Programowanie 2009

Lista zadań nr 2

Na zajęcia 10–11 marca 2009

Grupy zasadnicze

Zadanie 1 (1 pkt). Niech predykat `connection/2` (zdefiniowany przez zbiór faktów) oznacza, że istnieje bezpośrednie połączenie między dwoma miastami. Zdefiniuj predykat `trip/3`, który znajduje połączenie między dwoma miastami z dowolną liczbą przesiadek i nie zapętla się nawet wówczas, gdy w grafie połączeń są cykle. Pierwsze dwa parametry tego predykatu (wejściowe), to miasta początkowe i końcowe. Trzeci parametr (wyjściowy), to lista miast od początkowego do końcowego, przez które przebiega podróż, np.:

```
?- trip(gliwice, warszawa, T).
T = [gliwice, wroclaw, warszawa] ;
T = [gliwice, wroclaw, katowice, warszawa] ;
false.
```

Wskazówka: zdefiniuj wpierw predykat `trip/4`, którego dodatkowy parametr jest „akumulatorem" — listą zawierającą już odwiedzone miasta. Użyj predykatu `member/2` by sprawdzić, czy miasto, do którego chcemy przejść, było już wcześniej odwiedzone. Ścieżkę buduj od końca ku początkowi, by uniknąć konieczności odwracania listy po znalezieniu rozwiązania.

Zadanie 2 (1 pkt). Zaprogramuj w Prologu predykaty:

1. `filter(+LNum, ?LPos)`, spełniony, gdy `LPos` unifikuje się z podlistą listy `LNum` zawierającą wszystkie nieujemne elementy listy `LNum`.
2. `count(+Elem, +List, ?Count)`, spełniony, gdy `Elem` unifikuje się z dokładnie n elementami listy `List` i `Count` unifikuje się z liczbą n.
3. `exp(+Base, +Exp, ?Res)`, spełniony, gdy `Res` unifikuje się wynikiem podniesienia liczby `Base` do potęgi `Exp`.

Zadanie 3 (1 pkt). Zaprogramuj w Prologu predykaty:

1. `factorial(+N, ?M)` spełniony, gdy `M` unifikuje się z silnią liczby `N`.

Programming 2009

Problem set no. 2

Due March 11, 2009

Basic groups

Problem 1 (1 p). Let the predicate `connection/2` (defined by a set of facts) mean, that there is a direct connection between two cities. Define a predicate `trip/3` which finds a connection between two cities with an arbitrary number of changes and does not loop even if there are loops in the connection graph. First two parameters of this predicate (input) stand for the starting and ending cities. The third parameter (output) is a list of cities from the starting to the ending city by which the trip goes, eg.:

Hint: first define a predicate `trip/4`. Its additional parameter should be an "accumulator" — a list containing cities already visited. Use the predicate `member/2` to find out if a city we want to go has already been visited. Build the path from the end toward the beginning to avoid reversing the list when the solution is found.

Problem 2 (1 p). Define the following predicates in Prolog:

1. `filter(+LNum, ?LPos)`, satisfied when `LPos` unifies with a sublist of the list `LNum` that contains all nonnegative elements of the list `LNum`.
2. `count(+Elem, +List, ?Count)` satisfied when `Elem` unifies with exactly n elements of the list `List` and `Count` unifies with the number n.
3. `exp(+Base, +Exp, ?Res)` satisfied when `Res` unifies with the result of taking the number `Base` to the power of `Exp`.

Problem 3 (1 p). Define the following predicates in Prolog:

1. `factorial(+N, ?M)` satisfied when `M` unifies with the factorial of `N`.

2. `concat_number(+Digits, ?Num)`, spełniony, gdy lista `Digits` zawiera ciąg cyfr rozwinięcia dziesiętnego liczby, która unifikuje się z `Num`.
3. `decimal(+Num, ?Digits)`, spełniony, gdy `Digits` unifikuje się z z ciągiem cyfr rozwinięcia dziesiętnego liczby `Num`. Program ma zwracać poprawny wynik dla liczb nieujemnych.

2. `concat_number(+Digits, ?Num)` satisfied when the list `Digits` contains a sequence of digits of the decimal representation of the number that unifies with `Num`.
3. `decimal(+Num, ?Digits)` satisfied when `Digits` unifies with the list of digits of the decimal representation of the number `Num`. Your program should return the correct result for nonnegative numbers.

Zadanie 4 (1 pkt). Napisz zapytanie prologowe, które rozwiąże następującą łamigłówkę:

Problem 4 (1 p). Write a Prolog query that solves the following puzzle:

```
          U  S  A
+      U  S  S  R
-----------------
=  P  E  A  C  E
```

Zapytanie powinno zawierać zmienne `A`, `C`, `E`, `P`, `R`, `S` i `U` oraz powinno sprawdzić wszystkie możliwe podstawienia pod te zmienne parami różnych cyfr `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` i `9`. Cyfry podstawione pod zmienne `U` i `P` nie mogą być zerem. W zapytaniu możesz użyć następujących predykatów standardowych:

The query should contain variables `A`, `C`, `E`, `P`, `R`, `S` and `U`, and should try all possible substitutions for these variables pairwise different digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` and `9`. Digits substituted for variables `U` and `P` should differ from zero. You can use the following standard predicates in your query:

`permutation`/2, `length`/2, `is`/2, `\=`/2,

predykatu `concat_number`/2 z poprzedniego zadania oraz predykatu:

the predicate `concat_number`/2 from the previous problem and the predicate:

```
sublist([],[]).
sublist([_|S],T) :-
   sublist(S,T).
sublist([H|S],[H|T]) :-
   sublist(S,T).
```

Zapytanie powinno zwrócić dokładnie jedną odpowiedź, bez powtórzeń:

The query should return exactly one answer, without repetitions:

```
?- the query (omitted).
A = 2,
C = 7,
E = 0,
P = 1,
R = 8,
S = 3,
U = 9 ;
false.

?-
```

Zadanie 5 (1 pkt). Zaprogramuj predykat `select_min(+NumList, ?Min, ?Rest)` który wybiera najmniejszy element `Min` listy liczb `NumList` i zwraca pozostałe elementy na liście `Rest`. Wykorzystaj go do zaprogramowania predykatu `sel_sort`/2 sortującego listę liczb całkowitych używając algorytmu sortowania przez wybieranie.

Problem 5 (1 p). Define a predicate `select_min(+NumList, ?Min, ?Rest)` that selects the smallest element `Min` from the list `NumList` of integers and returns the remining elements in `Rest`. Use it to define a predicate `sel_sort`/2 which sorts a list of integers using the selection sort algorithm.

Zadanie 6 (1 pkt). Zaprogramuj predykat `insert(+NumList, +Elem, ?Res)`, który wstawia liczbę `Elem` do listy liczb całkowitych `NumList` nie zaburzając porządku i unifikuje wynik z `Res`. Wykorzystaj go do zaprogramowania predykatu `ins_sort`/2 sortującego listę liczb całkowitych za pomocą algorytmu sortowania przez wstawianie.

Problem 6 (1 p). Define a predicate `insert(+NumList, +Elem, ?Res)` which inserts the number `Elem` into the list `NumList` preserving the ordering of elements and unifies the result with `Res`. Use it to define a predicate `ins_sort`/2 which sorts a list of integers using the insertion sort algorithm.

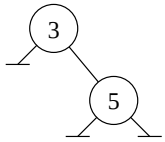
Zadanie 7 (1 pkt). *Drzewa binarne o etykietowanych wierzchołkach wewnętrznych* zapisujemy w Prologu używając funktora `node`/3 do reprezentowania wierzchołków wewnętrznych i atomu `leaf`/0 do reprezentowania liści. Dla przykładu struktura:

Problem 7 (1 p). We represent *binary trees with labelled internal nodes* in Prolog using the functor `node`/3 for internal nodes and the atom `leaf`/0 for leaves. E.g., the structure:

```
node(leaf, 3, node(leaf, 5, leaf))
```

reprezentuje drzewo:

represents the tree:



Zaprogramuj predykaty `mirror`/2 — tworzący lustrzane odbicie drzewa oraz `flatten`/2 — tworzący listę etykiet drzewa w porządku infiksowym.

Define predicates `mirror`/2, which creates the mirror image of a tree and `flatten`/2, which creates a list of the tree labels in the infix order.

Zadanie 8 (1 pkt). *Binarne drzewa poszukiwań* będziemy zapisywać podobnie, jak w poprzednim zadaniu. Zaprogramuj predykat `insert`/3 — wstawiający element do drzewa (z powtórzeniami). Zaprogramuj następnie algorytm *Treesort*, który wstawia wszystkie elementy listy do drzewa poszukiwań i następnie spłaszcza drzewo otrzymując posortowaną listę.

Problem 8 (1 p). We will represent *binary search trees* as in the previous problem. Define a predicate `insert`/3, which inserts an element to a tree (with repetitions). Next program the *Treesort* algorithm which inserts all the elements of a list into a tree and then flattens the tree to obtain the sorted list.

Grupy rozszerzone

Extended groups

Zadanie 1 (1 pkt). W tym zadaniu, inaczej niż w zadaniach 7–8 dla grup zasadniczych, będziemy zakładać, że drzewa poszukiwań nie zawierają dwóch równych etykiet. Zaprogramuj następujące predykaty: `insert`/3 — wstawiający element do drzewa (bez powtórzeń), `find`/2 — sprawdzający, czy element znajduje się w drzewie, `findMax`/2 — ujawniający największy element w drzewie, `delMax`/3 — ujawniający i usuwający największy element z drzewa, `delete`/3 — usuwający podany element z drzewa oraz `empty`/1 — sprawdzający, czy drzewo jest puste.

Problem 1 (1 p). In this problem, unlike in Problems 7–8 for the basic groups, we will assume that binary search trees do not contain two equal labels. Define predicates: `insert`/3 which inserts an element to a tree (without repetitions), `find`/2 which checks if an element occurs in a tree, `findMax`/2 which finds the biggest element in a tree, `delMax`/3 which returns and removes the biggest element from a tree, `delete`/3 which removes a particular element from a tree, and `empty`/1 which checks if a tree is empty.

Zadanie 2 (1 pkt). Zaprogramuj predykat `revall`/2 odwracający listę wraz z podlistami (tutaj przez podlistę rozumiemy element listy będący listą):

Problem 2 (1 p). Define a predicate `revall`/2 which reverses a list together with its sublists (here by a sublist we mean a member of a list being itself a list):

```
?- revall([1,[2,3,[4,5],6,[7,[8,9]]]], X).
X = [[[[9,8],7],6,[5,4],3,2],1]
```

Napisz następnie predykat `flatten`/2, który „spłaszcza” listę:

Then define a predicate `flatten`/2, which “flattens” a list:

```
?- flatten([1,[2,3,[4,5],6,[7,[8,9]]]], X).
X = [1,2,3,4,5,6,7,8,9]
```

Zadbaj o to, by Twoje predykaty nie generowały nieużytków!

Ensure your prediates do not generate any garbage!

Zadanie 3 (1 pkt). Napisz predykat `appn`/2, który, podobnie jak `append`/3, łączy listy, jednak nie dwie, tylko ich dowolną liczbę. Listy do połączenia są elementami listy będącej jego pierwszym parametrem, np.:

Problem 3 (1 p). Define a predicate `appn`/2 which, like `append`/3, concatenates lists, but any number of lists instead of just two. The lists to be concatenated are elements of a list being the first argument of the predicate:

```
?- appn([[1,2,3], [4,5], [6,7,8,9]], Y).
Y = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Predykat musi działać efektywnie. W szczególności wywołania rekurencyjne powinny być ogonowe. *Wskazówka:* użyj otwartych struktur danych.

The predicate has to be efficient. In particular it should be tail recursive. *Hint:* use open structures.

Zadanie 4 (1 pkt). Zaprogramuj w Prologu predykat `length`/2, który działa poprawnie we wszystkich trybach użycia, w szczególności nie zapętla się dla celu `length(X,5)` (ten cel powinien być spełniony dokładnie na jeden sposób z podstawieniem listy `[_,_,_,_,_]`) pod `X`.

Problem 4 (1 p). Define in Prolog a predicate `length`/2 which works correctly in any mode, in particular does not fall into endless loop for the goal `length(X,5)` (this goal should be satisfied exactly once with the list `[_,_,_,_,_]` substituted for `X`).

Zadanie 5 (1 pkt). Zaprogramuj ogonowe wersje predykatów opisanych w zadaniach 2–3 dla grup zasadniczych.

Problem 5 (1 p). Define tail recursive versions of the predicates described in Problems 2–3 for the basic groups.

Zadanie 6 (1 pkt). Zaprogramuj predykat `halve(+X, -L, -R)` który kopiuje pierwszą połowę listy `X` do listy `L` i unifikuje zmienną `R` z drugą połową listy `X`. Lista `L` powinna zostać skopiowana, zaś lista `R` — współdzielona. Jeśli lista `X` ma nieparzystą liczbę elementów, to dłuższa powinna być lista `R`.

Zaprogramuj predykat `merge`/3 który, zachowując porządek, łączy dwie listy liczb całkowitych uporządkowane rosnąco. Zadbaj o to, by predykat działał deterministycznie, a rekursja była ogonowa (tam, gdzie to niezbędne, użyj odcięć).

Użyj predykatów `halve`/3 i `merge`/3 do zdefiniowania predykatu `merge_sort`/2 implementującego algorytm sortowania *Mergesort*.

Problem 6 (1 p). Define a predicate `halve(+X, -L, -R)` which copies the first half of the list `X` to the list `L`, and unifies the variable `R` with the second half of the list `X`. The list `L` should be copied, but `R` — shared. If the list `X` has an odd length, the list `R` should be longer.

Define a predicate `merge`/3 which, preserving the order of elements, combines two lists of elements sorted increasingly. Ensure that the predicate is deterministic and tail recursive (use cuts when needed).

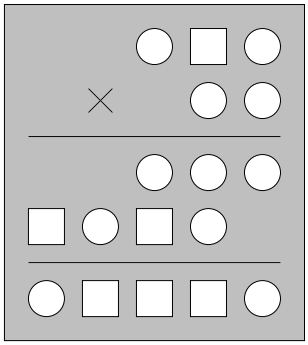
Use the predicates `halve`/3 and `merge`/3 to define a predicate `merge_sort`/2 that implements the *Mergesort* algorithm.

Zadanie 7 (1 pkt). Zdefiniuj predykat `split(+List, +Med, -Small, -Big)` który rozdziela listę `List` liczb całkowitych na listy: `Small` — elementów mniejszych i `Big` — elementów nie mniejszych niż liczba `Med`. Wykorzystaj go do zdefiniowania predykatu `qsort`/2 implementującego algorytm *Quicksort*. Uwaga: w fazie łączenia posortowanych list nie należy generować nieużytków — nie używaj więc do tego celu predykatu `append`/3. Dodatkowy akumulator powinien wystarczyć.

Problem 7 (1 p). Define a predicate `split(+List, +Med, -Small, -Big)` which splits a list `List` of integers into lists: `Small` containing the elements which are less than, and `Big` — which are greater or equal to the integer `Med`. Use this predicate to define a predicate `qsort`/2 that implements the *Quicksort* algorithm. Warning: the concatenation of sorted lists should not generate any garbage — do not use the predicate `append`/3 for that purpose. An additional accumulator should suffice.

Zadanie 8 (1 pkt). Napisz w Prologu program rozwiązujący zadania takie jak to, które pochodzi z numeru 11/2000 miesięcznika „Wiedza i Życie": *Wpisz w kwadraty cyfry parzyste (0, 2, 4, 6, 8), w koła zaś nieparzyste (1, 3, 5, 7, 9) tak, by otrzymać poprawny zapis mnożenia:*

Problem 8 (1 p). Write in Prolog a program solving problems similar to the following one, which appeared in the 11/2000 issue of the monthly magazine "Knowledge & Life": *Fill in the squares with even digits (0, 2, 4, 6, 8) and the circles with odd digits (1, 3, 5, 7, 9) in a way to obtain a correct multiplication chart:*



Dane do programu (opis problemu) należy podać w postaci listy list atomów `s` i `c` reprezentujących kwadraty i koła. Np. dane do problemu z obrazka są następujące:

Input data (description of a problem) should be given in the form of a list of lists of atoms `s` and `c` representing squares and circles, respectively. E.g., the data from the figure are the following:

```
[[c,s,c], [c,c], [c,c,c], [s,c,s,c], [c,s,s,s,c]]
```

Grupa zaawansowana

Advanced group

Zadanie 1 (2 pkt). Zapiszmy predykat

Problem 1 (2 p). Let us write the predicate

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

w postaci formuły rachunku predykatów I rzędu nad sygnaturą $\Sigma = \{nil, cons, append\}$:

in the form of the first order predicate formula over the signature $\Sigma = \{nil, cons, append\}$:

$$\Phi \quad \equiv \quad append(nil, nil) \wedge \forall H, X, T, S \ (append(T, X, S) \Rightarrow append(cons(H, T), X, cons(H, S))) \, .$$

Niech H będzie zbiorem termów stałych nad sygnaturą $\{nil, cons\}$ (zwanym uniwersum Herbranda). Rozważmy struktury relacyjne nad sygnaturą Σ i uniwersum H (we wszystkich tych strukturach każdy term zamknięty oznacza samego siebie, a struktury te różnią się jedynie interpretacją ternarnego symbolu relacyjnego *append*). Wprowadzamy na nich częściowy porządek, porównując interpretacje symbolu relacyjnego *append* za pomocą relacji zawierania. Pokaż, że istnieje najmniejsza struktura $\mathfrak{M}(\Phi)$ wśród wszystkich struktur spełniających formułę Φ. Nazywamy ją *modelem* powyższego programu prologowego. Zauważ, że w strukturze $\mathfrak{M}(\Phi)$ relacja *append* faktycznie odpowiada spinaniu list.

Let H stands for the set of ground terms over the signature $\{nil, cons\}$ (call the Herbrand universe). Consider relational structures over the signature Σ and the universe H (in all of these structures any ground term stands for itself, and these structures differ only with respect to their interpretation of the ternary relational symbol *append*). We introduce a partial ordering on those structures comparing interpretations of the relational symbol *append* by inclusion. Prove that there exists the smallest structure $\mathfrak{M}(\Phi)$ among all structures satisfying the formula Φ. We call this structure *the model* of the Prolog program presented above. Note that the relation *append* in the structure $\mathfrak{M}(\Phi)$ actually corresponds to the concatenation of lists.

Zadanie 2 (2 pkt). Rozważmy specyfikację problemu sortowania list:

Problem 2 (2 p). Consider a specification of the list sorting problem:

```
sort(X,Y) :-
   perm(X,Y),
   monotone(Y).
monotone([]).
monotone([_]).
monotone([H1,H2|T]) :-
   H1 =< H2,
   monotone([H2|T]).
```

Niech `perm` będzie predykatem implementującym generowanie permutacji przez wstawianie (zadanie 2 z listy 1 dla grup rozszerzonych). Powyższy program odpowiada algorytmowi sortowania przez wstawianie, ale ma złożoność wykładniczą zamiast kwadratowej.

Mówimy, że dwa predykaty (będące częścią dwóch programów prologowych) są *semantycznie równoważne względem semantyki logicznej*, jeśli ich interpretacje w modelach tych programów są równe. Przebuduj powyższy program tak, by otrzymać program z zadania 6 dla grup zasadniczych i pokaż, że nowy predykat sortowania (posiadający kwadratową złożoność) jest semantycznie równoważny predykatowi `sort`/2 z powyższego programu. W dowodzie możesz korzystać z faktu, że interpretacja relacji `=<`/2 jest liniowym porządkiem.

Czy to, że oba predykaty są logicznie równoważne oznacza, że będą działać tak samo?

Let `perm` be the predicate implementing the permutation by insertion algorithm (problem 2 from list 1 for extended groups). The program shown above corresponds to the sorting by insertion algorithm, but has exponential rather than quadratic complexity.

We say that two prediactes (which are parts of two Prolog programs) are *semantically equivalent with respect to the logical semantics*, if their interpretations in the models of these programs are equal. Rebuild the program written above in a way to obtain the program from Problem 6 for basic groups and prove, that the new sorting predicate (which has quadratic complexity) is semantically equivalent to the predicate `sort`/2 from the program above. You can assume in your proof that the interpretation of the relation `=<`/2 is a linear order.

Does the logical equivalence of the programs imply that they will behave in the same way?

Zadanie 3 (1 pkt). *Kolekcje* to struktury danych służące do przechowywania elementów. Niech interfejs kolekcji składa się z następujących nazw predyktów:

- `put(+E,+S,-R)` wstawia element `E` do kolekcji `S` i zwraca nową kolekcję `R`;
- `get(+S,-E,-R)` usuwa element z kolekcji `S`, podstawia go pod `E` i zwraca nową kolekcję `R`;
- `empty(?S)` sprawdza lub tworzy pustą kolekcję;
- `addall(-E, +G, +S, -R)` wstawia wszystkie wyniki podstawień pod zmienną `E`, które spełniają cel `G` (w którym zmienna `E` występuje) do kolekcji `S` i zwraca nową kolekcję `R` (ten predykat przypomina standardowe predykaty `findall`/3 i `findall`/4).

Podaj dwie implementacje tego interfejsu, jedną dla stosu (użyj list zamkniętych do reprezentowania kolekcji), drugą dla kolejek FIFO (tu użyj list różnicowych).

Problem 3 (1 p). *Collections* are data structures for storing elements. Let the interface for collections consists of the following predicate names:

- `put(+E,+S,-R)` inserts an element `E` to a collection `S` and returns a new collection `R`;
- `get(+S,-E,-R)` removes an element from a collection `S`, substitutes this element for `E` and returns a new collection `R`;
- `empty(?S)` checks if a collection is empty or generates an empty collection;
- `addall(-E, +G, +S, -R)` adds all results of substitutions for the variable `E` which satisfy the goal `G` (in which the variable `E` occurs) to a collection `S` and returns a new collection `R` (this predicate resembles the standard predicates `findall`/3 and `findall`/4).

Give two implementations of this interface, one for stacks (use closed lists to represent collections), the other one for FIFO queues (use difference lists).

Zadanie 4 (1 pkt). Rozważmy skierowany graf $G = \langle V, E \rangle$. Jego wierzchołki ze zbioru V będziemy reprezentować w postaci atomów i liczb, a relację krawędzi E — za pomocą binarnego predykatu `e`/2. Dane są wierzchołki $v_1, v_2 \in V$. Ścieżkę z wierzchołka v_1 do wierzchołka v_2 w grafie G można znaleźć używając algorytmu przeszukiwania sparametryzowanego kolekcją przechowującą oczekujące wierzchołki:

1. Jeśli kolekcja przechowująca oczekujące wierzchołki jest pusta, to zakończ pracę.
2. W przeciwnym razie wyjmij wierzchołek v z kolekcji. Jeśli v nie został wcześniej odwiedzony, to odwiedź go i wstaw wszystkich jego `e`-sąsiadów do kolekcji. W przeciwnym razie odrzuć wierzchołek v. Powtórz całą procedurę.

Zaprogramuj powyższy algorytm tak, by znajdował w grafie ścieżki między podanymi wierzchołkami używając interfejsu kolekcji z poprzedniego zadania. Następnie użyj stosów, by otrzymać DFS oraz kolejek, by otrzymać BFS.

Przypuśćmy, że wierzchołkom grafu dodajemy *wagi* i używamy kolejek priorytetowych do przechowywania oczekujących wierzchołków. Zbadaj zachowanie takiego algorytmu.

Zadanie 5 (1 pkt). Ulepsz predykat `halve`/2 z zadania 6 dla grup rozszerzonych tak, by nie kopiował także lewej połowy listy. Użyj do tego celu list różnicowych. Zaprogramuj następnie algorytm *Mergesort* w postaci predykatu sortującego listy różnicowe.

Zadanie 6 (1 pkt). Zaprogramuj predykat `prime/1` implementujący sito Eratostenesa, który działa poprawnie zarówno w przypadku generowania (tj. wywołany z nieukonkretnioną zmienną jako parametrem), jak i sprawdzania. W tym drugim przypadku jego argumentem może być dowolny term arytmetyczny (niekoniecznie literał całkowitoliczbowy). W przypadku wywołania z innym parametrem powinien zostać zgłoszony błąd arytmetyczny. Podczas przesiewania kandydat na liczbę pierwszą powinien być porównywany z wcześniej wygenerowanymi liczbami pierwszymi w kolejności od najmniejszej do największej. Użyj listy różnicowej aby sprawnie zaimplementować wstawianie liczby na koniec listy.

Problem 4 (1 p). Consider a directed graph $G = \langle V, E \rangle$. We will represent vertices from V using atoms or numbers and the edge relation E using the binary predicate `e`/2. Given two vertices $v_1, v_2 \in V$. We can find a path from v_1 to v_2 in G using the searching procedure parametrized with a collection storing pending vertices:

1. If the collection for pending vertices is empty, then we are done.
2. Otherwise get a vertex v from this collection. If v has not been visited, then visit it and add all its `e`-neighbours to the collection. Otherwise throw v away. Repeat the procedure.

Implement this algorithm so that it finds a path between given vertices using the interface for collections from the previous problem. Then use stacks to obtain DFS, and queues to obtain BFS.

Let us add *weights* to vertices and use the priority queues for storing pending vertices. Research the behaviour of this algorithm.

Problem 5 (1 p). Improve the predicate `halve`/2 from Problem 6 for extended groups so that it does not make a copy of the left half of the list as well. Use difference lists for that purpose. Next define the *Mergesort* algorithm in the form of a predicate for sorting difference lists.

Problem 6 (1 p). Define a predicate `prime/1` that implements the sieve of Erathostenes and works correctly for generating (when called with uninstantiated variable as a parameter), and for checking as well. In the second case it can be called with any arithmetic term, not necessarily a number. When called with other kinds of arguments it should report the arithmetic error. During sieving the candidate for the next prime number should be compared with previously generated primes, from the smallest to the biggest. Use difference lists to implement the insertion into the end of a list efficiently.