Programowanie 2009 Programming 2009

Lista zadań nr 1 Problem set no. 1

Na zajęcia 3–4 marca 2009 Due March 4, 2009

Grupy zasadnicze Basic groups

Zadanie 1 (1 pkt). Dane są predykaty:

Problem 1 (1 p). Given the predicates:

`parent/2` `male/1` `female/1`

Cel `parent`(a, b) jest spełniony wówczas, gdy a jest rodzicem b, zaś cele `male`(a) i `female`(a) — gdy a jest (odpowiednio) mężczyzną bądź kobietą.

Korzystając z powyższych predykatów zdefiniuj predykaty:

A goal `parent`(a, b) is satisfied when a is the parent of b. Goals `male`(a) and `female`(a) — when a is (respectively) a male or a female.

Using aforementioned predicates define the following ones:

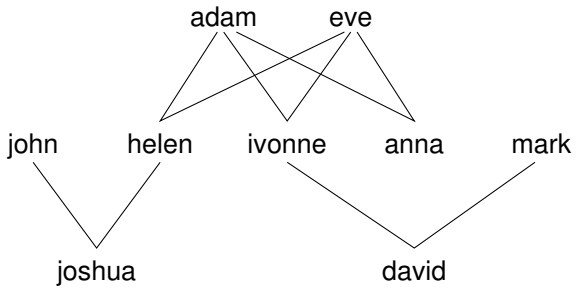
`sibling/2` `sister/2` `grandson/2` `cousin/2` `descendant/2`

Cel `sibling`(a, b) jest spełniony wówczas, gdy a i b są rodzeństwem, `sister`(a, b) — gdy a jest siostrą b, `grandson`(a, b) — gdy a jest wnukiem b, `cousin`(a, b) — gdy a jest kuzynem b (tj. synem ciotki, wuja lub stryja b), `descendant`(a, b) — gdy a jest potomkiem b.

Dołącz do programu zbiór faktów definiujących predykaty `parent`, `male` i `female` dla następujących zależności rodzinnych:

A goal `sibling`(a, b) is satisfied when a and b are siblings, `sister`(a, b) — when a is a sister of b, `grandson`(a, b) — when a is a grandson of b, `cousin`(a, b) — when a is a cousin of b (i.e. a son of an aunt or an uncle of b), `descendant`(a, b) — when a is a descendant of b.

Add a collection of facts defining predicates `parent`, `male` and `female` to your program for the following family relationships:



i zadaj Prologowi następujące pytania:

1. Czy John jest potomkiem Marka?
2. Kto jest potomkiem Adama?
3. Kto jest siostrą Ivonne?
4. Kto ma w tej rodzinie kuzyna i kim ten kuzyn jest?

Narysuj prologowe drzewa poszukiwań dla powyższych zapytań.

and ask Prolog the following questions:

1. Is John Mark's descendant?
2. Who is Adam's descendant?
3. Who is Ivonne's sister?
4. Who has a cousin in this family and who is this cousin?

Draw Prolog search trees for these queries.

Zadanie 2 (1 pkt). Zbuduj prologową bazę danych o bezpośrednich połaczeniach kolejowych między miastami (fakt, że istnieje połączenie z miasta A do miasta B nie implikuje, że istnieje połączenie odwrotne):

Problem 2 (1 p). Build a Prolog database containing facts about direct train connections between cities (the fact that there is a connection from a city A to a city B does not imply that there is a connection in the reverse direction):

from	to
wroclaw	warszawa
wroclaw	krakow
wroclaw	szczecin
szczecin	lublin
szczecin	gniezno
warszawa	katowice
gniezno	gliwice
lublin	gliwice

Zapytaj maszynę prologową:

1. czy istnieje bezpośrednie połączenie z Wrocławia do Lublina?
2. z jakimi miastami ma Wrocław bezpośrednie połączenie?
3. z jakich miast można dojechać do Gliwic z dokładnie jedną przesiadką?
4. z jakich miast można dojechać do Gliwic z co najwyżej dwiema przesiadkami? Czemu niektóre miasta są wymienione więcej niż raz?

Ask the Prolog machine:

1. if there is a direct connection from Wrocław to Lublin?
2. what cities has Wrocław a direct connection to?
3. from which cities one can reach Gliwice with exactly one change?
4. from which cities one can reach Gliwice with at most two changes? Why some cities are enumerated more than once?

Na wzór predykatu `descendant`/2 z poprzedniego zadania zaprogramuj predykat `connection`/2 spełniony wówczas, gdy istnieje połączenie pomiędzy podanymi miastami z dowolną liczbą przesiadek. Wyjaśnij dlaczego Twój predykat może działać niezgodnie z oczekiwaniem (nie musisz umieć zaprogramować poprawnego predykatu — powinieneś jedynie wyjaśnić, czemu taka naiwna implementacja jest niedobra).

Following the example of the predicate `descendant`/2 from the previous problem define the predicate `connection`/2 satisfied when there is a connection between two given cities with an arbitrary number of changes. Explain why your predicate may diverge from what is expected (you may not know how to define a proper predicate — you should only explain why such a naïve implementation is inappropriate).

Zadanie 3 (1 pkt). Przypomnijmy predykat `append`/3:

Problem 3 (1 p). Recall the `append`/3 predicate:

```
append([], X, X).
append([H|T], X, [H|Y]) :-
   append(T, X, Y).
```

Narysuj prologowe drzewo poszukiwań dla celu:

Draw the Prolog search tree for the goal:

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

Zadanie 4 (1 pkt). Przypomnijmy predykat `select`/3:

Problem 4 (1 p). Recall the `select`/3 predicate:

```
select(H, [H|T], T).
select(X, [H|T], [H|S]) :-
   select(X, T, S).
```

Narysuj prologowe drzewo poszukiwań dla celu:

Draw the Prolog search tree for the goal:

```
?- select(x, L, [a,b,c]).
```

Zadanie 5 (1 pkt). Zadaj Prologowi następujące pytania:

1. Jakie pary list mają tę własność, że druga jest wynikiem konkatenacji dwóch kopii pierwszej z nich?
2. Jaki element należy usunąć z listy `[a,b,c,d]` by otrzymać listę `[a,c,d]`?
3. Jaką listę należy dołączyć do listy `[a,b,c]`, by otrzymać listę `[a,b,c,d,e]`?

Zadanie 6 (1 pkt). Zaprogramuj w Prologu poniższe predykaty:

1. `even(X)`, spełniony gdy lista `X` ma parzystą liczbę elementów;
2. `palindrom(X)`, spełniony gdy lista `X` jest palindromem;
3. `singleton(L)`, spełniony gdy `X` jest listą jednoelementową.

Zadanie 7 (1 pkt). Zaprogramuj w Prologu poniższe predykaty:

1. `head`(H, L), spełniony gdy H jest pierwszym elementem (głową) listy L;
2. `deah`(L, H), spełniony gdy H jest ostatnim elementem listy L;
3. `tail`(T, L), spełniony gdy T jest listą wszystkich z wyjątkiem pierwszego elementów (ogonem) listy L;
4. `liat`(L, T), spełniony gdy T jest listą wszystkich z wyjątkiem ostatniego elementów listy L;
5. `prefix`(P, L), spełniony gdy P jest listą początkowych elementów (prefiksem) listy L;
6. `suffix`(L, S), spełniony gdy S jest listą końcowych elementów (sufiksem) listy L.

Problem 5 (1 p). Ask Prolog the following questions:

1. Which pairs of lists have the following property: the second one is the result of concatenation of two copies of the first one?
2. Which element has to be removed from the list `[a,b,c,d]` to obtain the list `[a,c,d]`?
3. What list has to be appended to the list `[a,b,c]` to obtain the list `[a,b,c,d,e]`?

Problem 6 (1 p). Define in Prolog the predicates listed below:

1. `even(X)`, satisfied when the list `X` is of even length;
2. `palindrom(X)`, satisfied when the list `X` is a palindrome,
3. `singleton(X)`, satisfied when `X` is a list containing exactly one element.

Problem 7 (1 p). Define in Prolog the predicates listed below:

1. `head`(H, L), satisfied when H is the first element (the head) of the list L;
2. `deah`(L, H), satisfied when H is the last element of the list L;
3. `tail`(T, L), satisfied when T is the list of all but the first of the elements (the tail) of the list L;
4. `liat`(L, T), satisfied when T is the list of all but the last of the elements of the list L;
5. `prefix`(P, L), satisfied when P is the list of leading elements (the prefix) of the list L;
6. `suffix`(L, S), satisfied when S is the list of trailing elements (the suffix) of the list L.

Grupy rozszerzone

Zadanie 1 (1 pkt). Zaprogramuj w Prologu taki predykat `sublist`/2, że cel `sublist`(l_1, l_2) jest spełniony, gdy lista l_2 jest *podlistą* listy l_1. Przy kolejnych nawrotach predykat powinien wygenerować wszystkie podlisty w dowolnej kolejności. Podlistą listy $[x_1, \ldots, x_n]$ nazywamy dowolną listę postaci $[x_{i_1}, \ldots, x_{i_k}]$ dla $1 \leq i_1 < \ldots < i_k \leq n$. Podlista powstaje zatem przez usunięcie z listy niektórych elementów. Lista n-elementowa ma 2^n podlist.

Extended groups

Problem 1 (1 p). Define in Prolog a predicate `sublist`/2 such that the goal `sublist`(l_1, l_2) is satisifed when the list l_2 is a *sublist* of the list l_1. During backtracking the predicate should generate all sublists in an arbitrary order. A sublist of the list $[x_1, \ldots, x_n]$ is any list of the form $[x_{i_1}, \ldots, x_{i_k}]$ for $1 \leq i_1 < \ldots < i_k \leq n$. A sublist is thus created by removing some elements from the given list. An n-element list has 2^n sublists.

Zadanie 2 (1 pkt). *Permutacją* listy $[x_1, \ldots, x_n]$ nazywamy dowolną listę postaci $[x_{i_1}, \ldots, x_{i_n}]$, gdzie $i_1, \ldots, i_n$ są parami różnymi liczbami z przedziału $1, \ldots, n$. Permutacja powstaje zatem przez przestawienie kolejności elementów na liście. Lista n-elementowa ma $n!$ permutacji.

Permutacje można generować *przez wybieranie*, zgodnie z następującym schematem rekurencyjnym:

- Jedyną permutacją listy pustej jest lista pusta.
- Aby wygenerować permutację pewnej niepustej listy, wybierz z niej dowolny element. Wybrany element będzie głową tworzonej permutacji. Jej ogonem będzie natomiast permutacja listy pozostałej po wybraniu tego elementu.

Zaprogramuj predykat `perm`/2 implementujący powyższy algorytm.

Permutacje można także generować *przez wstawianie*, zgodnie z następującym schematem rekurencyjnym:

- Jedyną permutacją listy pustej jest lista pusta.
- Aby wygenerować permutację pewnej niepustej listy, wygeneruj dowolną permutację jej ogona i następnie wstaw jej głowę w dowolne miejsce w wygenerowanej wcześniej permutacji.

Zaprogramuj predykat `perm`/2 implementujący powyższy algorytm.

Zadanie 3 (2 pkt). Rozważmy sygnaturę zawierającą symbol stałej `[]`/0 i binarny symbol funkcyjny `./2`. Potraktuj dwie definicje predykatu `perm`/2 z poprzedniego zadania jako formuły rachunku predykatów. Formuły te interpretujemy w modelu, w którym `[]` oznacza listę pustą, zaś `./2` — operację dołączenia głowy do listy. Ustal szczegóły tego modelu i udowodnij, że formuły odpowiadające obu wersjom predykatu `perm`/2 są w nim równoważne. Czy oznacza to, że oba programy będą działać tak samo?

Zadanie 4 (1 pkt). Najprostsza implementacja predykatu `reverse`/2:

Problem 2 (1 p). We call a *permutation* of the list $[x_1, \ldots, x_n]$ any list of the form $[x_{i_1}, \ldots, x_{i_n}]$, where $i_1, \ldots, i_n$ are pairwise different numbers from the set $1, \ldots, n$. A permutation is thus created by changing the order of elements in the given list. An n-element list has $n!$ permutations.

Permutations can be generated *by selection*, according to the following recursive scheme:

- The only permutation of the empty list is the empty list.
- In order to generate a permutation of some nonempty list, pick an arbitrary element from it. The selected element will be the head of the created permutation. Its tail is a permutation of the list left after selecting this element.

Define a predicate `perm`/2 which implements the algorithm stated above.

Permutations can also be generated *by insertion*, according to the following recursive scheme:

- The only permutation of the empty list is the empty list.
- In order to generate a permutation of some nonempty list, generate an arbitrary permutation of its tail, then insert its head into an arbitrary position in the generated permutation.

Define a predicate `perm`/2 which implements the algorithm stated above.

Problem 3 (2 p). Consider the signature containing the constant symbol `[]`/0 and the binary function symbol `./2`. Treat two definitions of the predicate `perm`/2 from the previous problem as the first order formulas. We interpret those formulas in the model, in which `[]` stands for the empty list and `./2` stands for an operation of adding a head to a list. Fix the details of this model and prove that formulas corresponding to the two versions of the predicate `perm`/2 are equivalent in this model. Does it mean that both programs will always work in the same way?

Problem 4 (1 p). The simplest implementation of the `reverse`/2 predicate:

```
reverse(X,Y) :-
   reverse(X,[],Y).

reverse([],A,A).
reverse([H|T],A,Y) :-
   reverse(T,[H|A],Y).
```

zapętla się w trybie `(-,+)`, tj. gdy pierwszy argument jest nieukonkretniony, a drugi ukonkretniony. Relacja „jest odwróceniem" jest symetryczna, oczekiwalibyśmy więc, że obliczenie celów `reverse`(a, b) oraz `reverse`(b, a) powinno dawać dokładnie ten sam efekt. Zaprogramuj predykat `reverse`/2 w taki sposób, by nie zapętlał się dla żadnych danych.

Zadanie 5 (1 pkt). Relacja „być permutacją" także jest symetryczna. Niestety w trybie `(-,+)` implementacje predykatu `perm`/2 z zadania 1 również nie działają zgodnie z intuicją. Popraw je tak, by wywołania `perm`(a, b) oraz `perm`(b, a) zawsze dawały ten sam efekt.

Zadanie 6 (1 pkt). Wskaż przykład formuły rachunku zdań, która nie jest równoważna żadnemu zbiorowi klauzul hornowskich. Podaj dowód.

loops when used in `(-,+)` mode, *i.e.* when its first argument is not instantiated while the second one is. Relation "is the reversion of" is symmetric, so we would rather expect the computation of goals `reverse`(a, b) and `reverse`(b, a) to give exactly the same outcome. Write the `reverse`/2 predicate in such a way that it never loops for any input data.

Problem 5 (1 p). The "to be a permutation" relation is symmetric as well. Unfortunately both implementations of the predicate `perm`/2 from the Problem 1 also work contrary to our intuition when called in the `(-,+)` mode. Correct them so that any pair of calls `perm`(a, b) and `perm`(b, a) gives the same outcome.

Problem 6 (1 p). Give an example of a propositional formula which is not equivalent to any set of Horn clauses. Give a proof.

Grupa zaawansowana

Rozważmy termy pierwszego rzędu (nad pewną niepustą sygnaturą i nieskończonym przeliczalnym zbiorem zmiennych). Termy będziemy nazywać *literałami pozytywnymi*, termy zanegowanie — *literałami negatywnymi*. *Klauzulami* nazywaliśmy na kursie logiki zbiory literałów. W tym zadaniu klauzulami będą *ciągi* literałów. Klauzula jest hornowska, jeśli zawiera co najwyżej jeden literał pozytywny. Literał pozytywny jest zawsze pierwszym elementem ciągu. Klauzule hornowskie dzielimy dalej na *klauzule określone*, zawierające dokładnie jeden literał pozytywny i *cele*, nie zawierające literałów pozytywnych. Będziemy rozważać zbiory klauzul hornowskich $\mathcal{C} \cup \{G\}$ zawierające dokładnie jeden cel G. Następująca reguła wnioskowania nazywa się *regułą SLD-rezolucji*:

$$\frac{\langle \neg t_1, \ldots, \neg t_n \rangle}{\langle \neg s_1, \ldots, \neg s_k, \neg t_2, \ldots, \neg t_n \rangle \theta} \; \langle s_0, \neg s_1, \ldots, \neg s_k \rangle \in \mathcal{C}, \theta = \mathrm{mgu}(t_1, s_0)$$

Zadanie 1 (1 pkt). Udowodnij, że jeśli istnieje ciąg celów otrzymanych przy pomocy powyższej reguły zaczynający się celem G i kończący klauzulą pustą, to zbiór klauzul $\mathcal{C} \cup \{G\}$ jest sprzeczny.

Zadanie 2 (5 pkt). Udowodnij, że jeśli zbiór klauzul $\mathcal{C} \cup \{G\}$ jest sprzeczny, to istnieje ciąg celów otrzymanych przy pomocy powyższej reguły zaczynający się celem G i kończący klauzulą pustą.

Zadanie 3 (1 pkt). Opisz system dowodzenia twierdzeń wykorzystujący regułę SLD-rezolucji. Czym różni się maszyna prologowa od tego systemu, tzn. czemu maszyna prologowa nie jest systemem dowodzenia twierdzeń?

Advanced group

Consider first order terms (over some nonempty signature and infinite countable set of variables). Terms will be called *positive literals*, negated terms — *negative literals*. At the logic course by a *clause* we meant a set of literals. In this problem a clause will be a *sequence* of literals. A *Horn clause* is a sequence of literals containing no more than one positive literal. The positive literal is always the first in the sequence. We divide Horn clauses further into *definite clauses* containing exactly one positive literal, and *goals* containing no positive literals at all. We will consider sets of clauses $\mathcal{C} \cup \{G\}$ containing exactly one goal G. The following inference rule is called *the rule of SLD-resolution*:

Problem 1 (1 p). Prove that if there exists a sequence of goals obtained by the inference rule stated above that starts with G and ends with the empty clause, then the clause set $\mathcal{C} \cup \{G\}$ is unsatisfiable.

Problem 2 (5 p). Prove that if a clause set $\mathcal{C} \cup \{G\}$ is unsatisfiable, then there exists a sequence of goals obtained by the inference rule stated above that starts with G and ends with the empty clause.

Problem 3 (1 p). Describe a theorem prover based on the SLD-resolution principle. What is the difference between the Prolog machine and this theorem prover, *i.e.* why the Prolog machine is not a theorem prover?