Uniwersytet Łódzki

Wydział Matematyki i Informatyki

Informatyka

Języki programowania sztucznej inteligencji

Piotr Fulmański

Łódź, 2010

Spis treści

Spis treści iii

Przedmowa xiii

I Prolog

Wykład 1

1 Podstawy 3

1.1 Zdumiewający początek 3

1.2 Obiekty i relacje 4

1.2.1 Obiekt „klasyczny” 4

1.2.2 Obiekt w Prologu 6

1.3 Program w Prologu 10

1.3.1 Struktura i składnia 10

1.3.2 Praca z programem – zapytania 12

1.4 Pytania i odpowiedzi 14

2 Składnia Prologa 17

2.1 Termy 17

2.2 Klauzule, program i zapytania 20

2.3 Pytania i odpowiedzi 25

3 Ewaluacja zapytania 29

3.1 Dopasowywanie wyrażeń 29

3.2 Obliczanie celu 31

3.3 Pytania i odpowiedzi . . . 39

4 Listy **43**

4.1 Składnia . . . 43

4.2 Głowa i ogon . . . 44

4.3 Pytania i odpowiedzi . . . 48

5 Odcięcie **53**

5.1 Wiele rozwiązań: mechanizm nawracania . . . 53

5.2 Odcięcie . . . 60

5.3 Pytania i odpowiedzi. . . . 63

6 Powtarzamy wiadomości **69**

6.1 O rekurencji raz jeszcze . . . 69

6.1.1 Sposób 1 . . . 69

6.1.2 Sposób 2 . . . 69

6.1.3 Sposób 3 . . . 70

6.1.4 Sposób 4 . . . 70

6.2 Akumulator . . . 70

6.2.1 Przykład z listą . . . 70

6.2.2 Przykład z liczbami . . . 72

6.3 Z góry na dół czy od dołu do góry? . . . 73

6.3.1 Z góry na dół . . . 73

6.3.2 Z dołu do góry . . . 74

6.4 Pytania i odpowiedzi . . . 75

7 Od problemu do jego (efektywnego) rozwiązania **79**

7.1 Rebus i jego pierwsze rozwiązanie . . . 79

7.2 Rozwiązanie drugie . . . 81

7.3 Rozwiązanie trzecie . . . 81

7.4 Rozwiązanie czwarte . . . 82

7.5 Rozwiązanie piąte . . . 83

7.6 Rozwiązanie szóste . . . 85

7.7 Rozwiązanie siódme . . . 86

II Prolog

Ćwiczenia 89

8 Ćwiczenie 1 91

8.1 Zadanie 91

8.1.1 Odpowiedzi 91

8.2 Zadanie 93

8.2.1 Odpowiedzi 93

8.3 Zadanie 94

8.3.1 Odpowiedzi 94

8.4 Zadanie 95

8.4.1 Odpowiedzi do zadania 8.4 95

8.5 Zadanie 95

8.5.1 Odpowiedzi 95

9 Ćwiczenie 2 97

9.1 Zadanie 98

9.1.1 Odpowiedzi 98

9.1.2 Zadanie 99

9.1.3 Odpowiedzi 99

9.1.4 Zadanie 99

9.1.5 Odpowiedzi 99

9.1.6 Zadanie 100

9.1.7 Odpowiedzi 100

9.1.8 Zadanie 100

9.1.9 Odpowiedzi 100

9.1.10 Zadanie 100

9.1.11 Odpowiedzi 100

9.1.12 Zadanie 100

9.1.13 Odpowiedzi 100

10 Ćwiczenie 3 101

10.1 Zadanie 101

10.1.1 Odpowiedzi 101
10.2 Zadanie 102
10.2.1 Odpowiedzi 102
10.3 Zadanie 102
10.3.1 Odpowiedzi 102
10.4 Zadanie 102
10.4.1 Odpowiedzi 102
10.5 Zadanie 102
10.5.1 Odpowiedzi 103
10.6 Zadanie 103
10.6.1 Odpowiedzi 103
10.7 Zadanie 103
10.7.1 Odpowiedzi 103
10.8 Zadanie 103
10.8.1 Odpowiedzi 103
10.9 Zadanie 104
10.9.1 Odpowiedzi 104
10.10Zadanie 104
10.10.1Odpowiedzi 104
10.11Zadanie 104
10.11.1Odpowiedzi 104

11 Ćwiczenie 4 105

11.1 Zadanie 105
11.1.1 Odpowiedzi 105
11.2 Zadanie 105
11.2.1 Odpowiedzi 105
11.3 Zadanie 105
11.3.1 Odpowiedzi 106
11.4 Zadanie 106
11.4.1 Odpowiedzi 106
11.4.2 Odpowiedzi do zadania **??** 106

11.5 Zadanie 106
11.5.1 Odpowiedzi 107
11.6 Zadanie 107
11.6.1 Odpowiedzi 107
11.7 Zadanie 107
11.7.1 Odpowiedzi 107
11.7.2 Odpowiedzi do zadania ?? 107

12 Ćwiczenie 5 **109**
12.1 Zadanie 109
12.1.1 Odpowiedzi 109
12.2 Zadanie 109
12.2.1 Odpowiedzi 110

III Lisp **111**

13 Podstawy **113**
13.1 Dlaczego Lisp? 113
13.2 Charakterystyka języka 115
13.3 Wpływ na programowanie 116

14 Podstawy składni **119**
14.1 s-wyrażenia 119
14.1.1 Listy 119
14.1.2 Atomy – liczby 119
14.1.3 Atomy – napisy 121
14.1.4 Atomy – nazwy 121
14.2 Ewaluacja wyrażeń 123
14.2.1 Symbole 123
14.3 Lista – szczególne przypadki 123
14.3.1 Prawda, fałsz i równość 125
14.4 Formatowanie kodu programu 128

15 Funkcje 131
15.1 Definiowanie funkcji . . . 131
15.2 Parametry funkcji . . . 132
15.2.1 Parametry wymagane . . . 132
15.2.2 Parametry opcjonalne . . . 132
15.2.3 Dowolna ilość parametrów . . . 134
15.2.4 Parametry nazwane . . . 134
15.2.5 „Mieszanie” parametrów . . . 135
15.3 Zwracanie wartości . . . 137
15.4 Funkcja jako zmienna . . . 137
15.5 Funkcje anonimowe . . . 140

16 Zmienne 143
16.1 Podstawowe wiadomości . . . 143
16.2 Zmienne leksykalne . . . 148
16.3 Zmienne dynamiczne . . . 149
16.4 Stałe . . . 151
16.5 Przypisania . . . 151

17 Makra 155
17.1 `if` oraz `when` i `unless` . . . 155
17.2 `cond` . . . 156
17.3 `and`, `or`, `not` . . . 157
17.4 `dolist`, `dotimes` . . . 157
17.5 `do` . . . 158
17.6 `loop` . . . 159
17.7 Zrozumieć makra . . . 161
17.7.1 Rozwinięcie i wykonanie makra . . . 161
17.7.2 Etapy tworzenia makra . . . 162
17.7.3 Wzorzec podstawienia . . . 162
17.7.4 Problemy związane z makrami . . . 165

18 Listy 171

IV Lisp

Ćwiczenia **175**

V Haskell **179**

VI Haskell

Ćwiczenia **181**

19 Proste zadania **183**

VII Jess **185**

20 Wprowadzenie **187**

20.1 Czym jest silnik regułowy? 187
20.2 Jess – historia 187
20.3 Instalacja, sposoby wykorzystania, wydajność 187
20.4 Part I: Introducing rule-based systems 188
20.4.1 Typical unalgorithmic problem 188
20.5 What are rule-based systems? (część 1, rozdzial 2 (13) 189
20.5.1 Programowanie proceduralne (imperatywne) vs. deklaratywne 189
20.6 Architecture of a rule-based system (część 1, rozdzial 2.3 (s. 19) 190

21 Jess – wiadomości podstawowe **191**

21.1 Składnia – podstawowe elementy 191
21.2 Listy 192
21.3 Zmienne 192
21.4 Sterowanie przepływem 193
21.4.1 `foreach` 193
21.4.2 `while` 194
21.4.3 `if-then-else` 194
21.4.4 `progn` 194
21.4.5 `progn` 195
21.4.6 `apply` 195

21.4.7 `eval/build` . . . 195
21.5 Funkcje . . . 196
21.5.1 Definiowanie funkcji . . . 196
21.5.2 Zmiana zachowania funkcji . . . 196

22 Java w Jess **197**
22.1 Tworzenie obiektów . . . 197
22.2 Wywoływanie metod . . . 197
22.3 Tablice . . . 198
22.4 Dostep do publicznych składowych klasy . . . 198
22.5 Obsługa wyjątków . . . 199

23 Fakty **201**
23.1 czym jest fakt . . . 201
23.2 Manipulacje na przestrzeni roboczej . . . 201
23.3 Różne rodzaje faktów . . . 203
23.3.1 Luźne fakty . . . 203
23.3.2 Fakty uporządkowane . . . 205
23.3.3 Fakty zakryte . . . 205

24 Reguły **207**

25 Przykłady **211**
25.1 Rebus . . . 211
25.1.1 Rozwiązanie 1 . . . 212
25.1.2 Rozwiązanie 2 . . . 214
25.1.3 Rozwiązanie 3 . . . 216
25.1.4 Rozwiązanie 4 . . . 217
25.1.5 Rozwiązanie 5 . . . 219
25.2 Zawody . . . 221
25.3 Pytania i odpowiedzi . . . 222

VIIIDrools
Ćwiczenia 223

26 Symulacja zachowania botów 225

IX Podstawy logiki 229

27 Rachunek zdań 231
27.1 Postać normalna . . . 237
27.1.1 Przekształcanie do postaci normalnej . . . 238
27.2 Formuła Horna . . . 240
27.3 twierdzenie tutu . . . 241
27.4 Rezolucja . . . 241
27.5 Pytania i odpowiedzi. . . . 243

28 Rachunek kwantyfikatorów 245
28.1 Postać normalna . . . 251
28.1.1 Algorytm przekształcania do postaci klauzulowej . . . 253
28.2 Nierozstrzygalność . . . 255
28.3 Model Herbranda . . . 256
28.4 Rezolucja . . . 259
28.5 Unifikacja . . . 262
28.6 Ograniczenia na rezolucje . . . 264

Bibliografia 267

Spis rysunków 269

Spis tabel 270

Skorowidz 271

Przedmowa

Uważam, że wiedza o tym czym jest Prolog i Lisp (lub inne języki realizujące podobne paradygmaty) i umiejętność elementarnego posługiwania się tymi językami jest koniecznym elementem wykształcenia współczesnego *informatyka*. Doskonale zdaje też sobie sprawę, że otaczająca nas rzeczywistość mówi coś dokładnie innego. Obecnie na rynku pracy jest głównie zapotrzebowanie na programistów Javy, C++ i .NET a „na topie" są tzw. aplikacje webowe. Studia nie mają jednak na celu wykształcenia programistów/administratorów/sieciowców, ale przede wszystkim ludzi myślących, posiadających rozległą i uniwersalną wiedzę. Stąd istotna jest świadomość, że algorytm nie koniecznie musi być z mozołem zapisywany w jednym z imperatywnych języków prgramowania (np. C, Java, Pascal), ale często można to zrobić dużo prościej i szybciej posługując się deklaratywnym językiem programowania np. Prologiem (programowanie w logice) czy Lispem (programowanie funkcyjne*). Oczywiście nie mówimy tutaj o całkowitym zastąpieniu tych pierwszych języków drugimi.

Bardzo interesująca jest możliwość łączenia różnych języków ze sobą czy realizowanie różnych idei w tym samym języku. Osiągamy wówczas pewnego rodzaju elastyczność mając możliwość napisania różnych elementów aplikacji w najoptymalniejszy sposób†. Odbicie tego sposobu myślenia odnajdujemy w najnowszych tendencjach

- C++
- Jess
- Drools

Moim zdaniem żaden z języków deklaratywnych nie nadaje się do stworzenia kompletnej aplikacji biznesowej‡. To jednak nie powinno dziwić, gdyż nie do tego zostały one zaprojektowane. Niestety

*Nazywane także funkcjonalnym.

†Nie oszukujmy się: nie ma jednego uniwerslanego języka programowania.

‡Nie oznacza to, że nie można. Jak bardzo się uprzemy to jest to wykonalne (np. system operacyjny [23]: *This project aims to develop a Lisp based operating system for general purpose computer architectures.*). Pytanie tylko, po co?

także próby posiłkowania się nimi w „normalnej aplikacji" często są nieskuteczne. Teoretycznie jest to możliwe, ale w praktyce okazuje się, że jest odwrotnie, bo „coś" nie działa.

Do czego więc są? Mówiąc ogólnie, są to języki szczególnie użyteczne wszędzie tam, gdzie stworzenie algorytmu jest trudne. Znany jest problem, znane są warunki jakie muszą być spełnione i pewne zależności a mimo to algorytmu albo nie potrafimy ułożyć, albo jest on strasznie „toporny". Do klasy zagadnień nie poddających się łatwemu opisowi algorytmicznemu, takich w których wiemy co chcemy zrobić, ale zupełnie nie mamy pojęcia jak, należy m.in. (szeroko rozumiana) sztuczna inteligencja.

Niniejszy podręcznik jest opisem narzędzi jakie możemy wykorzystać rozwiązując problemy z zakresu sztucznej inteligencji (ale nie tylko). Ponieważ narzędziem w tym przypadku jest język programowania, więc stąd tytuł: *języki programowania sztucznej inteligencji*. Zasadniczym celem jaki przed sobą stawiamy jest zmiana sposobu myślenia o problemie oraz umiejętność jego rozwiązania w inny niż algorytmiczny sposób. Służyć temu mają części I-V, w których prezentujemy kolejno Prolog, Lisp oraz Haskell. Wspomniane języki w swej pierwotnej postaci są już dosyć wiekowe i słabo przystają do obecnej rzeczywistości. Dlatego w kolejnych rozdziałach przedstawimy współczesne rozwiązania bazujące na koncepcjach obecnych we wspomnianych językach, czyli tzw. *silniki regułowe*. Część VII omawia „mały" silnik regułowy Jess, natomiast część VIII poświęcona jest systemowi zarządzania regułami biznesowymi (ang. *business rule management system – BRMS*) Drools. W żadnym razie nie jest to opracowanie aspirujące do roli „manuala" w omawianych zagadnieniach – osoby zainteresowane np. dokładnym poznaniem Lispa czy Drools odsyłamy do publikacji im poświęconych. Tym bardziej, że choćby samo Drools nie jest już żadną „akademicką zabawką" ale niezwykle rozbudowanym środowiskiem do tworzenia poważnych aplikacji biznesowych.

Piotr Fulmański

Łódź, 2010

Część I

Prolog
Wykład

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. Colmerauer badał możliwość przetwarzania* języka naturalnego, którego semantyka reprezentowana miała być za pomocą wyrażeń logicznych, natomiast narzędziem pozwalającym na wnioskowanie miała być rezolucja. Z tego powodu teoretyczne podstawy Prologa stanowi rachunek predykatów pierwszego rzędu (ale ograniczony tylko do klauzul Horna).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 części VII i VIII), to wciąż stanowi wspaniały model dydaktyczny.

*Przetwarzanie w sensie *rozumienie*.

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 algorytm, 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 takich jak np. 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 otaczają nas w rzeczywistym świecie posiadają dwie istotne cechy: **stan** w jakim w danej chwili się znajdują[†] 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...)

[†] Lub **cechy** jakie posiadają.

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);
Wektor w(2.5,-1.5);

p.Przesun(w);
```

Podsumowując:

- Programowanie zorientowane obiektowo (*ang. object-oriented programming – OOP*) to para-

dygmat (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 i polimorfizmu.

Wiedząc czym są, najlepiej chyba wszystkim znane, obiekty z języków zorientowanych obiektowo 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 tej cechy jak i żadnej innej.

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**[‡]

```
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

[‡]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.

```
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*:

```
?-  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[§] 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

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

musimy bardzo uważnie się zastanowić na temat tego

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

Wybór formy relacji (reguły) powinien być wystarczająco precyzyjny aby mógł być potraktowany jak **definicja** w problemie jaki rozwiązujemy. Musimy przy tym zdawać sobie jasno sprawę z tego, że z punktu widzenia języka, obiekty nie są rozróżnialne semantycznie. Oznacza to, że obiekt `slon` i `slonia` w poniższych faktach

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

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

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[¶] 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).
  ```

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

- 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>.
  ```

  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 aplika-

cji, gdyż programy Prologu z natury są raczej interakcyjne. Bardziej adekwatnym określeniem zamiast *uruchamianie* wydaje się być *formułowanie zapytań* lub też *interakcyjny tryb zapytanie–odpowiedź*[‖].

Zapisany program wczytujemy poleceniem (znak `?-` jest tzw. znakiem zachęty wyświetlanym przez interpreter)

```
?- [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 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 formułować jako

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

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

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

Mając 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*[**]*. 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*[††] *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*[‡‡]*: podaje się zbiór predykatów spełnianych przez definiowane obiekty.*

Uwaga: *Obiekty mogą pozostawać we wzajemnych zależnościach wyrażanaych 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.*

[**] 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.

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

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

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) .

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.*

Ważna jest także przynależność 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 utworzone zdanie jest nie tyle nieprawdziwe co pozbawione sensu.

ROZDZIAŁ 2

Składnia Prologa

2.1 Termy

Program Prologa 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**[*]. 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

[*]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).

- 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 :- .

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 Marcin 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

Fakt (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 właściwe podstawienie 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_rozniczkowa).
```

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_rozniczkowa ;

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 przykałd 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 okre-*

ślenie 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*[†].
- *Banan jest żółty.*
- *Samochód Ferrari jest żółty*[‡].

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

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

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

[‡]Zdaniem miłośników marki, prawdziwe Ferrari powinno być pomalowane na czerwono. Wzięło się to ze zwyczaju założyciela, który malował swoje samochody wyścigowe na kolor 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.

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
```

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 występują 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* 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†.
   b) Znajdź fakt lub regułę unifikującą się z termem‡. 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.

*Wewnątrz (w ciele) reguły, czyli po prawej stronie operatora `:-`

†Zwykle termy wybierane są od lewej do prawej.

‡Zwykle fakty i reguły przeszukiwane są w kolejności ich występowania w pliku.

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

```
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 = b    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 term `a(b,Z)` i w wyniku unifikacji z faktem `a(b,c)` otrzymujemy podstawienie

```
Z=c
```

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

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

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

```
Y=d
```

Rezolucja: ponieważ unifikacja dotyczyła faktu, więc rozpatrywany term 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.

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

```
1 mniej(p1,p2).
2 mniej(p2,p3).
3 mniej(p3,p4).
4
5 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 co się stało z parą (`A=p2, B=p4`)? Wytłumaczenie tego jest następujące (równolegle z opisem proszę śledzić drzewo wywołań, bo choć nie jest pozbawione wad, może ułatwić kontrolowanie co i kiedy jest wywoływane).

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 (5) (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 fakt 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 (linia 5 programu, 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 5 (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 fakt 3 (linia 17 w d.w.). Zatem wiemy już, że `Z` ma wartość `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 (5) (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 (5) (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 (5) (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, kiedy to 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 częsciowe przekonstruowanie reguły/faktów, np. w następujący sposób

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

```
3 mniej(p3,p4).
4
5 jestMniejszy(X,Y) :- mniej(X,Y).
6 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*[§]*.*

[§]Różnie nie w sensie „odmiennie", ale w sensie „specyficznie".

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***[¶]*, 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 $\eta = \{t_1|x_1, ..., t_n|x_n\}$*[‖] *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*

$$E_1 = x^2,$$

$$E_2 = y^3$$

[¶]Czyli najpierw wartość węzła a potem jego dzieci.

[‖]Zapis $t_i|x_i$ należy czytać: wyrażenie t_i podstawione za zmienną x_i.

unifikatorem jest

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

Prolog *W Prologu unifikacja oznacza proces, w którym dla dwóch atomów (jednego z zapytania, drugiego będącego faktem lub głową reguły*[**]*) 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

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

```
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
```

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* **??***).*

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,[])
```

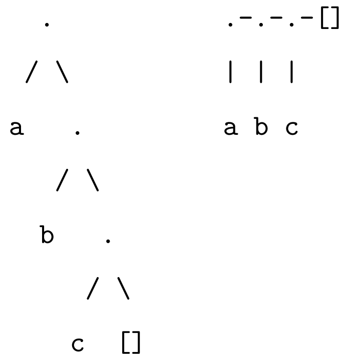
co na rysunku można przedstawić w następujący sposób

```
  .            .-[]
 / \           |
a   []         a
```

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* 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,
```

*Używany także do konstruowania listy.

```
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

- 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 Prologa. 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 nawracania

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, albo mówiąc inaczej takie same rozwiązania do których dochodzimy na dwa różne sposoby.

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 . . .

```
        a(X).
        |
1       +-a(0).
        |
2       +-a(X):-a(Y),X is Y+1.---------+
                |                      |(2.2.2)
2.1             +-a(0).-+              +-X=1+1--+-X=2+1--+-X=3+1
```

```
               |       |              .        .        .
2.1.1          |       +-X=0+1.       .        .        .
               |                      .        .        .
2.2            +-a(X):-a(Y),X is Y+1.-+--------+--------+
                       |              .        |        |
2.2.1                  +-a(0).-+      .        +-X=1+1  +-X=2+1
                       |       |      .        .        .
2.2.1.1                |       +------X=0+1    .        .
                       |                       .        .
2.2.3                  +-a(X):-a(Y),X is Y+1.--+--------+
                               |               .        |
                               +-a(0).-+       .        +-X=1+1
                               |       |       .        .
                               |       +-------X=0+1.   .
                               |                        .
                               +-a(X):-a(Y),X is Y+1.---+------
                                 |                      .
                                 +-a(0).-+              .
                                 |       |              .
                                 |       +--------------X=0+1.
                                 |
                                 +-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 przykaładzie.*** *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

Powtarzamy wiadomości

6.1 O rekurencji raz jeszcze

W ćwiczeniu 8 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.

6.1.1 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*.

6.1.2 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.

6.1.3 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).

6.1.4 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 kolejnościa celów w regule. Z przyczyn opisanych powyżej, taki sposób zapisu nie jest zalecany ze względu na możliwe obniżenie wydajności.

6.2 Akumulator

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

6.2.1 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 dokładniej co 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([b,c],LenTail), (10) Len is LenTail + 1
                              (9) LenTail = 2
      len([_|Tail],Len)                                      1    + 1
  (2) len([b,c]   ,LenTail) :- len([c],LenTail), (8) Len is LenTail + 1
                                   (7) LenTail = 1
      len([_|Tail],Len)                                      0    + 1
  (3) len([c]     ,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).

6.2.2 Przykład z liczbami

W ćwiczeniach z rozdziału 9 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)))))
```

6.3 Z góry na dół czy od dołu do góry?

6.3.1 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.

6.3.2 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. Liniową złożoność ma także rozwiązanie 8.5.1 z zadania 8.5

rozdziału 8, choć tam nie jest użyty akumulator.

6.4 Pytania i odpowiedzi

Pytanie 6.1. *Co to jest akumulator? Podaj przykład*[*]. *Akumulatorem nazywamy mechanizm wykonywania właściwych obliczeń podczas schodzenia w dół rekurencji zamiast wykonywania tych obliczeń podczas powrotu jak to ma miejsce w zwykłej bezakumulatorowej rekurencji. Rozważmy przykład programu obliczającego sumę wszystkich liczb podanych na liście.*

Wersja bez akumulatora

```
suma([], 0).
suma([H|T], R):-suma(T,R1), R is R1 + H.
```

Dla wywołania

```
suma([1,2,3],0,R).
```

wykonanie przebiega w następujący sposób.

1. *Nastąpi wywołanie reguły* `suma` *z ogonem kolejki i inną resztą czyli* `suma([2,3],R1)`*, przy czym* `H`*(ead) na tym poziomie ma wartość 1.*

2. *Nastąpi wywołanie reguły* `suma` *z ogonem kolejki i inną resztą czyli* `suma([3],R1)`*, przy czym* `H`*(ead) na tym poziomie ma wartość 2.*

3. *Nastąpi wywołanie reguły* `suma` *z ogonem kolejki i inną resztą czyli* `suma([],R1)`*, przy czym* `H`*(ead) na tym poziomie ma wartość 3.*

4. *W tym momencie wywołanie dopasowywuje się do faktu* `suma([], 0)`*, czyli doszliśmy na sam dół rekurencji i na poziom wyżej zwracamy 0.*

5. *Znajdujemy się teraz na poziomie kroku 3, gdzie* `H = 3`*. Ponieważ wywołanie sumy z ogonem zakończyło się sukcesem, przechodzimy do drugiego wyrażenia w regule czyli do wyniku* `R` *(przekazanego z poprzedniego kroku) dodajemy aktualne* `H` *czyli* `R = 0 + 3 = 3`*. Następnie zwracamy ten wynik poziom wyżej.*

[*] Odpowiedź p. Adama Domagalskiego.

6. *Znajdujemy się teraz na poziomie kroku 2, gdzie* `H = 2`. *Ponieważ wywołanie sumy z ogonem zakończyło się sukcesem, przechodzimy do drugiego wyrażenia w regule czyli do wyniku* `R` *(przekazanego z poprzedniego kroku) dodajemy aktualne* `H` *czyli* `R = 3 + 2 = 5`. *Następnie zwracamy ten wynik poziom wyżej.*

7. *Znajdujemy się teraz na poziomie kroku 1, gdzie* `H = 1`. *Ponieważ wywołanie sumy z ogonem zakończyło się sukcesem, przechodzimy do drugiego wyrażenia w regule czyli do wyniku* `R` *(przekazanego z poprzedniego kroku) dodajemy aktualne* `H` *czyli* `R = 5 + 1 = 6` *co jest pożądanym wynikiem .*

Wersja z akumulatorem

```
sumaAku([], R, R).
sumaAku([ H| T ], A, R) :- A1 is A + H,  sumaAku(T, A1, R).
```

Dla wywołania

`sumaAku([1,2,3],0,R).`

wykonanie przebiega w następujący sposób.

1. *Do akumulatora* `A` *dodajemy wartość z głowy kolejki:* `0 + 1 = 1`, *czyli* `A = 1`, *po czym następuje wywołanie z ogonem kolejki (czyli* `[2,3]`*)* `sumaAku([2,3],1,R).`

2. *Do* `A` *dodajemy aktualną głowę kolejki (czyli* `2`*)* `A = 1 + 2 = 3`, *po czym następuje wywołanie z ogonem kolejki (czyli* `[3]`*)* `sumaAku([3],3,R).`

3. *Do* `A` *dodajemy aktualną głowę kolejki (czyli* `3`*)* `A = 3 + 3 = 6`, *po czym następuje wywołanie z ogonem kolejki (czyli* `[]`*)* `sumaAku([3],6,R).`

4. *Ponieważ mamy zdefiniowany fakt dla wywołania sumy z pustą kolejką (*`sumaAku([], R, R).`*) nasze aktualne wywołanie mające postać* `sumaAku([],6,R)` *zwróci nam wartość* `6` *co jest pożądanym wynikiem.*

Jak widać zastosowanie akumulatora znacząco skraca czas działania, jak również upraszcza program.

Pytanie 6.2. ***Obliczenia „z góry na dół" czy „z dołu do góry"? Dlaczego?***[†] *Jeśli nie ma możliwości zastosowania podejścia akumulatorowego stosujemy metodę „z góry na dół", gdyż*

[†]Odpowiedź p. Adama Domagalskiego.

zazwyczaj jest ona szybsza i bezpieczniejsza niż zwykła „z dołu do góry”. Jeśli jednak możemy zastosować akumulatory, stosujemy metodę „z dołu do góry”, gdyż akumulator pozwoli nam na rozwiązywanie problemu w ramach schodzenia w dół rekurencji przez co pozwoli na pominięcie etapu „scalania” wyniku.

ROZDZIAŁ 7

Od problemu do jego (efektywnego) rozwiązania

7.1 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 zalżny od posiadanego systemu. Dlatego będziemy posługiwać się pewnymi abstrakcyjnymi jednostkami po-

zwalającymi oddać zależności procentowe co pozwoli ocenić zysk stosowanych rozwiązań niezależny od sprzętu. Dlatego przyjmujemy, że czas wykonania pierwszej wersji programu wynosi 100%.

7.2 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`

$$\begin{aligned} T &= 2D && \text{jeżeli} \quad D < 5, \\ T &= 2D - 10 && \text{jeżeli} \quad D > 4. \end{aligned} \tag{7.1}$$

Uwzględnienie tak prostego warunku, poprzez wprowadzenie niewielkiej modyfikacji do programu pozwala na zmniejszenie czasu poszukiwania rozwiązania do wartości 11.38s co stanowi 50.98% pierwszego rozwiązania.

```
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.

7.3 Rozwiązanie trzecie

Warunek (7.1) 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 9.15s (40.00%).

7.4 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 (7.1) 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

$$\underbrace{L + L + 1}_{\text{suma powiększona o 1}} = R$$

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.

7.5 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).
```

7.6 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).

7.7 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.

Część II

Prolog

Ćwiczenia

ROZDZIAŁ 8

Ćwiczenie 1

Zapoznajemy się ze składnią Prologa. Piszemy proste fakty i reguły.

8.1 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.

8.1.1 Odpowiedzi

```
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: ) .
?-
```

8.2 Zadanie

Utworzyć plik zawierający reguły opisujące następujące zdania o świecie:

1. Każdy człowiek je tylko mięso lub ser lub owoce.
2. Jeśli X jest babcią Y, to Y jest wnukiem lub wnuczką.
3. Dwie osoby są rodzeństwem, jeśli mają tych samych rodziców.
4. X jest figurą jeśli jest trojkatem lub kwadratem lub okręgiem.

8.2.1 Odpowiedzi

- Każdy człowiek je tylko mięso lub ser lub owoce.

  Raczej nie w ten sposób

  ```
  mieso(schab).
  ser(gouda).
  owoce(banan).
  je(czlowiek,X) :- mieso(X); ser(X); owoce(X).
  ```

  Poniższe jest lepszym rozwiązaniem

  ```
  jedzenie(mieso).
  jedzenie(ser).
  jedzenie(owoce).
  je(czlowiek,X) :- jedzenie(X).
  ```

- Jeśli X jest babcią Y, to Y jest wnukiem lub wnuczką X.

  ```
  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).
```

8.3 Zadanie

Napisać bardziej rozbudowaną wersję programu z zadania podpunkt c): użyć inne reguły, np. `jest_matka(X,Y)`, `kobieta(X)`, `rodzic(X,Y)`.

8.3.1 Odpowiedzi

```
jestMezczyzna(piotr).
jestKobieta(anna).
jestDzieckiem(anna,piotr).
jestDzieckiem(marek,piotr).


jestRodzicem(X,Y):-jestDzieckiem(Y,X).
jestOjcem(X,Y):-jestMezczyzna(X),jestDzieckiem(Y,X).
jestCorka(X,Y):-jestKobieta(X),jestRodzicem(Y,X).
jestMatka(X,Y):-jestKobieta(X),jestDzieckiem(Y,X).
jestSynem(X,Y):-jestMezczyzna(X),jestRodzicem(Y,X).
saRodzenstwem(X,Y):-jestDzieckiem(X,Z),jestDzieckiem(Y,Z).
```

8.4 Zadanie

Napisać program obliczający silnię.

8.4.1 Odpowiedzi do zadania 8.4

```
silnia(0,F) :- F is 1.

silnia(N1,F1) :- N1>0,
                 N2 is N1-1,
                 silnia(N2,F2),
                 F1 is N1*F2.
```

8.5 Zadanie

Napisać program obliczający n-ty wyraz ciągu Fibonacciego.

8.5.1 Odpowiedzi

- Rozwiązanie 1.

```
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).
```

- Rozwiązanie 2.

```
fib(0,1).
fib(1,1).

fib(N,F):- N>1,
           N1 is N-1,
           N2 is N-2,
```

```
        fib(N1,F1),
        fib(N2,F2),
        F is (F1 + F2).
```

- Rozwiązanie 3.

```
fib(0, 1, 0).
fib(1, 1, 1).

fib(X, A, B):-X1 is X-1, fib(X1, B, C), A is B+C.
```

ROZDZIAŁ 9

Ćwiczenie 2

W ćwiczeniu tym praktykować będziemy posługiwanie się termami złożonymi.

Giuseppe Peano zaproponował następujące warunki (tzw. postulaty lub aksjomaty Peano), które musi spełniać dowolna konstrukcja zbioru liczb naturalnych

1. 0 jest liczbą naturalną.
2. Każda liczba naturalna a ma swój następnik, oznaczany $f(a)$.
3. 0 nie jest następnikiem żadnej liczby naturalnej.
4. Różne liczby naturalne mają różne następniki, tzn. $a \neq b \Rightarrow f(a) \neq f(b)$.
5. Jeśli zero ma daną własność i następnik dowolnej liczby naturalnej o tej własności również ma tę własność, to każda liczba naturalna ma tę własność (zasada indukcji matematycznej).

Kolejne $n + 1$ liczb naturalnych to

- $a_0 = 0$,
- $a_1 = 1 = 0 + 1$,
- $a_2 = 2 = 1 + 1$,
- $a_3 = 3 = 2 + 1$,
- $a_4 = 4 = 3 + 1$,
- $\dots$,

- $a_n = a_{n-1} + 1$

Zgodnie z podaną aksjomatyką liczby te, przyjmując, że mamy funkcję następnika $f(\cdot)$ dającą element następny, oraz symbol początkowy 0, możemy 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).
```

Działanie

```
?- lN(f(f(f(0)))). % 3 jest liczba naturalna?

Yes
?-
```

Przy tak przyjętej definicji liczby naturalnej możemy wprowadzić np. relację większe lub równe (ang. gEq – *greater or equal*).

```
gEq(X, 0) :- lN(X).
gEq(f(X), f(Y)) :- gEq(X, Y).
```

9.1 Zadanie

Proszę napisać relację większy.

9.1.1 Odpowiedzi

```
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).
```

9.1.2 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.

9.1.3 Odpowiedzi

```
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
?-
```

9.1.4 Zadanie

Prosze napisać relację mnożenia (pamiętajmym, że mnożenie to wielokrotne dodawanie).

9.1.5 Odpowiedzi

```
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
?-
```

9.1.6 Zadanie

Prosze zapisać potęgowanie (pamiętajmy, że potęgowanie to wielokrotne mnożenie).

9.1.7 Odpowiedzi

```
pow(f(X), 0, 0).
pow(0, f(X), f(0)).
pow(f(N), X, Y) :- pow(N, X, Z), mul(Z, X, Y).
```

9.1.8 Zadanie

Proszę napisać program przeliczający zapis w postaci ciągu następników na liczbę naturalną.

9.1.9 Odpowiedzi

```
value(0,0).
value(f(X),Wartosc) :- value(X,WartoscTmp),Wartosc is WartoscTmp+1.
```

9.1.10 Zadanie

Proszę napisać program zamieniający liczbę naturalną na ciąg następników.

9.1.11 Odpowiedzi

```
repr(0,0).
repr(N,f(X)) :- N>0, N1 is N-1, repr(N1,X).
```

9.1.12 Zadanie

Napisz program mnożący dwie liczby naturalne podane jako ciąg następników, który będzie zwracał konkretną wartość liczbową (czyli nie ciąg następników).

9.1.13 Odpowiedzi

```
mulVal(X,Y,V) :- mul(X,Y,Z), value(Z,V).
```

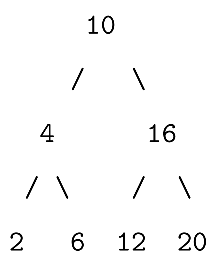
ROZDZIAŁ 10

Ćwiczenie 3

W ćwiczeniu tym w dalszym ciągu doskonalimy posługiwanie się termami złożonymi – w tym przypadku tematy zadań związane są z drzewem binarnym.

10.1 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`.

10.1.1 Odpowiedzi

```
(10,(4,(2,pusty,pusty),(6,pusty,pusty)),(16,(12,pusty,pusty),(20,pusty,pusty)))
```

10.2 Zadanie

Napisać predykat sprawdzajacy, czy przekazany argument jest drzewem binarnym (w sensie takiego drzewa, w którym każdy węzeł ma maksymalnie dwoje dzieci).

10.2.1 Odpowiedzi

```
drzewoB(pusty).
drzewoB((_,Lewe,Prawe)) :- drzewoB(Lewe), drzewoB(Prawe).
```

10.3 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).

10.3.1 Odpowiedzi

```
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.
```

10.4 Zadanie

Napisać predykat sprawdzajacy, czy dany element należy do drzewa.

10.4.1 Odpowiedzi

```
elementB(X,(X,_,_)).
elementB(X,(Y,Lewe,Prawe)) :- \+ (X=Y), (elementB(X,Lewe);elementB(X,Prawe)).
```

10.5 Zadanie

Napisać predykat obliczający sumę elementów znajdujących się w węzłach drzewa binarnego.

10.5.1 Odpowiedzi

```
suma(pusty,0).
suma((Element,Lewe,Prawe),Suma) :-
        suma(Lewe,SumaL),
        suma(Prawe,SumaP),
        Suma is SumaL+SumaP+Element.
```

10.6 Zadanie

Napisać predykat zliczający ilość węzłów w drzewie binarnym

10.6.1 Odpowiedzi

```
ileWezlow(pusty,0).
ileWezlow((_,Lewe,Prawe),Ile) :-
        ileWezlow(Lewe,IleL),
        ileWezlow(Prawe,IleP),
        Ile is IleL+IleP+1.
```

10.7 Zadanie

Napisać predykat przechodzący po drzewie w porządku preorder (tj. węzeł, lewy podwęzeł, prawy podwęzeł).

10.7.1 Odpowiedzi

```
preorder((X,L,P),Xs) :- preorder(L,Ls), preorder(P,Ps), append([X|Ls],Ps,Xs).
preorder(pusty,[]).
```

10.8 Zadanie

Napisać predykat przechodzący po drzewie w porządku inorder (tj. lewy podwęzeł, węzeł, prawy podwęzeł).

10.8.1 Odpowiedzi

```
inorder((X,L,P),Xs) :- inorder(L,Ls), inorder(P,Ps), append(Ls,[X|Ps],Xs).
inorder(pusty,[]).
```

10.9 Zadanie

Napisać predykat przechodzący po drzewie w porządku postorder (tj. lewy podwęzeł, prawy podwęzeł, węzeł).

10.9.1 Odpowiedzi

```
postorder((X,L,P),Xs) :- postorder(L,Ls),postorder(P,Ps),
                         append(Ps,[X],Ps1),append(Ls,Ps1,Xs).
postorder(pusty,[]).
```

10.10 Zadanie

Napisać predykat zwracający element maksymalny z drzewa binarnego.

10.10.1 Odpowiedzi

```
max(pusty, fail).
max((Element,_,pusty),Element).
max((_,_,Prawe),Max) :- \+ (Prawe=pusty), max(Prawe,Max).
```

10.11 Zadanie

Napisać predykat zliczający ilość liści w drzewie.

10.11.1 Odpowiedzi

```
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.
```

ROZDZIAŁ 11

Ćwiczenie 4

11.1 Zadanie

Napisać predykat zliczający ilość elementów na liście.

11.1.1 Odpowiedzi

```
zlicz([],0).
zlicz([H|T],X):- zlicz(T,X1), X is X1+1.
```

11.2 Zadanie

Napisać predykat zliczający ilość wszystkich elementów na liście włącznie z elementami podlist.

11.2.1 Odpowiedzi

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

zliczAll([],0).
zliczAll(H,1) :- \+lista(H).
zliczAll([H|T],X):- zliczAll(H,X1), zliczAll(T,X2), X is X1+X2.
```

11.3 Zadanie

Napisać predykat usuwający pojedyńcze wystąpienie elementu na liście.

11.3.1 Odpowiedzi

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

Wynik działania

```
?- usun(1,[1,2,3,1,2,3,1,2,3],R).
R = [2, 3, 1, 2, 3, 1, 2, 3] ;
R = [1, 2, 3, 2, 3, 1, 2, 3] ;
R = [1, 2, 3, 1, 2, 3, 2, 3] ;
false.
```

11.4 Zadanie

Napisać predykat usuwający wszystkie wystąpienia elementu na liście.

11.4.1 Odpowiedzi

```
usun(X,[],[]).
usun(X,[X|Ys],Zs) :- usun(X,Ys,Zs).
usun(X,[Y|Ys],[Y|Zs]) :- \+ X=Y, usun(X,Ys,Zs).
```

Wynik działania

```
?- usun(1,[1,2,3,1,2,3,1,2,3],R).
R = [2, 3, 2, 3, 2, 3] ;
false.
```

11.4.2 Odpowiedzi do zadania ??

```
usunAll(_,[],[]).
usunAll(X,[X|T],Z) :- !, usunAll(X,T,Z).
usunAll(X,[Y|T1],[Y|T2]) :- usunAll(X,T1,T2).
```

11.5 Zadanie

Napisać predykat odwracający listę.

11.5.1 Odpowiedzi

```
odwroc([],[]).
odwroc([X|Xs],Zs) :- odwroc(Xs,Ys), polacz(Ys,[X],Zs).
```

11.6 Zadanie

Napisać predykat zwracający ostatni element listy.

11.6.1 Odpowiedzi

```
ostatni([X],X).
ostatni([_|T],X) :- ostatni(T,X).
```

11.7 Zadanie

Napisać predykat zastępujący wszystkie wystąpienia elementu innym elementem.

11.7.1 Odpowiedzi

```
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).
```

11.7.2 Odpowiedzi do zadania ??

```
%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).
```

ROZDZIAŁ 12

Ćwiczenie 5

12.1 Zadanie

Napisać predykat sprawdzający czy dwa podane elementy sąsiadują ze sobą na liście.

12.1.1 Odpowiedzi

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

```
%sprawdza czy x jest siasiadem y
glowa([H|_],H).
sasiad(X,Y,[X|T]):-glowa(T,X1),X1=Y.
sasiad(X,Y,[H|T]):-X\=H,sasiad(X,Y,T).
```

```
sasiad2(X,Y,[X,Y|_]).
sasiad2(X,Y,[H|T]):-X\=H,sasiad2(X,Y,T).
```

12.2 Zadanie

Napisać predykat usuwający z listy wszystkie powtórzenia.

12.2.1 Odpowiedzi

```
%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).
```

```
usun(X,[] ,[]).
usun(X,[X|Ys],Zs):- usun(X,Ys,Zs).
usun(X,[Y|Ys],[Y|Zs]):- \+ X=Y, usun(X,Ys,Zs).
skroc([],[]).
skroc([H|T], [H|Tmp2]):-usun(H,T,Tp),skroc(Tp,Tmp2).
```

Część III

Lisp

ROZDZIAŁ 13

Podstawy

13.1 Dlaczego Lisp?

Nazwa *Lisp*, podobnie jak *Prolog*, zawiera w sobie informacje o zasadniczym przeznaczeniu języka. W tym przypadku nazwa pochodzi od angielskich wyrazów **lis***t* **p***rocessing* co na język polski możemy przetłumaczyć jako *prztwarzanie list*. Lista miała być narzędziem pozwalającym na efektywne przetwarzanie danych symbolicznych*. I jak niebawem zobaczymy, faktycznie przetwarzanie list to jest to, w czym Lisp jest całkiem niezły.

Lisp jest drugim z kolei pod względem wieku językiem programowania wysokiego poziomu pozostającym w użyciu (starszy jest tylko Fortran†). Choć pierwsza oficjalna wzmianka o Lispie pochodzi z 4 marca 1959 roku ([9]), to zwykle narodziny tego języka umiejscawia się w roku 1958‡. W roku 1960 John McCarthy opublikował swój projekt w Communications of the ACM, w artykule pod tytułem *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* ([10]) (*Rekursywne funkcje wyrażeń symbolicznych i ich maszynowe obliczanie, część I*; części II nigdy nie opublikowano), w którym pokazał, że za pomocą kilku operatorów i notacji dla funkcji można otrzymać kompletny w sensie Turinga język programowania (ang. *Turing-complete language*)§.

* *The LISP language is designed primarily for symbolic data processing.*[11]

† Robocza wersja specyfikacji *The IBM Mathematical Formula Translating System* pochodzi z roku 1954. Pierwszy tzw. manual ukazał się w październiku 1956, a pierwszy kompilator w kwietniu 1957 (źródło: `http://en.wikipedia.org/wiki/Fortran`, dostęp: 2010-02-15).

‡ W [17] znajdujemy informację, że Lisp został stworzony około roku 1956.

§ W teorii obliczalności o maszynie lub języku programowania mówimy, że jest kompletny w sensie Turinga lub zupełny, jeśli można za jego pomocą rozwiązać identyczną klasę problemów obliczeniowych, jak na uproszczonym modelu programowalnego komputera zwanego maszyną Turinga. W praktyce oznacza to, że jeśli dany język lub maszyna potrafi wykonać lub wyrazić każdy algorytm, określany jest mianem zupełnego, przy czym nie jest wymagane, by algorytm ten realizowany był prosto, wydajnie bądź efektywnie.

Pierwszą implementację Lispu opracował Steve Russel na komputerze IBM 704.

Pierwszy kompletny kompilator Lispu stworzony w Lispie napisali w 1962 Tim Hart i Mike Levin na MIT[14].

W dalszej części będziemy zajmować się Common Lispem, czyli współczesnym „potomkiem" Lispa. Co ciekawe, Common Lisp nie jest konkretną implementacją, ale specyfikacją stworzoną przez ANSI¶ mającą na celu ujednolicenie powstałych do tego czasu implementacji Lispa. Obecnie dostepnych jest kilka implementacji Common Lisp, zarówno zamkniętych, jak i dostępnych jako FOSS (ang. *free and open source software*).

Powiedzmy jeszcze kilka zdań o twórcy języka a stanie się jasne, dlaczego bedziemy mówić o Lispie. Urodzony w roku 1927 John McCarthy jest pionierem sztucznje inteligencji. Za swój wkład w rozwój tej dyscypliny otrzymał w 1971 roku Nagrodę Turinga. W roku 2003 otrzymał medal Instytutu Benjamina Franklina z informatyki i nauk poznawczych (Benjamin Franklin Medal in Computer and Cognitive Science). To własnie on po raz pierwszy użył terminu *sztuczna inteligencja*, który sformułował w 1956 r. na konferencji w Dartmouth.

Zakończmy ten podrozdział kilkoma wypowiedziami osób, których nie sposób uznać za informatycznych laików.

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

Eric Steven Raymond, [16]

Historically, languages designed for other people to use have been bad: Cobol, PL/I, Pascal, Ada, C++. The good languages have been those that were designed for their own creators: C, Perl, Smalltalk, Lisp.

Paul Graham, [20]

Anyone could learn Lisp in one day, except that if they already knew Fortran, it would take three days.

Marvin Minsky, [20]

¶ANSI INCITS 226-1994 (R2004), (formalnie X3.226-1994 (R1999)).

13.2 Charakterystyka języka

Podamy teraz bardzo krótką charkaterystykę Lispa aby każdy, już teraz, mógł bez trudu wskazać cechy odróżniające go od innych języków. O Lispie mówimy, że jest **językiem funkcjonalnym** lub **językiem programowania funkcyjnego**. Programowanie funkcyjne (lub programowanie funkcjonalne) to paradygmat programowania będący odmianą **programowania deklaratywnego**, w którym podstawowym elementem języka jest funkcja, a nacisk kładzie się na wartościowanie (często rekurencyjnych) funkcji, a nie na wykonywanie poleceń. Podstawą teoretyczną programowania funkcyjnego jest opracowany w latach 30. XX wieku przez Alonzo Churcha *rachunek lambda*, a dokładnie rachunek lambda z typami.

Lisp jest językiem, którego składnia opiera się na wyrażeniach (ang. *expression-oriented language*). W przeciwieństwie do większości innych języków, nie ma w nim podziału na wyrażenia (ang. *expressions*) i instrukcje (ang. *statements*). Wynikiem ewaluacji (wartościowania) wyrażeń jest wartość (lub lista wartości), która może być użyta jako argument dla innego wyrażenia. Pierwotnie John McCarthy wprowadził dwa typy wyrażeń:

- S-wyrażenia (ang. *S-expressions*), czyli wyrażenia symboliczne (ang. *Symbolic Expressions*) nazywane także sexpami (ang. *sexps*). Odzwierciedlały one wewnętrzną reprezentację kodu i danych.
- M-wyrażenia (ang. *M-expressions*), czyli meta wyrażenia (ang. *Meta Expressions*). Opisywały one funkcje S-wyrażeń.

I choć w założeniach to M-wyrażenia miały tworzyć składnię Lispa, to okazało się, że S-wyrażenia zdobyły bardzo dużą popularność. Co ciekawe, i dla wielu przeciwników tego języka szokujące, uznawane one były za wygodniejsze od składni Fortrana czy Algola

Another reason for the initial acceptance of awkwardnesses in the internal form of LISP is that we still expected to switch to writing programs as M-expressions. The project of defining M-expressions precisely and compiling them or at least translating them into S-expressions was neither finalized nor explicitly abandoned. It just receded into the indefinite future, and a new generation of programmers appeared who preferred internal notation to any FORTRAN-like or ALGOL-like notation that could be devised.[12]

Nieodłącznym elementem wyrażeń w Lispie są nawiasy. To dzięki nim nazwa Lisp rozwijana jest

też jako *Lots of Irritating Superfluous Parentheses* (wiele irytujących zbytecznych nawiasów)[15] czy też *Lost In Stupid Parentheses* (zagubiony w głupich nawiasach). Jednak nawet przeciwnicy Lispa muszą przyznać, iż składnia oparta na S-wyrażeniach leży u podstaw jego możliwości – jest ona niezwykle regularna, co znakomicie ułatwia jej przetwarzanie przez komputer.

Podstaowoym elementem składni Lispa jest, a jakże by inaczej, lista. Zapisywana jest ona jako elementy rozdzielone białymi znakami i otoczone wspomnianymi już nawiasami. Na przykład,

```
(1 2 foo)
```

to lista, której elementami są trzy atomy: (domyślnie typowane) wartości 1, 2, i `foo`[‖].

Wyrażenia zapisywane są jako listy z wykorzystaniem **notacji polskiej**[**]. Co ciekawe lista jako struktura danych służy w Lispie także do zapisu kodu źródłowego

Oto kilka przykładów kodu w Common Lispie.

- Klasyczny program wyświetlający napis *Hello world*

```
(print "Hello world")
```

- Obliczenie silni danej liczby

```
(defun factorial (n)
  (if (<= n 1)
    1
    (* n (factorial (- n 1)))))
```

13.3 Wpływ na programowanie

Paul Graham wymienia m.in. następujące idee, które Lisp wcielił w życie a które (przynajmniej część z nich) są częścią dzisiejszej rzeczywistości programisty[21]

1. Instrukcja warunkowa. Instrukcja warunkowa typu `if-then-else` jest dziś czymś oczywistym. Mało kto wie jednak, że narodziła się ona podczas prac Johna McCarthyego nad Lispem w bardziej ogólnej formie jako konstrukcja `cond`. Fortran w tamtym czasie dysponował jedynie instrukcją skoku warunkowego wzorowaną na rozkazach maszynowych. McCarthy, jako członek

[‖] Czym jest *foo*, patrz [13].

[**] Także: zapis przedrostkowy (ang. *prefix notation*), Notacja Łukasiewicza — sposób zapisu wyrażeń logicznych (a później arytmetycznych), podający najpierw operator, a potem operandy (argumenty). Został przedstawiony w 1920 roku przez polskiego filozofa i logika Jana Łukasiewicza. Obecnie informatyka jest jedynym polem, gdzie notacja ta jest wciąż popularna, zwłaszcza w wersji tzw. **odwrotnej notacji polskiej**.

komitetu opracowującego Algol, wprowadził instrukcję warunkową do tego języka, skąd trafiła ona do języków współczesnych.

2. Typ funkcyjny. W Lispie funkcje reprezentowane są za pomocą specjalnego typu w taki sam naturalny sposób jak zwykle reprezentujemy liczby całkowite czy napisy. Funkcje możemy przypisywac do zmiennych, przekazywac jako argumenty itd.

3. Rekurencja. Oczywiście pojęcie rekurencji istniało na długo przed tym jak pojawił się Lisp, ale Lisp był pierwszym językiem programowania, który umożliwiaj jej wykorzystanie.

4. Garbage-collection. Garbage collection (zbieranie nieużytków) to architektura zarządzania pamięcią, w której proces zwalniania nieużywanych jej obszarów odbywa się automatycznie.

5. Program składa się z wyrażeń. Program Lispa to inaczej mówiąc drzewo wyrażeń, z których każde zwraca wartość[††]. Większość współczesnych języków rozróżnia wyrażenia (ang. *expressions*) i instrukcje (ang. *statements*).

   - Wyrażenie (w języku programowania) to ewaluowana (wartościowana) zgodnie z określonymi w danym języku regułami kombinacja wartości, zmiennych, operatorów i funkcji, która zwraca inną wartość. Mówi się, że wyrażenie ewaluuje się do tej wartości. Podobnie jest w matematyce: wyrażenie jest reprezentacją pewnej wartości. Wyrażenia mogą (choć nie muszą) mieć efekty uboczne. Brak efektów ubocznych jest jedną z zasad programowania funkcyjnego, języki całkowicie nie obsługujące efektów ubocznych nazywami językami czysto funkcyjnymi.
   - Instrukcja (języka programowania) to najmniejszy samodzielny element imperatywnego języka programowania. Program jest tworzony jako zbiór różnych instrukcji. Instrukcja może zawierać wewnętrzne komponenty (np. wyrażenia). Wiele języków (np. C) odróżnia instrukcje i definicje: instrukcja zawiera kod wykonywalny, a definicja deklarację identyfikatora. W większości języków instrukcje różnią się od wyrażeń tym, że niekoniecznie zwracają wyniki i mogą być wykonywane dla osiągnięcia określonych skutków ubocznych, podczas gdy wyrażenia zawsze zwracają wynik i zwykle nie powodują żadnych efektów ubocznych.

[††]W niektórych implementacjach możliwe jest zwrócenie wielu wartości.

Co ciekawe, operując językiem zbudowanym w oparciu o wyrażenia identyczne rzeczy (w sensie uzyskanych efektów) można zapisać na wiele róznych sposobów. Można więc napisać (posługujemy się tutaj składnią typową dla języka Arc, czyli jak okresla go jego twórca: niedokończonego dialektu Lispa[22])

```
(if foo (= x 1) (= x 2))
```

ale również i tak

```
(= x (if foo 1 2))
```

6. Bez względu na to co się dzieje z programem, cały czas mamy dostęp do wszystkich aspektów języka. Oznacza to, że nie ma rozróżnienia np. pomiędzy etapem kompilacji a wykonania. Kod może być kompilowany lub wykonywany gdy jest odczytywany, czytany lub wykonywany gdy jest kompilowany czy też odczytywany lub kompilowany gdy jest wykonywany. Możliwość wykonywania kodu podczas jego odczytu umożliwia użytkownikom zmianę jego składni. Wykonywanie kodu podczas kompilacji jest podstawą zaawansowanych makr pozwalających na tworzenie kodu źródłowego „w locie". Kompilacja podczas wykonywania umożliwia na wykorzystanie języka jako rozszerzenia w innych programach (np. Emacs).

It seems to me that there have been two really clean, consistent models of programming so far: the C model and the Lisp model. These two seem points of high ground, with swampy lowlands between them. As computers have grown more powerful, the new languages being developed have been moving steadily toward the Lisp model. A popular recipe for new programming languages in the past 20 years has been to take the C model of computing and add to it, piecemeal, parts taken from the Lisp model, like runtime typing and garbage collection.

Paul Graham, The Roots of Lisp, May 2001

ROZDZIAŁ 14

Podstawy składni

W rozdziale tym poznamy podstawowe zasady rządzące składnią w Lispie. Naszym celem jest opisanie składni w taki sposób, aby jak najszybciej można było rozpocząć pisanie programów w tym języku. Nie bedziemy mówić tutaj o detalach, pozostawiając szczegóły do nastepnych rozdziałów.

14.1 s-wyrażenia

14.1.1 Listy

Składnia Lispa należy do jednej z najprostszych wśród języków programowania i opiera się o tzw. s-wyrażenia. Podstawowymi elementami tworzącymi s-wyrażenie są **lista** i **atom**. Listy ograniczone są nawiasami okrągłymi ((i)) i zawierają dowolną ilość elementów bedących s-wyrażeniami i rozdzielonych białymi znakami. Resztę stanowią atomy.

I to w zasadzie wszystko. Pozostało wyjaśnić jak wygląda składnia atomów. W dalszej części zajmiemy się najpopularniejszymi ich rodzajami, tj. liczbami (ang. *numbers*), napisami* (ang. *strings*) i nazwami (ang. *names*).

14.1.2 Atomy – liczby

Mając doświadczenie z innych języków programowania, można trafnie przewidywać, że liczba to nic innego jak ciąg cyfr, poprzedzonych ewentualnie znakiem i zawierający opcjonalny separator części całkowitej i ułamkowej. Można także spodziewać się, że dopuszczalne jest użycie tzw. notacji

*W języku polskim zwykle używamy własnie określenia *napis*, choć chcąc być precyzyjniejszym powinno się mówić o *ciągach znaków*.

naukowej, czyli podanie wykładnika. I rzeczywiście, wszystko to jest prawdą. Poniżej podajemy kilka przykładów poprawnych liczb zapisanych w Lispie[†]

```
1        ; liczba całkowita jeden
+1       ; liczba całkowita (dodatnia) (plus) jeden
-1       ; liczba całkowita (ujemna) minus jeden
1.0      ; liczba zmiennoprzecinkowa (rzeczywista) jeden o domyślnej precyzji
1.0e0    ; liczba zmiennoprzecinkowa (rzeczywista) jeden o domyślnej precyzji
         ; zapisana w notacji naukowej
1.0d0    ; liczba zmiennoprzecinkowa (rzeczywista) jeden o podwójnej precyzji
         ; zapisana w notacji naukowej
1.0e-4   ; liczba zmiennoprzecinkowa (rzeczywista) jedna dziesięciotysięczna
         ; o domyślnej precyzji zapisana w notacji naukowej
```

Najprawdopodobniej jednak nie odgadniemy pozostałych możliwości jakie daje nam Lisp czyli np. liczb ułamkowych czy też liczb zespolonych.

```
3/7      ; ułamek trzy-siódme
-1/2     ; ułamek minus jedna-druga
-2/4     ; inny sposób zapisu ułamka minus jedna-druga
2/2      ; inny sposób zapisu liczby całkowitej jeden
#c(10 5) ; liczba zespolona 10 + 5i (czyli o części rzeczywistej
         ; równej dziesięć i części urojonej równej pięć
```

Przyglądając się powyższym przykładom widać, że te same liczby można zapisywać na wiele sposobów, gdyż Lisp sprowadza je do pewnej postaci kanonicznej właściwej dla typu liczby reprezentowanej przez napis. Na przykład ułamki zawsze są upraszczane przez co $\frac{2}{2}$ jest tym samym[‡] co 1 podobnie jak 1.0 jest tym samym co $1.0e0$ Z drugiej natomiast strony 1.0 nie jest tym samym co $1.0d0$ lub 1. Więcej informacji o liczbach podamy w rozdziale **??**.

[†]W przykładzie widzimy także użycie średnika jako znaku rozpoczynającego jednolinijkowy komentarz.
[‡]Zarówno biorąc pod uwagę typ jak i wartość.

14.1.3 Atomy – napisy

Napis jest dowolnym ciągiem znaków ujętym w cudzysłów[§]. W takiej sytuacji znakiem wymagającym specjalnego traktowania jest znak cudzysłowu, który, jeśli ma być częścią tekstu, należy poprzedzić znakiem backslash[¶]. Ponieważ backslash pełni rolę znaku zmieniającego znaczenie znaku za nim następujacego[‖], więc gdy chcemy go wypisać, to także jego należy poprzedzić znakiem backslash.

```
"foo"     ; ciąg znaków zawierający znaki f, o oraz o
"fo\o"    ; ciąg znaków zawierający znaki f, o oraz o
"fo\\o"   ; ciąg znaków zawierający znaki f, o, \ oraz o
"fo\"o"   ; ciąg znaków zawierający znaki f, o, " oraz o
```

14.1.4 Atomy – nazwy

W Lispie nazwy to zarówno np. `FOO` jak i `hello-world` czy `*foo*`. Mówiąc ogólnie, nazwy pełnią rolę symboli mogących reprezentować np. zmienną czy funkcję. Dlatego też w dalszej części wymiennie używać będziemy pojęcia *nazwa* i *symbol*. Z samego „wygladu” nazwy niesposób wywnioskować czego są reprezentacją (co można było zrobić np. w Prologu, gdzie zmienne zaczynają się zawsze z dużej litery). Prawie każdy znak może w dowolny sposób wchodzić w skład nazwy. Wyjątkami są poniższe grupy.

- **Znak spacji.** Znak spacji ze względu na rolę separatora elementów listy jest znakiem o szczególnym znaczeniu i dlatego nie może być częścią nazwy.
- **Cyfry.** Cyfry mogą być częścią nazwy tak długo jak tylko nazwa nie może być zinterpretowana jako liczba.
- **Kropka.** Nazwa może zawierać kropkę, ale nie może składać się tylko z kropek.
- **Znaki specjalne.** Jest dziesięć znaków mających specjalne znaczenie i jako takie nie powinny być częścią nazwy: nawiasy okrągłe, cudzysłów i apostrof, backtick, przecinek, średnik, backslash i kreska pionowa. Nie powinny, ale mogą stać się częścią nazwy gdy zostaną poprzedzone znakiem backslash lub zostaną umieszczone pomiędzy kreskami pionowymi.

[§] Nazywany także *podwójnymi uszami* dla odróżnienia od znaku apostrofu, czyli *pojedyńczego ucha*.

[¶] Nazywany „po naszemu” *w-tył-ciach*.

[‖] W Lispie backslash wpływa tylko na znak cudzysłowu. W innych językach jest więcej kombinacji backslash-znak, np. w C mamy znak przejścia do nowego wiersza `\n` czy też znak tabulacji `\t`.

Zasadniczo wielkość użytych liter nie ma znaczenia, gdyż są one zawsze konwertowane na duże litery przez co np. nazwy `foo`, `Foo` oraz `FOO` rozumiane są jako ten sam symbol: `FOO`. Inaczej sprawa wygląda gdy użyjemy znaków specjalnych. Na przykład łańcuchy `\f\o\o` lub `|foo|` reprezentują nazwę `foo`.

Z powyższego widać, że nazwy w Lispie mogą być znacznie „bogatsze” niż w innych językach jak choćby C czy Java. I jak w większości języków, także tutaj obowiązują pewne konwencje związane z nazewnictwem.

- Nazwy wielowyrazowe łączone są myślnikiem, np. `hello-world`.
- Zmienne globalne, to nazwy zaczynające się i kończące znakiem `*`.
- Stałe, to nazwy zaczynające się i kończące znakiem `+`.
- Czasem nazwy niskopoziomowych funkcje w programie lispowym poprzedza się jednym lub dwoma znakami `%`.

Składnia list, liczb, napisów i nazw daje wystarczająco dobre rozeznanie w tym co i jak można w Lispie napisać. Pozostałe elementy jak np. wektory czy tablice nie odbiegają w znaczący sposób od opisanych reguł i zapoznanie z nimi możemy na jakiś czas odłożyć (patrz rozdział ??). Dla nas w tym momencie istotna jest umiejętność łączenia podanych reguł. Przypatrzmy się następującym przykładom

```
x              ; nazwa (symbol) X
()             ; lista pusta
(1 2 3)        ; lista zawierająca trzy liczby
("foo" "bar")  ; lista zawierająca dwa napisy
(x y z)        ; lista zawierająca trzy symbole
(x 1 "foo")    ; lista zawierająca symbol, liczbę i napis
(+ (* 2 3) 4)  ; lista zawierająca symol, listę i liczbę
```

I troszeczkę bardziej złozony przykład listy zawierającej cztery elementy (dwa symbole, listę pustą i listę) w typowym dla Lispa „nieczytelnym” zapisie

```
(defun hello-world ()
  (format t "hello, world"))
```

14.2 Ewaluacja wyrażeń

14.2.1 Symbole

Atomy: symbole

Atomy: liczby i napisy

self-evaluating symbols

14.3 Lista – szczególne przypadki

Zajmiemy się teraz trzema przypadkami dotyczacymi składni listy mających szczególne znaczenie. W kolejności opisywania będą to

- wywołania funkcji (ang. *function call forms*),
- operatory specjalne (ang. *special forms*),
- wywołania makr (ang. *macro forms*).

Dalsze szczegóły związane wyżej wymienionymi elementami opisane zostaną w rozdziale (Funkcje), oraz .

Wywołania funkcji

Podstawowa składnia wywołania funkcji ma postać

```
(function-name argument*)
```

gdzie `function-name` jest nazwą funkcji a `argument*` jest ciągiem wyrażeń będących argumentami. Istotne jest w tym miescu to, że argumentem może być każde poprawne wyrażenie języka. Reguły ewalucji takiej listy powodują ewaluację wszytkich argumentów listy i przekazanie ich wyników do funkcji jako efektywnych argumentów. Zatem wyrażenie

```
(+ 1 2)
```

ewaluowane jest w następujący sposób

1. Ewaluacja wyrażenia 1. W wyniku ewaluacji otrzymujemy wartość liczbową 1.
2. Ewaluacja wyrażenia 2. W wyniku ewaluacji otrzymujemy wartość liczbową 2.

3. Ponieważ nie ma więcej wyrażeń będących argumentami, zatem wszystkie otrzymane do tej pory wyniki ewaluacji przekazywane są do funkcji + jako jej (efektywne) argumenty. W wyniku ewaluacji funkcji + otrzymujemy wartość liczbową 3.

Podobny schemat zachowany jest także przy bardziej złożonych wyrażeniach, np.

```
(* (+ 1 2) (- 3 4))
```

W tym przypadku ewaluowane są najpierw argumenty `(+ 1 2)` oraz `(- 3 4)` po czym otrzymane wyniki stają się argumentem dla funkcji `*`.

Operatory specjalne

Funkcje są bardzo użyteczne, ale nie rozwiązują one wszystkich napotykanych problemów/sytuacji. Pewnych operacji nie można, choć mogłoby wydawać się inaczej, opisać jako funkcje a przynajmniej nie można postępować z nimi w sposób opisany powyżej.

```
(if x (format t "yes") (format t "no"))
```

Składniowo operatory specjalne wyglądają jak wywołanie funkcji, ale trzeba sobie zdawać sprawę z tego, że działają odmiennie (w znaczeniu: są ewaluowane w inny sposób niż funkcje). W przypadku operatora `if` działanie określone jest w prosty sposób. Jeśli pierwsze wyrażenie w wyniku ewaluacji przyjmie wartość różną od `NIL`, wówczas ewaluowane jest drugie wyrażenie i jego wartość jest wartością zwracaną przez `if`. W przeciwnym razie `if` zwraca wartość trzeciego wyrażenia lub `NIL` jeśli ono nie występuje. Innymi słowy składnia `if` wygląda jak poniżej

```
(if test-form then-form [ else-form ])
```

Makra

Makra są sposobem na rozszerzanie syntaktyki języka we wszytkich przypadkach, gdy zastosowanie funkcji jest niemożliwe lub niewskazane. Makra, choć składniowo podobne do funkcji, pełnią całkiem odmienną rolę. Widać to najlepiej w ewaluacji makr. Przyjmują one jako swój argument s-wyrażenia i wyrażenie jest także wynikiem ich działania. Dalej jednak wynik ten jest ewaluowany w miejscu wystąpienia makra. Tak więc ewaluacja makra przebiega w dwóch etapach

1. Przekazanie do makra argumentów, które nie pdlegają jednak ewaluacji jak w przypadku funkcji.
2. Wyrażenie zwrócone przez makro, nazywane **rozwinięciem makra** (ang. *macro expansion*) ewaluowane jest w miejscu wystapienia makra według normalnych reguł ewalucaji wyrażeń.

14.3.1 Prawda, fałsz i równość

Prada, fałsz

Nieodzownym elementem programowania jest zwykle potrzeba wyrażenia prawdziwości lub fałszywości pewnego wyrażenia a także stwierdzenia identyczności dwóch obiektów. W Lispie „prawdziwość” i „fałszywość” przedstawia się według wyjątkowo prostych reguł: symbol `nil` oznacza fałsz a wszystko inne jest prawdą. W przypadku gdy nie mamy akurat żadnego wyrażenia różnego od `nil`, które mogłoby zostać zwrócone jako wynik działania, można posłużyć się symbolem `T`. `nil` jest jedynym obiektem, który jest jednocześnie atomem (reprezentuje fałsz) i listą (reprezentuje pustą listę).

Porównanie: eq

eq sprawdza czy dwa obiekty są identyczne. Niestety w tym przypadku identyczność jest mocno zależna od implemetacji. Może się zdarzyć, że wyrażenie `(eq x x)`, w którym `x` jest znakiem lub liczbą, bedzie ewaluowane zarówno do prawdy jak i do fałszu. Oto kilka przykładów

```
(eq 'a 'b)                                  ==> false
(eq 'a 'a)                                  ==> true
(eq 3 3)                                    ==> true or false
(eq 3 3.0)                                  ==> false
(eq 3.0 3.0)                                ==> true or false
(eq #c(3 -4) #c(3 -4))                      ==> true or false
(eq #c(3 -4.0) #c(3 -4))                    ==> false
(eq (cons 'a 'b) (cons 'a 'c))              ==> false
(eq (cons 'a 'b) (cons 'a 'b))              ==> false
(eq '(a . b) '(a . b))                      ==> true or false
(progn (setq x (cons 'a 'b)) (eq x x))      ==> true
(progn (setq x '(a . b)) (eq x x))          ==> true
(eq #\A #\A)                                ==> true or false
(eq "Foo" "Foo")                            ==> true or false
(eq "Foo" (copy-seq "Foo"))                 ==> false
(eq "FOO" "foo")                            ==> false
```

Porównanie: `eql`

`eql` oferuje identyczne działanie jak eq z tym wyjątkiem, że dla dwóch obiektów tej samej klasy reprezentujących taką samą wrtość liczbową lub taki sam znak zwróci wartość prawda. Dlatego wyrażenie `(eql 1 1)` zawsze zwraca prawdę, natomiast `(eql 1 1.0)` fałsz. Oto kilka przykładów

```
(eql 'a 'b)                                 ==> false
(eql 'a 'a)                                 ==> true
(eql 3 3)                                   ==> true
(eql 3 3.0)                                 ==> false
(eql 3.0 3.0)                               ==> true
(eql #c(3 -4) #c(3 -4))                     ==> true
(eql #c(3 -4.0) #c(3 -4))                   ==> false
(eql (cons 'a 'b) (cons 'a 'c))             ==> false
(eql (cons 'a 'b) (cons 'a 'b))             ==> false
(eql '(a . b) '(a . b))                     ==> true or false
(progn (setq x (cons 'a 'b)) (eql x x)) ==> true
(progn (setq x '(a . b)) (eql x x))         ==> true
(eql #\A #\A)                               ==> true
(eql "Foo" "Foo")                           ==> true or false
(eql "Foo" (copy-seq "Foo"))                ==> false
(eql "FOO" "foo")                           ==> false
```

Oczywiście powstaje pytanie czy i kiedy używać eq lub `eql`? Każde rozwiązanie ma swoich zwolenników.

- Jeśli to tylko możliwe, korzystać z eq, gdyż
  - w ten sposób sygnalizujemy, że nie będziemy porównywac liczb i znaków;
  - eq jest (nieznacznie) wydajniejsze, gdyż nie musi sprawdzać czy przypadkiem argument nie jest liczbą lub znakiem.
- Nigdy nie należy korzystać z eq, gdyż

– każdorazowe natrafienie na eq zmusza nas do zastanowienia się nad tym, czy faktycznie użyto tego porównania zgodnie z przeznaczeniem (tzn. czy przypadkiem argumenty nie bądą liczbą lub znakiem);

– zysk z wydajności jest pozorny, gdyż zwykle ulega stracony przez występowanie innych wąskich gardeł.

Porównanie: `equal`

Predykat `equal` jest prawdziwy gdy jego argumenty są strukturalnie podobnymi obiektami. Dla ułatwienia można przjąć regułę, że `equal` zwraca prawdę dla obiektów, które posiadają identyczną postać wizualną (są wypisywane w taki sam sposób). Oto kilka przykładów

```
(equal 'a 'b)                            ==> false
(equal 'a 'a)                            ==> true
(equal 3 3)                              ==> true
(equal 3 3.0)                            ==> false
(equal 3.0 3.0)                          ==> true
(equal #c(3 -4) #c(3 -4))                ==> true
(equal #c(3 -4.0) #c(3 -4))              ==> false
(equal (cons 'a 'b) (cons 'a 'c))        ==> false
(equal (cons 'a 'b) (cons 'a 'b))        ==> true
(equal '(a . b) '(a . b))                ==> true
(progn (setq x (cons 'a 'b)) (equal x x)) ==> true
(progn (setq x '(a . b)) (equal x x))    ==> true
(equal #\A #\A)                          ==> true
(equal "Foo" "Foo")                      ==> true
(equal "Foo" (copy-seq "Foo"))           ==> true
(equal "FOO" "foo")                      ==> false
```

Porównanie: `equalp`

`equalp` działa podobnie do `equal` z tą różnicą, że

- przy porównywaniu napisów nie jest brana pod uwagę wielkość liter;
- liczby są równe jeśli reprezentują taką samą wartość (z matematycznego punktu widzenia).

Oto kilka przykładów

```
(equalp 'a 'b)                            ==> false
(equalp 'a 'a)                            ==> true
(equalp 3 3)                              ==> true
(equalp 3 3.0)                            ==> true
(equalp 3.0 3.0)                          ==> true
(equalp #c(3 -4) #c(3 -4))                ==> true
(equalp #c(3 -4.0) #c(3 -4))              ==> true
(equalp (cons 'a 'b) (cons 'a 'c))        ==> false
(equalp (cons 'a 'b) (cons 'a 'b))        ==> true
(equalp '(a . b) '(a . b))                ==> true
(progn (setq x (cons 'a 'b)) (equalp x x)) ==> true
(progn (setq x '(a . b)) (equalp x x))    ==> true
(equalp #\A #\A)                          ==> true
(equalp "Foo" "Foo")                      ==> true
(equalp "Foo" (copy-seq "Foo"))           ==> true
(equalp "FOO" "foo")                      ==> true
```

14.4 Formatowanie kodu programu

W Lispie, podobnie jak w większości języków programowania, trudno mówić aby formatowanie kodu miało jakieś znaczenie sematyczne lub syntaktyczne[**]. Mimo to zaleca się przestrzeganie pewych reguł co ma oczywiście ujednolicić tworzony kod. Podobnie jak w Pythonie kluczem jest tutaj właściwe stosowanie wcięć[††]. Wcięcia powinny odzwierciedlać strukture programu a w szczególności ułatwiać kontrolowanie występowania nawiasów. Każdy kolejny poziom zagnieżdżenia powinien posiadać większe wcięcie.

```
(defun print-list (list)
  (dolist (i list)
    (format t "item:␣~a~%" i)))
```

[**]Jako wyjątek można podać np. Python, gdzie wielkość wcięć jest istotnym elementem składniowym i znaczeniowym.

[††]Choć w Pythonie jest to wymagane a tutaj „dobrowolne”.

Inną ważną regułą jest wstawianie nawiasów zamykających w tym samym wierszu, w którym występuje ostatni element listy zamykanej przez dany nawias. Dlatego zamiast

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d.␣hello~%" i)
  )
)
```

należy pisać

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d.␣hello~%" i)))
```

Komentarze

Choć komentarz rozpoczyna pojedyńczy znak średnika, to w Lispie istnieje zwyczajowa hierarchia komentarzy zależna od ich „zasięgu”. Przypatrzmy się następującemu przykładowi.

```
;;;; W ten sposob komentujemy tekst odnoszacy sie do calego pliku

;;; W ten sposob komentujemy tekst rozpoczynajacy nowa czesc kodu
;;; Zwykle dotyczy on wiekszej sekcji

(defun foo (x)
  (dotimes (i x)
    ;; W ten sposob komentujemy tekst majacy zastosowanie do
    ;; kodu nastepujacego bezposrednio za tym komentarzem.
    ;; Zauwazmy przy tym, ze komentarz ma takie samo wciecie
    ;; jak kod
    (some-function-call)
    (another-fun arg)          ; Ten komentarz dotyczy tylko tej linii
    (and-another)              ; a ten tylko tej
    (baz)))
```

ROZDZIAŁ 15

Funkcje

W rozdziale tym dowiemy się jak zapisywać funkcje, czyli konstrukcje programistyczne, które dostarczają najbardziej elementarnego mechanizmu abstrakcji.

15.1 Definiowanie funkcji

Funkcje w Lispie definiowane są za pomocą makra `defun` o nastepującej składni

```
(defun name (parameter*)
  "Optional␣documentation␣string."
  body-form*)
```

gdzie `parameter*` to opcjonalna lista parametrów, `"Optional documentation string."` to opis funkcji a `body-form*` jest ciągiem wyrażeń tworzących ciało funkcji.

Nazwa Praktycznie każdy symbol lub ich ciąg może być nazwą funkcji. Zwyczajowo jest to ciąg liter rozdzielany w razie potrzeby znakiem myślnika jeśli nazwa jest wielowyrazowa. Tak więc piszemy raczej `nazwa-funkcji` niż `nazwa_funkcji` czy też `nazwaFunkcji`. Nic nie stoi jednak na przeszkodzie aby np. funkcje konwertującą liczby dziesiętne na binarne nazwać `decimal->binary`.

Argumenty

Opis funkcji

Ciało funkcji

Przyglądając się poniższej funkcji `hello-world` bez trudu odnajdujemy poszczególne elementy ją tworzące.

```
(defun hello-world () (format t "hello,␣world"))
```

Jej nazwa to `hello-world`, lista parametrów jest pusta, nie zawiera ciągu ją opisującego a ciało zawiera tylko jedno wyrażenie

```
(format t "hello,␣world")
```

Funkcja będąca kolejnym przykładem zawiera wszytkie opisane elementy

```
(defun verbose-sum (x y)
  "Sum␣any␣two␣numbers␣after␣printing␣a␣message."
  (format t "Summing␣~d␣and␣~d.~%" x y)
  (+ x y))
```

Po kolei mamy

- nazwę funkcji: `verbose-sum`;
- listę parametrów `(x y)` z którymi związne zostaną argumenty;
- opis funkcji: `Sum any two numbers after printing a message.`;
- ciało zawierające dwa wyrażenia; wartość zwrócona przez drugie (ostatnie) wyrażenie jest wartością zwracaną przez funkcję.

15.2 Parametry funkcji

15.2.1 Parametry wymagane

Jeśli lista parametrów (argumentów) jest listą nazw zmiennych, wówczas argumenty takie nazywamy argumentami wymaganymi (ang. *required parameters*). Jak sama nazwa wskazuje, takie argumentu muszą być podane przy wywołaniu funkcji.

```
(defun foo (a b) (list a b))
```

15.2.2 Parametry opcjonalne

W celu zdefiniowania funkcji z parametrami opcjonalnymi (ang. *optional parameters*) używamy symbolu `&optional`. Symbol ten występuje za wszystkimi parametrami wymaganymi, ale poprzedza wszystkie parametry opcjonalne.

```
(defun foo (a b &optional c d) (list a b c d))
```

```
(foo 1 2)     ==> (1 2 NIL NIL)
(foo 1 2 3)   ==> (1 2 3 NIL)
(foo 1 2 3 4) ==> (1 2 3 4)
```

Oczywiście domyślnie przypisywana wartość dla parametru opcjonalnego jaką jest `nil` może zostać zmieniona na inna bardziej odpowiednią. W tym celu nazwę parametru opcjonalnego należy zastąpić listą zawierającą nazwę parametru oraz dowolne wyrażenie. W przypadku gdy użytkownik nie wyspecyfikuje argumentu, wówczas jego wartością będzie wartość zwrócona przez to wyrażenie.

```
(defun foo (a &optional (b 10)) (list a b))
```

```
(foo 1 2) ==> (1 2)
(foo 1)   ==> (1 10)
```

Lisp daje nam jeszcze większą swobodę w ustalaniu wartości argumentów – możemy bowiem uzależnić wartość argumentu od innych argumentów. Typowy porzykład funkcji gdzie utworzenie takiej zależności jest naturalnym rozwiązaniem jest np. funkcja tworząca obiekt prostokąta. Jak wiadomo kwadrat jest szczególnym przypadkiem prostokąta i dlatego nie warto tworzyć dla niego odrębnej funkcji. Z drugiej strony konieczność wyspecyfikowania długości obu boków, gdy wiadomo, że są identyczne bywa irytująca. W Lispie można jednak napisać listę parametrów jak w poniższym przykładzie

```
(defun make-rectangle (width &optional (height width)) ...)
```

co powoduje, że opcjonalnym parametr `height` w razie jego niewyspecyfikowania przyjmuje wartość `width`.

Czasem warto wiedzieć skąd argument pochodzi: czy jego wartość jest wynikiem wyspecyfikowania jej przez użytkownika czy domyślnego podstawienia wynikającego z definicji funkcji. Na przykład funkcja

```
(defun foo (a b &optional (c 3 c-supplied-p))
  (list a b c c-supplied-p))
```

zwróci następujące rezultaty

```
(foo 1 2)   ==> (1 2 3 NIL)
(foo 1 2 3) ==> (1 2 3 T)
(foo 1 2 4) ==> (1 2 4 T)
```

15.2.3 Dowolna ilość parametrów

maksymalna ilość parametrów funkcji zależna jest od implementacji, ale wynosi minimum 50. W większości przypadków wynosi ona od 4096 do 536,870,911. Można ją sprawdzić za pomocą stałej `CALL-ARGUMENTS-LIMIT`.

Przykłady funkcji wywoływanych ze zmienną ilością parametrów

```
(format t "hello, world")
(format t "hello, ~a" name)
(format t "x: ~d y: ~d" x y)
(+)
(+ 1)
(+ 1 2)
(+ 1 2 3)
```

```
(defun format (stream string &rest values) ...)
(defun + (&rest numbers) ...)
```

15.2.4 Parametry nazwane

```
(defun foo (&key a b c) (list a b c))
```

Jeśli parametr nazwany nie zostanie podany, wówczas zostanie nadana jemu wartość domyślna podobnie jak dla parametru opcjonalnego. Ze względu na posługiwanie się wyróżnikiem jakim jest nazwa, parametry nazywane mogą występowac w dowolnej kolejności. Oto kilka przykładów wywołań

```
(foo)                ==> (NIL NIL NIL)
(foo :a 1)           ==> (1 NIL NIL)
(foo :b 1)           ==> (NIL 1 NIL)
(foo :c 1)           ==> (NIL NIL 1)
(foo :a 1 :c 3)      ==> (1 NIL 3)
(foo :a 1 :b 2 :c 3) ==> (1 2 3)
(foo :a 1 :c 3 :b 2) ==> (1 2 3)
```

Podobnie jak w przypadku parametrów opcjonalnych, parametry nazywane mogą mieć przypisane wartości domyślne i zmienną określającą „pochodzenie” argumentu.

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b)))
  (list a b c b-supplied-p))
```

```
(foo :a 1)           ==> (1 0 1 NIL)
(foo :b 1)           ==> (0 1 1 T)
(foo :b 1 :c 4)      ==> (0 1 4 T)
(foo :a 2 :b 1 :c 4) ==> (2 1 4 T)
```

Ponad to możemy zmienić domyślne zachowanie, tak aby parametry miały inną nazwę niż zmienne użyte w ciele funkcji. W poniższym przykładzie zapis `(:aaa a)` oznacza: parametr a należy nazywać aaa Poniższa definicja funkcji `foo`

```
(defun foo (&key ((:aaa a)) ((:bbb b) 0) ((:ccc c) 0 c-supplied-p))
  (list a b c c-supplied-p))
```

pozwala na następujące wywołanie

```
(foo :aaa 10 :bbb 20 :ccc 30) ==> (10 20 30 T)
```

15.2.5 „Mieszanie” parametrów

tutu – uzupelnic

kolejnosc

```
(defun foo (x &optional y &key z) (list x y z))
```

Pewne wywołania będą działały bez problemu, np.

```
* (foo 1 2 :z 3)

(1 2 3)
```

oraz

```
* (foo 1)

(1 NIL NIL)
```

Niestety w pewnym momencie „funkcja przestanie działać”

```
* (foo 1 :z 3)

debugger invoked on a SB-INT:SIMPLE-PROGRAM-ERROR in thread #<THREAD
"initial thread" RUNNING {AA5E819}>:
  odd number of &KEY arguments

Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.

(FOO 1 :Z 3)[:OPTIONAL]
0]
```

Stanie się tak, gdyż klucz `:z` zostanie potraktowany jako wartość opcjonalnego arametru y pozostawiając jako argumenty niezwiązane jedynie 3. Z definicji funkcji, Lisp spodziewa się natomiast, że trzeci argument będzie albo parą klucz-wartość albo bedzie pusty. Stąd otrzymany wynik. Na szczęście niektóre interpretery syganlizują możliwość wystąpienia problemów*

```
* (defun foo (x &optional y &key z) (list x y z))
; in: LAMBDA NIL
;     (SB-INT:NAMED-LAMBDA FOO (X &OPTIONAL Y &KEY Z) (BLOCK FOO (LIST X Y Z)))
; ==>
;   #'(SB-INT:NAMED-LAMBDA FOO (X &OPTIONAL Y &KEY Z) (BLOCK FOO (LIST X Y Z)))
;
; caught STYLE-WARNING:
;   &OPTIONAL and &KEY found in the same lambda list: (X &OPTIONAL Y &KEY Z)
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition

FOO
```

*Wynik zwrócony przez SBCL [Steel Bank Common Lisp] 1.0.29.11.debian, an implementation of ANSI Common Lisp.

15.3 Zwracanie wartości

We wszystkich przykładach jakie pojawiły się do tej pory jako wynik działania funkcji zwracana była wartość ewaluacji ostatniego wyrażenia. Jeśli to zachowanie domyślne nie jest dla nas odpowiednie, możemy skorzystać z operatora `return-from` powodującego natychmiastowe zwrócenie przez funkcję określonej wartości. Pierwszym argumentem `return-from` jest nazwa bloku (funkcji) z jakiego ma nastąpić powrót. Ponieważ argument ten nie podlega ewaluacji, więc nie ma potrzeby zaznaczania, że jest to nazwa.

Poniższy przykład wykorzystuje pętle zagnieżdżone w celu znalezienia pary liczb mniejszych od 10, których iloczyn jest większy od argumentu funkcji. Wykorzystując `return-from` zwracamy pierwszą znalezioną parę spełniającą podane warunki

```
(defun foo (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from foo (list i j))))))
```

15.4 Funkcja jako zmienna

Choć zasadniczym sposobem wykorzystania funkcji jest ich wywoływanie, to jednak zdarzają się sytuacje, gdy wygodnie jest potraktować funkcję jak daną (zmienną). Klasycznym przykładem jest tutaj funkcja sortująca, która jako jeden ze swoich argumentów przyjmuje funkcję wykorzystywaną do porównywania elementów celem ustalenia, który z nich powinien wystąpić jako pierwszy. Innym przykładem mogą być algorytmy przeszukiwania grafu wszerz i w głąb. Oba wspomniane algorytmy mają identyczną strukturę tylko operują na innych strukturach danych. W jednym z nich para funkcji `push`, `pop` operuje na stosie, w drugim na kolejce. „Podmieniające” te funkcje i pozostawiając algorytm całkowicie bez zmian uzyskujemy odmienne zachowanie programu.

W Lispie funkcja jest rodzajem obiektu. Definiując funkcję za pomocą `defun` w rzeczywistości

- tworzymy nowy obiekt reprezentujący funkcję
- i nadajemy temu obiektowi konkretną nazwę.

Po zdefiniowaniu funkcji

```
* (defun foo (x) (* 2 x))

FOO
```

za pomocą operator `function` możemy otrzymać obiekt ją reprezentujący, pisząc

```
* (function foo)

#<FUNCTION FOO>
```

lub w inny równoważny sposób

```
* #'foo

#<FUNCTION FOO>
```

Mając obiekt repreznetujący funkcję możemy teraz skorzystać z niego wywołując ją. Common Lisp dostarcza dwóch funkcji wykorzystujących do wywoływania funkcji obiekt ją reprezentujący: `funcall` oraz `apply` a różniących się sposobem przekazywania argumentów do funkcji.

`funcall`

Funkcję `funcall` używamy wówczas, gdy w momencie tworzenia kodu znamy liczbę argumentów jakie zostaną przekazane do wywoływanej funkcji. Pierwszym argumentem `funcall` jest nazwa obiektu funkcyjnego związanego z funkcją jaką chcemy wywołać, natomiast pozostałe argumenty są argumentami przekazywanymi w wywołaniu jakie ma nastapić. Poniższe dwie linie są sobie równoważne

```
(foo 1 2 3)
(funcall #'foo 1 2 3)
```

Poniższy kod zawiera bardziej dydaktyczny przykład użycia `funcall`. Oto funkcja `plot` przyjmuje cztery argumenty

- `fn` – nazwę jednoargumentowej funkcji zwracającej liczbę rzeczywistą
- `min` – tutu – uzupełnić
- `max` –

- `step` –

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
        (loop repeat (funcall fn i) do (format t "*"))
        (format t "~%")))
```

tutu – opisać działanie

Przykładowe uzycie może wyglądać jak poniżej

```
* (plot #'exp 0 4 1/2)
*
*
**
****
*******
************
********************
*********************************
*****************************************************
NIL
```

`apply`

W wielu przypadkach lista argumentów nie jest stała a tym bardziej nie jest możliwe określenie „z góry" liczby argumentów. Wówczas zamiast `funcall` należy skorzystać z `apply`. Pierwszym argumentem `apply` jest nazwa wywoływanej funkcji, drugim zaś lista zawierająca argumenty

```
* (defun f (x) (* 2 x))

F
* (apply #'f ?????? tutu uzupelnic przyklad ?????)
*
*
**
```

```
****
*******
************
********************
*********************************
******************************************************
NIL
```

15.5 Funkcje anonimowe

Gdy zaczniemy posługiwać się funkcjami traktowanymi jak argumenty dla innych funkcji, dosyć szybko irytować zacznie nas konieczność definiowania i nazywania funkcji tylko po to aby jeden jedyny raz móc z niej skorzystać. Lawina nazw rozpoczynających się od `help`, `tmp` czy podobnie może bardzo szybko nas pochłonąć. Jeśli każdorazowa konieczność definiowania funkcji za pomocą `defun` jest dla nas męcząca, wówczas możemy skorzystać z wyrażeń lambda (ang. *lamda expression*)

```
(lambda (parameters) body)
```

O wyrażeniach lambda można mysleć jak o funkcjach, których nazwa definiuje ich działanie†. Takie podejście wyjaśnia dlaczego lambda wyrażenie można wykorzystać w miejscu nazwy funkcji za operatorem `#'`

```
* (funcall #'(lambda (x y) (+ x y)) 2 3)

5
```

Powyższe można napisać w jeszcze bardziej zwartej formie, traktując `lambda` jak nazwę funkcji

```
* ((lambda (x y) (+ x y)) 2 3)

5
```

Jednak zasadniczym przeznaczeniem wyrażeń lambda jest przekazywanie do funkcji argumentów bedących prostymi funkcjami‡. Zamiast definiować funkcję

```
(defun double-exp (x) (* 2 (exp x)))
```

†Lub inaczej, w których ciało jest nazwą (jej częścią).
‡Prostymi w sensie: krótkimi.

aby ją wywołać w następujący sposób

```
* (plot #'double-exp 0 3 1/2)
**
****
******
*********
***************
*************************
*****************************************
NIL
```

prościej jest napisać

```
(plot #'(lambda (x) (* 2 (exp x))) 0 3 1/2)
```

ROZDZIAŁ 16

Zmienne

Pierwotnie rozdział ten miał pojawić się w dalszej części, jednak okoliczności skłoniły mnie do zmiany kolejności. Już bowiem na pierwszych ćwiczeniach z Lispa wiele osób pytało o możliwość posługiwania się zmiennymi. Zmienne w Lispie są jak najbardziej dostępne, choć nie one stanowią o specyfice tego języka, dlatego chciałem o tym mówić później. Skoro jednak trudno wyobrazić sobie bez nich program, to spróbujemy teraz się nimi zająć.

16.1 Podstawowe wiadomości

Common Lisp należy do języków **typowanych dynamicznie** (ang. *dynamically typed*) – to znaczy takich, w których błędy związane z niezgodnością typów eykrywane są na etapie wykonania programu. Podobną cechę posiada większość języków skryptowych jak np. PHP, Python czy bash.

Jednym z najbardziej elementarnych sposobów wprowadzenia zmiennych do programu (pomijając zmienne występujące na liście parametrów w definicji funkcji) jest użycie operatora `let` postaci

```
(let (variable*)
  body-form*)
```

`(variable*)` to lista zawierająca zmienne wraz z przypisaną im wartością lub same zmienne, jeśli mają mieć nadaną wartość domyslną `nil`.

```
* (let ((x 10) (y 20) z)
  (format t "Variables: x=~a, y=~a, z=~a" x y z))
Variables: x=10, y=20, z=NIL
```

```
NIL
```

Jako wartość `let` zwracana jest wartość ostatniego wyrażenia. Zakres działania zmiennych wprowadzonych przez `let` ograniczony jest przez samo wyrażenie. Dla programu jak poniżej

```
(defun foo (x)
  (let ((y 2))
    (format t "x=~a y=~a" x y))
  (format t "x=~a y=~a" x y))  ; The variable Y is unbound.
```

już podczas próby wczytania otrzymujemy ostrzeżenia

```
* (load "test")


; in: LAMBDA NIL
;     (FORMAT T "x=~a y=~a" X Y)
;
; caught WARNING:
;   undefined variable: Y
;
; compilation unit finished
;   Undefined variable:
;     Y
;   caught 1 WARNING condition


T
```

a błąd podczas próby wykonania

```
* (foo 1)
x=1 y=2
debugger invoked on a UNBOUND-VARIABLE in thread #<THREAD "initial thread" RUNNING {AA5E8
  The variable Y is unbound.


Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.
```

```
restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.


(FOO 1)
0] 0
```

W tym przypadku chodzi oczywiście o brak związania zmiennej y w drugim wywołaniu funkcji `format`.

W Lispie, podobnie jak w innych językach programowania, mamy do czynienia z przykrywaniem zmiennych (ang. *variable shadow*) w sytuacji gdy w „blokach wewnętrznych" definiowana jest taka sama zmienna jak w „blokach zewnętrznych". Potwierdzeniem tego jest np. poniższy program

```
(defun foo (x)                                ; scope
  (format t "Function:␣x=~a~%" x)             ; |<------ x is argument
  (let ((x 2))                                ; |
    (format t "Outer␣let:␣x=~a~%" x)          ; | |<---- x is 2
    (let ((x 3))                              ; | |
      (format t "Inner␣let:␣x=~a~%" x))       ; | | |<-- x is 3
    (format t "Outer␣let:␣x=~a~%" x))         ; | |
  (format t "Function:␣x=~a~%" x))            ; |
```

który zwraca następujące efekty

```
* (foo 1)
Function: x=1
Outer let: x=2
Inner let: x=3
Outer let: x=2
Function: x=1
NIL
```

Podobnie do `let` działa `let*` z tą różnicą, że w definicji zmiennych można odwołać się do zmiennych, które zostały już zdefiniowane. Tak więc można napisać

```
* (defun foo (x)
  (let* ((y 20)
         (z (+ y 30)))
    (list x y z)))
```

```
FOO
* (foo 10)

(10 20 50)
```

ale nie

```
* (defun foo (x)
  (let ((y 20)
         (z (+ y 30)))
    (list x y z)))

;      (+ Y 30)
;
; caught WARNING:
;   undefined variable: Y
;
; compilation unit finished
;   Undefined variable:
;     Y
;   caught 1 WARNING condition

FOO
* (foo 10)

debugger invoked on a UNBOUND-VARIABLE in thread #<THREAD "initial thread" RUNNING {AA5E8
  The variable Y is unbound.

Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.
```

```
(FOO 10)
0] 0
```

Aczkolwiek podobny efekt jak za pomocą `let*` można uzyskać przez wielokrotne wykorzystanie `let`

```
* (defun foo (x)
  (let ((y 20))
    (
      let ((z (+ y 30)))
      (list x y z)
    )
  )
)

FOO
* (foo 10)

(10 20 50)
```

Oczywiście wspomniane ograniczenie nie dotyczy zmiennych istniejących w momencie przetwarzania `let`, jak np. zmiennej `x` w poniższym przykładzie.

```
* (defun foo (x)
  (let ((y 20)
        (z (+ x 30)))
    (list x y z)))

FOO
* (foo 10)

(10 20 40)
```

16.2 Zmienne leksykalne

W Common Lisp występują dwa rodzaje zmiennych: leksykalne (ang. *lexical*) oraz dynamiczne (ang. *dynamic*). Domyślnie w Common Lispie wszystkie związania zmiennych mają zasięg leksykalny (ang. *lexically scoped variables*). Na podstawie dotychczasowego doświadczenia z imperatywnymi językami programowania jak C, Java czy Python, zasięg leksykalny zmiennych można wyjaśnić w następujący sposób: wszystkie odwołania mające miejsce w obszarze bloku w którym zdefiniowano zmienną*. W Lispie sprawa nie jest jednak taka prosta. Rozważmy taki fragment kodu

```
(let ((x 0)) #'(lambda () (setf x (1+ x))))
```

tutu – wyjaśnić W tym przypadku funkcja anonimowa nazywana jest domknieciem (ang. *closure*) ponieważ *it „closes over" the binding created by the* `let`†. Inne wyjaśnienie pochodzące z [19] to: *[closure] mean a subroutine that holds some memory but without some disadvantages of modifying a global variable*. Ogólnie domknięcie jest sposobem powiązania funkcji oraz środowisko w jakim ta funkcja ma działać. Środowisko przechowuje wszystkie obiekty wykorzystywane przez funkcję, nie będące dostępne w globalnym zakresie widoczności. Realizacja domknięcia jest zdeterminowana przez język, jak również przez kompilator. Niejako z definicji domknięcia występują głównie w językach funkcyjnych, w których funkcje mogą zwracać inne funkcje, wykorzystujące zmienne utworzone lokalnie. Na przykład w języku Python możemy napisać

```
def createMultiplyFunction(y):
    def funkcja(x):
        return x * y

    return funkcja

makeDouble = createMultiplyFunction(2)

print makeDouble(5)
```

Funkcja `podwojenie` oprócz argumentu formalnego `x` ma dostęp (poprzez domknięcie) do kopii wartości `y` przekazanej w wywołaniu `createMultiplyFunction` (w tym przykładzie `y` to liczba 2). Efektem wykonania `print makeDouble(5)` będzie wypisanie wartości 10.

*Nie jest to do końca precyzyjne wyjaśnienie, ale jedyne uniwerslane na jakie wpadł autor.
†Świadomie ten fragment jest w języku angielskim, jako, że tłumaczenie mogłoby zatrzeć sens zdania.

16.3 Zmienne dynamiczne

Zmienne lokalne, czyli zmienne, których zasięg działania ogranicza się jedynie do pewnego fragmentu kodu, dla którego mają znaczenie są całkiem dobrym pomysłem pozwalającym w jakimś sensie porządkować kod. Często jednak zachodzi konieczność użycia zmiennych o nieograniczonym w żaden sposób zasięgu i właściwie każdy język posiada taką „funkcjonalność”. W Lispie zmienne globalne nazywane są zmiennymi dynamicznymi (ang. *dynamic variable*) lub zmiennymi specjalnymi (ang. *special variable*).

W Lispie zmienne globalne możemy utworzyć na dwa sposoby: za pomocą `defvar` oraz `defparameter`. W obu przypadkach podajemy najpierw nazwę zmiennej, wartość początkową i opcjonalny opis.

```
(defvar *foo* 1
  "Very important global variable.")

(defparameter *bar* 2
  "Another very important global variable.")
```

W przypadku `defvar`, przypisanie wartości początkowej do zmiennej następuje tylko wtedy, gdy zmienna jest niezdefiniowana, natomiast używając `defparameter` przypisanie wartości początkowej do zmiennej ma miejsce zawsze. Ponad to `defvar` może zostać użyte bez wartości początkowej co powoduje utworzenie niezwiązanej zmiennej globalnej. Ze zdefiniowanych w jeden z podanych sposobów zmiennych można oczywiście korzystać w dalszej części programu (w dowolnym jego miejscu), np.

```
(defvar *foo* 1
  "Very important global variable.")

(defparameter *bar* 2
  "Another very important global variable.")

(defun fb () (+ *foo* *bar*))
```

i efekt działania

```
* (fb)

3
```

Mechanizm przykrywania zmiennych globalnych. — tutu wyjasnić problem — Ostatnio wykonane związanie zmiennej z wartością przykrywa wszystkie poprzednie związania. Spróbujemy zilustrować ten mechanizm następującym przykładem

```
* (defvar *x* 10)
(defun foo () (format t "x=~a " *x*))
*X*
* (foo)
x=10
NIL
* (let ((*x* 20)) (foo))
x=20
NIL
* (foo)
x=10
NIL
```

— tutu opisać ten przykład — Zamykając powyższy przykład w pojedyńczej funkcji,

```
(defun bar ()
  (foo)
  (let ((*x* 20)) (foo))
  (foo))
```

otrzymujemy analogiczny efekt

```
* (bar)
x=10 x=20 x=10
NIL
```

— tutu opisać ten przykład —

Rozbudujmy teraz trochę nasz przykład zmieniając definicję funkcji `foo`

```
(defun foo ()
  (format t "x=~a~%" *x*)
  (setf *x* (+ 1 *x*))
  (format t "x=~a~%" *x*))
```

Działanie samej funkcji `foo` nie jest raczej niczym niezwykłym

```
* (foo)
x=10
x=11
NIL
```

natomiast wywołanie `bar` daje dosyć ciekawe wyniki

```
* (bar)
x=11
x=12
x=20
x=21
x=12
x=13
NIL
```

— tutu opisać ten przykład —

16.4 Stałe

Stałe (ang. *constants*), często w mowie potocznej nazywane zmiennymi stałymi (ang. *constant variable*), definiujemy za pomocą `defconstant` o składni analogicznej do opisanego `defparameter`, tj. najpierw nazwa stałej, wartość początkową i opcjonalny opis.

16.5 Przypisania

Wiedząc już jak zmienne można utworzyć, naturalne są dwie dalsze operacje: odczytanie (wykorzystanie) obecnej wartości zmiennej i przypisanie do niej nowej wartości. Obie te operacje już wykonywaliśmy, gdyż bez nich trudno by było podać jakieś przykłady. Odniesienie się do wartości zmiennej następuje po prostu przez podanie jej nazwy. Nadanie zmiennej nowej wartości uzyskujemy za pomocą makra `setf` według schematu

```
(setf place value)
```

`setf` używa pierwszego swojego argumentu (`place`) do zdefiniowania miejsca w pamięci, w którym zapisuje efekt ewaluacji argumentu drugiego (`value`). Przypisanie do zmiennej `x` wartości 10 uzyskamy pisząc

```
(setf x 10)
```

Oczywiście zgodnie z zasadami opisanymi wcześniej, związanie zmiennej następujące w wyniku przypisania nie wpływa na związania w innym „bloku" programu. Dlatego `setf` w funkcji

```
(defun foo (x) (setf x 10))
```

w żaden sposób nie wpływa na otoczenie o czym przekonujemy się pisząc

```
* (let ((y 20))
  (foo y)
  (print y))
```

20

Takie zachowanie nie powinno nas jednak dziwić. W podobny sposób zachowa się każdy cywilizowany język programowania. Na przykład efektem działania ponizszego fragmentu kodu języka C

```
tutu uzupelnic
```

będzie

```
tutu uzupelnic
```

Dobrą wiadomością w codziennym użytkowaniu `setf` jest możliwość dokonania kilku przypisań. I tak zamist

```
(setf x 1)
(setf y 2)
```

wystarczy napisać

```
(setf x 1 y 2)
```

Ponieważ wartością `setf` jest wartość ostatniego przypisania, dlatego można napisać

```
* (setf x (setf y 1 z 2))

;     (SETF X (SETF Y 1 Z 2))
; ==>
;   (SETQ X (SETF Y 1 Z 2))
;
; caught WARNING:
;   undefined variable: X

;     (SETF Y 1 Z 2)
; --> PROGN SETF
; ==>
;   (SETQ Y 1)
;
; caught WARNING:
;   undefined variable: Y

; ==>
;   (SETQ Z 2)
;
; caught WARNING:
;   undefined variable: Z
;
; compilation unit finished
;   Undefined variables:
;     X Y Z
;   caught 3 WARNING conditions

2
* (print x)

2
```

```
2
* (print y)

1
1
* (print z)

2
2
*
```

Uogólnione przypisania

```
(setf x (+ x 1))

(setf x (- x 1))

(incf x)    === (setf x (+ x 1))
(decf x)    === (setf x (- x 1))
(incf x 10) === (setf x (+ x 10))
```

Zamiast pisać

```
(setf (aref *array* (random (length *array*)))
      (1+ (aref *array* (random (length *array*)))))
```

znacznie wygodniejszą formą jest

```
(incf (aref *array* (random (length *array*))))
```

ROZDZIAŁ 17

Makra

Makra są niejako warstwą abstrakcji nad rdzeniem Lispa stanowiąc w pewnym sensie interfejs dla niego. Tworząc nowe makra możemy w nieskończoność rozbudowywać funkcjonalność języka dostosowując go do naszych potrzeb nie ingerując jednocześnie w jego najbardziej elementarne mechanizmy. Okazuje się, co zobaczymy na początku, że konstrukcje znane z innch języków jako podstawowe, czy też tworzące właśnie jego rdzeń (jak np. instrukcja warunkowa) tutaj nie definiują języka (a przynajmniej nie muszą).

17.1 `if` oraz `when` i `unless`

```
(if condition then-form [else-form])
```

```
(if (> 2 3) "X" "Y") ==> "Y"
(if (> 2 3) "X")     ==> NIL
(if (> 3 2) "X" "Y") ==> "X"
```

single Lisp form

```
(if (condition)
    (form 1)
    (form 2))
```

```
(if (condition)
    (progn
```

```
      (form 1)
      (form 2)))

(when (condition)
  (form 1)
  (form 2))

(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))

(defmacro unless (condition &rest body)
  `(if (not ,condition) (progn ,@body)))
```

17.2 cond

Można tak

```
(if a
    (do-x)
    (if b
       (do-y)
       (do-z)))
```

tylko po co? A będzie jeszcze gorzej gdy w wyniku spełnienia pewnego warunku trzeba będzie wykonać kilka wyrażeń co wymusi użycie `progn`.

```
(cond
  (test-1 form*)
      .
      .
      .
  (test-N form*))
```

Stąd równoważnik powyższych „zagnieżdżonych” if-ów

```
(cond (a (do-x))
      (b (do-y))
```

```
    (t (do-z)))
```

17.3 and, or, not

```
(not nil)             ==> T
(not (= 1 1))         ==> NIL
(and (= 1 2) (= 3 3)) ==> NIL
(or (= 1 2) (= 3 3))  ==> T
```

17.4 dolist, dotimes

```
dolist

(dolist (var list-form)
  body-form*)

* (dolist (x (list 1 2 3)) (print x))

1
2
3
NIL
```

lub równoważnie

```
* (dolist (x '(1 2 3)) (print x))

1
2
3
NIL

* (dolist (x '(1 2 3)) (print x) (if (eq x 2) (return)))

1
```

```
2
NIL
```

```
dotimes
```

```
(dotimes (var count-form)
  body-form*)
```

```
* (dotimes (i 5) (print i))

0
1
2
3
4
NIL
```

17.5 do

Jak napisaliśmy wcześniej, `dolist` oraz `dotimes` to nic innego jak makra „opakowujące" niejako bardziej ogólne makro jakim jest `do`. `do` pozwala na związanie dowolnej ilości zmiennych i ich kontrolowanie podczas kolejnych iteracji. Ponadto możemy określić warunek końca pętli i wyrażenie którego wartość będzie zwracana po jej zakończeniu.

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

Każde `variable-definition*` to wyrażenie postaci

```
(var init-form step-form)
```

Część `step-form` jest opcjonalna – w przypadku jej braku zmienna musi zostać zmieniona jawnie wewnątrz pętli. W przypadku, gdy nie określono `init-form` zmienna ma wartość `nil`. Ewaluacja wyrażenia `end-test-form` w każdej iteracji następuje po przypisaniu nowych wartości do zmiennych. Tak długo jak jej wynikiem jest `nil` wykonywane jest `statement*`. Ewaluacja jego do wartości *true*,

pociąga za sobą ewaluację `result-form*` i zwrócenie wyniku ostatniego z jego wyrażeń jako wyniku pętli `do`.

Wyrażenia `step-form` obliczane są, w kolejności występowania, zanim nastąpi jakiekolwiek przypisanie, co można wykorzystać pisząc program obliczający dziesiąty wyraz ciągu Fibonacciego

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

Odpowiednik pętli `dotimes` wypisującej liczby całkowite od 0 do 4

```
(dotimes (i 5) (print i))
```

zapisany z wykorzystaniem pętli `do` może wygladać jak poniżej

```
(do ((i 0 (1+ i)))
    ((>= i 5))
  (print i))
```

17.6 `loop`

Podstawowa składnia pętli `loop` nie jest niczym zaskakującym

```
(loop
  body-form*)
```

Jest to chyba najprostsza z możliwych do wyobrażenia pętli iterująca po `body-form*` do momentu wykonania `return`. Dla przykładu odpowiednik wcześniejszej pętli `do` wypisującej liczby od 0 do 4 może wyglądać tak

```
petla loop wypisujaca liczby od 0 do 5 --> cwiczenia
```

Jednak to rozszerzona składnia tej petli ukazuje z czym tak naprawdę mamy do czynienia. Pętla `loop` jest bowiem dosyć nietypowym „tworem” natomiast idealnym przykładem ukazującym możliwości makr. W pewnym sensie `loop` zrywa ze standardową składnią Lispa co skutkuje tym, że nie przez wszystkich jest ona akceptowana. Mówiąc najogólniej `loop` zamiast „*mnóstwa głupich nawiasów*” operuje na czytelnym, praktycznie dla każdego, sposobie zapisu. Dzięki temu zadania takie jak np.

zliczanie, sumowanie czy przejście po wszystkich elementach listy roalizuje się „od ręki”. Aby nie być gołosłownym, podajmy kilka przykładów.

- Utworzenie listy zawierającej liczby całkowite od 1 do 10

```
* (loop for i from 1 to 10 collecting i)

(1 2 3 4 5 6 7 8 9 10)
```

- Sumowanie pierwszych 10 wartości wyrażenia

```
* (loop for x from 1 to 10 summing (expt x 2))

385
```

- Zliczanie ilości wystąpień litery „o” w napisie*

```
* (loop for x across "The quick brown fox jumps over the lazy dog"
      counting (find x "o"))

4
```

- Obliczenie dziesiątego wyrazu ciągu Fibonacciego

```
(loop for i below 10
      and a = 0 then b
      and b = 1 then (+ b a)
      finally (return  a))
```

W ten sposób `loop` ukazuje potęgę makr. W tym przypadku użyto makra w celu rozszerzenia podstawowej składni języka, w żaden sposób nie wpływając na jego pozostałe cechy. Słowa kluczowe pętli `loop` analizowane są zgodnie z jej specyficzną składnią, ale pozostała część to nic innego ja typowe wyrażenia Lispa. Co także istotne `loop` cały czas pozostaje „jedynie” makrem – nie jest rdzeniem (czy w rdzeniu) języka.

*Tekst „*The quick brown fox jumps over the lazy dog*” jest chyba angielskim odpowiednikiem polskiego „*Ala ma kota*” i jest tak samo często używany jak wspomniane już wcześniej *foo* i *bar*.

17.7 Zrozumieć makra

Makro definiujemy za pomocą `defmacro`

```
(defmacro name (parameter*)
  "Optional documentation string."
  body-form*)
```

Podobnie jak w `defun` mamy tutaj nazwę, listę argumentów, opcjonalny opis i ciało złożone z wyrażeń. Choć programy wywołują makra dokładnie w taki sam sposób jak funkcje to jednak ich zachowanie (makr) jest zupełnie inne. Ciało nie zwraca wartości, ale wyrażenie języka Lisp, które w dalszej części będzie podlegało ewaluacji. Argumenty makra nie podlegają ewalucji. Dlatego przekazując do makra listę `(* 2 (+ 3 4))`, argument w ciele makra będzie zawsze listą `(* 2 (+ 3 4))` a nie wartością 14. Poniższe proste makro[†]

```
(defmacro square (x)
  `(* ,x ,x))
```

można rozumieć w następujący sposób: za każdym razem, gdy preprocesor napotka w kodzie zapis `(square X)` dokonuje jego zastąpienia przez `(* X X)`.

17.7.1 Rozwinięcie i wykonanie makra

Kluczem do zrozumienia makr jest świadomość różnicy pomiędzy generowaniem kodu (czy też kodem generującym kod) a wykonaniem kodu (programu). Pisząc makro, piszemy program, który zostanie wykorzystany przez kompilator w celu wygenerowania kodu, który zostanie skompilowany. Program zostanie wykonany tylko wtedy, gdy wszystkie makra zostaną rozwinięte i zastąpione kodem podlegającym kompilacji. Moment działania makr nazywany jest własnie rozwijaniem makr (ang. *macro expansion time*) dla odróżnienia od momentu wykonywania kodu zarówno tego napisanego „recznie" jak i wygenerowanego przez makra (ang. *runtime*). Rozróżnienie to jest o tyle istotne, że podczas rozwijania makra nie ma możliwości operowania danymi, które będą istniały podczas wykonania.

Program

```
(defun foo (x)
  (when (> x 10) (print 'big)))
```

[†]Uwaga na zapis: w makrze użyto nie znaku quote ('), ale backtick (`).

spowoduje wywołanie makra

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

które wygeneruje następujący kod

```
(if (> x 10) (progn (print 'big)))
```

— tutu opisać —

Istotne jest aby pamiętać, że Lisp (zależnie od implementacji) może rozwijać makra w różnym czasie.

- Makro może być rozwijane tylko raz, podczas kompilacji programu.
- Makro może być rozwijane podczas pierwszego użycia.
- Makro może być rozwijane za każdym razem gdy jest ono używane.

Dobrze napisane makro powinno zadziałać w każdej z powyższych sytuacji.

17.7.2 Etapy tworzenia makra

1. Napisanie przykładowego wywołania makra oraz kodu do jakiego makro powinno zostać rozwinięte.
2. Napisanie kodu makra działającego dla przykładowego wywołania i porównanie efektu jego rozwięcia z kodem napisanym w poprzednim kroku.
3. Sprawdzić, czy makro faktycznie działa poprawnie.

17.7.3 Wzorzec podstawienia

```
* '(suma liczb 3 i 5 to (+ 3 5))

(SUMA LICZB 3 I 5 TO (+ 3 5))

* `(suma liczb 3 i 5 to ,(+ 3 5))

(SUMA LICZB 3 I 5 TO 8)
```

```
* (defmacro repeat-code (times &body body)
    `(dotimes (x ,times)
       ,@body))

REPEAT-CODE
* (macroexpand-1 '(repeat-code 3 (print "ala ma kota") (print "i psa")))

(DOTIMES (X 3) (PRINT "ala ma kota") (PRINT "i psa"))
T
* (repeat-code 3 (print "ala ma kota") (print "i psa"))

"ala ma kota"
"i psa"
"ala ma kota"
"i psa"
"ala ma kota"
"i psa"
NIL
* (defmacro repeat-code (times &body body)
    `(dotimes (x ,times)
       ,body))

REPEAT-CODE
* (macroexpand-1 '(repeat-code 3 (print "ala ma kota") (print "i psa")))

(DOTIMES (X 3) ((PRINT "ala ma kota") (PRINT "i psa")))

T
* (repeat-code 3 (print "ala ma kota") (print "i psa"))
; in: LAMBDA NIL
;     ((PRINT "ala ma kota") (PRINT "i psa"))
;
```

```
; caught ERROR:
;   illegal function call
;
; compilation unit finished
;   caught 1 ERROR condition
...
```

Wiedząc to, możemy teraz spróbować napisać program, którego nie udało się nam napisać na ostatnich ćwiczeniach, a mianowicie kod wykonujący operację zamiany zmiennych.

```
* (defmacro swap (x y)
  `(let ((temp ,x))
     (setf ,x ,y)
     (setf ,y temp)))

SWAP
* (let ((x 3)
        (y 7))
    (swap x y) ; macro call
    (list x y))

(7 3)
* (macroexpand-1 '(swap x y))

(LET ((TEMP X))
  (SETF X Y)
  (SETF Y TEMP))
T
* (macroexpand-1 '(swap 3 5))

(LET ((TEMP 3))
  (SETF 3 5)
  (SETF 5 TEMP))
```

```
T
* (macroexpand-1 '(swap a 5))

(LET ((TEMP A))
  (SETF A 5)
  (SETF 5 TEMP))
T
```

— tutu znaczenie at —

17.7.4 Problemy związane z makrami

Najczęściej spotykane problemy związane z makrami dotyczą zwykle ich implementacji i co ciekawe występują w makrach, które wydają się działać poprawnie.

1. Próba ewaluacji argumentów podczas wykonania.
2. Wielokrotna ewaluacja argumentów.
3. Konflikt nazw.

Próba ewaluacji argumentów podczas wykonania

Makra rozwijane są w trakcie pracy preprocesora i dlatego ponieważ wartości zmiennych nie są dostępne, to kod na nich bazujący nie bedzie działał poprawnie. To jest powodem, dla którego makro

```
(defmacro square (X)
  (* X X))
```

```
* (macroexpand-1 '(square 3))

9
T
* (macroexpand-1 '(square x))

debugger invoked on a SIMPLE-TYPE-ERROR in thread #<THREAD "initial thread" RUNNING {AA5E
  Argument X is not a NUMBER: X
```

```
Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.
```

zadziała dla wywołania `(square 3)`, ale będzie powodem błędów dla `(square x)`. Poprawna postać tego makra jest następująca

```
(defmacro square (X)
  `(* ,X ,X))
```

```
* (macroexpand-1 '(square 3))

(* 3 3)
T
* (macroexpand-1 '(square x))

(* X X)
T
```

Wielokrotna ewaluacja argumentów

```
* (defmacro square (x)
  `(* ,x ,x))

SQUARE
* (square 3)

9
* (macroexpand-1 '(square x))

(* X X)
T
* (macroexpand-1 '(square 3))

(* 3 3)
T
```

```
* (macroexpand-1 '(square (foo 2)))

(* (FOO 2) (FOO 2))
T
* (macroexpand-1 '(square (rand 100)))

(* (RAND 100) (RAND 100))
T
```

Dlatego makro square powinno być raczej napisane w następujący sposób

```
(defmacro square-improved (x)
  `(let ((temp ,x))
     (* temp temp)))
```

dzięki czemu działanie zgodne jest z oczekiwaniem

```
* (macroexpand-1 '(square-improved (rand 100)))

(LET ((TEMP (RAND 100)))
  (* TEMP TEMP))
```

Konflikt nazw

```
(defmacro repeat-code (times &body body)
    `(dotimes (x ,times)
       ,@body))
```

```
* (repeat-code 3 (print "ala ma kota"))

"ala ma kota"
"ala ma kota"
"ala ma kota"
NIL
* (macroexpand-1 '(repeat-code 3 (print "ala ma kota")))

(DOTIMES (X 3) (PRINT "ala ma kota"))
```

```
T
* (defparameter text "reksio")

TEXT
* (repeat-code 3 (print text))

"reksio"
"reksio"
"reksio"
NIL
* (macroexpand-1 '(repeat-code 3 (print text)))

(DOTIMES (X 3) (PRINT TEXT))
T
* (defparameter x "filemon")

X
* (repeat-code 3 (print x))

0
1
2
NIL
* (macroexpand-1 '(repeat-code 3 (print x)))

(DOTIMES (X 3) (PRINT X))
T
```

```
(defmacro repeat-code (times &body body)
  (let ((x (gensym "foo-")))
    `(dotimes (,x ,times)
       ,@body)))
```

```
* (repeat-code 3 (print x))

"filemon"
"filemon"
"filemon"
NIL
* (macroexpand-1 '(repeat-code 3 (print x)))

(DOTIMES (#:|foo-687| 3) (PRINT X))
T
```

ROZDZIAŁ 18

Listy

Listy stanowią pewnego rodzaju abstrakcję nad instancjami obiektów pogrupowanych w pary nazywane ang. *cons cells* jako, że do ich utworzenia wykorzystywana jest funkcja `cons`.

```
* (cons 1 2)

(1 . 2)
```

Dwa obiekty tworzące komórkę `cons` nazywane są `car` i `cdr`

```
* (car (cons 1 2))

1
* (cdr (cons 1 2))

2

* (defparameter *cons* (cons 1 2))

*CONS*
* *cons*

(1 . 2)
* (setf (car *cons*) 10)
```

```
10
* *cons*

(10 . 2)
* (setf (cdr *cons*) 20)

20
* *cons*

(10 . 20)

* (cons 1 nil)

(1)
* (cons 1 (cons 2 nil))

(1 2)
* (cons 1 (cons 2 (cons 3 nil)))

(1 2 3)
```

Zauwazmy, że

```
* (cons 1 2)

(1 . 2)
* (cons 1 (cons 2 nil))

(1 2)
```

Diagramy

```
|CAR|CDR| -> ...
```

```
| | |->|3| |->|4|nil|
 \
  \-->|1| |->|2|nil|

* (cons 1 2)

(1 . 2)
* (cons 1 (cons 2 nil))

(1 2)
* (cons 1 (cons 3 (cons 4 nil)))

(1 3 4)
* (cons (cons 1 2) (cons 3 (cons 4 nil)))

((1 . 2) 3 4)
* (cons (cons 1 (cons 2 nil)) (cons 3 (cons 4 nil)))

((1 2) 3 4)
* (defparameter *list2* (cons (cons 1 (cons 2 nil)) (cons 3 (cons 4 nil))))

*LIST2*
* (car *list2*)

(1 2)
* (cdr *list2*)

(3 4)
```

Dla odróżnienia lista zawierająca dwie listy, czyli lista, która po „narysowaniu" wygląda jak poniżej

```
| | |->| |nil|
 |       |
 |       |--|3| |->|4|nil|
```

```
 |
 |-->|1| |->|2|nil|

* (defparameter *list3* (cons (cons 1 (cons 2 nil)) (cons (cons 3 (cons 4 nil)) nil)))

((1 2) (3 4))
* (defparameter *list3* (cons (cons 1 (cons 2 nil)) (cons (cons 3 (cons 4 nil)) nil)))

*LIST3*
* (car *list3*)

(1 2)
* (cdr *list3*)

((3 4))
```

Część IV

Lisp

Ćwiczenia

(defun f (x) (* 2 x)) (defun g (x) (if (¿= x 0) x (- x))) (defun silnia1(n) (if (= n 0) 1 (* n (silnia1 (- n 1))))) (defun silnia2(n a) (if (= n 0) a (silnia2 (- n 1) (* a n)))) (null '(a b)) (null '()) (defun dllist(lst a) (if (null lst) 0 (dllist (cdr lst) (+ 1 a))))

fibonaci silnia iloczyn dwoch modul (wartosc bezwzlgedna) pierwiastki trojmianu sasiedztwo elementow na liscie

Część V

Haskell

Część VI

Haskell

Ćwiczenia

ROZDZIAŁ 19

Proste zadania

1. Funkcja zwracająca ilość elementów na liście.
2. Funkcja powielająca elementy listy.
3. Funkcja zwracająca przedostatni element listy.
4. Przyjmując, że pierwszy element listy ma indeks 0, znajdź k-ty jej element.
5. Funkcja zastępująca element powtarzajacy się pojedyńczym jego wystąpieniem.
6. Funkcja zwracająca podlistę listy.
7. Funkcja wstawiająca zadany element na określoną pozycję w liście.

Część VII

Jess

ROZDZIAŁ 20

Wprowadzenie

Jak nie trudno się zorientować, cały rozdział o Jess czeka na napisanie – proszę o cierpliwość.

20.1 Czym jest silnik regułowy?

20.2 Jess – historia

20.3 Instalacja, sposoby wykorzystania, wydajność

```
http://www.jessrules.com/
```

```
java -classpath jess.jar jess.Main
```

In 1994, I was working in the Scientific Computing department at Sandia National Laboratories in Livermore, California. We had an impressive (for the time) array of heterogeneous computing equipment

I was writing software agents that managed dynamically distributed computations across this network. Agents were running on each machine, and they used a sort of "post and bid" method to decide which machines would run which piece of a computation, based on machine capabilities and load balancing.

Their "brains" were rule engines—software systems that used rules to derive conclusions from premises.

That project led to others, and soon I developed an interest in mobile agents— software entities that can travel from node to node in a computer network, main- taining their state as they go. Thus was born the idea for a rule engine whose state could be packaged up, sent across a wire, and reconstituted. The newly released Java language seemed to be a perfect vehicle for this rule engine—and such was the origin of JessTM, the rule engine for the Java Platform.1 Jess is a general-purpose rule engine, developed at Sandia National Laborato- ries. Written in the Java programming language, Jess offers easy integration with other Java-based software.

Jess has evolved quite a bit since its original introduction in 1997,

20.4 Part I: Introducing rule-based systems

What are rule-based systems? What are they good for? Where did they come from? Are they right for you? What should you do if you want to build one?

20.4.1 Typical unalgorithmic problem

* A foursome of golfers is standing at a tee, in a line from left to right. Each golfer wears different colored pants; one is wearing red pants. The golfer to Fred's immediate right is wearing blue pants. * Joe is second in line. * Bob is wearing plaid pants. * Tom isn't in position one or four, and he isn't wearing the hideous orange pants. * In what order will the four golfers tee off, and what color are each golfer's pants?"

To solve problem you're going to: 1 Choose a way to represent the possible combinations of men's names, positions, and pants colors. 2 Write one rule that describes the problem.

Datastructures (5)

The first step is to define data structures to represent the smallest useful pieces of a possible solution to the problem: a link between a name and either a position or a color: (deftemplate pants-color (slot of) (slot is)) (deftemplate position (slot of) (slot is))

Whereas a Java class is a definition of a type of object, a template is a definition for a type of fact

Fact and rules (5)

20.5 What are rule-based systems? (część 1, rozdzial 2 (13)

There's an old saying that "when all you've got is a hammer, everything looks like a nail." In computer programming, too, it's important to choose the right tool for the right job. Some problems can be solved easily using traditional pro- gramming techniques, whereas writing a rule-based system is the easiest way to solve others. Other problems are in the middle, and either technique will work equally well. This chapter will help you understand which problems are well suited to being solved with a rule-based system, and what this type of software is all about.

20.5.1 Programowanie proceduralne (imperatywne) vs. deklaratywne

This time, the instructions are filled with many context-sensitive decisions— in fact, there are more decisions than actions. The control flow includes many loops and branches. It would be next to impossible to write a single list of instructions covering every situation that might possibly arise, considering that circumstances interact to constantly change the correct outcomes of each decision. For example, seeing a child's ball roll into the road is not a serious situation when the car isn't moving, but it requires prompt and decisive action when the car is driving toward the bouncing ball. If it's impossible to write a procedural program to control your robot, how can you do it? You can use declarative programming instead.

Procedural programming

- In procedural programming, the programmer tells the computer **what to do, how to do it, and in what order.**
- Procedural programming is **well suited for problems in which the inputs are well specified and for which a known set of steps can be carried out to solve the problem.** Mathematical computations, for example, are best written procedurally.

Declarative programming

- A purely declarative program, in contrast, **describes what the computer should do, but omits much of the instructions on how to do it.**
- In a rule-based program, you write only the individual rules.

- Another program, the rule engine, determines which rules apply at any given time and executes them as appropriate.
- As a result, a rule-based version of a complex program can be shorter and easier to understand than a procedural version.
- Writing and modifying the program is simpler, because you can concentrate on the rules for one situation at a time.
- A rule is a kind of instruction or command that applies in certain situations. (2.2 (s. 17))
- The domain of a rule is the set of all information the rule could possibly work with.
- A rule-based system is a system that uses rules to derive conclusions from premises.

20.6 Architecture of a rule-based system (część 1, rozdzial 2.3 (s. 19)

A typical rule engine contains:

- Blok wnioskowania (An inference engine) The inference engine controls the whole process of applying the rules to the working memory to obtain the outputs of the system.
  - A pattern matcher (Your inference engine has to decide what rules to fire, and when. The purpose of the pattern matcher is to decide which rules apply, given the current contents of the working memory.)
  - An agenda (Once your inference engine figures out which rules should be fired, it still must decide which rule to fire first. The list of rules that could potentially fire is stored on the agenda.)
  - An execution engine
- Baza reguł (A rule base)
- Przestrzeń roboczą (A working memory)

ROZDZIAŁ 2

Jess – wiadomości podstawowe

21.1 Składnia – podstawowe elementy

Formatowanie

Jak w większości współczesnych języków programowania sposób formatowania kodu nie ma żadngo znaczenia syntaktycznego i semantycznego. Tak więc można pisać zarówno w taki sposób

```
( t u t u )
```

jak i taki

```
( t u t u )
```

Ponieważ jednak składnia Jess bazuje na składni Lispa, więc preferowaną formą zapisu jest sposób drugi.

Dopuszczalne znaki

Liczby

Napis

Komentarze

21.2 Listy

Wywołania funkcji

21.3 Zmienne

Zmienne

```
Jess> (bind ?x "Ala ma kota")
"Ala ma kota"
```

```
Jess> (bind ?x (+ 2 3))
5
Jess> ?x
5
Jess> (+ ?x 4)
9
```

Zmienne globalne

```
(defglobal [?*<nazwa zmiennej globalnej>* = <wartość>]+)
```

```
Jess> (defglobal ?*x* = 3)
TRUE
Jess> ?*x*
3
Jess> (bind ?*x* 4)
4
Jess> ?*x*
4
```

```
Jess> (reset) ; Jess przywróci początkową wartość ?*x*, jaką było 3
TRUE
Jess> ?*x*
3
Jess> (bind ?*x* 4)
4
Jess> (set-reset-globals nil)
FALSE
Jess> (reset) ; ?*x* pozostanie niezmienione
TRUE
Jess> ?*x*
4
```

Zmienne wielowartościowe

Zmienne wielowartościowe (ang. *multifield*)

```
$?x
```

Szczególne znaczenie ma tylko w przypadku wykorzystania w definicji argumentów funkcji lub po lewej stronie reguły.

21.4 Sterowanie przepływem

21.4.1 foreach

```
(foreach <variable> <list> <expression>+)
```

```
Jess> (bind ?grocery-list (create$ eggs milk bread))
(eggs milk bread)
Jess> (foreach ?e ?grocery-list
    (printout t ?e crlf))
eggs
milk
bread
```

21.4.2 while

```
(while <Boolean expression> do <expression>+)
```

```
Jess> (bind ?i 1)
1
Jess> (bind ?sum 0)
0
Jess> (while (<= ?i 10) do
    (bind ?sum (+ ?sum ?i))
    (bind ?i (+ ?i 1)))
FALSE
Jess> ?sum
55
```

21.4.3 if-then-else

```
(if <Boolean expression> then <expression>+ [else <expression>+])
```

```
Jess> (bind ?grocery-list (create$ eggs milk bread))
(eggs milk bread)
Jess> (if (member$ eggs ?grocery-list) then
      (printout t "I need to buy eggs" crlf)
    else
      (printout t "No eggs, thanks" crlf))
I need to buy eggs
```

21.4.4 progn

```
(if <Boolean expression> then <expression>+ [else <expression>+])
```

```
Jess> (bind ?grocery-list (create$ eggs milk bread))
(eggs milk bread)
Jess> (if (member$ eggs ?grocery-list) then
      (printout t "I need to buy eggs" crlf)
    else
```

```
      (printout t "No eggs, thanks" crlf))
I need to buy eggs
```

21.4.5 progn

```
(progn <expression>+)
```

```
Jess> (bind ?n 2)
2
Jess> (while (progn (bind ?n (* ?n ?n)) (< ?n 1000)) do
    (printout t ?n crlf))
4
16
256
FALSE
```

21.4.6 apply

```
(apply <function-name> <expression>+)
```

```
Jess> (apply (read) 1 2 3)
+
6
```

21.4.7 eval/build

```
(apply <function-name> <expression>+)
```

```
Jess> (bind ?x "(+ 1 2 3)")
"(+ 1 2 3)"
Jess> (eval ?x)
6
```

21.5 Funkcje

21.5.1 Definiowanie funkcji

```
(deffunction <name> (<parameter>*) [<comment>] <expression>*)
```

```
Jess> (deffunction distance (?X1 ?Y1 ?Z1 ?X2 ?Y2 ?Z2)
      "Compute the distance between two points in 3D space"
      (bind ?x (- ?X1 ?X2))
      (bind ?y (- ?Y1 ?Y2))
      (bind ?z (- ?Z1 ?Z2))
      (bind ?distance (sqrt (+ (* ?x ?x) (* ?y ?y) (* ?z ?z))))
      (return ?distance))
TRUE
Jess> (distance 10 0 0 2 0 0)
8
```

Wykorzystanie zmiennej wielowartościowej

```
Jess> (deffunction min ($?args)
     "Compute the smallest of a list of positive numbers"
     (bind ?minval (nth$ 1 ?args))
     (foreach ?n ?args
(if (< ?n ?minval) then
    (bind ?minval ?n)))
     (return ?minval))
TRUE
Jess> (min 10 100 77 6 43)
6
```

21.5.2 Zmiana zachowania funkcji

`defadvice before ?argv after ?retval undefadvice`

ROZDZIAŁ 22

Java w Jess

22.1 Tworzenie obiektów

```
Jess> (bind ?prices (new java.util.HashMap))
<External-Address:java.util.HashMap>
```

lub

```
Jess> (import java.util.*)
TRUE
Jess> (bind ?prices (new HashMap))
<External-Address:java.util.HashMap>
```

22.2 Wywoływanie metod

```
Jess> (call ?prices put bread 0.99)
Jess> (call ?prices put peas 1.99)
Jess> (call ?prices put beans 1.79)
Jess> (call ?prices get peas)
1.99
```

```
(call ?prices put bread 0.99)
```

```
(?prices put bread 0.99)
```

```
((bind ?prices (new HashMap)) put bread 0.99)
```

When you call a static method, you must include the call function name

```
Jess> (call Thread sleep 1000)
(pause for one second)
Jess>
```

```
Jess> (bind ?b (new javax.swing.JButton))
<External-Address:javax.swing.JButton>
Jess> (?b setText "Press Me") ;; or...
Jess> (set ?b text "Press Me")
Jess> (?b getText ) ;; or...
"Press Me"
Jess> (get ?b text)
"Press Me"
```

22.3 Tablice

```
Jess> (bind ?grocery-list ((?prices keySet) toArray))
(bread peas beans)
Jess> (import javax.swing.JComboBox)
Jess> (bind ?jcb (new JComboBox ?grocery-list))
<External-Address:javax.swing.JComboBox>
```

22.4 Dostep do publicznych składowych klasy

```
Jess> (bind ?pt (new java.awt.Point))
<External-Address:java.awt.Point>
Jess> (set-member ?pt x 37)
37
Jess> (set-member ?pt y 42)
42
Jess> (get-member ?pt x)
37
```

22.5 Obsługa wyjątków

```
Jess> (deffunction parseInt (?string)
    (try
       (bind ?i (call Integer parseInt ?string))
       (printout t "The answer is " ?i crlf)
     catch
       (printout t "Invalid argument" crlf)))
TRUE
Jess> (parseInt "10")
                           Lowercase \ell",
The answer is 10
                           uppercase \oh"
Jess> (parseInt "l0")
Invalid argument

Jess> (import java.io.*)
TRUE
Jess> (bind ?file nil)
Jess> (try
          (bind ?file
              (new BufferedReader
                  (new java.io.FileReader "data.txt")))
          (bind ?file
          (while (neq nil (bind ?line (?file readLine)))
              (printout t ?line crlf))
       catch
```

Licensed to Piotr Fulmanski <fulmanp@math.uni.lodz.pl>


```
              (printout t "Error processing file" crlf)
           finally
              (if (neq nil ?file) then
                  (?file close)))
```

```
Error processing file
```

ROZDZIAŁ 23

Fakty

23.1 czym jest fakt

Przykłady prostych faktów

```
(initial-fact)
(little red Corvette)
(groceries milk eggs bread)
```

23.2 Manipulacje na przestrzeni roboczej

watch

```
Jess> (watch facts)
TRUE
Jess> (reset)
==> f-0 (MAIN::initial-fact)
TRUE
Jess> (unwatch facts)
TRUE
```

facts

```
Jess> (facts)
```

```
f-0 (MAIN::initial-fact)
For a total of 1 facts.
```

assert

```
Jess> (reset)
TRUE
Jess> (assert (groceries milk eggs bread))
<Fact-1>
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::groceries milk eggs bread)
For a total of 2 facts.
```

retract

```
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::groceries milk eggs bread)
For a total of 2 facts.
Jess> (retract 1)
TRUE
Jess> (facts)
f-0 (MAIN::initial-fact)
For a total of 1 facts.
Jess> (bind ?f (fact-id 0))       Fetch Fact object
<Fact-0>                          with fact-id 0
Jess> (retract ?f)
TRUE
Jess> (facts)
For a total of 0 facts.
```

`reset` **i** `clear`

`deffacts`

```
Jess> (clear)
TRUE
Jess> (deffacts catalog "Product catalog"
    (product 354 sticky-notes "$1.99")
    (product 355 paper-clips "$0.99")
    (product 356 blue-pens "$2.99")
    (product 357 index-cards "$0.99")
    (product 358 stapler "$5.99"))
TRUE
Jess (facts)
For a total of 0 facts.
Jess> (reset)
TRUE
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::product 354 sticky-notes "$1.99")
f-2 (MAIN::product 355 paper-clips "$0.99")
f-3 (MAIN::product 356 blue-pens "$2.99")
f-4 (MAIN::product 357 index-cards "$0.99")
f-5 (MAIN::product 358 stapler "$5.99")
For a total of 6 facts.
```

23.3 Różne rodzaje faktów

23.3.1 Luźne fakty

```
Jess> (deftemplate person "People in actuarial database"
    (slot name)
    (slot age)
    (slot gender))
```

```
TRUE
Jess> (assert (person (age 34) (name "Bob Smith")
                (gender Male)))
<Fact-1>
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::person (name "Bob Smith") (age 34)
    (gender Male))
For a total of 2 facts.
```

```
Jess> (assert (person (age 30) (gender Female)))
<Fact-2>
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::person (name "Bob Smith") (age 34)
    (gender Male))
f-2 (MAIN::person (name nil) (age 30)
    (gender Female))
For a total of 3 facts.
```

```
Jess> (clear)
TRUE
Jess> (deftemplate person "People in actuarial database"
    (slot name (default "OCCUPANT"))
    (slot age)
    (slot gender))
TRUE
```

```
Jess> (assert (person (age 30) (gender Female)))
<Fact-0>
Jess> (facts)
f-0 (MAIN::person (name "OCCUPANT") (age 30)
    (gender Female))
For a total of 1 facts.
```

```
Jess> (clear)
TRUE
Jess> (deftemplate person "People in actuarial database"
    (slot name (default OCCUPANT))
    (slot age)
    (slot gender)
    (multislot hobbies))
TRUE
Jess> (assert (person (name "Jane Doe") (age 22)
    (hobbies snowboarding "restoring antiques")
    (gender Female)))
<Fact-0>

Jess> (modify 0 (age 23))
<Fact-0>
Jess> (facts)
f-0 (MAIN::person (name "Jane Doe") (age 23)
    (gender Female)
    (hobbies snowboarding "restoring antiques"))
For a total of 1 facts.

Jess> (duplicate 0 (name "John Doe") (gender Male))
<Fact-1>
Jess> (facts)
f-0 (MAIN::person (name "Jane Doe") (age 23)
    (gender Female) (hobbies snowboarding "restoring antiques"))
f-1 (MAIN::person (name "John Doe") (age 23)
    (gender Male) (hobbies snowboarding "restoring antiques"))
For a total of 2 facts.
```

23.3.2 Fakty uporządkowane

23.3.3 Fakty zakryte

ROZDZIAŁ 24

Reguły

Najprostsza reguła

```
Jess> (defrule null-rule
  "Najprostsza regula"
  =>
  )
TRUE

Jess> (defrule change-baby-if-wet
     "If baby is wet, change its diaper."
     ?wet <- (baby-is-wet)
     =>
     (change-baby)
     (retract ?wet))
TRUE

Jess> (clear)
TRUE
Jess> (watch all)
TRUE
Jess> (reset)
==> f-0 (MAIN::initial-fact)
```

```
TRUE
Jess> (deffunction change-baby ()
     (printout t "Baby is now dry" crlf))
TRUE
Jess> (defrule change-baby-if-wet
     "If baby is wet, change its diaper."
     ?wet <- (baby-is-wet)
     =>
     (change-baby)
     (retract ?wet))
change-baby-if-wet: +1+1+1+t
TRUE
Jess> (assert (baby-is-wet))
==> f-1 (MAIN::baby-is-wet)
==> Activation: MAIN::change-baby-if-wet : f-1
<Fact-1>
Jess> (run)
FIRE 1 MAIN::change-baby-if-wet f-1
Baby is now dry
<== f-1 (MAIN::baby-is-wet)
1


Jess> (clear)
TRUE
Jess> (defrule literal-values
     (letters b c)
     =>)
TRUE
Jess> (watch activations)
TRUE
Jess> (assert (letters b d))      This doesn't activate the rule...
<Fact-0>
```

```
                                    ... but this does
Jess> (assert (letters b c))
==> Activation: MAIN::literal-values: f-1
<Fact-1>
```

```
    Jess> (defrule simple-variables
         (a ?x ?y)
         =>
         (printout t "Saw 'a " ?x " " ?y "'" crlf))
```

```
    Jess> (defrule repeated-variables
         (a ?x)
         (b ?x)
         =>
         (printout t "?x is " ?x crlf))
TRUE
Jess> (watch activations)
TRUE
Jess> (deffacts repeated-variable-facts
     (a 1)
     (a 2)
     (b 2)
     (b 3))
TRUE
Jess> (reset)
==> Activation: MAIN::repeated-variables : f-2, f-3
TRUE
Jess> (run)
?x is 2
1
```

```
  (defrule any-shopping-cart
      (shopping-cart (contents $?items))
      =>
```

```
    (printout t "The cart contains " ?items crlf))

  (defrule cart-containing-milk
    (shopping-cart (contents $?before milk $?after))
    =>
    (printout t "The cart contains milk." crlf))
```

```
Jess> (defrule small-order
    (shopping-cart (customer-id ?id)
                      (contents $?c&:(< (length$ $?c) 5)))
      (checking-out-now ?id)
      =>
      (printout t "Wouldn't you like to buy more?" crlf))
   TRUE
```

```
Jess> (defrule large-order-and-no-dairy
     (shopping-cart (customer-id ?id)
                   (contents $?c&
                    :(and (> (length$ $?c) 50)
                          (not (or (member$ milk $?c)
                                   (member$ butter $?c))))))
     (checking-out-now ?id)
     =>
     (printout t "Don't you need dairy products?" crlf))
TRUE
```

```
   (item (price ?x))
   (item (price ?y&:(eq ?y (* ?x 2))))
```

lub

```
   (item (price ?x))
   (item (price =(* ?x 2)))
```

ROZDZIAŁ 25

Przykłady

W rozdziale tym spróbujemy zebrać wszystko to, czego dowiedzieliśmy się do tej pory, podając przykładowe rozwiązania pewnych zadań.

25.1 Rebus

W rozdziale 7 rozwiązywaliśmy rebus, w którym należało znaleźć takie przyporządkowanie cyfr do liter aby poniższy zapis

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

był prawdziwym działaniem matematycznym. Przyporządkowanie jest różnowartościowe, tzn. każdej literze odpowiada inna cyfra. Jak wiemy, rozwiązanie tego rebusu przedstawia się następująco

```
A, B, D, E, G, L, N, O, R, T
4, 3, 5, 9, 1, 8, 6, 2, 7, 0

   526485
 + 197485
=========
 = 723970
```

Spróbujmy teraz rozwiązać ten sam problem, ale korzystając z Jess.

25.1.1 Rozwiązanie 1

Na początek przykładowe rozwiązanie podobnego problemu, mianowicie

```
    TWO
 +  TWO
========
 = FOUR
```

W tym przypadku istnieje wiele rozwiązań, np.

```
Jess> (solve)
T:4
W:8
O:9
Jess> (solve)
T:3
W:7
O:7
Jess> (solve)
T:3
W:3
O:6
Jess> (solve)
T:3
W:9
O:7
Jess> (solve)
T:4
W:5
O:9
```

Podane wyniki wygenerowane zostały przez poniższy program.

```
(deffunction ttf (?t ?w ?o)
"Funkcja␣sprawdzajaca␣czy␣dana␣trojka␣liczb␣rozwiazuje␣problem"
(bind ?f 0)
(bind ?u 0)
(bind ?r 0)
(bind ?r (+ ?o ?o))
(bind ?flagO 0)
(if (> ?o 4) then
        (bind ?flagO 1)
)
(if (> ?o 4) then
        (bind ?r (- ?r 10))
)
(bind ?u (+ ?w (+ ?w ?flagO)))
(bind ?flagW 0)
(if (> ?w 4) then
        (bind ?flagW 1)
)
(if (> ?w 4) then
        (bind ?u (- ?u 10))
)
(if (> ?t 4) then (bind ?f 1))
(bind ?wynik 0)
(if (eq ?o (+ ?flagW (+ ?t ?t))) then(bind ?wynik 1))
return ?wynik)

(deffunction losuj ()
"Funkcja␣losujaca␣liczby␣z␣przedzialu␣0-9"
(bind ?zmienna 11)
(bind ?flaga 11)

(while (> ?flaga 10) do
(bind ?zmienna (random))
(if (> ?zmienna 0)     then (bind ?flaga 0  ))
(if (> ?zmienna 6001)  then (bind ?flaga 1  ))
(if (> ?zmienna 12001) then (bind ?flaga 2  ))
(if (> ?zmienna 18001) then (bind ?flaga 3  ))
(if (> ?zmienna 24001) then (bind ?flaga 4  ))
(if (> ?zmienna 30001) then (bind ?flaga 5  ))
```

```
(if (> ?zmienna 36001) then (bind ?flaga 6  ))
(if (> ?zmienna 42001) then (bind ?flaga 7  ))
(if (> ?zmienna 48001) then (bind ?flaga 8  ))
(if (> ?zmienna 54001) then (bind ?flaga 9  ))
(if (> ?zmienna 60001) then (bind ?flaga 11 ))
);nawias konca petli while

(bind ?zmienna ?flaga)
return ?zmienna)

(deffunction solve ()
"Funkcja rozwiazujaca problem TTF"
(bind ?tmp 0)
(bind ?zm1 0)
(bind ?zm2 0)
(bind ?zm3 0)

(while (eq ?tmp 0) do

(bind ?zm1 (losuj))
(bind ?zm2 (losuj))
(bind ?zm3 (losuj))

(bind ?tmp (ttf ?zm1 ?zm2 ?zm3))

);koniec petli while

(printout t "T:")(printout t ?zm1 crlf)
(printout t "W:")(printout t ?zm2 crlf)
(printout t "O:")(printout t ?zm3 crlf)
)
```

25.1.2 Rozwiązanie 2

czas ¿ 60s

```
(clear)

(deffacts cyfry "Zbior wszystkich cyfr a wiec mozliwych podstawien za pojedyncza litere"
(cyfra 0)
```

```
(cyfra 1)
(cyfra 2)
(cyfra 3)
(cyfra 4)
(cyfra 5)
(cyfra 6)
(cyfra 7)
(cyfra 8)
(cyfra 9))

(reset)

(deffunction sprawdz (?a ?b ?d ?e ?g ?l ?n ?o ?r ?t)
"Funkcja sprawdzajaca czy zadane przyporzadkowanie liczb do liter rozwiazuje problem"
(bind ?x1 (+ (* ?d 100000) (* ?o 10000) (* ?n 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x2 (+ (* ?g 100000) (* ?e 10000) (* ?r 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x3 (+ (* ?r 100000) (* ?o 10000) (* ?b 1000) (* ?e 100) (* ?r 10) ?t))
(if (= ?x3 (+ ?x1 ?x2)) then
   return TRUE
 else
   return FALSE)
)

(bind ?t1 (time))

(defrule solution
"Poszukuje rozwiazania"
(cyfra ?a)
(cyfra ?b&~?a)
(cyfra ?d&~?b&~?a)
(cyfra ?e&~?d&~?b&~?a)
(cyfra ?g&~?e&~?d&~?b&~?a)
(cyfra ?l&~?g&~?e&~?d&~?b&~?a)
(cyfra ?n&~?l&~?g&~?e&~?d&~?b&~?a)
(cyfra ?o&~?n&~?l&~?g&~?e&~?d&~?b&~?a)
(cyfra ?r&~?o&~?n&~?l&~?g&~?e&~?d&~?b&~?a)
(cyfra ?t&~?r&~?o&~?n&~?l&~?g&~?e&~?d&~?b&~?a)
(test (sprawdz ?a ?b ?d ?e ?g ?l ?n ?o ?r ?t))
=>
```

```
(printout t "A="?a "␣B="?b "␣D="?d "␣E="?e "␣G="?g "␣L="?l "␣N="?n "␣O="?o "␣R="?r "␣T="?t c
)

(bind ?t2 (time))

(- ?t2 ?t1)
```

25.1.3 Rozwiązanie 3

czas 30 s

```
(clear)

(deffacts cyfry "Zbior␣wszystkich␣cyfr␣a␣wiec␣mozliwych␣podstawien␣za␣pojedyncza␣litere"
(cyfra 0)
(cyfra 1)
(cyfra 2)
(cyfra 3)
(cyfra 4)
(cyfra 5)
(cyfra 6)
(cyfra 7)
(cyfra 8)
(cyfra 9))

(deffacts uklady-d-t "Wszystkie␣mozliwe␣uklady␣dla␣liter␣D␣i␣T"
(uklad-d-t 1 2)
(uklad-d-t 2 4)
(uklad-d-t 3 6)
(uklad-d-t 4 8)
(uklad-d-t 5 0)
(uklad-d-t 6 2)
(uklad-d-t 7 4)
(uklad-d-t 8 6)
(uklad-d-t 9 8))

(reset)

(deffunction sprawdz (?a ?b ?d ?e ?g ?l ?n ?o ?r ?t)
"Funkcja␣sprawdzajaca␣czy␣zadane␣przyporzadkowanie␣liczb␣do␣liter␣rozwiazuje␣problem"
```

```
(bind ?x1 (+ (* ?d 100000) (* ?o 10000) (* ?n 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x2 (+ (* ?g 100000) (* ?e 10000) (* ?r 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x3 (+ (* ?r 100000) (* ?o 10000) (* ?b 1000) (* ?e 100) (* ?r 10) ?t))
(if (= ?x3 (+ ?x1 ?x2)) then
   return TRUE
 else
   return FALSE)
)

(bind ?t1 (time))

(defrule solution
"Poszukuje␣rozwiazania"
(uklad-d-t ?d ?t)
(cyfra ?a&~?d&~?t)
(cyfra ?b&~?a&~?d&~?t)
(cyfra ?e&~?d&~?b&~?a&~?t)
(cyfra ?g&~?e&~?d&~?b&~?a&~?t)
(cyfra ?l&~?g&~?e&~?d&~?b&~?a&~?t)
(cyfra ?n&~?l&~?g&~?e&~?d&~?b&~?a&~?t)
(cyfra ?o&~?n&~?l&~?g&~?e&~?d&~?b&~?a&~?t)
(cyfra ?r&~?o&~?n&~?l&~?g&~?e&~?d&~?b&~?a&~?t)
(test (sprawdz ?a ?b ?d ?e ?g ?l ?n ?o ?r ?t))
=>
(printout t "A="?a "␣B="?b "␣D="?d "␣E="?e "␣G="?g "␣L="?l "␣N="?n "␣O="?o "␣R="?r "␣T="?t c
)

(bind ?t2 (time))

(- ?t2 ?t1)
```

25.1.4 Rozwiązanie 4

czas 1s

```
(clear)

(deffacts cyfry "Zbior␣wszystkich␣cyfr␣a␣wiec␣mozliwych␣podstawien␣na␣pojedyncza␣litere"
(cyfra 0)
(cyfra 1)
```

```
(cyfra 2)
(cyfra 3)
(cyfra 4)
(cyfra 5)
(cyfra 6)
(cyfra 7)
(cyfra 8)
(cyfra 9))

(reset)

(deffunction carry (?x ?y)
"Funkcja zwraca wartosc przeniesienia"
(if (> (+ ?x ?y) 9) then
   return 1
 else
   return 0)
)

(deffunction sprawdz (?a ?b ?d ?e ?g ?l ?n ?o ?r ?t)
"Funkcja sprawdzajaca czy zadane przyporzadkowanie liczb do liter rozwiazuje problem"
(bind ?x1 (+ (* ?d 100000) (* ?o 10000) (* ?n 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x2 (+ (* ?g 100000) (* ?e 10000) (* ?r 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x3 (+ (* ?r 100000) (* ?o 10000) (* ?b 1000) (* ?e 100) (* ?r 10) ?t))
(if (= ?x3 (+ ?x1 ?x2)) then
   return TRUE
 else
   return FALSE)
)

(bind ?t1 (time))

(defrule solution
"Poszukuje rozwiazania"
(cyfra ?d)
(cyfra ?t&~?d)
(test (= (+ ?d ?d) (+ ?t (* 10 (carry ?d ?d)))))
(cyfra ?l&~?d&~?t)
(cyfra ?r&~?l&~?d&~?t)
```

```
(test (= (+ ?l ?l (carry ?d ?d)) (+ ?r (* 10 (carry ?l ?l)))))

(cyfra ?a&~?r&~?l&~?d&~?t)
(cyfra ?b&~?a&~?r&~?l&~?d&~?t)
(cyfra ?e&~?b&~?a&~?r&~?l&~?d&~?t)
(cyfra ?g&~?e&~?b&~?a&~?r&~?l&~?d&~?t)
(cyfra ?n&~?g&~?e&~?b&~?a&~?r&~?l&~?d&~?t)
(cyfra ?o&~?n&~?g&~?e&~?b&~?a&~?r&~?l&~?d&~?t)

(test (sprawdz ?a ?b ?d ?e ?g ?l ?n ?o ?r ?t))
=>
(printout t "A="?a "␣B="?b "␣D="?d "␣E="?e "␣G="?g "␣L="?l "␣N="?n "␣O="?o "␣R="?r "␣T="?t c
)

(bind ?t2 (time))

(- ?t2 ?t1)
```

25.1.5 Rozwiązanie 5

czas 0s

```
(clear)

(deffacts cyfry "Zbior␣wszystkich␣cyfr␣a␣wiec␣mozliwych␣podstawien␣na␣pojedyncza␣litere"
(cyfra 0)
(cyfra 1)
(cyfra 2)
(cyfra 3)
(cyfra 4)
(cyfra 5)
(cyfra 6)
(cyfra 7)
(cyfra 8)
(cyfra 9))

(reset)

(deffunction carry (?x ?y)
"Funkcja␣zwraca␣wartosc␣przeniesienia"
```

```
(if (> (+ ?x ?y) 9) then
   return 1
 else
   return 0)
)

(deffunction sprawdz (?a ?b ?d ?e ?g ?l ?n ?o ?r ?t)
"Funkcja␣sprawdzajaca␣czy␣zadane␣przyporzadkowanie␣liczb␣do␣liter␣rozwiazuje␣problem"
(bind ?x1 (+ (* ?d 100000) (* ?o 10000) (* ?n 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x2 (+ (* ?g 100000) (* ?e 10000) (* ?r 1000) (* ?a 100) (* ?l 10) ?d))
(bind ?x3 (+ (* ?r 100000) (* ?o 10000) (* ?b 1000) (* ?e 100) (* ?r 10) ?t))
(if (= ?x3 (+ ?x1 ?x2)) then
   return TRUE
 else
   return FALSE)
)

(bind ?t1 (time))

(defrule solution
"Poszukuje␣rozwiazania"
(cyfra ?d)
(cyfra ?t&~?d)
(test (= (+ ?d ?d) (+ ?t (* 10 (carry ?d ?d)))))

(cyfra ?l&~?d&~?t)
(cyfra ?r&~?l&~?d&~?t)
(test (= (+ ?l ?l (carry ?d ?d)) (+ ?r (* 10 (carry ?l ?l)))))

(cyfra ?a&~?r&~?l&~?d&~?t)
(cyfra ?e&~?a&~?r&~?l&~?d&~?t)
(test (= (+ ?a ?a (carry ?l ?l)) (+ ?e (* 10 (carry ?a ?a)))))

(cyfra ?b&~?a&~?e&~?r&~?l&~?d&~?t)
(cyfra ?g&~?b&~?a&~?e&~?r&~?l&~?d&~?t)
(cyfra ?n&~?g&~?b&~?a&~?e&~?r&~?l&~?d&~?t)
(cyfra ?o&~?n&~?g&~?b&~?a&~?e&~?r&~?l&~?d&~?t)

(test (sprawdz ?a ?b ?d ?e ?g ?l ?n ?o ?r ?t))
```

```
=>
(printout t "A="?a "␣B="?b "␣D="?d "␣E="?e "␣G="?g "␣L="?l "␣N="?n "␣O="?o "␣R="?r "␣T="?t c
)

(bind ?t2 (time))

(- ?t2 ?t1)
```

25.2 Zawody

W zadaniu ?? ćwiczenia ?? rozwiązywaliśmy zagadkę o następującej treści.

Pięcioro przyjaciół rywalizowało na bieżni. Wincenty ze smutkiem opowiedział, ze mu się nie udało zająć pierwszego miejsca. Grzegorz przybiegł na metę jako trzeci po Dymitrze.* Wincenty zauważył na marginesie, ze Dymitr nie zajął drugiego miejsca, a Andrzej nie był ani pierwszym ani ostatnim. Borys powiedział, że przybiegł na metę w ślad za Wincentym. Pytanie: kto przybiegł na jakim miejscu?

```
(clear)

(deffacts miejsca "Zbior␣mozliwych␣do␣zajecia␣miejsc"
(miejsce 1)
(miejsce 2)
(miejsce 3)
(miejsce 4)
(miejsce 5)
)

(reset)

(defrule solution
"Poszukuje␣rozwiazania"
(miejsce ?w&:(not (= ?w 1))) ; Wincenty ze smutkiem opowiedzial, ze mu sie nie udalo zajac p
(miejsce ?d&~?w&:(not (= ?d 2))) ; Wincenty zauwazyl na marginesie, ze Dymitr nie zajal drug
(miejsce ?a&~?w&~?d&:(and (not (= ?a 1)) (not (= ?a 5)))); Andrzej nie byl ani pierwszym ani
(miejsce ?b&~?a&~?w&~?d&:(= ?b (+ ?w 1))) ; Borys powiedzial, ze przybiegl na mete w slad za
(miejsce ?g&~?b&~?a&~?w&~?d&:(and (= ?g 3) (> ?g ?d))) ; Grzegorz przybiegl na mete jako trz
=>
(printout t "Andrzej=" ?a "␣Borys=" ?b "␣Dymitr=" ?d "␣Grzegorz=" ?g "␣Wincenty=" ?w crlf)
```

*Grzegorz przybiegł na metę jako trzeci i był później niż Dymitr.

)

25.3 Pytania i odpowiedzi

Pytanie 25.1. ***Podaj rozwiązanie następujących zagadek logicznych***

- ***Zagadka 1: narzeczeni*** *W ogrodzie siedziały cztery pary narzeczonych i zajadały sliwki. Andzia zjadła 2 sliwki, Beata – 3, Celina – 4, Danusia – 5. Ich narzeczeni równiez nie próznowali. 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 sliwki. Jakie sa imiona par narzeczonych?*

- ***Zagadka 2: morderstwo na czwartym piętrze*** *Pani K została zabita przez jedna z czterech sąsiadek, inna sąsiadka z tej uroczej czwórki była pomagierką. Oto ich zeznania:*
  - *A: Jezeli B jest winna czegos, C musi byc niewinna.*
  - *B: Jezeli A jest niewinna , C musi byc winna.*
  - *C: Jezeli B była zabójca, D nie ma nic wspólnego z morderstwem.*
  - *D: Ja jestem niewinna.*

  Pamietajcie ze winne zawsze kłamia, a niewinne zawsze mówia prawde. Kto zabił p. K i z czyja pomoca?

- ***Zagadka 3: pracownicy naukowi*** *Trzech pracowników naukowych (jeden z nich to kobieta): chemik – p. Siwek, humanista – p. Czarnota i historyk – p. Blond, siedzi przy stole i rozmawia.*
  - *— Czy nie jest to dziwne – zauwaza kobieta, ze nasze nazwiska brzmia Czarnota, Blond i Siwek, i że jeden z nas ma włosy czarne, drugi blond, trzeci siwe?*
  - *— To prawda – powiedziała osoba majaca włosy czarne – ale nie mniej ciekawe, że nazwisko żadnego z nas nie odpowiada kolorowi jego włosów.*
  - *— Istotnie – wykrzyknął p. Siwek.*

  Jeżeli kobieta nie jest blondynką, to jaki jest kolor jej włosów i jak brzmi jej nazwisko?

Część VIII

Drools

Ćwiczenia

ROZDZIAŁ 20

Symulacja zachowania botów

Celem projektu* jest napisanie programu, w którym poczynania cyfrowych postaci (nazwijmy je Cyfrakami) w cyfrowym świecie kontrolowane będą za pomocą systemu regułowego. Przyjmujemy następujące założenia

1. Cyfrowy świat to dowolny obszar podzielony na komórki układające się w wiersze i kolumny (obszar nie musi być regularny: może być prostokątem, może przypominać trójkąt, elipsę lub dowolna inną figurę).
2. Komórki dzielimy na
    - PUSTE – do tej komórki może wejść Cyfrak.
    - NEDOSTEPNE – do tej komórki Cyfrak nie może wejść.
    - JEDZENIE – komórka z pożywieniem dla Cyfraka. Ilość jedzenia w tej komórce ustalana jest losowo na początku symulacji i podlega zmniejszaniu w efekcie korzystania z tego źródła przez Cyfraki uzupełniające swoje punkty życia.
    - ZAJETE – w danej komórce znajduje się Cyfrak. W dowolnej ustalonej chwili czasowej w danej komórce może znajdować się co najwyżej jeden Cyfrak.
3. Początkowa liczba Cyfraków jest parametrem programu.
4. Cyfraki mogą przemieszczać się na sąsiednie pola. Każda komórka ma maksymalnie osiem komórek sąsiednich, tj. tych, które bezpośrednio z nią sąsiaduja.

*Dla osób zainteresowanych podaję, że projekt bazuje na tzw. *automacie Moore'a*.

5. Wraz z przemieszczaniem, Cyfrak traci pewną ilość punktów życia.

6. Początkowy stan liczby punktów życia jest losowy, ale większy od pewnej stałej.

7. W zależności od ilości punktów i sytuacji w jakiej znajduje się Cyfrak, możemy przypisać jemu jeden z następujących stanów.

   - PATROL – stan początkowy polegający na beztroskim zwiedzaniu okolicy. Sposób poruszania się Cyfraka w tym stanie może być losowy, albo bardziej systematyczny (według dowlnych założeń). Jeśli podczas patrolu Cyfrak wejdzie na pole z jedzeniem, to bez względu na to czy je czy nie, to zapamiętuje informacje o tym polu (położenie i ilość pożywienia).
   - SPOTKANIE – gdy w sąsiedztwie Cyfraka znajdzie się inny Cyfrak.
   - ROZMOWA – po napotkaniu osobnika dysponującego nowymi informacjami o punktach z pożywieniem, następuje krótka konwersacja i wymiana danych.
   - POSZUKIWANIE – stan, w którym osobnik korzystając z wiedzy zdobytej w czasie PATROLU lub wymiany informacji z innymi osobnikami w stanie ROZMOWA, szuka miejsca, w którym mógłby odnowić swój zasób punktów życia.
   - JEDZENIE – uzupełnienie punktów życia.
   - SMIERC – po wyczerpaniu puli punktów życia osobnik ginie.

W trakcie symulacji mogą wystąpić następujące komunikaty:

- K1: mała liczba punktów życia (1 - 4999),
- K2: duża liczba punktów życia (5000 - 9999),
- K3: maksymalna liczba punktów życia (10000),
- K4: wyczerpany zasób punktów życia (0),
- K5: spotkanie,
- K6: koniec rozmowy,
- K7: odnaleziono miejsce z pożywieniem,
- K8: wyczerpanie zasobów pożywienia.

Powyższe komunikaty pozwalają stworzyć następujące reguły według schematu

$$(biezacy\ stan,\ symbol\ wejsciowy) \rightarrow (nowy\ stan)$$

- $(PATROL,\ K5) \rightarrow (ROZMOWA)$;
- $(PATROL,\ K1) \rightarrow (POSZUKIWANIE)$;
- $(ROZMOWA,\ K6) \rightarrow (PATROL)$;
- $(POSZUKIWANIE,\ K7) \rightarrow (JEDZENIE)$;
- $(POSZUKIWANIE,\ K4) \rightarrow (SMIERC)$;
- $(JEDZENIE,\ K3 \vee (\ K8 \wedge K2)) \rightarrow (PATROL)$;
- $(JEDZENIE,\ K8 \wedge K1) \rightarrow (POSZUKIWANIE)$.

Brakujące parametry należy dodefiniować samodzielnie a całość najlepiej zilustrować w formie graficznej. Poza tym, należy dodać jeszcze trzy samodzielnie wymyślone komunikaty i stworzyć dla nich odpowiednie reguły. Nowe komunikaty mogą np. uwzględniać rywalizację między Cyfrakami (walka), możliwość powiększania się ich liczebności, aktywny udział w odnowieniu zapasów jedzenia itp.

Część IX

Podstawy logiki

ROZDZIAŁ 27

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

$$p = \textit{Europa jest kontynentem.}$$
$$q = \textit{Ziemia krąży wokół Słońca.}$$

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

$$p' = \textit{Europa jest kontynentem.}\ (\textit{fałsz})$$
$$q' = 2 + 2 = 4\ (\textit{prawda})$$

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 27.1 (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 27.2 (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** (ang. *truth-table*)

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 27.3. *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 27.4. *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 27.5. *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 27.6. *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 27.1. *Formuła f jest tautologią wtedy i tylko wtedy, gdy $\neg f$ jest niespełnialna.*

Dowód. Dowód pozostawiamy jako ćwiczenie. □

Twierdzenie 27.2. *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 27.3. *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. *$(\bigwedge_{i=1}^n f_i) \to g$ jest tautologią.*

3. *$(\bigwedge_{i=1}^n f_i) \wedge g$ jest niespełnialne.*

Dowód. Dowód tutu. □

27.1 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 27.7. *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 27.8 (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 27.4. *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//. □

27.1.1 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 nastęujące 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 $\vee$.

27.2 Formuła Horna

Definicja 27.9 (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*. Zgodnie z tym co powiedzieliśmy wcześniej, zapis

$$\neg p \vee g$$

równoważny jest

$$p \to q.$$

Tak więc, na przykład formuła

$$f = (p \vee \neg q) \wedge (\neg p \vee \neg q \vee r) \wedge (\neg p \vee \neg q) \wedge s$$

równoważna jest formule

$$f = (q \to p) \wedge (p \wedge q \to r) \wedge (p \wedge q \to 0) \wedge (1 \to s)$$

co można czytać jako f *zachodzi jeśli prawdą jest, że z q wynika p oraz prawdą jest, że . . .* lub inaczej f *zachodzi jeśli prawdą jest, że jeśli q to p 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).

Algorytm

1. Jeśli w formule f występuje podformuła postaci $1 \to p$, wówczas oznacz każde wystąpienie formuły atomowej p w f.

2. Dopóki w f występują podformuły g postaci $(n \geqslant 1)$

$$(p_1 \wedge \cdots \wedge p_n) \to q$$

lub

$$(p_1 \wedge \cdots \wedge p_n) \to 0,$$

gdzie formuły atomowe $p_1, \ldots, p_n$ zostały już oznaczone natomiast q nie jest oznaczone, wykonuj

   a) Jeśli g jest postaci $(p_1 \wedge \cdots \wedge p_n) \to q$ wówczas oznacz wszystkie wystąpienia q w f; w przeciwnym razie zwróć wynik: *formuła niespełnialna* i zakończ algorytm.

3. Zwróć wynik: *formuła spełnialna*.

Twierdzenie 27.5. *Powyższy algorytm jest poprawny dla formuł Horna i zawsze kończy się po co najwyżej n iteracjach kroku 2, gdzie n jest ilością formuł atomowych w f.*

27.3 twierdzenie tutu

Twierdzenie 27.6. *Zbiór formuł S jest spełnialny wtedy i tylko wtedy gdy każdy skończony podzbiór zbioru S jest spełnialny.*

Dowód. Dowód

tutu

. □

27.4 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 $\{l_{i,1}, \ldots, l_{i,n_i}\}$. 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 27.10. *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 27.7. *Jeśli r jest resolwentą dwóch klauzul Horna, wówczas r także jest klauzulą Horna.*

Dowód. Dowód

tutu

. □

Twierdzenie 27.8. *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 27.11. *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

$$Res^*(f) = \bigcup_{k \geqslant 0} Res^k(g).$$

Twierdzenie 27.9. *Dla każdego skończonego zbioru klauzul f istnieje $k \geqslant 0$ takie, że*

$$Res^k(f) = Res^{k+1}(f) = \cdots = Res^*(f).$$

Dowód. Dowód

tutu

. □

Twierdzenie 27.10. *Zbiór klauzul f jest niespełnialny wtedy i tylko wtedy, gdy klauzula pusta należy do $Res^*(f)$.*

Dowód. Dowód

tutu

. □

27.5 Pytania i odpowiedzi.

Pytanie 27.1. ***Podaj dowód twierdzenia 27.2.***

Pytanie 27.2. ***Podaj dowód twierdzenia 27.4.***

Pytanie 27.3. ***Stosując tablicę prawdy sprawdź, czy zadana formuła f jest tautologią.***

ROZDZIAŁ 28

Rachunek kwantyfikatorów

Rachunek kwantyfikatorów nazywany także **rachunkiem predykatów pierwszego rzędu** (ang. *first order predicate calculus*) lub **logiką pierwszego rzędu** może być postrzegany jako naturalne rozszerzenie rachunku zdań. Za pomocą rachunku zdań możemy zapisać pewne sformułowania (zdania), związki między nimi oraz sposoby wnioskowania nowych zdań na podstawie zdań innych i sprawdzać w sposób formalny czy zdania te są prawdziwe czy nie. Przyjrzyjmy się następującym zdaniom

- Pada.
- Jest słonecznie.
- Wieje.
- Jeśli pada to nie jest słonecznie.

Każde z nich jest prostym stwierdzeniem, wyraża prosty **fakt**. Zdania te formalnie możemy zapisać w następujący sposób

- `pada`
- `słonecznie`
- `wieje`
- `pada -> not słonecznie`

Zauważmy w tym momencie, że za pomocą tych faktów stworzyliśmy pewną bazę wiedzy. Wiedzą tą możemy zarządzać, w szczególności wykorzystać ją, np. do stwierdzenia, że z faktu iż *pada deszcz* wynika, że *nie jest prawdą, że jest słonecznie*. Tak więc mamy sposób na reprezentację wiedzy i wykorzystanie tej reprezentacji. Co jest ważne, sposób ten posiada dosyć mocne i sprawdzone podstawy teoretyczne dające nadzieję na poprawne funkcjonowanie systemu o niego opartego. Rachunek zdań posiada jednak znaczne ograniczenia, które dosyć łatwo jest pokazać. Przyjrzyjmy się takim oto zdaniom

- Zosia jest kobietą.
- Asia jest kobietą.

i ich odpowiednikom wyrażonym formalnie

- `zosiakobieta`
- `asiakobieta`

Fakty, które stworzyliśmy nie pozwalają na wyciągnięcie **żadnych** wniosków co do „podobieństw" między nimi. Nie możemy dokonywać rozbioru znaczeniowego podanych faktów (pomijamy kwestię tego, że przecież znaczenie „wyrazów" dla komputera nie jest znane), gdyż równie dobrze, formalnie, oba zdania mogły by zostać zapisane jako

- `z`
- `a`

A przecież wiemy, że w języku naturalnym fakty te wyrażają stwierdzenie, że dwa różne „obiekty": `zosia` i `asia` mają cechę wspólną: są `kobietą`. Używając więc rachunku zdań nie mamy możliwości odnoszenia się do **obiektów**, nie możemy mówić o **relacjach** jakie je łączą i nie możemy mówić, że pewna cecha zachodzi **dla wszystkich** takich to a takich obiektów lub, że **istnieją** obiekty pewną cechę posiadające. Tym czasem powyższe możliwości są koniecznością, np. w świecie matematyki. Nie są nam przecież obce takie zapisy

$$\text{Dla każdego } \varepsilon > 0 \text{ istnieje taka liczba } \delta > 0, \text{ że dla każdego } x$$
$$\text{jeśli } |x - x_0| < \delta \text{ to } |f(x) - g| < \varepsilon,$$

w których pojawiają się pojęcia dotyczące zakresu (*dla wszystkich*, *istnieje*), relacji ($<$, $>$) czy też funkcji (f, $-$).

Podobnie jak w przypadku rachunku zdań, rozważania na temat przydatności rachunku predykatów zaczniemy od syntaktyki a więc od tego jak mogą wyglądać poprawnie budowane „napisy” rachunku kwantyfikatorów.

Definicja 28.1 (Syntaktyka rachunku kwantyfikatorów)**.** *Do konstruowania wyrażeń rachunku kwantyfikatorów wykorzystujemy symbole*

1. ***zmiennej*** *(ang.* variable*) oznaczane w dalszej części zwykle przy pomocy liter u, v, w, x, y, z z ewentualnymi indeksami. Zmienna oznacza wielkość, która może przyjmować rozmaite (zależne od problemu) wartości, należące na ogół do pewnego zbioru;*
2. ***predykatu*** *(ang.* predicate*) lub* ***relacji*** *oznaczane w dalszej części zwykle przy pomocy litery P_i^j, gdzie ewentualny indeks dolny (i) stosowany jest dla odróżnienia dwóch predykatów, natomiast indeks górny (j) określa argumentowość (lub* ***arność****, ang.* arity*) predykatu*[*]*;*
3. ***funkcji*** *(ang.* function*) oznaczane w dalszej części zwykle przy pomocy liter f, g, h z ewentualnymi indeksami dolnymi i górnymi o znaczeniu analogicznym jak dla predykatów. Symbol funkcyjny o argumentowości 0 nazywamy* ***stałą****. W dalszej części stałe będziemy oznaczać literami a, b, c, d.*

Przy pomocy powyższych symboli definiujemy ***termy*** *w następujący sposób.*

1. *Każda zmienna jest termem.*
2. *Jeśli f jest symbolem funkcyjnym o argumentowości k a $t_1, \ldots, t_k$ termami, wówczas $f(t_1, \ldots, t_k)$ jest termem.*

Formuły *rachunku kwantyfikatorów definiujemy w następujący sposób.*

1. *Jeśli P jest symbolem predykatu o argumentowości k a $t_1, \ldots, t_k$ termami, wówczas $P(t_1, \ldots, t_k)$ jest formułą. Formułę taką nazywamy* ***formułą atomową****.*
2. *Dla każdej formuły F, $\neg F$ jest formułą.*
3. *Dla wszystkich formuł F i G, $F \wedge G$ jest formułą.*
4. *Dla wszystkich formuł F i G, $F \vee G$ jest formułą.*

---

[*]Jeśli nie będzie to prowadziło do nieporozumień to indeksy będziemy opuszczać. W szczególności pomijany będzie indeks argumentowości w przypadku gdy będzie on wynosił 0 lub gdy z zapisu jasno będzie wynikała argumentowość.

5. *Jeśli x jest zmienną i F formułą, wówczas $\exists_x F$ jest formułą.*

6. *Jeśli x jest zmienną i F formułą, wówczas $\forall_x F$ jest formułą.*

O zmiennych występujących w formule możemy powiedzieć, że są ***związane*** *(ang.* bound*) lub* ***wolne*** *(ang.* free*). Zmienna staje się związana od miejsca w formule, w którym użyto symbolu $\exists$ lub $\forall$. Jeśli (albo zanim) to nie nastąpi, zmienna jest wolna. Symbole $\exists$ lub $\forall$ nazywamy* ***kwantyfikatorami*** *(ang.* quantifiers*), przy czym pierwszy z nich nazywamy* ***kwantyfikatorem egzystencjalnym*** *(ang.* existential quantifier*) lub* ***szczegółowym*** *drugi zaś* ***kwantyfikatorem uniwersalnym*** *(ang* universal quantifier*). Formuła, w której nie występują zmienne wolne nazywana jest* ***zamkniętą*** *(ang.* closed*).* ***Macierzą formuły*** *(ang.* matrix of a formula*) F nazywamy formułę F^* otrzymaną z formuły F przez usunięcie wszystkich kwantyfikatorów.*

Wiedząc jak wyglądają poprawne zdania tego „języka", musimy teraz nadać im sens a więc określić znaczenie. Definicję sematyczną podzielimy ze względów praktycznych na dwie części. Postępujemy tak, gdyż

- bez tego zabiegu jedna definicja jest strasznie długa i czytając jej koniec można zapomnieć co było na początku;
- w części pierwszej wprowadzimy istotne pojęcie, które będzie wykorzystywane w drugiej i zanim do częsci drugiej przejdziemy, będziemy chcieli zilustrować je przykładem.

Definicja 28.2 (Semantyka rachunku kwantyfikatorów (część pierwsza))**.** *Strukturą (ang.* structure*) naywamy parę postaci $\mathcal{S} = (U_\mathcal{S}, I_\mathcal{S})$. $U_\mathcal{S}$ jest pewnym niepustym zbiorem nazywanym tutu (ang.* ground set, universe*). $I_\mathcal{S}$ jest odwzorowaniem przekształcającym*

- *n argumentowy symbol predykatu P na n argumentowy predykat na $U_\mathcal{S}$;*
- *n argumentowy symbol funkcji f na n argumentową funkcję na $U_\mathcal{S}$;*
- *każdą zmienną x na element zbioru $U_\mathcal{S}$.*

Definicja ta wydaje się być mało intuicyjna, więc pokażemy jej sens na przykładzie. Rozważmy formułę F postaci

$$F = \forall_x P(x, f(x)) \wedge Q(g(a, z))$$

Widzimy, że

- P jest predykatem binarnym, Q jest predykatem unarnym;
- f jest funkcją unarną, g jest funkcją binarną, a jest stałą;
- zmienna x jest związana, zmienna z jest wolna.

Przykładem struktury odpowiadającej F jest struktura zdefiniowana poniżej

- $U_{\mathcal{S}} = \{1, 2, 3, \dots\}$
- $I_{\mathcal{S}}(P) = \{(m, n) : m, n \in U_{\mathcal{S}}, m < n\}$
- $I_{\mathcal{S}}(Q) = \{n \in U_{\mathcal{S}} : n \text{ jest liczbą pierwszą}\}$
- $I_{\mathcal{S}}(f) =$ następnik elementu, tzn. $f(n) = n + 1$, $n \in I_{\mathcal{S}}$
- $I_{\mathcal{S}}(g) =$ suma elementów, tzn. $g(m, n) = m + n$, $m, n \in I_{\mathcal{S}}$
- $I_{\mathcal{S}}(a) = 2$
- $I_{\mathcal{S}}(z) = 3$

Definicja 28.3. *Niech F będzie formułą a $\mathcal{S} = (U_{\mathcal{S}}, I_{\mathcal{S}})$ strukturą. Strukturę $\mathcal{S}$ nazywamy* ***odpowiednią*** *(ang.* suitable*) dla F jeśli $I_{\mathcal{S}}$ jest określone dla wszystkich symboli predykatów, symboli funkcyjnych i dla wszystkich zmiennych wolnych występujących w F.*

Definicja 28.4 (Semantyka rachunku kwantyfikatorów (część druga))**.** *Niech F będzie formułą a $\mathcal{S} = (U_{\mathcal{S}}, I_{\mathcal{S}})$ odpowiednią strukturą. Dla każdego termu t z F definiujemy jego wartość (ang.* value*) ze względu na strukturę $\mathcal{S}$, co zapisujemy, $\mathcal{S}(t)$ w następujący sposób.*

1. *Jeśli t jest zmienną (czyli $t = x$), wówczas $\mathcal{S}(t) = I_{\mathcal{S}}(x)$.*
2. *Jeśli t jest postaci $f(t_1, \dots, t_k)$, gdzie $t_1,\dots,t_k$ to termy a f jest symbolem funkcyjnym o argumentowości k wówczas $\mathcal{S}(t) = I_{\mathcal{S}}(f(\mathcal{S}(t_1), \dots, \mathcal{S}(t_k)))$.*

Analogicznie jak to miało miejsce w przypadku definicji semantycznej z rozdziału 27 określimy teraz kiedy formuła rachunku predykatów F jest prawdziwa ze względu na strukturę $\mathcal{S}$.

1. *Jeśli F ma postać $F = P(t_1, \dots, t_k)$, gdzie $t_1,\dots,t_k$ są termami a P predykatem, wówczas*

$$\mathcal{S}(F) = \begin{cases} 1, & \text{jeśli} \quad (\mathcal{S}(t_1), \dots, \mathcal{S}(t_k)) \in I_{\mathcal{S}}(P) \\ 0, & \text{w przeciwnym razie} \end{cases}$$

2. *Jeśli F jest postaci $F = \neg G$, wówczas*

$$\mathcal{S}(F) = \begin{cases} 1, & \text{jeśli} \quad \mathcal{S}(G) = 0 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

3. *Jeśli F jest postaci $F = (G \wedge H)$, wówczas*

$$\mathcal{S}(F) = \begin{cases} 1, & \text{jeśli} \quad \mathcal{S}(G) = 1 \quad \text{i} \quad \mathcal{S}(H) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

4. *Jeśli F jest postaci $F = (G \vee H)$, wówczas*

$$\mathcal{S}(F) = \begin{cases} 1, & \text{jeśli} \quad \mathcal{S}(G) = 1 \quad \text{lub} \quad \mathcal{S}(H) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

5. *Jeśli F jest postaci $F = \forall_x G$, wówczas*

$$\mathcal{S}(F) = \begin{cases} 1, & \text{jeśli dla wszystkich} \quad u \in U_{\mathcal{S}} \quad \text{zachodzi} \quad \mathcal{S}_{[x|u]}(G) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

6. *Jeśli F jest postaci $F = \exists_x G$, wówczas*

$$\mathcal{S}(F) = \begin{cases} 1, & \text{jeśli istnieje takie} \quad u \in U_{\mathcal{S}} \quad \text{że} \quad \mathcal{S}_{[x|u]}(G) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

Definicja 28.5. *Powiemy, że formuła F jest prawdziwa w $\mathcal{S}$ (ze względu na strukturę $\mathcal{S}$) lub, że $\mathcal{S}$ jest* ***modelem*** *dla F (co zapiszemy $\mathcal{S} \models F$) jeśli F jest formułą, $\mathcal{S}$ odpowiednią strukturą dla F oraz $\mathcal{S}(F) = 1$.*

Definicja 28.6. *Jeśli każda odpowiednia struktura dla F jest modelem dla F, wówczas zapiszemy to jako $\models F$ i powiemy, że F jest prawdziwa (ang.* valid*).*

Definicja 28.7. *Jeśli dla fomuły F istnieje co najmniej jeden model, wówczas F nazywamy* ***spełnialną*** *(ang* satisfiable*). W przeciwnym razie nazywamy ją* ***niespełnialną*** *(ang.* unsatisfiable*) lub* ***sprzeczną*** *(ang.* contradictory*).*

Analogicznie jak w rozdziale poświęconym rachunkowi zdań, będziemy dążyć do pewnego sposobu normalizacji formuł. Zanim do tego dojdziemy musimy poznać sposoby przekształcania formuł w taki sposób aby otrzymać inne „wizualnie” wyrażenie, ale równoważne zadanemu. Zauważmy, że jeśli nie mamy kwantyfikatorów to w mocy pozostają wszystkie równoważności z twierdzenia 27.2. Uwzględniając natomiast kwantyfikatory uzyskujemy kilka nowych równoważności

Twierdzenie 28.1. *Niech F i G będą dowolnymi formułami. Zachodzą następujące równoważności*

1.

$$\neg\forall_x F \equiv \exists_x \neg F$$

$$\neg\exists_x F \equiv \forall_x \neg F$$

2. Jeśli zmienna x nie występuje jako wolna w formule G, wówczas

$$(\forall_x F \wedge G) \equiv \forall_x (F \wedge G)$$

$$(\forall_x F \vee G) \equiv \forall_x (F \vee G)$$

$$(\exists_x F \wedge G) \equiv \exists_x (F \wedge G)$$

$$(\exists_x F \vee G) \equiv \exists_x (F \vee G)$$

3.

$$(\forall_x F \wedge \forall_x G) \equiv \forall_x (F \wedge G)$$

$$(\exists_x F \vee \exists_x G) \equiv \exists_x (F \vee G)$$

4.

$$\forall_x \forall_y F \equiv \forall_y \forall_x F$$

$$\exists_x \exists_y F \equiv \exists_y \exists_x F$$

Dowód. Dowód tutu. □

28.1 Postać normalna

Definicja 28.8. *Niech F będzie formułą, x zmienną a t termem. Zapis*

$$F_{[x|t]}$$

oznaczać będzie formułę otrzymaną z formuły F przez podstawienie za wszystkie wolne zmienne x termu t, gdzie $[x|t]$ to zapis podstawienia (ang. substitution*). Zapis*

$$F_{[x|t_1][y|t_2]}$$

rozumiemy jako wielokrotne podstawienie w tym sensie, że

1. *Najpierw wykonywane jest podstawienie* $[x|t_1]$*, przy czy nie jest wykluczone, że* t_1 *zawiera zmienną* y.

2. *Następnie wykonywane jest podstawienie* $[y|t_2]$.

Twierdzenie 28.2. *Niech dana będzie formuła* F *postaci*

$$F = Q_x G,$$

gdzie Q *jest dowolnym kwantyfikatorem. Niech* y *będzie zmienną, która nie występuje jako wolna w formule* G*. Wówczas zachodzi równoważność*

$$Q_x G \equiv Q_y G_{[x|y]}$$

Dowód. Dowód tutu. □

Twierdzenie 28.3. *Dla każdej formuły istnieje równoważna jej formuła w tzw. postaci (??!!)(ang.* rectified form*) taka, że*

- *żadna zmienna nie występuje jako związana i wolna,*
- *każdy kwantyfikator wiąże inną zmienną.*

Dowód. Dowód tutu. □

Twierdzenie 28.4. *Dla każdej formuły istnieje równoważna jej formuła w tzw. (??!!)(ang.* prenex form*) postaci*

$$Q_{1y_1} \dots Q_{ny_n} F,$$

gdzie $Q_1,\dots,Q_n$ *są kwantyfikatorami,* $y_1,\dots,y_n$ *zmiennymi a formuła* F *nie zawiera kwantyfikatorów.*

Dowód. Dowód tutu. □

Twierdzenie 28.5. *Niech* F *będzie formułą, natomiast* $x_1, \dots, x_n$ *zmiennymi wolnymi w* F*. Wówczas*

1. *formuła* F *jest prawdziwa wtedy i tylko wtedy, gdy prawdziwa jest formuła* $\forall_{x_1} \dots \forall_{x_n} F$*;*

2. *formuła* F *jest spełnialna wtedy i tylko wtedy, gdy spełnialna jest formuła* $\exists_{x_1} \dots \exists_{x_n} F$.

Dowód. Dowód tutu. □

Definicja 28.9. *Dla każdej formuły F w postaci RPF (ang.* rectified and prenex form*) definiujemy odpowiadającą jej* ***formułę Skolema*** *(ang.* Skolem formula*) jako formułę w której wszystkie zmienne wprowadzone przez kwantyfikator egzystencjalny (∃) zostały zastąpione przez nowy symbol funkcyjny (nie występujący w F) zależny od zmiennych wprowadzonych przez kwantyfikator uniwersalny (∀). Zamiast bowiem mówić, że istnieje obiekt o pewnych własnościach, można utworzyć taki obiekt i nadać jemu nazwę.*

Pomimo, że formuła Skolema, ze względu na wprowadznie nowych symboli funkcyjnych, jedynie odpowiada pewnej formule a nie jest jej równoważna to zachodzi następujące twierdzenie

Twierdzenie 28.6. *Formuła F postaci RPF jest spełnialna wtedy i tylko wtedy gdy odpowiadająca jej formuła Skolema jest spełnialna.*

Dowód. Dowód tutu. □

W tym momencie mamy już cały aparat potrzebny do tego aby dowolne wyrażenie rachunku kwantfikatorów przekształcić w jedną uniwersalną postać, którą jest znana nam już postać klauzulowa. Proces przekształcania składa się z dwóch zasadniczych kroków

1. Przekształcenie formuły rachunku kwantyfikatorów F do odpowiadającej jej formuły Skolema.
2. Otrzymaną w poprzednim kroku formułę Skolema przekształcamy do postaci CNF zgodnie z metodami poznanymi w rozdziale poświęconym rachunkowi zdań.

Przekształcanie do formuły Skolema

Proces przekształcania formuły do formuły Skolema ma cechy jak najbardziej algorytmiczne i nadaje się do zautomatyzowania. Biorąc pod uwagę, że także przekształcanie do postaci CNF daje się w prosty sposób opisać, całość można połączyć w nieskomplikowany algorytm.

//algorytm skolemizacji tutu//

28.1.1 Algorytm przekształcania do postaci klauzulowej

Dla dowolnej formuły rachunku predykatów F wykonaj.

1. Zamiana formuły F na równoważną jej formułę F_{rect} w postaci rectified form. Krok ten wiąże się ze zmianą nazw zmiennych związanych.

2. Wszystkie zmienne wolne $x_1,\ldots,x_n$ występujące w formule F_{rect} zostają wprowadzone kwantyfikatorem egzystencjalnym

$$F_{exists} = \exists_{x_1} \ldots \exists_{x_n} F_{rect}$$

3. Zamiana formuły F_{exists} na równoważną jej formułę F_{prenex} w postaci prenex form. Krok ten wiąże się z „wyprowadzeniem" wszystkich kwantyfikatorów na zewnątrz formuły.

4. Eliminacja kwantyfikatorów egzystencjalnych występujących w formule F_{prenex} przez sprowadzenie do formuły Skolema F_{skol}.

5. W tym kroku o wszystkich zmiennych można powiedzieć, że są wprowadzone przez kwantyfikator i kwantyfikatorem tym jest kwantyfikator uniwersalny. Skoro tak to „usuwamy" te kwantyfikatory.

6. Traktując otrzymaną w poprzednim kroku formułę jak poprawną formułą rachunku zdań, przekształcamy ją do postaci CNF po czym zapisujemy jako zbiór klauzul.

W przypadku Prologa realizacja kroku 4 dokonywana jest przez wprowadzenie nowych symboli stałych (inaczej funkcji 0 argumentowych), nazywanych też **stałymi Skolema**, w miejsce zmiennych wprowadzanych za pomocą kwantyfikatorów egzystncjalnych. Przykładowo

```
exists(X, a(X) & b(X))
```

zostaje przekształcone na

```
a(g103) & b(g103)
```

gdzie g103 to pewna niepowtarzalna stała. Stąd właśnie pochodzą pojawiające się często „tajemnicze" wyniki złożone z symboli podobnych do g103.

Przykład

Niech dana będzie następująca formuła

$$F = \neg\exists_x(P(x,z) \vee \forall_y Q(x, f(y))) \vee \forall_y P(g(x,y), z))$$

1. Zamiana formuły F na równoważną jej formułę F_{rect} w postaci rectified form. Krok ten wiąże się ze zmianą nazw zmiennych związanych.

$$F = \neg\exists_x(P(x,z) \vee \forall_y Q(x, f(y))) \vee \forall_w P(g(x,w), z))$$

2. Wszystkie zmienne wolne występujące w formule F_{rect} zostają wprowadzone kwantyfikatorem egzystencjalnym

$$F = \exists_z(\neg\exists_x(P(x,z) \vee \forall_y Q(x, f(y))) \vee \forall_w P(g(x,w), z)))$$

3. Zamiana formuły F_{exists} na równoważną jej formułę F_{prenex} w postaci prenex form. Krok ten wiąże się z „wyprowadzeniem" wszystkich kwantyfikatorów na zewnątrz formuły.

$$F = \exists_z\forall_x\exists_y\forall_w((\neg(P(x,z) \wedge \neg Q(x, f(y))) \vee P(g(x,w), z)))$$

4. Eliminacja kwantyfikatorów egzystencjalnych występujących w formule F_{prenex} przez sprowadzenie do formuły Skolema F_{skol}.

$$F = \forall_x\forall_w((\neg(P(x,a) \wedge \neg Q(x, f(h(x)))) \vee P(g(x,w), a)))$$

W tym przypadku za z podstawiono stałą a a za y funkcję $h(x)$.

5. W tym kroku o wszystkich zmiennych można powiedzieć, że są wprowadzone przez kwantyfikator i kwantyfikatorem tym jest kwantyfikator uniwersalny. Skoro tak to „usuwamy" te kwantyfikatory.

$$F = (\neg(P(x,a) \wedge \neg Q(x, f(h(x)))) \vee P(g(x,w), a))$$

6. Traktując otrzymaną w poprzednim kroku formułę jak poprawną formułą rachunku zdań, przekształcamy ją do postaci CNF po czym zapisujemy jako zbiór klauzul.

$$\{\{\neg P(x,a), P(g(x,w), a)\}, \{\neg Q(x, f(h(x))), P(g(x,w), a)\}\}$$

28.2 Nierozstrzygalność

Celem zarówno tego rozdziału jak i poprzedniego jest pokazanie, że sposób działania pewnych języków (w tym przypadku Prologa) nie jest „przypadkowy". Innymi słowy, chcemy pokazać (udowodnić), że „algorytm" według którego pracują jest poprawny i może faktycznie dać nam odpowiedź w kwestii spełnialności (ang. *satisfiability*) czy nawet prawdziwości (ang. *validity*) zadanej formuły. W tym podrozdziale pokażemy, że nie ma takiego algorytmu, który by działał dla dowolnych formuł rachunku predykatów. Mówiąc inaczej, rachunek predykatów jest **nierozstrzygalny** (ang. *undecidable*). Dowód tego faktu wymaga sprecyzowania co rozumiemy pod pojęciem obliczeń (ang. *computations*) i algorytmu (ang. *algorithm*). W teorii obliczeń funkcję nazywamy **obliczalną** (ang. *computable*) (lub

o problemie, który ona reprezentuje mówimy **rozstrzygalny** (ang. *decidable*)) jeśli istnieje dla niego abstrakcyjna maszyna (tzw. maszyna Turinga), która rozpoczynając swoje działanie od danych wejściowych będących dziedziną funkcji, zatrzymuje się w skończonej liczbie kroków w jednym ze stanów: akceptującym lub odrzucającym. Jeśli taka maszyna nie istnieje, to funkcję nazywamy nieobliczlną (ang. *non-computable*) (lub odpowiadający jej problem nierozstrzygalnym (ang. *undecidable*)).

Twierdzenie 28.7 (Church)**.** *Problem prawdziwości formuł rachunku predykatów jest nierozstrzygalny.*

Dowód. Dowód tutu. □

Tak więc ogólne podejście do rachunku predykatów nie pozwoli znaleźć uniwersalnego (i dającego się opisać algorytmicznie a przede wszystkim zautomatyzować) sposobu „oceny" formuły. Rodzi się w tym momencie pytanie, czy wyrażenia rachunku predykatów można przedstawić w pewnej tzw. postaci kanonicznej (tj. postaci uniwersalnej) jak np. postać CNF, dla której problem ten nie będzie występował. Okazuje się, że zachodzi następujące twierdzenie.

Twierdzenie 28.8. *Jeśli F jest zamkniętą formułą Skolema, wówczas F jest spełnialna wtedy i tylko wtedy gdy F posiada model Herbranda.*

Dowód. Dowód tutu. □

Dzięki temu twierdzeniu możemy mieć nadzieję, że uda się znaleźć warunki, w których formuła jest rozstrzygalna. Zanim przejdziemy dalej, musimy jednak wyjaśnić czym jest model Herbranda (ang. *Herbrand model*).

28.3 Model Herbranda

Definicja 28.10 (Uniwersum Herbranda)**.** ***Uniwersum Herbranda*** *(ang.* Herbrand universe*) dla zamkniętej formuły Skolema F, oznaczane jako $\mathcal{H}(F)$, definiujemy w następujący sposób.*

1. *Każda stała występująca w F należy do $\mathcal{H}(F)$. Jeśli F nie zawiera żadnej stałej, wówczas przyjmujemy, że istnieje pewna stała, np. c, która należy do $\mathcal{H}(F)$.*

2. *Jeśli termy $t_1, \ldots, t_n \in \mathcal{H}(F)$ oraz $f \in \mathcal{H}(F)$ jest funkcją n-argumentową, wówczas $f(t_1, \ldots, t_n) \in \mathcal{H}(F)$.*

Tak więc uniwersum Herbranda składa się z wszystkich zamkniętych termów złożonych ze stałych i symboli funkcyjnych występujących w formule. Jeśli formuła nie zawiera żadnych stałych dodaje się do uniwersum dowolną stałą, żeby nie było ono puste. Z definicji wynika także, że jeśli formuła zawiera choć jeden symbol funkcyjny o argumentowości większej niż 0, uniwersum Herbranda jest zbiorem nieskończonym. Uniwersum Herbranda jest zawsze co najwyżej przeliczalne. Przyjrzyjmy się kilu przykładom.

1.
$$\begin{aligned} F &= P(x) \vee P(y) \\ \mathcal{H}(F) &= \{a\} \end{aligned}$$

2.
$$\begin{aligned} F &= P(x, a) \vee P(b, y) \\ \mathcal{H}(F) &= \{a, b\} \end{aligned}$$

3.
$$\begin{aligned} F &= P(x, a) \wedge \neg P(b, f(y)) \\ \mathcal{H}(F) &= \{a, f(a), f(f(a)), f(f(f(a))), \dots\} \end{aligned}$$

4.
$$\begin{aligned} F &= P(x, a) \wedge P(b, f(y)) \wedge P(f(a), g(x)) \\ \mathcal{H}(F) &= \{a, b, f(a), f(b), g(a), g(b), f(f(a)), f(g(a)), g(f(a)), \dots\} \end{aligned}$$

5.
$$\begin{aligned} F &= P(x, f(y), g(z, x)) \\ \mathcal{H}(F) &= \{a, f(a), g(a, a), f(g(a, a)), f(f(a)), g(a, f(a)), g(f(a), a), g(f(a), f(a)), \dots\} \end{aligned}$$

Definicja 28.11 (Struktura Herbranda)**.** *Niech F będzie zamkniętą formułą Skolema. Każdą strukturę $\mathcal{S} = (U_{\mathcal{S}}, I_{\mathcal{S}})$ spełniającą warunki*

1. $U_{\mathcal{S}} = \mathcal{H}(F)$,

2. dla wszystkich termów $t_1, \dots, t_n \in \mathcal{H}(F)$ oraz wszystkich n-argumentowych funkcji f z F zachodzi równość

$$f^{\mathcal{S}}(t_1, \dots, t_n) = f(t_1, \dots, t_n)$$

*nazywamy **strukturą Herbranda** (ang.* Herbrand structure*).*

Definicja 28.12 (Model Herbranda)**.** *Strukturę Herbranda dla formuły F nazywamy **modelem Herbranda** (ang.* Herbrand model*) dla F jeśli struktura ta jest modelem dla F.*

Definicja 28.13 (Rozszerzenie Herbranda)**.** *Niech dana będzie zamknięta formuła Skolema F*

$$F = \forall_{x_1} \dots \forall_{x_n} G.$$

Wówczas ***rozszerzenie Herbranda*** *(ang. Herbrand expansion)* $\mathcal{E}(F)$ *definiujemy jako*

$$\mathcal{E}(F) = \{G_{[x_1|t_1]\dots[x_n|t_n]} : t_1, \dots, t_n \in \mathcal{H}(F)\}.$$

Zatem formuły w $\mathcal{E}(F)$ tworzymy zastępując, na wszystkie możliwe sposoby, zmienne w G termami pochodzącymi z $\mathcal{H}(F)$. Na przykład dla formuły

$$F = \forall_x \forall_y \forall_z P(x, f(y), g(z, x))$$

otrzymujemy następujące elementy zbioru $\mathcal{E}(F) = \{F_1, F_2, \dots\}$

$$\begin{array}{rcccl}
F_1 & = & P(x, f(y), g(x, z))_{[x|a][y|a][z|a]} & = & P(a, f(a), g(a, a)), \\
F_2 & = & P(x, f(y), g(x, z))_{[x|f(a)][y|a][z|a]} & = & P(f(a), f(a), g(f(a), a)), \\
F_3 & = & P(x, f(y), g(x, z))_{[x|a][y|f(a)][z|a]} & = & P(a, f(f(a)), g(a, a)), \\
F_4 & = & P(x, f(y), g(x, z))_{[x|a][y|a][z|f(a)]} & = & P(a, f(a), g(a, f(a))), \\
F_5 & = & P(x, f(y), g(z, x))_{[x|g(a,a)][y|a][z|a]} & = & P(g(a, a), f(a), g(g(a, a), a)), \\
 & & \dots & &
\end{array}$$

Zauważmy, że formuły występujące w $\mathcal{E}(F)$ możemy potraktować jak formuły rachunku zdań, gdyż nie zawierają one zmiennych.

Twierdzenie 28.9 (Gödel – Herbrand – Skolem)**.** *Każda zamknięta formuła Skolema F jest spełnialna wtedy i tylko wtedy, gdy zbiór formuł* $\mathcal{E}(F)$ *(traktowany jak zbiór formuł rachunku zdań) jest spełnialny.*

Dowód. Dowód tutu. □

Twierdzenie 28.10 (Herbrand)**.** *Zamknięta formuła Skolema F jest niespełnialna wtedy i tylko wtedy, gdy istnieje skończony podzbiór* $\mathcal{E}(F)$ *(traktowany jak zbiór formuł rachunku zdań), który jest niespełnialny.*

Dowód. Dowód tutu. □

Oparając się na twierdzeniu Herbranda, możemy mówić o **półrozstrzygalności** lub **częściowej rozstrzygalności** (ang. *semi-decidability*), rozumianej jako procedura dająca odpowiedź w skończone

ilości kroków na pytanie, na które istniej odpowiedź twierdząca. Zaprezentowany poniżej algorytm posiadą tą własność, że zatrzymuje się po skończonej ilości kroków, jeśli tylko zadana formuła F jest niespełnialna. Jeśli jest spełnialna, wówczas się nie zatrzymuje (nie ma takiej gwarancji). Dzięki temu, testując formułę $\neg F$ mamy możliwość sprawdzić jej prawdziwość.

Algorytm Gilmore'a

Daną wejściową dla algorytmu jest zamknięta formuła Skolema F, $F_i \in \mathcal{E}(F)$, dla $i = 1, 2, \ldots$

1. $n := 0$
2. $n := n + 1$
3. Jeśli $F_1 \wedge \cdots \wedge F_n$ nie jest niespełnialna, przejdź do 2.
4. Wypisz *niespełnialna* i zakończ algorytm.

28.4 Rezolucja

Realizacja kroku 3 algorytmu Gilmore'a możliwa jest w oparciu o środki udostępniane przez rachunek zdań, np. tablice prawdy. Niestety tablice prawdy są efektywne jedynie dla „małych" formuł, tzn. takich gdzie liczba formuł atomowych skutkuje „akceptowalnym" czasem działania algorytmu[†]. Zamiast nich, skorzystać możemy z wniosków płynących z podrozdziału 27.4 a w szczególności twierdzenia 27.10, które prowadzą do następującego algorytmu.

Algorytm rezolucji dla rachunku predykatów

Daną wejściową dla algorytmu jest zamknięta formuła Skolema F z macierzą F^* w postaci CNF, $F_i \in \mathcal{E}(F)$, dla $i = 1, 2, \ldots$

1. $n := 0$, $S := \emptyset$
2. $n := n + 1$
3. $S := S \cup F_n$
4. $S := Res^*(S)$

[†]Pamiętamy, że dla n formuł atomowych mamy 2^n różnych przyporządkowań. „Akceptowalność" oznacza w tym przypadku czas jaki jesteśmy skłonni poczekać na otrzymanie odpowiedzi.

5. Jeśli klauzula pusta nie należy do S, przejdź do 2.

6. Wypisz *niespełnialna* i zakończ algorytm.

Celem wskazania problemów na jakie możemy natrafić rozważmy dwa przykłady.

Przykład 1

Niech dana będzie następująca (niespełnialna) formuła

$$F = \forall_x (P(x) \wedge \neg P(f(x))).$$

Zgodnie z definicją macierzy dla formuły, mamy

$$F^* = (P(x) \wedge \neg P(f(x))),$$

co w postaci klauzulowej zapisujemy jako

$$F^* = \{\{P(x)\}, \{\neg P(f(x))\}\}.$$

Rozszerzeniem Herbranda w tym przypadku jest

$$\mathcal{E}(F) = \{(P(a) \wedge \neg P(f(a))), (P(f(a)) \wedge \neg P(f(f(a)))), \dots\}.$$

Zauważmy, że w wyniku podstawienia $[x|a]$ do zbioru S trafią klauzule $\{P(a)\}$ oraz $\{\neg P(f(a))\}$. Kolejne podstawieni spowoduje dodanie klauzul $\{P(f(a))\}$ oraz $\{\neg P(f(f(a)))\}$. To już wystarczy aby wykazać niespełnialność formuły F, gdyż w wyniku zastosowania rezolucji na klauzulach $\{\neg P(f(a))\}$ oraz $\{P(f(a))\}$ otrzymujemy klauzulę pustą. Zwykle jednak sytuacja jest znacznie bardziej skomplikowana, co pokazuje drugi przykład.

Przykład 2

Niech dana będzie następująca (niespełnialna) formuła

$$F = \forall_x \forall_y ((\neg P(x) \vee \neg P(f(a)) \vee Q(y)) \wedge (P(y)) \wedge (\neg P(g(b, x)) \vee \neg Q(b))).$$

Zgodnie z definicją macierzy dla formuły, mamy

$$F^* = ((\neg P(x) \vee \neg P(f(a)) \vee Q(y)) \wedge (P(y)) \wedge (\neg P(g(b, x)) \vee \neg Q(b))),$$

co w postaci klauzulowej zapisujemy jako

$$F^* = \{\{\neg P(x), \neg P(f(a)), Q(y)\}, \{P(y)\}, \{\neg P(g(b, x)), \neg Q(b)\}\}.$$

Wypisywanie elementów rozszerzenia Herbranda w tym przypadku jest już mniej celowe niż poprzednio, dlatego wskażemy jedynie „operacje" jakie należy wykonać aby uzyskać klauzulę pustą. Ponumerujmy otrzymane do tej pory klauzule

1. $\{\neg P(x), \neg P(f(a)), Q(y)\}$
2. $\{P(y)\}$
3. $\{\neg P(g(b, x)), \neg Q(b)\}$

Z klauzuli pierwszej przez podstawienie $[x|f(a)][y|b]$ otrzymujemy klauzulę czwartą

$$\{\neg P(f(a)), Q(b)\}$$

Z klauzuli drugiej przez podstawienie $[y|f(a)]$ oraz $[y|g(b, a)]$ otrzymujemy odpowiednio klauzulę piątą i szóstą

$$\{P(f(a))\}$$

$$\{P(g(b, a))\}$$

Z klauzuli trzeciej przez podstawienie $[x|a]$ otrzymujemy klauzulę siódmą

$$\{\neg P(g(b, a)), \neg Q(b)\}$$

Z klauzuli czwartej i piątej otrzymujemy klauzulę ósmą

$$\{Q(b)\}$$

Z klauzuli szóstej i siódmej otrzymujemy klauzulę dziewiątą

$$\{\neg Q(b)\}$$

Z klauzuli ósmej i dziewiątej otrzymujemy klauzulę dziesiątą – pustą.

Zauważmy, że w tym przypadku jedna klauzula ($\{P(y)\}$) posłużyła do otrzymania dwóch różnych klauzul ($\{P(f(a))\}$, $\{P(g(b, a))\}$). Ponad to w wyniku zastosowania podstawienia, klauzula może zmniejszyć liczebność elementów jakie zawiera (klauzula $\{\neg P(x), \neg P(f(a)), Q(y)\}$ „zmieniła się" w klauzulę $\{\neg P(f(a)), Q(b)\}$).

Z podanych przykładów widać, że proces dopasowywania klauzul do siebie wcale nie musi być prosty i oczywisty. Musimy więc usystematyzować w jakiś sposób procedurę dopasowywania, co jest celem następnego podrozdziału.

28.5 Unifikacja

Definicja 28.14. *Podstawienie sub nazywamy* ***unifikatorem*** *(ang.* unifier*) dla pewnego skończonego zbioru literałów* $L = \{L_1, \ldots, L_n\}$ *jeśli*

$$L_{1sub} = \cdots = L_{nsub}.$$

Unifikator sub nazywamy ***najbardziej ogólnym unifikatorem*** *(ang.* most general unifier*) jeśli dla każdego unifikatora* sub' *istnieje podstawienie* s *takie, że* $sub' = sub\ s$.

Twierdzenie 28.11 (Robinson)**.** *Dla każdego unifikowalnego zbioru literałów istnieje najbardziej ogólny unifikator.*

Dowód. Dowód tutu. □

Algorytm unifikacji

//tutu//

Przykład działania algorytmu unifikacji

Czy można zunifikować literał

$$L_1 = \neg P(f(z, g(a, y)), h(z))$$

oraz

$$L_2 = \neg P(f(f(u, v), w), h(f(a, b)))$$

Krok 1. Po zastosowaniu podstawienia $[z|f(u, v)]$ otrzymujemy

$$\neg P(f(f(u, v), g(a, y)), h(f(u, v)))$$
$$\neg P(f(f(u, v), w), h(f(a, b)))$$

Krok 2. Po zastosowaniu podstawienia $[w|g(a, y)]$ otrzymujemy

$$\neg P(f(f(u, v), g(a, y)), h(f(u, v)))$$
$$\neg P(f(f(u, v), g(a, y)), h(f(a, b)))$$

Krok 3. Po zastosowaniu podstawienia $[u|a]$ otrzymujemy

$$\neg P(f(f(a, v), g(a, y)), h(f(a, v)))$$
$$\neg P(f(f(a, v), g(a, y)), h(f(a, b)))$$

Krok 4. Po zastosowaniu podstawienia $[v|b]$ otrzymujemy

$$\neg P(f(f(a,b),g(a,y)),h(f(a,b)))$$
$$\neg P(f(f(a,b),g(a,y)),h(f(a,b)))$$

Zatem ostatecznie otrzymujemy podstawienie

$$sub = [z|f(u,v)][w|g(a,y)][u|a][v|b],$$

które jest najbardziej ogólnym unifikatorem dla $L = \{L_1, L_2\}$. Ponad to

$$L_{sub} = \{\neg P(f(f(a,b),g(a,y)),h(f(a,b)))\}.$$

Dzięki pojęciu unifikacji możemy teraz zdefiniować rezolucję dla przypadku rachunku predykatów.

Definicja 28.15 (Rezolucja dla rachunku predykatów). *Niech C_1, C_2 i R będą klauzulami rachunku predykatów. Wówczas R nazywane jest resolwentą C_1 i C_2 jeśli zachodzą następujące warunki.*

1. *Istnieją podstawienia sub_1 i sub_2 takie, że C_{1sub_1} oraz C_{2sub_2} nie zawierają takich samych zmiennych.*

2. *Istnieją zbiory literałów $L_1^1, \ldots, L_m^1 \in C_{1sub_1}$ oraz $L_1^2, \ldots, L_n^2 \in C_{2sub_2}$ takie, że zbiór $L = \{\overline{L_1^1}, \ldots, \overline{L_m^1}, L_1^2, \ldots, L_n^2\}$ jest unifikowalny. Niech sub będzie najbardziej ogólnym unifikatorem dla L.*

3. *R jest postaci*

$$R = ((C_{1sub_1} \setminus \{L_1^1, \ldots, L_m^1\}) \cup (C_{2sub_2} \setminus \{L_1^2, \ldots, L_n^2\}))_{sub}$$

Przykład

Niech

$$C_1 = \{P(f(x)), \neg Q(z), P(z)\}$$

$$C_2 = \{\neg P(f(x)), R(g(x), a)\}$$

Wówczas

$$\begin{aligned} L_1^1 &= P(f(x)) \\ L_2^1 &= P(z) \\ L_1^2 &= \neg P(x) \\ sub_1 &= [] \\ sub_2 &= [x|u] \\ sub &= [z|f(x)][u|f(x)] \\ R &= \{\neg Q(f(x)), R(g(f(x)), a)\} \end{aligned}$$

W dalszej części posługiwać się będziemy także notacją zapożyczoną z rachunku zdań z definicji 27.11.

Twierdzenie 28.12 (Twierdzenie o rezolucji dla rachunku predykatów)**.** *Niech F będzie zamkniętą formułą Skolema z odpowiadającą jej macierzą F^* w postaci CNF. Wówczas F jest niespełnialne wtedy i tylko wtedy, gdy formuła pusta należy do $Res^*(F^*)$.*

Dowód. Dowód tutu. □

28.6 Ograniczenia na rezolucje

Rezolucja w swojej ogólnej postaci może prowadzić do olbrzymiego wzrostu zbioru rozważanych klauzul przez co może być bardzo trudno „trafić" na takie klauzule, z których będzie można otrzymać klauzulę pustą. Dlatego też wprowadza się pewne ograniczenia związane z tym jakiej postaci muszą być klauzule aby rezolucja mogła być wykonana. Tym samym znacznie redukuje się ilość rozważanych przypadków.

Definicja 28.16. *Rezolucję nazywamy* ***zupełną*** *(ang.* complete*) jeśli dla każdej niespełnialnej formuły F można wykazać jej niespełnialność za pomocą rezolucji.*

Okazuje się, że wyjątkowo dobre w kontekście ograniczania gwałtownego przyrostu klauzul w wyniku stosowania rezolucji są klauzule Horna. Zachodzą następujące twierdzenia.

Twierdzenie 28.13. *Unit resolution is complete for the class of Horn clauses.*

Dowód. Dowód tutu. □

Twierdzenie 28.14. *SLD-resolution is complete for the class of Horn clauses.*

Dowód. Dowód tutu. □

Twierdzenie 28.15. *Input resolution is complete for the class of Horn clauses.*

Dowód. Dowód tutu. □

Występujące w twierdzeniach nazwy //tutu// określają różne sposoby zawężania zbioru rozpatrywanych klauzul.

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] J. Lu, J. J. Mead. *Prolog. A Tutorial Introduction*, Computer Science Department Bucknell University Lewisburg, PA 17387.

[6] U. Endriss, *Lecture notes. An Introduction to Prolog Programming*, University of Amsterdam, Version: 7 July 2007.

[7] U. Schöning, *Logic for Computer Scientists*, Reprint of the 1989 Edition, Birkhäuser, Boston, 2008.

[8] L. Sterling, E. Shapiro. *The Art of Prolog*, 2nd edition, MIT Press, 1994.

[9] J. McCarthy, *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Artificial Intelligence Project — RLE and MIT Computation Center, Memo 8, March 4, 1959.

[10] J. McCarthy, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, Communications of the ACM, Volume 3, Issue 4 (April 1960), pp. 184 - 195, 1960.

[11] J. McCarthy, *LISP 1.5 Programmer's Manual*, The MIT Press, 1962.

[12] J. McCarthy, *History of Lisp*, Artificial Intelligence Laboratory, Stanford University, 12 February 1979: http://www-formal.stanford.edu/jmc/history/lisp/lisp.html Podstrona http://www-formal.stanford.edu/jmc/history/lisp/node3.html, ostatnia modyfikacja: John McCarthy, Fri Jul 26 22:37:29 PDT 1996 (dostep: 2010-02-17).

[13] D. Eastlake 3rd, C. Manros, E. Raymond, *RFC 3092: Etymology of „Foo"*, http://www.ietf.org/rfc/rfc3092.txt

[14] T. Hart, M. Levin, *AI Memo 39-The new compiler*.

[15] E. S. Raymond, *The Jargon File*, version 4.4.7, http://www.catb.org/~esr/jargon/, http://www.catb.org/~esr/jargon/html/L/LISP.html

[16] E. S. Raymond, *How To Become A Hacker*, http://www.catb.org/~esr/faqs/hacker-howto.html

[17] P. Seibel, *Practical Common Lisp*, Apress, 2005. Cała książka dostępna jest także pod adresem http://gigamonkeys.com/book/ (dostep 2010-02-14).

[18] D. S. Touretzky, *Common Lisp: A Gentle Introduction to Symbolic Computation*, (książka dostępna pod adresem http://www.cs.cmu.edu/~dst/LispBook/index.html, dostep 2010-02-20).

[19] L. Xah, *Jargons of Info Tech industry (A Love of Jargons)*, http://xahlee.org/UnixResource_dir/writ/jargons.html, dostęp 2010-03-23.

[20] http://lispers.org/

[21] http://www.paulgraham.com/diff.html

[22] http://www.paulgraham.com/arcll1.html

[23] Lisp Operating System, http://sourceforge.net/projects/losak/ (dostep 2010-02-15).

[24] E. Friedman-Hill, *Jess in Action. Java Rule-based Systems*, Manning Publications Co., 2003. Książka dostępna na stronie wydawnictwa: http://www.manning.com/friedman-hill/ w wersji PDF e-book.

[25] http://www.jessrules.com/

Spis rysunków

Spis tabel

Skorowidz

„z dołu do góry”, 74
„z góry na dół”, 73
Łukasiewicz, Jan, 104

aksjomaty Peano, 85
akumulator, 70
atom, 17

Church, Alonzo, 103
ciało, 20

domknięcie, 136
dopasowywanie wyrażeń, 29

enkapsulacja danych, 5

fakt, 16, 20
forma zdaniowa, 27
funkcja zdaniowa, 27

głowa, 20, 44

instrukcja warunkowa, 104

język
- deklaratywny, 4, 103
- funkcjonalny, 103
- funkcyjny, 103
- obiektowy, 5
- opisowy, 4

klasa, 5
klauzula, 20

Lisp
- atom, 107
- domknięcie, 136
- fałsz, 113
- liczba, 107
- lista, 107
  - szczególne przypadki, 111
- napis, 109
- nazwa (symbol), 109
- nil, 113
- prawda, 113
- stałe, 139
- symbol (nazwa), 109
- typowanie dynamiczne, 131
- zmienna dynamiczna, 136
- zmienna globalna
  - niezwiązana, 137
- zmienna leksykalna, 136

lista, 43
- głowa, 44
- ogon, 44

Mechanizm nawracania, 53
mechanizm odcięcia, 60

nawracanie, 42, 53
Notacja Łukasiewicza, 104
notacja polska, 104
- odwrotna, 104

obiekt, 4

obliczanie celu, 31
odcięcie, 60
odwrotna notacja polska, 104
ogon, 44

paradygmat obiektowości, 4
Peano
    aksjomaty, 85
    Giuseppe, 85
predykat, 26
programowanie
    deklaratywne, 103
    funkcjonalne, 103
    funkcyjne, 103
    zorientowane obiektowo, 5
Prolog
    „z dołu do góry”, 74
    „z góry na dół”, 73
    akumulator, 70
    atom, 17
    dopasowywanie wyrażeń, 29
    fakt, 16, 20
    forma zdaniowa, 27
    funkcja zdaniowa, 27
    klauzula, 20
    lista, 43
        głowa, 44
        ogon, 44
    nawracanie, 42, 53
    obliczanie celu, 31
    odcięcie, 60
    predykat, 26
    program, 21
    reguła, 12, 16, 20
        ciało, 20
        głowa, 20
    relacja, 15
    rezolucja, 32, 41
    stała, 25
    struktura, 19, 26
    term, 17
        złożony, 19
    termy pasujące, 29
    unifikacja, 30, 39
    zapytanie, 13, 21
    zmienna, 18, 25
    zmienna anonimowa, 18

rachunek lambda, 103
rachunek predykatów pierwszego rzędu, 27
reguła, 12, 16, 20
    ciało, 20
    głowa, 20
relacja, 15
rezolucja, 32, 41

stała, 25
stałe, 139
struktura, 19, 26

term, 17
    złożony, 19
termy pasujące, 29
typowanie dynamiczne, 131

unifikacja, 30, 39

zapis przedrostkowy, 104
zapytanie, 13, 21
zmienna, 18, 25
zmienna anonimowa, 18
zmienna dynamiczna, 136
zmienna globalna
    niezwiązana, 137
zmienna lepsykalna, 136