Wykład 1

Prolog — język przetwarzania symbolicznego przeznaczony do rozwiązywania problemów dotyczących obiektów i relacji miedzy nimi.

Program Prologowy składa się z klauzul argumentami relacji mogą być konkretne obiekty, stałe lub ogólne obiekty (zmienne).

Zakończeniem każdego pytania czy reguły jest znak [**.**]

Pytania składają się z jednego lub więcej celów. Ciąg celów należy połączyć odpowiednik spójnikiem logicznym (np. koniunkcja)

Przykład:

?- rodzic(X, ala), rodzic(X, ola).

Odpowiedzią na zadane pytanie jest:

1. Na pytanie szczególne np. X = ania
2. Na pytanie ogólne: True lub False

W prologu jeżeli nie jest możliwa weryfikacja odpowiedzi w oparciu o zdefiniowane fakty to odpowiedz brzmi False.

Jeżeli istnieje kilka możliwych odpowiedzi Prolog może wygenerować ich tyle ile zażyczy sobie użytkownik

Program prologowy może zawierać również fakty, opisujące wybrane cechy obiektów.
Wykorzystuje się do tego „relacje unarne”

np. kobieta(ewa).

Fakty prologowe są **zawsze** (bezwarunkowo) prawdziwe.
Reguły Prologowe są prawdziwe, jeżeli spełniony jest pewien warunek.

Semantyka:

dziecko(Y,X) :- rodzic(X,Y).

Lewa część reguły (przed znakiem [:-]) to część konkluzyjna. Prawa strona reguły (za znakiem [:-]) to część warunkowa.

Brak bezpośrednich zawsze prawdziwych faktów powoduje odwołanie do reguł.
Reguła jako uogólnienie może być zastosowana do konkretnych obiektów wiążąc zmienne.

Koniunkcja warunków odbywa się za pomocą operatora [**,**]

Przykład:

matka(X,Y) :- rodzic(X,Y), kobieta(X).

Programy prologowe można rozszerzyć poprzez definiowanie nowych klauzul.
Klauzula prologowa składa się z nagłówka i ciała;
Ciało to lista celów oddzielonych przecinkami, które traktujemy jako koniunkcję
Istnieją trzy rodzaje klauzul: fakty, reguły, pytania.

Fakty opisują to co zawsze jest bezwarunkowo prawdziwe. Fakty to klauzule które maja nagłówek i puste ciało;

Reguły opisują to czego prawdziwość zależy od pewnego warunku. Reguły to klauzule które mają zarówno nagłówek jak i niepuste ciało

Za pomocą pytań można się dowiedzieć co jest prawdą a co fałszem.

W trakcie działania zmienne mogą być zamieniane przez inne obiekty; proces ten określamy jako unifikację.

Zakładamy że wszystkie zmienne klauzuli są poprzedzone kwantyfikatorem uogólnionym.

Rekurencja

Rekurencja jest podstawową metodą programowania w prologu (nie ma w prologu pętli)

Przykład:

```
przodek(X,Z) :- rodzic(X,Z).                 %przodek bezpośredni
przodek(X, Z) :- rodzic(X,Y), przodek(Y,Z).  %przodek pośredni
```

Generowanie odpowiedzi na postawione pytanie odbywa się w następujący sposób:

1. Pytanie (cel lub ciąg celów)
2. Sprawdzić czy cel jest spełniony
3. Czy cel jest prawdziwy przy założeniu, że zdefiniowanie relacje są prawdziwe?
4. Dowieść że cel jest logiczną konsekwencją zdefiniowanych faktów i reguł
5. Jeśli cel zawiera zmienne, podać dla jakich wartości jest spełniony

Generowanie odpowiedzi (nieformalna interpretacja deklaratywna):

1. Znany fakt: rodzic(robert, ola).
2. Na podstawie reguły prb wnioskujemy, że przodek(robert, ola) bo:

rodzic(robert, ola) —> przodek(robert, ola).

3. Znany fakt: rodzic(tomek, robert).
4. Korzystając z faktów:

przodek(robert, ola) i rodzic(tomek, robert) wnioskujemy na podstawie reguły prp, że przodek(tomek, ola):

rodzic(tomek, robert) i przodek(robert, ola) —> przodek(tomek, ola)

Generowanie odpowiedzi (nieformalna interpretacja proceduralna)

1. Cel: przodek(tomek, ola).
2. Poszukiwanie odpowiednich klauzul
3. Dopasowanie do nagłówków klauzul prb i prp
4. Klauzula prb: przodek(X, Z) :- rodzic(X, Z). wiązanie zmiennych X = tomek, Z = ola

5. Nowy cel: rodzic(tomek, ola)
6. Brak klauzuli pasującej do celu rodzic(tomek, ola)
7. Nawrót do pierwotnego celu (przodek(tomek, ola)) i analiza drugiej pasującej klauzuli, prp:

przodek(X,Z) :- rodzic(X,Y), przodek(Y,Z).

8. Wiązanie zmiennych: X = tomek, Z = ola
9. Nowy cel (dwa pod cele):

rodzic(tomek, Y), przodek(Y, ola)

10. Pierwszy pod cel: rodzic(tomek, Y).
11. Dopasowanie do klauzuli rodzic(tomek, robert)

Wiązanie zmiennych: Y = robert

12. Drugi pod cel: przodek(robert, ola)
13. Dopasowanie do klauzul prb i prp
14. Klauzula prb:

przodek(X', Z') :- rodzic(X', Z').
wiązanie zmiennych: X' = robert, Z' = ola

15. Nowy cel: rodzic(robert, ola)
16. Dopasowanie do klauzuli (faktu)

rodzic(robert, ola)

17. Koniec wnioskowania - cel główny spełniony

Interpretacja deklaratywna pozwala określić jaki będzie rezultat programu prologowego

Interpretacja proceduralna pozwala określić w jaki sposób ten rezultat zostanie otrzymany czyli w jaki sposób zostaną użyte relacje zdefiniowane w programie

Programowanie w języku prolog powinno opierać się tylko - o ile to możliwe - na interpretacji deklaratywnej programu prologowego.

Podsumowanie 1 wykładu

Programowanie w prologu opiera się na definiowaniu relacji i zadawaniu pytań dotyczących relacji

Program prologowy składa się z klauzul. Mamy trzy rodzaje klauzul: fakty, reguły, pytania

Relacje definiowane są za pomocą faktów które maja postać n-tek zawierających obiekty, które relacje spełniają lub reguły które je opisują

Pytania dotyczące relacji przypominają zadawanie pytań w systemach baz danych odpowiedzi składają się ze zbioru obiektów, spełniających cel zawarty w pytaniu

Sprawdzanie czy dany obiekt spełnia cel opiera się złożonym procesie obliczeniowym wykorzystującym mechanizmy logicznego wnioskowania i nawroty; wszystkie szczegóły tego procesu pozostawają ukryte przed programista

Wyróżniamy dwie interpretacje programu prologowego: deklaratywna i proceduralna; interpretacja deklaratywna powinna być jedyna wykorzystywana przez programista; są jednak sytuacje kiedy nie da się uniknąć interpretacji proceduralnej

Podstawowe pojęcia:
klauzula, fakt, reguła, pytanie, nagłówek i ciało klauzuli, reguła rekurencyjna, definicja rekurencyjna, zmienna, wiązanie zmiennej, cel, cel spełniony, cel niespełniony, nawroty, interpretacja deklaratywna i proceduralna.

Wykład 2

Złożone struktury danych:

Dane dzieli się na:

1. proste: stałe(symboliczne lub numeryczne) i zmienne
2. złożone (listy, sekwencje i struktury które tworzy sam programista)

Stałe symboliczne (atomy, stałe atomowe) składają się z dużych liter, małych liter, cyfr, znaków specjalnych i są ciągami (konwencja zakłada że nazwa zaczyna się z małej litery, dopuszcza się ciągi złożone z samych znaków specjalnych dlatego że mechanizmy metadanych pozwalają programistom definiować własne operatory). Istnieją też łańcuchy znakowe używające apostrofa zamiast cudzysłowie np. ‚Ania'

Zmienne to ciągi liter, cyfr i znaków podkreślenia zaczynające się od dużej litery lub podkreślenia

Zmienna anonimowa [_] taka zmienna reprezentuje obiekt w relacji ale jego wartość nie wymaga zapisania do przyszłego użytku. Jej wartość nie jest istotna do spełnienia danej relacji.

Przykład:

```
?- ma_dziecko(X) :- rodzic(X, _ ).
```

Nie interesuje nas imię dziecka, obchodzi nas tylko czy X ma dziecko

Każde wystąpienie zmiennej anonimowej jest unikalnym wystąpieniem zmiennej.
Każde takie wystąpienie reprezentuje inna zmienna.

Przykład:

```
?- ktos_ma_dziecko :- rodzic( _ , _ ).
```

Jest równoważne:

```
?- ktos_ma_dziecko :- rodzic(X, Y).
```

A nie:

```
?- ktos_ma_dziecko :- rodzic(X, X).
```

Użycie zmiennej anonimowej w zapytaniu sprawi że jej wartość nie będzie wyświetlona.

Przykład:

```
?- rodzic(X,_).
X = tomek;

?- rodzic(_, _).
true;
```

W odpowiedzi dostaniemy czy istnieje jakikolwiek rodzic. Tak naprawdę używając tego zapytania pytamy czy istnieje JAKIKOLWIEK rodzic.

Zasięg identyfikatora zmiennej jest ograniczony tylko do jednej klauzuli
Każda reguła jest oddzielnym fragmentem wiedzy przedmiotowej, nie ma żadnego przekazywania wartości zmiennej pomiędzy jedna reguła a druga reguła.

Przykład:

```
ma_dziecko(X) :- rodzic(X, _).
dziecko(X, Y) :- rodzic(Y,X).
```

Dane złożone mają charakter strukturalny. Składają się z kilku elementów, tworzących jeden obiekt; w szczególności składowymi struktury mogą być również dane złożone

Dane złożone tworzone są za pomocą funktora i argumentów traktowanych razem jako jeden obiekt programu

Przykład:

```
date(1, october, 2000)
```

Mimo wyglądu predykatu jest to struktura danych. Wszystko zależy od specyficznego kontekstu w którym dany obiekt się pojawia.

Składowymi struktury danych mogą być stałe ale także zmienne. Dane strukturalne w prologu są czymś w rodzaju szablonu daty.

Przykład:

```
date(Day ,march, 2000)
Day = zmienna
```

Nie jest to nic niezwykłego i może być przetwarzane w czasie działania programu.

Wszystkie dane strukturalne w prologu maja drzewiasta strukturę, nazywamy ja postacią kanoniczna tej struktury. Funktorem główny reprezentuje korzeń drzewa, natomiast w gałęziach znajdują się elementy struktury.

Przykład trójkąta na płaszczyźnie:

```
P1 jako point(1,1)
P2 jako point(2,3)
S jako seg(P1, P2) czyli seg(point(1,1), point(2,3))
T jako triangle(point(4,2), point(6,4), point(7,1))
```

Każda z tych struktur maja swoja reprezentacja w interpreterze. W języku prolog nie ma ograniczenia ile składowych może być w strukturze lub jak głęboko mogą być one zagnieżdżone. Wszystko zależy od programisty i jego potrzeb.

Dane strukturalne identyfikowane są na podstawie 2 cech:

1. Nazwa symboliczna funktora
2. Arność

Oznacza to ze dwa funktory o tej samej nazwie lecz innej arności są różnymi obiektami.

Struktury oparte na polaczeniu binarnym można odtworzyć w prologu za pomocą skomplikowanych sieci połączeń

Pojecie formalne termu

Wszelkie obiekty danych w Prologu. Do termów zaliczamy:

1. Stałe i zmienne prologowe

2. Struktury reprezentowane za pomocą funktorów o dowolnej arności, których argumentami są inne termy (należy zwrócić uwagę na to ze znajduje się rekurencyjne definiowanie tych struktur). Dzięki temu możemy tworzyć struktury o dowolnej komplikacji.

Mechanizm Uzgadniana

W prologu nie ma możliwości deklarowania obiektów danych. Dlatego należy korzystać z mechanizmu który weryfikuje z jaka dokładnie struktura mamy do czynienia.
Jeśli zgodność nie występuje to mechanizm wykrywanie uzgodnienia aby termy były identyczne

O uzgodnieniu mówimy gdy:

1. Termy są identyczne
2. Zmienne zawarte w obu termach mogą przyjąć wartości, dla których termy te stana się identyczne

Przykład:

date(D, M, 1993) oraz date(D1, may, Y1) można uzgodnić:

D = D1
M = may
Y1 = 1993

Proces uzgadniana nie musi kończyć się sukcesem, może być zakończony porażką, nie należy traktować tego jako błąd, czasami powód dla którego nie dochodzi do identycznego uzgodnienia może być prozaiczny a innym razem może to być skomplikowane zagadnienie.

NIE można uzgadniać struktur o tej samej arności ale innych nazwach funktorów

Przykład:

date(X,Y,Z) oraz point(X,Y,Z)

Proces uzgadniana termu może być realizowany na życzenie programisty za pomocą znaku [=] który można wywołać dla dwóch struktur aby zażądać ich uzgodnienia. Wartości które zostana uzgodnione są specyficzne ponieważ mechanizm uzgadniana poszukuje najbardziej ogólnego podstawienia, wiec stara się nie narzucać wartości zmiennym, jedyne co musi być spełnione to tożsamość tych zmiennych

Formalnie mechanizm opiera się na algorytmie unifikacji termów (wyznacza najogólniejszy możliwy unifikator) czyli zbiór podstawień który najmniej ogranicza zbiór wartości zmiennych. Przebiega on systematycznie wewnątrz drzewa i towarzyszy mu propagacja związanych wartości zmiennych do wszystkich wystąpień zmiennych w dalszych częściach struktur.

Przebieg procesu unifikacji

1. Jeżeli dwa termy S i T , są stałymi to uzgodnienie zachodzi gdy są identyczne
2. Jeżeli S jest zmienna a T dowolnym termem, to uzgodnienie zachodzi gdy S przypiszemy wartość T;
3. Jeżeli S i T sa obiektami złożonymi to uzgodnienie zachodzi, gdy Si T maja ten sam funktor i zachodzi uzgodnienie pomiędzy wszystkimi ich składowymi (definicja rekurencyjna)

Ostateczny rezultat jest złożeniem wszystkich uzgodnień

Proces uzgadniana przebiega rekurencyjnie od korzenia do liści struktur drzewiastych

Drugim kontekstem wywołania mechanizmu uzgadniania jest próba spełnienia jakiegoś celu, interpreter wyszukując rozwiązania rozpoczyna od uzgadniania miedzy nagłówkiem klauzuli a realizowanym celem.

Niejawnie uzgodnienie realizuje pewnie przetwarzanie i to przetwarzanie nie musi dotyczyć calej struktury.

Przykład:

```
vertical(seg(point(2,3), P)).
P = point(2,Y)
```

Zapytanie jeszcze ogólniejsze:

```
vertical(S), horizontal(S).
S = seg(point(X,Y), point(X,Y)).
```

Deklaratywna i proceduralna interpretacja programu prologowego

Rozważmy klauzulę P w postaci P :- Q,R.

1. Interpretacja deklaratywna klauzuli P:

P jest prawdziwe wtedy, gdy Q i R są prawdziwe.

2. Interpretacja proceduralna klauzuli P:

Żeby osiągnąć cel P, najpierw musisz osiągnąć podcel Q, a potem podcel R.

Różnica: Interpretacja proceduralna określa nie tylko logiczny związek miedzy nagłówkiem reguły i jej ciałem ale również porządek w jakim maja być osiągane podcele.

Deklaratywna interpretacja programu prologowego

Deklaratywna interpretacja programu prologowego określa czy dany cel jest spełniony, to dla jakich wartości zmiennych.

Instancja klauzuli to taka klauzula w której za każdą zmienna podstawiono jakiś term.

FORMALNA DEFINICJA

Dla danego programu prologowego i celu G:
Cel G jest prawdziwy(spełniony albo wynika logicznie z programu) wtedy i tylko wtedy gdy:

1. istnieje w programie klauzula C, taka że:
2. istnieje instancja J klauzuli C taka, że:
   zachodzi uzgodnienie nagłówka instancji J z G, oraz
   wszystkie podcele w ciele instancji J są prawdziwe spełnione przy tym uzgodnieniu

W przypadku celów złożonych (ich koniunkcja), mówimy ze cala lista celów jest spełniona gdy wszystkie jej cele są rownocześnie spełnione dla tych samych wiązań (wyników uzgodnień) zmiennych.

Operator dysjunkcji celów [;]
Można go zastąpić wbudowanym mechanizmem nawrotów.

Koniunkcja ma wyższy priorytet niż dysjunkcja wiec należy stosować odpowiednie nawiasy okrągłe

Interpretacja proceduralna programu prologowego

Interpretacja proceduralna programu prologowego określa jak Prolog odpowiada na pytania (w jaki sposób spełnia cele).
Interpretacja proceduralna oznacza wykonanie procedury obliczeniowej która doprowadzi do spełnienia listy celów z uwzględnieniem danego programu prologowego.

FORMALNA DEFINICJA

1. Jeśli lista celów jest pusta to koniec z sukcesem

2. Jeśli lista celów nie jest pusta to kontynuuj operacje ANALIZA

3. ANALIZA:
   Przeszukiwanie całego programu(od początku do końca) do pierwszej klauzuli C, której nagłówek można uzgodnić z pierwszym celem G1. Jeśli nie ma takiej klauzuli to koniec z porażką. Jeśli taka klauzula jest i ma postać:

   H :- B1, B2, … , Bn.

   to zmiana nazw zmiennych w C na unikalne zmienne(nowa instancja tej klauzuli: C'), rozne od zmiennych zawartych w liście G1, G2, … , Gm:
   H' :- B1',B2',… Bn' instancji dla C oznaczona jako C'
   Uzgodnienie G1 z H' i rezultat w postaci zbioru podstawień S.
   W liście celu głównego G1,G2,…,Gm zamieniamy G1 na listę warunków B1', B2',…,Bn' i otrzymujemy nowa listę celów:
   B1',B2',…,Bn',G2,…,Gm
   (jeśli C jest faktem wtedy n = 0 i nowa lista celów jest krótsza niż lista pierwotna; zmniejszenie się tej listy doprowadzi w końcu do listy pustej i tym samym osiągnięcia celu głównego).

Zamieniamy zmienne na nowej liście celów zgodnie z podstawieniem ze zbioru S i otrzymujemy ostateczna listę:
B1",B2",…,Bn",G2',…,Gm'

4. Wykonujemy rekurencyjnie cala powyższą procedurę dla nowej listy celów B1',B2',…,Bn',G2', …,Gm'. Jeśli realizacja tego celu zakończy się sukcesem to realizacja celu pierwotnego G1,G2,…,Gm również kończy się sukcesem. Jeśli lista celów B1',B2',…,Bn',G2',…,Gm' zakończy się porażką, to porzucamy dalsze przetwarzanie tej listy i powracamy do kontynuowania operacji ANALIZA dla pozostałej części programu prologowego. Przeglądamy dalej program, poczynając od klauzuli następnej po klauzuli C(C jest klauzula która użyliśmy jako ostaniej) i próbujemy wykorzystać inna klauzule, której nagłówek można uzgodnić z celem G1.

UWAGI DO DEFINICJI

1. Procedura nie określa w jaki sposób otrzymywany jest ostateczny zbiór podstawień zmiennych S. Zbiór podstawień który prowadzi do spełnienia pierwszego celu z listy podlega najczęściej dalszym uściśleniom w wyniku realizacji kolejnych celów

2. Kiedy rekurencyjnie wywołanie procedury prowadzi do porażki dokonujemy nawrotu do tego miejsca w programie w którym wybrano złą klauzule (klauzule C), porzucając wszystkie rezultaty jej przetwarzania, włącznie z dokonanymi podstawieniami zmiennych. Taka realizacja celów gwarantuje systematyczna analiza wszystkich alternatywnych ścieżek przetwarzania prowadzących do spełnienia celu lub kończy się stwierdzeniem (po sprawdzeniu ich wszystkich), ze cel nie jest spełniony

   Zapamietać należy tylko to że interpreter prowadzi „dziennik" zmian który pozwala mu w sposób systematyczny dokonywania wiązań i wykonywać nawroty.

Problem Mały i Banana

„Małpa przy drzwiach wejściowych do pomieszczenia.
Małpa stoi na podłodze.
Pudło stoi pod oknem.
Małpa nie ma banana.
Banan wisi na środku sufitu.
Małpa nie może dosięgnąć banana."

Pytanie: **Czy małpa może zjeść banana?**

Możliwe ruchy małpy:

1. chwytanie banan
2. Wspinanie się na pudło
3. Przesuwanie pudla
4. Poruszanie się po pomieszczeniu

Nie wszystkie ruchy są dopuszczalne w każdym stanie. Każdy ruch wymaga spełnienia określonych warunków początkowych, niezbędnych do jego realizacji.

Reprezentacja w prologu ruchów:

move(<stan1>, <ruch>, <stan2>).

Przykład najprostszego ruchu:

```
move( state(middle, onbox, middle, hasnot), grasp,
state(middle, onbox, middle, has)).
```

Klauzuli definiującej ruch nie musi odpowiadać tylko jeden z ruchów możliwych do wykonania w modelowanym świecie, może to być cały zbiór ruchów, czyli …

Przykład trudniejszego ruchu:

```
move(  state(Pos1, onfloor, Box, Has), walk(Pos1,Pos2), state(Pos2, onfloor, Box, Has)).
```

Małpa może zmienić swoje położenie w poziomie niezależnie od położenia pudła i faktu schwytania banana:

1. Zmiana położenia z miejsca Pos1 na miejsce Pos2
2. Małpa znajduje się na podłodze zarówno przed jak i po wykonaniu ruchu
3. Pudło pozostaje w tym samym położeniu Box
4. Stan posiadania małpy Has również nie ulega zmianie

Wykorzystanie zmiennych uogólniło definicję na wszystkie przypadki, w których możliwe jest poruszanie się po pomieszczeniu.

Sformułowanie celu:

Klauzula: canget(<stan>) gdzie <stan> = stan docelowy

W dowolnym stanie w którym małpa ma juz banana klauzula canget powinna być zawsze prawdziwa - nie ma potrzeby wykonywania wtedy żadnych ruchów czyli:

```
canget(state(_, _, _, has)).
```

W każdym innym przypadku należy wykonać co najmniej jeden ruch małpa może schwytać banana w stanie State1, o ile istnieje ruch ze stanu State1 do takiego stanu State2, w którym małpa jest w stanie dosięgnąć banana:

```
canget(State1) :- move(State1, Move, State2), canget(State2).
```

canget jest rekurencyjną klauzulą

Pełne rozwiazanie problemu:

```
move( state(middle, onbox, middle, hasnot),
          grasp,
          state(middle, onbox, middle, has)).
move( state(P, onfloor, P, H),
          climb,
          state(P, onbox, P, H)).
move( state(P1, onfloor, P1, H),
          push(P1, P2),
          state(P2, onfloor, P2, H).
move( state(P1, onfloor, B, H),
          walk(P1, P2),
          state(P2, onfloor, B, H)).

canget(state(_, _, _, has)).
canget(State1) :-
                    move(State1, Move, State2),
                    canget(State2).
```

Porządek klauzul prologowych i celów

Porządek klauzul w programie prologowym ma kluczowe znaczenie dla efektywności i skuteczności programu

Deklaratywnie poniższa klauzula jest tautologią:

```
P :- P.
```

Jednak prowadzi to do nieskończonej pętli, wynika to z interpretacji proceduralnej programu.

Architektura von Neumanna używana w komputerach i procesorach nie pozwala na pierwotne abstrakcyjne wnioskowanie w logice matematycznej.

Pętle mogą być efektem niewłaściwej kolejności reguł i celów. Proceduralna interpretacja programu powoduje ze preferowane są reguły i/lub cel zdefiniowane wcześniej w programie. Wszystkie warunki sprawdzane są sekwencyjne. Oznacza to ze mogą się pojawić w poprawnym programie cykle tylko z powodu zaniedbań odnośnie kolejności klauzul.

Program prologowy może nie znaleźć rozwiązania nawet jeżeli rozwiazanie to istnieje (pętle nieskończone)

Program prologowy może być poprawny w sensie deklaratywnym ale błędny w sensie proceduralnym. Logika matematyczna nie zakłada sekwencyjności rozwiązywania, zakłada równoczesność co jest nierealne ze względu architektury komputera.

Istnieją ogólne metody eliminacji nieskończonych pętli (bezproduktywnych dróg poszukiwania rozwiązania), które można zastosować w programie prologowym.

W definicjach klauzuli prologowej należy najpierw korzystać z klauzul, opisujących proste relacje miedzy obiektami, zanim użyjemy relacji bardziej złożonych (rekurencyjnych).

Nowoczesne języki zmierzają do czystej interpretacji deklaratywnej, prolog jest jednym z kroków w tym kierunku. Uwalniając tym samym programistę od pewnych zmudnych czynności. Zadaniem programisty jest kreowanie pewnej przestrzeni rozwiązującej problemy i zadania.

Podsumowanie 2 Wykładu

Proste obiekty w Prologu to atomy, stałe, zmienne. OBiekty proste i złożone (strukturalne) to termy właściwe,

Termy składają się z funktora (operatora/symbolu funkcyjnego), określonego przez nazwę i arność

Typ obiektu jest rozpoznawany jedynie w oparciu o jego budowę składniową, a nie deklarację typu

Zakresem leksykalnym zmiennej jest jedna klauzula, zatem zmienne o tej samej nazwie w różnych klauzulach są różne

Termy można reprezentować w postaci drzew, zaś ich przetwarzanie traktować jako przetwarzanie struktur drzewiastych

Proces uzgadniania polega na sprawdzeniu czy dwa termy są identyczne i dla jakich podstawień zmiennych są takie same

Jeżeli proces uzgadniania zmiennych zakończy się sukcesem, to w wyniku otrzymujemy najogólniejsze możliwe podstawienie zmiennych (unifikator)

Semantyka deklaratywna w Prologu określa czy dla danego programu cel jest spełniony i jeśli tak to dla jakich podstawień zmiennych

Przecinek miedzy predykatami oznacza koniunkcje celów, a średnik dysjunkcje celów która ma niższy priorytet wykonania.

Semantyka proceduralna w Prologu to procedura spełniania listy celów w kontekście danego programu. Procedura ta stwierdza fałszywość lub prawdziwość listy celów i podaje ewentualne podstawienie zmiennych. Procedura korzysta z mechanizmu nawrotów i analizuje alternatywne rozwiązania

„Czysta” semantyka deklaratywna nie zależy od kolejności klauzul i kolejności celów w klauzulach

Semantyka proceduralna jest ściśle określona przez kolejność klauzul i celów, Kotra może mieć decydujący wpływ na efektywność i skuteczność programu

Mając dany poprawny w sensie deklaratywnym program możemy zwiekszyć jego efektywność poprzez zmianę kolejności klauzul i celów, nie naruszając przy tym jego poprawności deklaratywnej. Jest to dobra metoda wykrywania i eliminacji nieskończonych pętli w programie

Istnieją inne ogólne techniki (np. Z dziedziny Sztucznej Inteligencji) eliminacji pętli nieskończonych

Podstawowe pojęcia: atomy, liczby, zmienne, term prosty, term złożony, funktor, arność, uzgadnianie termów, najogólniejszy unifikator, semantyka deklaratywna, semantyka proceduralna, Instancja klauzuli

Wykład 3

Listy

Lista to dowolnej długości ciąg obiektów zapisywany w postaci:

[element1, element2, …]

Element może być **DOWOLNYM** termem.

Reprezentacja wewnętrzna listy odpowiada strukturze drzewiastej (zdegenerowane drzewa binarne)

Funktorem tworzącym jest „. . ”, jest to struktura rekurencyjna, pusta lista jest to para nawiasów kwadratowych [] (term symboliczny)

Budowa Listy (niepustej):

Głowa - pojedynczy obiekt z przodu listy (pierwszy)

Ogon - pozostałe elementy listy

Właśności:

1. Głowa listy może być dowolny obiekt języka prolog np inna lista, term, zmienna
2. Ogon listy jest zawsze lista i może być lista pusta
3. Lista jest struktura rekurencyjna jeżeli ogon jest niepusty, to również on składa się z głowy i ogona.

Przykładowa lista:

```
Hobby1 = [muzyka, kuchnia],
Hobby2 = [narty, taniec],
Lista = [tenis, Hobby1, film , Hobby2].

?- Lista = [ tenis, [muzyka, kuchnia], film, [narty, taniec]].
```

Elementem składowym listy może być inna lista.
Nie ma jednorodności, jest to lista termów która może zawierać dowolny term.

Można używać notacji kropkowej choć jest niewygodna:

```
?- Pets = .(dogs,.(cats,[])).
?- Pets = [dogs, cats]
```

Funktor może być wykorzystywany jawnie do tworzenia list.

Rekurencyjna notacja list:

[Head | Tail]

Head - może być ciągiem dowolnych elementów (termów) oddzielonych przecinkiem, a Tail - dowolna lista elementów

Przykład zapisów równoważnych:

?- [a, b, c] = [a | [b, c]]. — YES

?- [a, b, c] = [a, b | [c]]. — YES

?- [a, b, c] = [a, b, c | []]. — YES

?- [a, b, c] = [a, [b, c]]. — NO

Wybrane operacje na listach:

Sprawdzanie przynależności do listy:

Klauzula member/2 (member(X, L)) ma być prawdziwa jeżeli element X należy do listy L.

Implementacja:

```
member(X, [X | Tail]).
member(X, [Head | Tail]) :- member(X, Tail).
```

Operacja łączenia(konkatenacji) list:

Klauzula append(L1, L2, L,3) łączy listę L1 z lista L2 w listę wynikowa L3. (L1 + L2 = L3).

Przykłady:

```
?- append( [a, b], [c, d], [a, b, c, d] ) — jest prawdziwe
?- append( [a, b], [c, d], [a, b, a, c, d] ) — jest fałszywe
```

Nie ma znaczenia czy Lista L1 lub L2 jest pusta

Nie ma zabiegu w prologu który pozwala dostać się na koniec rekurencyjne listy.

Implementacja:

```
append( [] , L, L).
append( [H| T1], L2, [H|T2]) :- append(T1, L2, T2).
```

Dekompozycja listy używając append:

```
?- append(L1, L2, [a, b, c]).
L1 = []
L2 = [a, b, c];

L1 = [a]
L2 = [b, c];

L1 = [ a, b]
L2 = [c];

L1 = [a, b, c]
L2 = [];
```

Przeszukiwanie podlist używając append:

```
?- append( Przed, [sr | Po], [pon, wt, sr, czw, pt, sob, nd]).

Przed = [pon, wt]
Po = [czw, pt, sob, nd]
```

Usuwanie podlisty używając append:

```
?- L1 = [a, b, z, z, c, z, z, z, d, e], append(L2, [z, z, z | _ ], L1).

L1 = [a, b, z, z, c, z, z, z, d, e]
L2 = [a, b, z, z, c]
```

Alternatywna wersja member używając append:

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

Albo ze zmienna anonimową:

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

Dodawanie elementu na początek listy:

Najprostszy wariant - dodać element X na początku listy L (będzie głową nowej listy).

Klauzula w prologu:

```
add(X, L , [X|L]).
```

Identyczny efekt uzyskamy kiedy jawnie użyjemy operatora uzgadniania:

```
NewL = [X | L]
```

Usuwanie elementu z listy:

Klauzula del(X, L, L1) ma być prawdziwa, jeżeli lista L1 jest równa liście L pomniejszonej o element X. Położenie elementu X jest dowolne.

```
del( X, [X | Tail], Tail).
del(X, [H | Tail], [H | Tail1]) :- del(X, Tail, Tail1).
```

Operacja ta usuwa dowolne, ale tylko jedno wystąpienia X z listy L. Działanie takie (wielokrotne wykonanie z innym wynikiem) określamy mianem nie determinizmu predykatu.

Przykład:

```
del( a, [a, b, a, a], L).

L = [b, a , a];
L = [a, b, a];
L = [a, b, a];
NO
```

Podobne niedeterministyczne będzie np member(X, [a, b]).

Multimodalność — parametry predykatów procedur mogą być zarówno wejściowe i wyjściowe.

Dodawanie elementu do listy za pomocą usuwania z listy:

```
del(a, L, [1, 2, 3)
L = [a, 1, 2, 3];
L = [1, a, 2, 3];
L = [1, 2, a, 3];
L = [1, 2, 3, a];
NO
```

Implementacja

```
insert(X, L, BiggerL) :- del(X, BiggerL, L).
```

Operacje na podlistach:

Klauzula sublist(S, L) jest prawdziwa, jeśli lista S zawiera się w liście L.

Przykłady:

```
sublist([c, d, e], [a, b, c, d, e, f]) — jest prawdziwe
sublist([c, e], [a, b, c, d, e, f]) — jest fałszywe
```

Implementacja:

```
sublist(S, L) :-
        append(L1, L2, L), append(S, L3, L2).
```

Operacje generowania permutacji listy

Klauzula permut(L1, L2) jest prawdziwa jeśli lista L2 jest permutacją listy L1 (zawiera te same elementy ale w innej kolejności).

Przykład:

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

P = [a, b, c];
P = [b, c, a];
…
```

Alternatywna wersja używająca predykatu del:

Inna wersja klauzuli permut(L1, L2) — z zastosowaniem klauzuli del najpierw usuwamy element, na pozostałej reszcie dokonujemy permutacji i wstawiamy element na początek poddanej juz permutacji reszcie listy.

Implementacja:

```
permut([], []).
permut(L, [X | P]) :- del(X, L, L1),
                              permut( L1, P).
```

Ta operacja nie jest multimodalna, w pewnych przypadkach może prowadzić do nieskończonych pętli (dotyczy obu wersji)

Operatory Arytmetyczne

Wyrażenia arytmetyczne w prologu są termami, są drzewiastymi strukturami tak jak wszystkie inne funktory które poznaliśmy w prologu. Kanoniczna reprezentacja będzie wiec notacja prefiksowa

Przykład:

+(*(2, a),*(b, c))

Wyrażenia tworzą w reprezentacji wewnętrznej struktury drzewiaste (są termami), a operatory pełnią rolę funktorów (są atomami):

Dopuszczalna jest notacja infiksowa

Przykład:

2*a + b*c

Wyrażenia takie automatycznie przekształcane są do notacji prefiksowej(i na odwrót)

Pierwszeństwo operatorów decyduje o interpretacji wyrażeń w notacji infiksowej

W prologu wszystkie operatory arytmetyczne (czy te predefiniowane czy zbudowane przez programistę) są oparte o dyrektywę systemowa op

Dyrektywa ta ma postać:

op(<pierwszeństwo>, <składnia>, <symbol>)

gdzie:

<pierwszeństwo> — klasa pierwszeństwa operatora,
<składnia> — budowa operatora(prefix, infix, suffix),
<symbol> — oznaczenie operatora

Przykład:

op(600, xfx, has).

Można teraz zdefiniować fakt np.: Piotr has auto.

Definiowanie operatorów w Prologu

Zasady:
1. Definicja operatora **nie określa** żadnej operacji, która będzie wykonywana na argumentach operatora
2. Operatory definiowane w Prologu są tylko funktorami, służącymi do konstruowania bardziej złożonych struktur
3. Klasa pierwszeństwa operatora może przyjmować wartości z zakresu od 1 do 1200
4. Oznaczenie operatora musi być nazwą (stałą symboliczną)

Budowa składniowa operatora — notacje:

1. prefiksowa:
   fx fy
2. infiksowa:
   xfx xfy yfx
3. postifksowa:
   xf yf

Gdzie:

f — to operator
x — to argument o klasie pierwszeństwa < f
y — to argument o klasie pierwszeństwa <= f

Zasady pierwszeństwa argumentów i operatora:

Składnik wyrażenia	Klasa pierwszenstwa
Argument prosty	0
Argument w nawiasach	0
Operator	Wg definicji (dyrektywa op)
Argument zlozony	Wg pierwszenstwa operatora(funktora glownego)

Przykład:

op(500, yfx, -).

Reprezentacja wyrażenia a-b-c to (a-b)-c, a nie a-(b-c).

UWAGI

1. Definicja nowych operatorów nie określają żadnych nowych działań na argumentach, lecz wzbogacają notacje składniową o nowe sposoby tworzenia złożonych form reprezentacji
2. Operator o najwyższej klasie pierwszeństwa jest głównym operatorem wyrażenia złożonego; operatory o niższej klasie pierwszeństwa maja silniejsze wiązanie
3. Specyfika operatora zależy zarówno od jego położenia względem argumentów, jak i od zasad pierwszeństwa operatora oraz jego argumentów

Ewaluacja wyrażeń:

Operator przypisania:

= (znak równości) operacja uzgadniania (unifikacji) termów;
is operacja przypisania (ewaluacja wartości wyrażenia i unifikacji zmienne)

is składa się z dwóch kroków:
1. Najpierw dochodzi do ewaluacji wyrażenia
2. Wywołanie argumentu unifikacji dla zmiennej

Operatory arytmetyczne:

Oznaczenie Operatora	Operacja arytmetyczna
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie rzeczywiste
// lub div	Czesc calkowita z dzielenia
mod	Reszta calkowita z dzielenia
rem	czesc ulamkowa z dzielenia
** lub ^	Potegowanie
Operatory Porównania	**Operacja Porównania**
>	Wiekszy
<	Mniejszy
>=	Wiekszy lub rowny
=<	Mniejszy lub rowny
=:=	Równy
=\=	Różny
\=	Negacja uzgodnienia termow

Zastosowanie operacji arytmetycznych:

Operator =:= powoduje obliczenie wartości argumentów (ewaluacja) bez wiązania zmiennych (muszą być one juz związane)

Przykład:

NWD dwóch liczb: dla dwóch liczb całkowitych, dodatnich X i Y, największy wspólny dzielnikD:
1. równa się X, jeżeli X i Y są równe,
2. Równa się największemu wspólnemu dzielnikowi X i Y-X, jeżeli X<Y,
3. Równa się największemu wspólnemu dzielnikowi Y i X-Y, jeżeli Y<X.

Implementacja błędna:

```
nwd(X, Y, D) :- X<Y,  nwd(X, Y-X, D).
nwd(X, Y, D) :- X>Y,  nwd(X-Y, Y, D).
```

Zawsze operacje należy wykonywać przed, nigdy w regule, ponieważ nie dojdzie wtedy do ewaluacji wyrażenia.

Implementacja prawidłowa:

```
nwd(X, X, X).
nwd(X, Y, D) :- X<Y, Y1 is Y-X, nwd(X, Y1, D).
nwd(X, Y, D) :- X>Y, X1 is X-Y, nwd(X1, Y, D).
```

Podsumowanie 3 Wykładu:

Operacje z użyciem operatorów arytmetycznych wymagają zastosowania predefiniowanych procedur ewaluacji wyrażenia

Wykonanie operacji arytmetycznej jest możliwe po zastosowaniu procedury is

Operatory porównania również prowadzą do ewaluacji wartości porównanych wyrażeń

W trakcie ewaluacji wyrażenia wszystkie argumenty musza mieć przypisaną (związaną) wartość liczbową

Wykład 4:

Mechanizm odcięć:

Mechanizm ten:

1. Zapewnia weryfikacje alternatywnych sposobów osiągnięcia celu,
2. Analizuje systematycznie wszystkie kolejne alternatywny
3. może okazać się nieefektywny, jeśli nie zostanie zastosowana odpowiednia strategia nawracania
4. Jest automatyczny i gwarantuje systematyczne przeszukanie przestrzeni rozwiązań
5. Dopiero w momencie przeszukania całej przestrzeni rozwiązań interpreter odda wynik negatywny (w przypadku gdy nie znalazł rozwiązania)

Zakłada się że programista potrafi wywnioskować więcej niż jest to w stanie zrobić interpreter z kodu

W momencie gdy pewna reguła wyklucza inną regułę (reguły się uzupełniają) to jest pewne że któraś z nich zawsze zawsze zakończy się sukcesem.

Jeżeli warunki którejkolwiek z reguł są spełnione, nie ma potrzeby sprawdzać pozostałych alternatyw

Rozwiązanie w którym programista nie zatrzyma nawrotów interpretera jest rozwiązaniem nieefektywnym ponieważ spełnienie jednej z reguł wyklucza możliwość spełnienia drugiej.

Mechanizm nawrotów (symbol operatora odcięcia [**!**]), wskazuje miejsca w których nie potrzeba dalszego sprawdzania warunków reguł, nie potrzeba alternatywnych testów sprawdzania dla innych danych. Znak ten utrwala w pewnym sensie sposoby spełnienia poprzedzających odcięcie warunków przez zamrożenie poprzednich warunków. Cofniecie się przed znak odcięcia i zmiana warunków nie jest możliwa.

Można blokować w ten sposób sprawdzanie innych wariantów tej samej reguły, zamrażając nagłówek tej reguły. Inne warianty przestaną być brane pod uwagę w procesie wnioskowania.

Przykład:

```
f(X, 0) :- X<3, !.
f(X, 2) :- 3=<X, X<6, !.
f(X, 4) :- 6=<X.
```

Uzyskany rezultat jest wydajniejszy niż wariant:

```
f(X, 0) :- X<3.
f(X, 2) :- 3=<X, X<6.
f(X, 4) :- 6=<X.
```

Zastosowanie odcięć spowodowało zmiany tylko w interpretacji proceduralnej programu

Przykład 2:

```
f(X, 0) :- X<3, !.
f(X, 2) :- 3=<X, X<6, !.
f(X, 4) :- 6=<X.
```

Na zapytanie:

```
?- f(7,Y).
Odp. Y= 4
```

Jeżeli warunek pierwszej reguły nie jest spełniony (7<3), to pierwszy warunek drugiej reguły (3=<7) po nawrocie (!) jest na pewno prawdziwy, gdyż stanowi jego logiczne dopełnienie; warunek ten jest więc nadmiarowy, bo jeśli w ogóle jest sprawdzany, to tylko po nawrocie, do którego doszło tylko dlatego, że nie był spełniony warunek 7<3 (inaczej zadziałałby mechanizm odcinania). Podobna sytuacja ma miejsce w przypadku warunku trzeciej reguły (6=<X): nie spełnienie drugiego warunku drugiej reguły (X<6) oznacza nawrót i automatycznie prawdziwość warunku trzeciej reguły, więc sprawdzenie tego warunku (6=<X) jest nadmiarowe, bo dochodzi do niego tylko po nie wykonaniu odcięcia w drugiej regule.

Pozwala to uprościć funkcje do postaci:

```
f(X, 0) :- X<3, !.
f(X, 2) :- X<6, !.
f(X, 4).
```

Zmienia to jednak interpretację deklaratywną takiej funkcji. Jest to prologowy odpowiednik funkcji warunkowej dopełniających się przypadków znanej z obiektowych języków programowania (if else)

```
Jeżeli X<3, to Y=0,
W przeciwnym przypadku jeżeli X<6, to Y=2,
W przeciwnym przypadku Y=4
```

Brak odcięć w tej funkcji może prowadzić to nie determinizmu odpowiedzi co jest niebezpieczne. Istnienie odcięć zmienia interpretację proceduralną programu i może zmienić także interpretację deklaratywną. Może to odprowadzić w skrajnym przypadku do błędów utrudniających debugowanie programu.

Formalna definicja odcięć:

Odcięcie jest to systemowy cel czy warunek, który jest natychmiast spełniony, gdy tylko zostanie osiągnięty znak odcięcia. Wszystkie cele, które zostały do tej chwili spełnione (w klauzuli zawierającej odcięcie) nie będą analizowane powtórnie w celu weryfikacji alternatywnych definicji,

odpowiadających im klauzul (czyli alternatywnych sposobów ich spełnienia). Dotyczy to również nagłówka klauzuli.
Odcięcie nie musi być ostatnim warunkiem w regule.

Formalna interpretacja proceduralna odcięcia:

Załóżmy że dana jest klauzula H w postaci:

H :- B1, B2, … , Bm, !, … , Bn.

Oraz dany cel G, który został dopasowany do klauzuli. W momencie w którym zostanie osiągnięty znak odcięcia, wszystkie podcele:

B1, B2, … , Bm

Są już spełnione.
Po przekroczeniu znaku odcięcia rozwiązania tych pod celów zostają „zamrożone", zaś alternatywne sposoby ich spełnienia nie będą analizowane. Również sam cel G jest już możliwy do spełnienia tylko przez klauzule H i inne reguły zdefiniowane później niż H, nagłówek których pasuje do G, nie będą używane.

Przykład:

```
c :- p, q, r
c :- v.
a :- b, c, d.
```

Zapytanie:

```
?- a.
```

Odcięcie w pierwszej regule c uniemożliwia nawrót do innych dopasowań dla pod celów p, q, r, jak również do drugiej reguły c, jeżeli pierwsza byłaby spełniona. Nawroty są jednak nadal możliwe w ramach pod celów s, t, u. Podobnie odcięcie wpłynie tylko na sposób osiągania celu c, natomiast dla celu głównego a pozostaje ono „niewidoczne", więc nawrót w ramach listy warunków b, c, d jest nadal możliwy.

Przykłady wykorzystywania odcięć:

Obliczanie maksimum:

Definicja 1:

```
Max = X, o ile
X jest większe lub równe Y,
Lub
Max = Y, o ile
X jest mniejsze od Y
```

Implementacja:

```
max(X,Y,X) :- X>=Y.
max(X,Y,Y) :- X< Y.
```

Definicja 2:

Max = X, o ile
X jest większe lub równe Y
W przeciwnym przypadku
Max = Y.

Implementacja:

```
max(X,Y,X) :- X>=Y, !.
max(X,Y,Y).
```

Szukanie pierwszego wystąpienia na liście:

Definicja używając odcięć pozwala na znalezienie dowolnego elementu i potwierdzenie jego istnienia na liście. Następuje tutaj niejawne uzgodnienie głowy listy z szukanym elementem.

Implementacja:

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

Wstawianie elementu do listy bez powtórzeń:

Definicja:

Jeżeli X należy do L, to L1 = L
W przeciwnym przypadku L1
Jest równe L powiększonemu o X.

Implementacja:

```
add(X, L, L) :- member(X, L), !.
add(X, L, [X | L]).
```

Grupowanie w Kategorie:

Treść zadania:

Dysponujemy bazą danych o rozgrywkach tenisowych w pewnym klubie sportowym, reprezentowanych w postaci faktów:

beat(<zwyciezca>, <pokonany>).

Naszym celem jest podział graczy na trzy kategorie:

winner — gracze którzy wygrali wszystkie swoje mecze,
fighter — gracze, którzy część meczy wygrali a część meczy przegrali,
sportsman — gracze, którzy przegrali wszystkie swoje mecze.

Wynik ma być reprezentowany za pomocą relacji:

class(<gracz>, <kategoria>).

Definicje:

X należy do kategorii **fighter,** o ile
Istnieje taki Y, którego X pokonał
i istnieje taki Z, który pokonał X.

X należy do kategorii **winner**, o ile
Istnieje taki Y, którego X pokonał
i nie istnieje taki Z, który pokonał X.

X należy do kategorii **sportsman**, o ile
Nie istnieje taki Y, którego X pokonał i
Istnieje taki Z, który pokonał X.

Definicja bez negacji:

Jeżeli X pokonał kogoś i został przez kogoś pokonany,
To należy do kategorii **fighter**,
W przeciwnym przypadku jeżeli X pokonał kogoś,
To należy do kategorii **winner**,
W przeciwnym przypadku jeżeli X został przez kogoś pokonany,
To należy do kategorii **sportsman**.

Implementacja:

```
class(X, fighter) :- beat(X, _ ), beat( _ , X), !.
class(X, winner) :- beat(X, _ ), !.
class(X, sportsman) :- beat( _ , X).
```

Negacja przez niepowodzenie:

false i true

W prologu istnieje systemowy sterujący predykat fail, który nigdy nie jest spełniony i prowadzi do porażki celu nadrzędnego. Stosowany jest do wymuszenia następnych (alternatywnych) sposobów spełnienia celu. Jego niepowodzenie zawsze zmusi do wcześniejszych warunków i znalezienia alternatywnych sposobów spełnienia (nawrót).

Posiada on swoje przeciwieństwo, predykat systemowy true, który zawsze jest spełniony.

Przykład:

Mamy bazę danych o osobach zapisana w postaci relacji person. Należy znaleźć wszystkie osoby z naszej bazy.

Implementacja:

```
find :- person(X), write(X), fail.
find.
```

Przykład 2:

Predykat different(X, Y) jest spełniony, jeżeli X i Y są różne.

Różnicą nazwiemy:

1. X i Y to różne napisy
2. X i Y to niedopasowane termy
3. X i Y to wartości wyrażeń X i Y są różne

Dla interpretacji drugiej:

```
different(X, X) :- !, fail.
different(X, Y).
```

Albo używając true:

```
different(X, Y) :- X=Y, !, fail; true.
```

not(X)

W prologu istnieje systemowy predykat unarny not(X), który jest spełniony, jeżeli X nie jest spełnione (nie daje się wywieść). Negacja przez niepowodzenie nie jest dokładnym odzwierciedleniem negacji logicznej (matematycznej), co nie pozostaje bez wpływu na działanie programów prologowych. U podstaw teoretycznej negacji przez niepowodzeniem leży założenie o „zamkniętości świata", które mówi, że jeżeli prawdziwość faktu nie można wykazać za pomocą dostępnych danych, to negacja faktu jest prawdziwa.

Przykład zastąpienia odcięć negacją systemową:

```
class(X, fighter) :- beat(X, _ ), beat( _ , X).
class(X, winner) :- beat(X, _ ), not(beat( _ , X)).
class(X, sportsman) :- beat( _ , X), beat(X, _ ).
```

Problemy związane z zastosowaniem odcięć:

Zalety mechanizmu odcięć:

1. Zwiększenie efektywności programu, jawne pokazanie które rozwiązania alternatywne są niepotrzebne
2. Pozwalają zapisać wzajemnie wykluczające się relacje (jeżeli…to…w przeciwnym przypadku…)

Wady:

1. wpływają na interpretację proceduralną i deklaratywną prowadząc do znacznych różnic pomiędzy tymi interpretacjami
2. Zmiana porządku klauzul i celów wpływa jedynie na efektywność programu, gdy nie ma w nim odcięć, ale zmienia stronę deklaratywną, gdy odcięcia w nim występują

Przykład wpływu odcięć na interpretacje deklaratywną:

```
p :- a, b.
p :- c.
```

Interpretacja deklaratywna:

$$p \Leftrightarrow (a \wedge b) \vee c\,.$$

Zmiana kolejności klauzul nie zmieni tej interpretacji

```
p :- a, !, b.
p :- c.
```

Interpretacja deklaratywna:

$$p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)\,.$$

Jeżeli zmienimy kolejność reguł:

```
p :- c.
p :- a, !, b.
```

To interpretacja ma postać:

$$p \Leftrightarrow c \vee (a \wedge b)\,.$$

Wyróżniamy dwa rodzaje odcięć:

1. Odcięcia czerwone — to takie, które zmieniają interpretację deklaratywną programu, utrudniają jego zrozumienie i powodują utratę pewnych rozwiązań
2. Odcięcia zielone — takie, które nie wpływają na interpretację deklaratywną, nie zmniejszają jego czytelności i zachowują wszystkie rozwiązania (choć obcinają drzewo poszukiwań)

Zastosowanie odcięć czerwonych wymaga dużej ostrożności przy programowaniu

Problemy związane z zastosowaniem negacji:

Zastosowanie predykatu not niesie ze sobą wszystkie zagrożenia, wynikające z niewłaściwego zastosowania odcięć.

Przykład:

```
?- not(human(tom).
True
```

Odpowiedz nie oznacza, że tom nie jest człowiekiem, lecz nie ma dość informacji, żeby stwierdzić, że tom jest człowiekiem. Interpreter nie próbuje dowieść negacji celu, lecz dowodzi celu prostego i jeśli ten wywód się nie powiedzie, zakłada, że negacja jest prawdziwa. Jest to przejaw założenia o zamkniętości świata.

Wiązanie zmiennych zmienia „moc” negacji.

Przykład:

```
good_standard(jeanluis).
expensive(jeanluis).
good_standard(francesco).
reasonable(R) :- not(expensive(R)).

?- good_standard(X), reasonable(X).
X = francesco
?- reasonable(X), good_standard(X).
False
```

Różne odpowiedzi są efektem wiązania zmiennej X w pierwszym zapytaniu już w pierwszym celu i brakiem takiego wiązania w drugim zapytaniu.

Podsumowanie 4 Wykłady:

Odcięcie i negacja powinny być wykorzystywane z należytą ostrożnością. Nie oznacza to jednak, iż należy z nich całkowicie zrezygnować. Są one często pomocne w zwiększaniu efektywności programu, a czasami wręcz niezbędne w znalezieniu rozwiązania w sensownym czasie. Problemy, które wynikają z ich zastosowania występują również w innych językach deklaratywnych.

Wykład 5:

Operacje wejścia/wyjścia:

W języku prolog typ jest reprezentowany przez dwa rodzaje struktur pliki termów, drugi rodzaj to pliki o charakterze znakowym znane z innych języków programowania.
Przyjęto koncepcje strumienia danych, mamy strumień danych wejściowych i wyjściowych które są skojarzone ze sobą. W prologu istnieje domyślny strumien wejściowy i wyjściowy jest nim terminal użytkownika, strumienie można jednak kojarzyć z innymi plikami. Strumieniem wejściowym i/lub wyjściowym może być dowolny plik o dostępie sekwencyjnym (tekstowy).
W trakcie wykonywania programu w danej chwili realizowana może być operacja odczytu i zapisu odpowiednio z/do jednego strumienia wejściowego i jednego strumienia wyjściowego.

Operacje na plikach sekwencyjnych:

Otwieranie pliku:

Operacje zmiany aktualnego strumienia wejściowego:

see(<nazwa_pliku>)

Jeżeli plik jest już otwarty to nadal pozostanie w trybie odczytu (nie będzie błędu).

Operacja zmiany aktualnego strumienia wyjściowego:

tell(<nazwa_pliku>)

Jeżeli plik jest już otwarty to nadal pozostanie w trybie zapisuj (nie będzie błędu)

Zamykanie pliku:

Operacja zamknięcia aktualnego strumienia wejściowego:

seen

Predykat ten jest zawsze spełniony. Po wykonaniu strumieniem wejściowym zostaje terminal.

Operacja zamknięcia aktualnego strumienia wyjściowego:

told

Predykat ten jest zawsze spełniony. Po wykonaniu strumieniem wyjściowym zostaje terminal.

Standard przetwarzania przedstawiony do teraz nazywamy standardem edynburskim.

Identyfikacja strumieni:

Operacja identyfikacji aktualnego strumienia wejściowego:

seeing(Str)

Zmienna Str jest unifikowana z identyfikatorem strumienia (wygenerowanym automatycznie przez system).

Operacja identyfikacji aktualnego strumienia wyjściowego:

telling(Str)

Zmienna Str jest unifikowana z identyfikatorem strumienia (wygenerowanym automatycznie przez system).

Operacje otwarcia i zamknięcia (see, tell, seen, told) służą wyłącznie do przetwarzania plików tekstowych (sekwencyjnych).

Podczas przetwarzania plików sekwencyjnych każda operacja (zapis/odczyt) powoduje automatyczne przejście do następnej pozycji w pliku.

Osiągnięcie końca pliku sygnalizowane jest zawsze specjalną wartością.

Ogólny schemat przetwarzania pliku:

$$Otwarcie \rightarrow odczyt/zapis \rightarrow \ldots \rightarrow odczyt/zapis \rightarrow Zamkniecie$$

Sekwencyjny charakter przetwarzania plików jest niezgodny z deklaratywna naturą programowania za pomocą reguł.

Rodzaje plików tekstowych:

1. Pliki znakowe: plik składa się z pojedynczych bajtów/znaków
2. Pliki termów: podstawowym składnikiem pliku jest term

Operacje odczytu i zapisu dla plików znakowych:

Odczyt pojedynczego i nie białego znaku kodu ASCII z aktualnego strumienia:

get(X)

Odczyt jednego, dowolnego bajtu z aktualnego strumienia i unifikacje jego kodu ASCII ze zmienną X:

get_byte(X)

Zapis pojedynczego znaku kodu ASCII do aktualnego strumienia:

put(X)

Osiągnięcie końca pliku sygnalizowane jest wartością -1

Operacje odczytu i zapisu dla plików termów:

Odczyt pojedynczego termu z aktualnego strumienia wejściowego:

read(X)

Zapis pojedynczego termu do aktualnego strumienia wyjściowego:

write(X)

Osiągnięcie końca pliku sygnalizowane jest predefiniowanym atomem end_of_file

Pliki termów musza spełniać wymogi składni termów języka Prolog

Wczytanie informacji reprezentowanej w postaci listy kodów znaków w stałą symboliczną lub liczbę:

name(A, S)

A - stała, symboliczna lub liczba
S - lista kodów znaków

Konwersji można dokonać w obie strony.

Alternatywną reprezentacją list kodów w Prologu jest napis ograniczony cudzysłowiem np. „pies” = [112, 105, 101, 115].

Przetwarzanie plików termów:

Operacja zapisu termu do strumienia wyjściowego

Predykat read(X) oznacza odczyt pojedynczego termu z aktualnego strumienia wejściowego i unifikację termu ze zmienną X.

Brak uzgodnienia dla argumentu X predykatu read, nie będącego zmienną wolną doprowadzi do błędu. Nie nastąpi nawrót w celu odczytania ponownie termu.

Termy zawarte w pliku wejściowym muszą być zakończone znakiem kropki

Pełniejsza wersja operacji pozwala również wskazywać jakiego strumienia (Str) dotyczy operacja:

read(Str, X)

Operacja zapisu termu do strumienia wyjściowego:

Predykat write(X) oznacza zapis pojedynczego termu X do aktualnego strumienia wyjściowego. Należy pamiętać że term zostanie zapisany BEZ kropki, należy ją dopisać samemu.

Term X zostanie zapisany do pliku skojarzonego ze strumieniem wyjściowym w formie identycznej z wykorzystywaną dla domyślnego strumienia wyjściowego (terminala).

Termy zapisywane do pliku mogą mieć dowolny stopień złożoności (zagnieżdżenia)
Pełniejsza wersja operacji zapisu pozwala również wskazywać jakiego strumienia (Str) dotyczy operacja:

write(Str, X).

Pozostałe operacje dla strumienia wyjściowego:

Predykat append(X) otwiera plik X w trybie dopisywania i kojarzy go z aktualnym strumieniem wyjściowym.

Predykat tab(N) oznacza zapis N znaków odstępu do aktualnego strumienia wyjściowego, przy czym N musi być większe od 0.

Predykat nl (bezargumentowy) powoduje zapis znaku nowego wiersza (przejście do nowej lini) do pliku skojarzonego z aktualnym strumieniem wyjściowym.

Przykład przetwarzania plików tekstowych — odczyt z terminala:

Interaktywne obliczanie sześcianu liczb dla ciągu wartości wczytanych z terminala od użytkownika:

```
cube :- read(X), process(X).

process(stop) :- !.
process(X) :- N is X*X*X, write(N), cube.
```

Wyświetlanie elementów listy:

```
writelist([]).
writelist([ H | T]) :- write(H), nl, writelist(T).
```

Wczytywanie danych z terminala na listę:

```
readlist(L) :- read(X), process(X, L).

process(stop, []) :- !.
process(X, [X | T]) :- readlist(T).
```

Zagnieżdżenie rekurencyjnych struktur wymaga zdefiniowania oddzielnego predykatu przetwarzania na każdym poziomie złożonej (rekurencyjnej) struktury.

Przetwarzanie pliku dyskowego — wyświetlanie zawartości pliku z numerowaniem termów:

```
procfile :- showfile(1).
showfile(N) :- read(Term), show(Term, N).
show(end_of_file, _) :- !.
show(Term, N) :- write(N), tab(2), write(Term), nl, N1 is N+1.
```

Przetwarzanie termów z niestandardowego strumienia wejściowego i wyjściowego:

Katalog towarów zawarty jest w pliku składającym się z termów w postaci:

```
item(numer, opis, cena, dostawca)
```

Chcemy wygenerować nowy plik zawierający tylko towary od jednego wyznaczonego dostawcy Sup. Przetwarzanie wymaga odczytu z pliku np. dane i zapisu do drugiego pliku np. wyniki

Zapytanie celu głównego:

```
see(‚dane'), tell(‚wyniki'), Sup=lloyd,
makefile(Sup), told, seen.

makefile(Sup) :- write(Sup), write(, . '),  n1, makerest(Sup).
makerest(Sup) :- read(Item), proc(Item, Sup).
proc(end_of_file, _ ) :- !, write(item(N, D, P)), write(, . ,), nl,
makerest(Sup).
proc( _ , Sup) :- makerest(Sup).
```

Operacje odczytu i zapisu znaku z/do strumienia wyjściowego:

Usuwanie nadmiarowych znaków odstępu z wczytanego napisu wejściowego — predykat squeeze. Napis musi być zakończony kropką:

```
squeeze :- get_byte(C), put(C), do_rest(C).
do_rest(46) :- !.
do_rest(32) :- !, get(C), put(C), do_rest(C).
do_rest(C) :- squeeze.
```

Poprawiona definicja squeeze. Nie musimy znać kodów ASCII znaków szczególnych:

```
squeeze :- get_byte(C), put(C), do_rest(C).
do_rest(X) :- [X]=„.", !.
do_rest(X) :- [X]=„ ", !,
get(C), put(C), do_rest(C).
do_rest(C) :- squeeze.
```

Kompozycja i dekompozycja atomu:

Przekształcenie napisów języka naturalnego w następujące reprezentację wewnętrzną:

1. Każdy pojedynczy napis jest atomem
2. Całe zdanie jest listą atomów

Przykładowo:

> Input: Tomek był zadowolony z postępów robota.
> Output: [Tomek, był, zadowolony, z, postępów, robota]

Implementacja:

```
getsntc(Wlist) :-
get_byte(C), getrest(C, Wlist).

getrest(46, []) :- !.
getrest(32, Wlist) :- !, getsntc(Wlist).
getrest(L, [W | Wlist]) :- getlttrs(L, Ls, Next), name(W, Ls),
getrest(Next, Wlist).

getlttrs(46, [], 46) :- !.
getlttrs(32, [], 32) :- !.
getlttrs(L, [L | LS], Next) :- get_byte(C), getlttrs(C, Ls, Next).
```

Wczytywanie programów prologowych:

Predykat consult(F) ładuje wszystkie klauzule z pliku F, które następnie są wykorzystywane do osiągnięcia zadanego celu; ponowne wykonanie tego predykatu spowoduje dopisanie nowych klauzul do już istniejących.

Predykat reconsult(F) działa podobnie; jedynie w przypadku wystąpienia w pliku F klauzul relacji zawartych już w pamięci nastąpi ich redefinicja; pozostałe klauzule pozostaną nie zmienione

Wykład 6:

Metapredykaty — predefiniowane procedury systemowe, które traktują program prologowy jak dane, czyli obiekt którym możemy manipulować w trakcie przetwarzania programu. Mają różne role i mogą być różnych zadań

Sprawdzanie typu termów:

Prolog umożliwia manipulowanie termami (stałymi, zmiennymi, liczbami, atomami) za pomocą specjalnych procedur systemowych:

1. Predykat var(X) jest spełniony, jeżeli X jest zmienną wolną (nie związaną), X może jako argument być dowolnym termem.
2. Predykat nonvar(X) jest spełniony, jeżeli X jest termem innym niż zmienna lub zmienną związaną
3. Predykat atom(X) jest spełniony, jeżeli X jest stałą lub zmienną (związaną) atomową

4. Predykat integer(X) jest prawdziwy, jeżeli X jest stałą lub zmienną (związaną) całkowitoliczbową
5. Predykat float(X) jest prawdziwy, jeżeli X jest stałą lub zmienną (związaną) zmiennoprzecinkową (rzeczywistą)
6. Predykat atomic(X) jest prawdziwy, jeżeli X jest stałą lub zmienną (związaną) liczbową lub atomową

Przykłady:

```
?- var(Z), Z=2.
Z = 2
?- Z = 2, var(Z).
False
?- integer(X), X = 2.
False
?- Y=2, integer(Y), nonvar(Y).
Y = 2
?- atom(22).
False
?- atomic(22).
True
?- atom([]).
True
?- atomic(p(1)).
False
```

Predykaty dekompozycji:

W języku prolog mamy do dyspozycji predykaty systemowe przeznaczone do konstruowania i dekomponowania termów.

Predykat =.. (ang. univ) służy do konstruowania termu z listy atomów. Cel Term =..L jest spełniony, jeżeli lista L zawiera nazwę funktora termu Term i wszystkie jego kolejne argumenty. Jest on multimodalny I zapisywany w postaci infiksowej

Przykład:

```
?- f(a, b) =..L.
L = [f, a, b]
?- T=..[rectangle, 3, 5].
T = rectangle(3, 5).
?- Z =.. [p, X, f(X, Y)].
Z = p(X, f(X, Y)).
?- atomek =.. L.
L = [atomek]
```

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

Przykład:

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

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

Przykład:

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

Predykat functor może być także wykorzystany do tworzenia struktur danych. W poniższym przykładzie predykat functor generuje „szablon termu" rozpoczynający się funktorem date o arności 3, którego składowe początkowo są nieokreślone (niezwiązane). Zostają one ustalone dopiero na drodze spełnienia następnych celów wskutek wywołania predykatu arg.

Przykład:

```
?- functor(D, date, 3),
        arg(1, D, 29),
        arg(2, D, june),
        arg(3, D, 1982).
D = date(29, june, 1982)
```

Różne rodzaje operacji równości:

1. X = Y jest spełniony, gdy termy X i Y unifikują się
2. X is E jest spełniony, gdy X unifikuje się z wartością wyrażenia E (ewaluacja E potem proces uzgadniania z X)
3. E1=:= E2 jest spełniony, gdy wartości wyrażeń arytmetycznych E1 i E2 są równe
4. E1 =/= E2 jest spełniony, gdy wartości wyrażeń arytmetycznych E1 i E2 są różne
5. T1 == T2 jest spełniony gdy termy T1 i T2 są identyczne (unifikują się leksykalnie włącznie z nazwami zmiennych)

Operator == zachowuje się różnie w zależności czy termy są związane czy nie, jeżeli występuje zmienna wolna to operator nie wykonuje unifikacji więc wyrażenie otrzyma wartość fałszywą! To samo dotyczy dwóch obiektów wolnych

Manipulacja bazą danych:

Pewne klauzule traktują program prologowy jak bazę danych. Pozwalają dodawać i usuwać klauzule w trakcie dziania programu.

Program prologowy traktowany jako baza danych to:

1. Klauzula bezwarunkowa — fakty reprezentujące jawne relacje
2. Klauzule warunkowe — fakty reprezentujące niejawne relacje

Predykaty systemowe umożliwiające manipulowanie bazą klauzul:

assert(C) — zawsze spełniony cel, dodający klauzule C

asserta(C) —zawsze spełniony cel, dodający klauzule C na początku bazy

assertz(C) — zawsze spełniony cel, dodający klauzulę C na końcu bazy

retract(C) — zawsze spełniony cel, usuwający klauzule C

Pozwalają osiągnąć pamięć globalną w programie, tworzyć fakty w trakcie działania programu wypełnione danymi do dalszego użytku.

Przykład manipulacji bazą danych:

Zbiór klauzul:

```
nice :- sunshine, not(raining).
funny :- sunshine, raining.
disgusting :- raining, fog.
raining.
fog.
```

Dialog:

```
?- nice.
False
?- disgusting
True
?- retract(fog).
True
?- disgusting
False
?- assert(sunshine).
True
?- funny.
True
?- retract(raining).
True
?- nice.
True
```

Dodawać można także reguły za pomocą asercji, jednak wymaga to dodatkowej pary nawiasów, aby działać poprawnie.

Przykład:

```
assert((faster(X, Y) :- fast(X), slow(Y))).
```

Operacja retract jest realizowana w sposób niedeterministyczny — można usunąć cały zbiór klauzul dzięki mechanizmowi nawrotów.

Operacje asserta i assertz pozwalają wskazywać miejsce w którym zostanie dodana nowa klauzula, co ma znaczenie w przypadku nawrotów dokonywanych dla nowej klauzuli.

Operacje assert, asserta, assertz mogą zostać wykorzystywane do przechowywania wyników wcześniejszych obliczeń lub generowania faktów na potrzeby przyszłych zadań.

Predykaty dodawania i usuwania klauzul są bardzo użytecznym narzędziem programistycznym, lecz ich zastosowanie wymaga dużej ostrożności. Są to bowiem mechanizmy, które dokonują modyfikacji programu w trakcie jego działania, czyli samo modyfikacji, i jako takie mogą zmienić jego funkcjonowanie z upływem czasu. Utrudnia to zarówno zrozumienie programu, jego ewentualne. poprawki, jak i ogranicza nasze przekonanie co do jego prawidłowego działania.

Czasami takie zabiegi wymagają specjalnej dyrektywy **dynamic**

Manipulowanie przepływem sterowania:

Predykaty systemowe przeznaczone to modyfikacji sterowania:

1. odcięcie(!) — cel eliminujący nawroty
2. fail — cel, który zawsze jest niespełniony
3. true — cel, który zawsze jest spełniony
4. not(P) — negacja (przez niepowodzenie) celu P, ukrywa w sobie odcięcie
5. call(P) — cel spełniony, gdy wywołany cel P jest spełniony (pozwala traktować term nie jako strukturę danych tylko jako predykat, zaciera różnicę czym jest dana a czym jest reguła w Prologu)
6. repeat — cel zawsze spełniony; niedeterministyczny prowadzi poprzez nawroty do poszukiwania alternatywnych rozwiązań ze względu na definicję:

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

Efekt repeat często wykorzystuje się z odcięciem

Przykład:

```
makesqr :- repeat, read(X), proc(X).
proc(stop) :- !.
proc(X) :- integer(X), !, Y is X*X, write(Y), nl, fail.
```

Mechanizm odwoływania się (agregacji) do wcześniej wygenerowanych rozwiązań:

Mechanizm nawrotów stosowany w Prologu umożliwia sprawdzenie wszystkich obiektów lub relacji, które spełniają zadany cel. Po dokonaniu nawrotu nie jest jednak możliwe odwołanie się do wcześniej wygenerowanych rozwiązań (tych przed nawrotem). Efekt taki można jednak uzyskać za pomocą predykatów:

1. bagof(X, P, L) — generuje listę L wszystkich obiektów X takich, że cel P jest spełniony; ma sens kiedy P i X mają wspólne zmienne
2. setof(X, P, L) — podobnie jak bagof tyle, że lista L zostanie uporządkowana i pozbawiona powtórzeń elementów;
3. findall(X, P, L) — podobnie jak bagof tyle, że generowane są wszystkie obiekty niezależnie od wartości tych zmiennych w P, które nie występują w X

Przykład użycia bagof:

Dane są fakty:

```
age(peter, 7).
age( ann ,5).
age(pat, 8).
age(tom, 5).
```

Wszystkie dzieci w wieku 5 lat:

```
?- bagof(Ch, age(Ch,5), L).
L = [ann, tom]
```

Wszystkie dzieci w dowolnym wieku:

```
?- bagof(Ch ,age(Ch, Age), L).
Age = 7
L = [peter];
Age = 5
L = [ann, tom];
Age = 8
L = [pat];
False
```

Agregacja wyników dotyczy tylko imion dzieci a nie ich wieku

Wynik predykatu bagof może zawierać powtórzenia, jeżeli znaleziony obiekt wielokrotnie spełniał podany cel.

Predykat setof eliminuje powtórzenia i porządkuje obiekty alfabetycznie kiedy są atomami, a rosnąco gdy są liczbami. Jeśli obiekty są termami to porządkowanie alfabetyczne odnosi się do funktorów tych termów, a gdy funktory są takie same i termy złożone, to dotyczy skrajnie lewych, najmniej zagnieżdżonych w nich, różnych od siebie funktorów.

Przykład działania setof:

```
?- setof(Age/Child, age(Child, Age), L).
L = [5/ann, 5/tom, 7/peter, 8/pat]
```

Predykat findall(X, P, L) umożliwia uzyskanie na liście L obiektów X z wszystkich rozwiązań celu P niezależnie od wartości pozostałych zmiennych w P, które nie należą do X. Jeżeli nie istnieje żaden taki obiekt X, który spełniłby P, to w wyniku zwracana jest lista pusta. Różni się on działaniem ponieważ zawsze dokona wszystkich możliwych nawrotów.

Przykład użycia findall:

```
?- bagof(Ch, age(Ch, Age), L).
Age = 7
L = [peter];
Age = 5
L = [ann, tom];
Age = 8
L = [pat]
?- findall(Ch, age(Ch, Age), L).
L = [peter, ann ,pat, tom]
```

Wykład 7:

Ogólne zasady poprawnego programowania w Prologu:

Kryteria oceny programowania:

1. Poprawność — program realizuje przyjęte na początku założenia i generuje oczekiwane wyniki
2. Efektywność — program nie zużywa niepotrzebnie zasobów systemu komputerowego
3. Czytelność — program jest łatwy w interpretacji i zrozumieniu; nie jest bardziej skomplikowany niż to konieczne; struktura i budowa jest przejrzysta
4. Modyfikowalność — program łatwo poddaje się zmianom, rozszerzeniom i ulepszeniom

Dodatkowe kryteria:

5. Odpornośc — program powinien być przygotowany na pewne błędy i niepoprawne dane; zachowywać się racjonalnie w obliczu drobnych pomyłek

6. Dokumentacja — program powinien być właściwie opisany; minimalny wymóg to komentarz nagłówków reguł programu prologowego

Zależność poszczególnych kryteriów zależy od zadania, które realizuje program, od okoliczności w jakich program jest tworzony oraz od środowiska w jakim będzie wykorzystywany

Najważniejszym kryterium niezależnym od języka jest poprawność i żadne inne kryterium nie może być traktowane jako bardziej istotne

Punktem wyjścia do procesu pisania programu powinna być zawsze dogłębna analiza i zrozumienie problemu.

Ogólne zasady programowania w Prologu są tymi samymi zasadami które stosuje się w innych paradygmatach programowania.

Podstawowe techniki wykorzystywane podczas programowania w Prologu:

1. Rekurencja
2. Generalizacja
3. Reprezentacja graficzna rozwiązania

Rekurencja:

Problem zawsze można zredukować do przypadków należących do dwóch grup:

1. Przypadki trywialne, podstawowe lub brzegowe
2. Przypadki regularne, typowe, w których rozwiązanie jest konstruowanie na podstawie rozwiązania zredukowanego przypadku problemu pierwotnego

Technika ta jest podstawową metodą programowania w Prologu

Najważniejszą z przyczyn dlaczego stosujemy rekursję jest rekurencyjny charakter struktur danych wykorzystywanych w języku prolog.

Generalizacja:

Problem, który próbujemy rozwiązać często jest przypadkiem szczególnym innego, ogólniejszego problemu.

Znalezienie rekurencyjnego rozwiązania zadania ogólnego umożliwia rozwiązanie zadania pierwotnego, będącego jego przypadkiem szczególnym

Generalizacja wymaga z reguły wprowadzenia dodatkowych argumentów w opisie zadania

Zasadnicza trudność polega na konieczności głebokiej analizy problemu i znalezieniu odpowiedniego uogólnienia.

Reprezentacja graficzna rozwiązania problemu:

Reprezentacja graficzna problemu ułatwia zrozumienie zależności (relacji) występujących w zadaniu

Prolog jest szczególnie predestynowany do rozwiązywania problemów dotyczących obiektów i relacji między nimi

Dane strukturalne wykorzystywane w Prologu są reprezentowane w sposób naturalny za pomocą struktur drzewiastych

Interpretacja deklaratywna programu ułatwia zamianę reprezentacji graficznej w formę klauzul, gdyż kolejność opisu obrazów nie ma najczęściej znaczenia

Styl programowania:

Przyczyny zaleceń stylistycznych w programowaniu:

1. konieczność redukcji powtarzalnych błędów programistycznych
2. Poprawa czytelności programu, która decyduje o prostocie modyfikacji, poprawiania i ulepszania programu

Podstawowe zasady dobrego stylu programowania w Prolog:

1. Definicje klauzul powinny być krótkie, powinny się składać z nie więcej niż kilku pod celów (warunków)
2. Pojedynczy predykat powinien być reprezentowany za pomocą co najwyżej kilku klauzul; rozbudowane definicje są dopuszczalne, o ile mają dość jednolitą i powtarzalną strukturę
3. Nazwy predykatów, funktorów i zmiennych powinny być zrozumiałe — wyrażać przypisane im znaczenie
4. Struktura całego programu, powinna być czytelna i spójna; użycie odstępów, tabulacji oraz pustych linii powinno służyć zwiększeniu czytelności; klauzule tego samego predykatu powinny być grupowane razem; zalecane jest również umieszczanie każdego pod celu (warunku) w innym wierszu.
5. Konwencja stylistyczna może być różna dla różnych osób i/lub różnych programów, lecz powinna być spójna w ramach jednego projektu
6. Operator odcięcia powinien być wykorzystywany z należytą ostrożnością i tylko tam, gdzie to niezbędne; o ile to możliwe należy stosować odcięcia „zielone” a nie „czerwone”; te istatnie

powinny być ograniczone do dwóch przypadków: konstrukcji logicznej negacji i selekcja alternatyw (if… else)
7. Należy zawsze pamiętać o postaci definicji predykatu not; stosować go tam, gdzie użycie odcięć może zmniejszyć czytelność programu
8. Zmiany programu wywołane zastosowane predykatów assert i retract mogą w znaczący sposób ograniczyć zrozumienie jego działania — program może zachowywać się inaczej w innym czasie; jeśli zachowanie ma być powtarzalne musimy zadbać o odpowiednie odtwarzanie stanów sprzed użycia tych predykatów
9. Zastosowanie średnika (alternatywa celów) może czasami zmniejszyć czytelność klauzuli oraz zwiększyć liczbę przetwarzanych obiektów; można ją zwiększyć (liczbę obiektów odpowiednio zmniejszyć) poprzez rozbicie definicji klauzuli na alternatywne klauzule

Efektywność programów prologowych:

Zasadnicze aspekty dotyczące efektywności:

1. Brak zgodności między architekturą komputera a sposobem przetwarzania realizowanym przez mechanizm wnioskowania w Prologu, skutkiem czego szybciej napotkamy na ograniczenia czasowe lub pamięciowe
2. Jednak deklaratywny charakter programowania często skraca w stopniu znaczącym czas potrzebny na napisanie programu
3. Prostsza implementacja algorytmów opartych na przetwarzaniu symbolicznym i strukturalnych formach reprezentacji danych
4. Mniej efektywna w prologu implementacja algorytmów przetwarzania numerycznego

Ogólne metody poprawy efektywności:

1. Wybór, o ile to możliwe, kompilacji a nie interpretacji programu prologowego
2. Zmiana interpretacji proceduralnej programu prologowego:
   szukanie lepszego porządku klauzul, innej kolejności warunków, zastosowanie odcięć „zielonych”
3. Zmiana sposobu poszukiwania rozwiązania:
   — unikanie niepotrzebnych nawrotów,
   — unikanie analizy nadmiarowych alternatywnych ścieżek wnioskowania
4. Dobór lepszego z punktu widzenia efektywności form reprezentacji danych
5. Zastosowanie mechanizmów przechowywania wyników pośrednich

Słownik pojęć:

Klauzula — składająca się z nagłówka i ciała, ogólna forma przyjęta jako jedna z podstawowych jednostek prologowych. Klauzula dzieli się na fakty, reguły i pytania.

Ciało klauzuli — lista celów, najczęściej oddzielona znakiem koniunkcji (,)

Fakt prologowy — opisują to co zawsze jest bezwarunkowo prawdziwe. Fakty to klauzule które maja nagłówek i puste ciało

Reguła prologowa — opisują to czego prawdziwość zależy od pewnego warunku. Reguły to klauzule które mają zarówno nagłówek jak i niepuste ciało

Pytanie prologowe — klauzula która pozwala otrzymać odpowiedź szczególną lub ogólną

Unifikacja — proces (jawny lub niejawny) wiązania zmiennych w sposób który pozwala na dalszą logiczną ewaluację

Stałe symboliczne — (atomy, stałe atomowe) składają się z dużych liter, małych liter, cyfr, znaków specjalnych i są ciągami (konwencja zakłada że nazwa zaczyna się z małej litery, dopuszcza się ciągi złożone z samych znaków specjalnych dlatego że mechanizmy metadanych pozwalają programistom definiować własne operatory). Istnieją też łańcuchy znakowe używające apostrofa zamiast cudzysłowie np. ‚Ania’

Zmienne — ciągi liter, cyfr i znaków podkreślenia zaczynające się od dużej litery lub podkreślenia

Zmienna anonimowa — zmienna reprezentuje obiekt w relacji ale jego wartość nie wymaga zapisania do przyszłego użytku. Jej wartość nie jest istotna do spełnienia danej relacji

Term — wszelkie obiekty danych w Prologu. Do termów zaliczamy: stałe i zmienne prologowe oraz struktury reprezentowane za pomocą funktorów o dowolnej arności, których argumentami są inne termy (należy zwrócić uwagę na to ze znajduje się rekurencyjne definiowanie tych struktur). Dzięki temu możemy tworzyć struktury o dowolnej komplikacji

Instancja klauzuli — taka klauzula w której za każdą zmienna podstawiono jakiś term

Arność — ilośc argumentów którą przyjmuje funktor

Lista — podstawowa struktura danych w Prologu, składa się z dowolnych obiektów/termów

Metapredykaty — predefiniowane procedury systemowe, które traktują program prologowy jak dane, czyli obiekt którym możemy manipulować w trakcie przetwarzania programu. Mają różne role i mogą być różnych zadań

Multimodalność — parametry predykatów procedur mogą być zarówno wejściowe i wyjściowe.

Odcięcie — systemowy cel czy warunek, który jest natychmiast spełniony, gdy tylko zostanie osiągnięty znak odcięcia. Wszystkie cele, które zostały do tej chwili spełnione (w klauzuli zawierającej odcięcie) nie będą analizowane powtórnie w celu weryfikacji alternatywnych definicji.

Wszystkie operatory:

Arytmetyczne (wszystkie dokonują ewaluacji):

Operator	Operacja
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie rzeczywiste
// lub div	Dzielenie całkowitoliczbowe
mod	Reszta z dzielenia
Rem	Część ułamkowa z dzielenia
** lub ^	Potęgowanie

Porównania:

Opertor	Operacja	Dodatkowe infromacje
>	Większość	Następuje ewaluacja
<	Mniejszość	Następuje ewaluacja
>=	Większy lub równy	Następuje ewaluacja
=<	Mniejszy lub równy	Następuje ewaluacja
=	Unifikacja na żądanie	Jawna unifikacja
=:=	Równy (w sensie arytmetycznym)	Następuje ewaluacja
=\=	Różny (w sensie arytmetycznym)	Następuje ewaluacja
\=	Negacja uzgodnienia termów	Jawna unifikacja
==	Równy (w sensie leksykalnym)	zachowuje się różnie w zależności czy termy są związane czy nie, jeżeli występuje zmienna wolna to operator nie wykonuje unifikacji więc wyrażenie otrzyma wartość fałszywą! To samo dotyczy dwóch obiektów wolnych
is	Równy	Następuje ewaluacja a następnie unifikacja

Logiczne:

Operator	Operacja
,	Koniunkcja
;	Alternatywa

Reszta:

Operator	Operacja
:-	Definicja
!	Odcięcie
[|]	Podział Listy
.	Zakończenie klauzuli
%	Komentarz jednoliniowy

Paweł Koch