).
sprawdz([A|B],Expr) :- v(A),write(' '),write(A), sprawdz(B,Expr),calc(Expr),fail.

calc(Expr):- Expr,write(' -> T'),nl.
calc(Expr):- \+Expr,write(' -> F'),nl.
```

Niestety wyniki nie są czytelne

```
?- sprawdz([A,B,C],a(A,o(n(B),C))).
 true true true -> T
 fail -> F
 fail true -> T
 fail -> T
 fail true true -> F
 fail -> F
 fail true -> F
 fail -> F

No
```

Aby poprawić czytelność należy napisać program tak, aby powyższy wynik został wyświetlony w postaci

```
?- sprawdz([A,B,C],a(A,o(n(B),C))).
 true true true -> T
           fail -> F
      fail true -> T
           fail -> T
 fail true true -> F
           fail -> F
      fail true -> F
           fail -> F

No
```

14.7 Ćwiczenie 7

14.7.1 Odpowiedzi do zadania 13.7.1

```
wielomian(X,X).
wielomian(T,X) :- number(T).
wielomian(T,X) :- atom(T).
wielomian(W1+W2, X) :- wielomian(W1, X), wielomian(W2, X).
wielomian(W1-W2, X) :- wielomian(W1, X), wielomian(W2, X).
wielomian(W1*W2, X) :- wielomian(W1, X), wielomian(W2, X).
wielomian(W1/W2, X) :- wielomian(W1, X), (number(W2); atom(W2)).
wielomian(W^N, X) :- wielomian(W, X), integer(N), N>0.
```

14.7.2 Odpowiedzi do zadania 13.7.2

```
d(X,X,1) :- !.
d(C,X,0) :- number(C); atom(C).
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) :- (number(C);atom(C)),\+ C=X,d(U,X,A),!. %przypadek szczególny
d(U*V,X,A*V+U*B) :- d(U,X,A), d(V,X,B).
d(pow(U,C),X,C*pow(U,NC)*W) :- number(C),NC is C - 1,d(U,X,W).

r(W,W) :- (number(W);atom(W)),!.
r(W,N) :- W =..[Oper,L,R],
          r(L,Xl),
          r(R,Xr),
          red(Oper,Xl,Xr,N).

red(+,X,0,X).
red(+,0,X,X).
red(+,X,Y,Z) :- number(X),number(Y),Z is X+Y,!.
red(+,X,Y,X+Y).

red(-,X,0,X).
red(-,0,X,X).
red(-,X,Y,Z) :- number(X),number(Y),Z is X-Y,!.
red(-,X,Y,X-Y).

red(*,_,0,0).
red(*,0,_,0).

red(*,X,1,X).
red(*,1,X,X).

red(*,X,Y,X*Y).

dr(W,X,Wynik) :- d(W,X,WW),r(WW,Wynik).
```

14.8 Ćwiczenie 8

14.8.1 Odpowiedzi do zadania 13.8.1

14.8.2 Odpowiedzi do zadania 13.8.2

14.8.3 Odpowiedzi do zadania 13.8.3

14.9 Ćwiczenie 9

14.9.1 Odpowiedzi do zadania 13.9.1

Bibliografia

[1] I. Bratko, *Prolog Programming for Artificial Intelligence*, 3rd edition, Addison-Wesley Publishers, 2001.

[2] W. F. Clocksin, C. S. Mellish, *Prolog. Programowanie*, Wydawnictwo HELION, Gliwice, 2003.

[3] M. A. Covington, *Efficient Prolog: a practical tutorial*, Artificial Intelligence Review, 273-287, 5 (1991).

[4] M. A. Covington, *Research Report AI-1989-08. Efficient Prolog: A Practical Guide*, Artificial Intelligence Programs, The University of Georgia, Athens, Georgia 30602, August 16, 1989.

[5] U. Endriss, *Lecture notes. An Introduction to Prolog Programming*, University of Amsterdam, Version: 7 July 2007.

[6] L. Sterling, E. Shapiro. *The Art of Prolog*, 2nd edition, MIT Press, 1994.

[7] J. Lu, J. J. Mead. *Prolog. A Tutorial Introduction*, Computer Science Department Bucknell University Lewisburg, PA 17387.