Some Prolog programming exercises

Michał Szafraniuk

24 maja 2019

Workout

Natural numbers here are represented with `0` and successor function `s/1`. Type conditions have been mostly omitted for the sake of clarity.

nat/1

`nat(x)`: success iff x is a natural number

```
nat(0).
nat(s(X)) :- nat(X).
```

plus/3

`plus(x, y, z)`: success iff $x + y = z$

```
plus(0, X, X).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
```

minus/3

`minus(x, y, z)`: success iff $x - y = z$.

```
minus(X, X, 0).
plus(s(X), Y, s(Z)) :- minus(X, Y, Z).
```

le/2

`le(x, y)`: success iff $x \leqslant y$.

```
le(0, s(X)).
le(X, X).
le(s(X), s(Y)) :- le(X, Y).
```

lt/2

`lt(x, y)`: success iff $x < y$.

```
lt(0, s(X)).
lt(s(X), s(Y)) :- lt(X, Y).
```

times/3

`times(x, y,z)`: success iff $xy = z$.

```
times(0, X, 0).
times(s(X), Y, Z) :-
    times(X, Y, XY),
    plus(XY, Y, Z).
```

exp/3

`exp(n, x, y)`: success iff $x^n = y$.

```
exp(s(_), 0, 0).
exp(0, s(_), s(0)).
exp(s(N), X, Y) :-
    exp(N, X, Z),
    times(Z, X, Y).
```

fact/2

`factorial(k, n)`: success iff $k! = n$.
Simplistic, overly complex (left-recursive) solution:

```
factorial(0, s(0)).
factorial(s(K), N) :-
    factorial(K, M),
    times(M, s(K), N).
```

Tail recursive solution:

```
factorial(K, N) :- factorial(K, 1, N).
factorial(0, N, N) :- !.
factorial(K, Acc, N) :-
    K1 is K - 1,
    Acc1 is Acc * K,
    factorial(K1, Acc1, N).
```

odd/1

`odd(n)`: success iff $2 \nmid n$.

```
odd(s(0)).
odd(s(s(X))) :- odd(X).
```

even/1

`even(n)`: success iff $2 \mid n$.

```
even(0).
even(s(s(X))) :- even(X).
```

twice/2

`twice(x, y)`: success iff $y = 2x$.

```
twice(0, 0).
twice(s(X), s(s(Y))) :- twice(X, Y).
```

mod/3

`mod(x, y, z)`: success iff $x \mod y = z$

```
mod(X, Y, X) :- lt(X, Y).
mod(X, Y, Z) :-
    plus(X1, Y, X),
    mod(X1, Y, Z).
```

gcd/3

`gcd(x, y, z)`: success iff z is the greatest common divisor of x and y

```
gcd(X, 0, X) :- lt(X, 0).
gcd(X, Y, G) :-
    mod(X, Y, Z),
    gcd(Y, Z, G).
```

fib/2

- naive top-down (exponential)

```
fibo_td(0,0).
fibo_td(1,1).
fibo_td(N,F):-
  N>1, N1 is N-1, N2 is N-2,
  fibo_td(N1,F1), fibo_td(N2,F2),
  F is F1+F2.
```

- bottom-up (linear)

```
fibo_bu(N,F):-fibo_bu(0,0,1,N,F).
fibo_bu(N,F,_,N,F).
fibo_bu(N1,F1,F2,N,F):-
  N1<N, N2 is N1+1, F3 is F1+F2,
  fibo_bu(N2,F2,F3,N,F).
```

: success iff

Lists

list/1

`list(xs)`: success iff *xs* if Prolog list

```
list([]).
list([_|Xs]) :-
    list(Xs).
```

first/2

`first(x, xs)`: success iff `x` is the first element of `xs`

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

member/2

`member(x, xs)`: success iff `x` is an element of the list `xs`.

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

memberOnce/2

`memberOnce(x, xs)`: success iff `x` is an element of the list `xs`, with cut.

```
memberOnce(X, [X|_]) :- !.
memberOnce(X, [_|Ys]) :-
    memberOnce(X, Ys).
```

append/3

`append(xs, ys, xsys)`: success iff `xsys` is the result of concatenation of `xs` and `ys`.

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

prefix/2

`prefix(xs, ys)`: success iff `xs` is a prefix of `ys`.

```
prefix([], _).
prefix([X|Xs], [X|Ys]) :-
    prefix(Xs, Ys).
```

suffix/2

`suffix(xs, ys)`: success iff `xs` is a suffix of `ys`.

```
sufix(Xs, Xs).
sufix(Xs, [_|Ys]) :-
    sufix(Xs, Ys).
```

len/2

`len(xs, n)`: success iff `xs` has length `n`.
Using terms:

```
len([], 0).
len([_|Xs], s(N))
    :- len(Xs, N).
```

Using numbers, left-recirsive:

```
len([], 0).
len([_|Xs], N) :-
    len(Xs, N1),
    N is N1 + 1.
```

Using numbers, tail-recursive with accumulator:

```
len(Xs, N) :-
    len(Xs, 0, N).
len([], N, N).
len([_|Xs], Acc, N) :-
    Acc1 is Acc + 1,
    len(Xs, Acc1, N).
```

sublist/2

`sublist(xs, ys)`: success iff `xs` is a sublist of `ys`.

- using `append`:

```
sublist(Xs, AsXsBs) :-
    append(AsXs, _, AsXsBs),
    append(_, Xs, AsXs).
```

- sublist as a sufix of some prefix:

```
sublist2(Xs, AsXsBs) :-
    prefix(AsXs, AxXsBs),
    suffix(Xs, AsXs).
```

- sublist as a prefix of some sufix:

```
sublist3(Xs, AsXsBs) :-
    suffix(XsBs, AsXsBs),
    prefix(Xs, XsBs).
```

- recursively:

```
sublist4(Xs, Ys) :-
    prefix(Xs, Ys).
sublist4(Xs, [Y|Ys]) :-
    sublist4(Xs, Ys).
```

reverse/2

`reverse(xs, ys)`: success iff `xs` is a reversed version of the list `ys`.

```
reverse(Xs, Ys) :-
    reverse(Xs, [], Ys).
reverse([], Ys, Ys).
reverse([X|Xs], Acc, Ys) :-
    reverse(Xs, [X|Acc], Ys).
```

last/2

`last(x, xs)`: success iff `x` is the last element of `xs`.

```
last(X, Xs) :-
    append(_, [X], Xs).

last2(X, Xs) :-
    suffix([X], Xs).
```

exlast/2

`exlast(xs, ys)`: success iff `xs` is the list `ys` ex last element.

```
exlast([], [X]).
exlast([X|Xs], [X|Ys]) :-
    exlast(Xs, Ys).
```

len/2

`len(xs, n)`: success iff `xs` has `n` elements.

- without built-in arithmetic

```
len([], 0).
len([X|Xs], s(N)) :-
    len(Xs, N).
```

- using built-in airthemtic, left-recursive (less efficient)

```
len2([], 0).
len2([_|Xs], N) :-
    len2(Xs, M),
    N is M+1.
```

- using built-in airthemtic, tail-recursive (more efficient)

```
len3(Xs, N) :- len3(Xs, 0, N).
len3([_|Xs], A, N) :-
    A2 is A+1,
    len3(Xs, A2, N).
len3([], A, A).
```

maxElement/2

`maxElement(xs, z)`: success iff `z` is the maximal element of the integer list `xs`.

```
maxElement([X|Xs], Z) :-
    maxElement(Xs, X, Z).
maxElement([X|Xs], Acc, Z) :-
    X > Acc,
    maxElement(Xs, X, Z).
maxElement([X|Xs], Acc, Z) :-
    X =< Acc,
    maxElement(Xs, Acc, Z).
maxElement([], Acc, Acc).
```

middle/2

`middle(x, ys)`: success iff `x` is the middle element of the odd length list `ys`.

```
middle(X,[X]).
middle(X, [Y|Ys]) :-
    exlast(Zs, Ys),
    middle(X, Zs).
```

```
middle2(M,[M]).
middle2(M, [Z1, Z2|Zs]) :-
    middle2(M, Z2, [], Zs).
middle2(M, M, As, [Z]).
middle2(M, X, [], [Z1, Z2|Zs]) :-
    middle2(M, Z1, [Z2], Zs).
middle2(M, X, [A|As], [Z1, Z2|Zs]) :-
    append(As, [Z1, Z2], As2),
    middle2(M, A, As2, Zs).
```

separate/3

`separate(xs, os, es)`: success iff the lists `os` and `es` consist of `xs`'s elements of odd and even indexes respectively. Indexing starting at 1 is assumed.

```
separate([X], [X], []).
separate([X|Xs], [X|Os], Es) :-
    separate(Xs, Es, Os).
```

subseq/2

`subseq(xs, ys)`: success iff `xs` is a subsequence of `ys`.

```
subseq([], _).
subseq([X|Xs], [X|Ys]) :-
    subseq(Xs, Ys).
subseq([X|Xs], [Y|Ys]) :-
    subseq([X|Xs], Ys).
```

justBefore/3

`justBefore(x, y, zs)`: success iff `x` is an element just before `y` in `zs`.

```
justBefore(X, Y, [X|Zs]) :-
    first(Y, Zs).
justBefore(X, Y, [_|Zs]) :-
    justBefore(X, Y, Zs).
```

before/3

`before(x, y, zs)`: success iff `x` is an element before `y` in `zs`.

```
before(X, Y, [X|Zs]) :-
    element(Y, Zs).
before(X, Y, [_|Zs]) :-
    before(X, Y, Zs).
```

palindrome/1

`palindrome(xs)`: success iff `xs` is a palindrome

```
palindrome(Xs) :-
    reverse(Xs, Xs).
```

plFlag/2

`plFlag(ls, fs)`: success iff `ls` is a list of constants `a, b` and `fs` is sorted version of `ls`.

```
plFlag(Ls, Fs) :-
    plFlag(Ls, [], Fs).
plFlag([b|Ls], Acc, [b|Fs]) :-
    plFlag(Ls, Acc, Fs).
plFlag([c|Ls], Acc, Fs) :-
    plFlag(Ls, [c|Acc], Fs).
plFlag([], Acc, Acc).
```

nlFlag/2

`nlFlag(ls, fs)`: success iff `ls` is a list of constants `a, b, c` and `fs` is sorted version of `ls`.

```
nlFlag(Ls, Fs) :-
    nlFlag(Ls, [], [], Fs).
nlFlag([c|Ls], Bs, Ns, [c|Fs]) :-
    nlFlag(Ls, Bs, Ns, Fs).
nlFlag([b|Ls], Bs, Ns, Fs) :-
    nlFlag(Ls, [b|Bs], Ns, Fs).
nlFlag([n|Ls], Bs, Ns, Fs) :-
    nlFlag(Ls, Bs, [n|Ns], Fs).
nlFlag([], [b|Bs], Ns, [b|Fs]) :-
    nlFlag([], Bs, Ns, Fs).
nlFlag([], [], Ns, Ns).
```

delete/3

`deleteAll(z, xs, ys)`: success iff `ys` is a list `xs` with all occurances of `z` removed

```
deleteAll(Z, [Z|Xs], Ys) :-
    deleteAll(Z, Xs, Ys).
deleteAll(Z, [X|Xs], [X|Ys]) :-
    Z \= X,
    deleteAll(Z, Xs, Ys).
deleteAll(Z, [], []).
```

select/3

`select(z, xs, ys)`: success iff `ys` is a list `xs` with exactly one occurance of `z` removed

```
select(X, [X|Xs], Xs).
select(Z, [X|Xs], [X|Ys]) :-
    select(Z, Xs, Ys).
```

insert/3

`insert(z, xs, ys)`: success iff `ys` is `xs` with element `z` arbitrarily inserted

```
insert(Z, Xs, Ys) :-
    select(Z, Ys, Xs).
```

permutation/2

`permutation(xs, ys)`: success iff `xs` is a permutation of `ys`

- using `append`

```
permutation([], []).
permutation([X], [X]) :-!.
permutation([X|Xs], Ys) :-
    permutation(Xs, Zs),
    append(L1, L2, Zs),
    append(L1, [X], X1),
    append(X1, L2, Ys).
```

- using `select`

```
permutation2(Xs, [Z|Zs]) :-
    select(Z, Xs, Ys),
    permutation2(Ys, Zs).
permutation2([], []).
```

- using `insert`

```
permutation3([X|Xs], Zs) :-
    permutation3(Xs, Ys),
    insert(X, Ys, Zs).
permutation3([], []).
```

ordered/1

`ordered(xs)`: success iff `xs` is a list of integers in nondecreasing order.

```
ordered([]).
ordered([X]).
ordered([X, Y| Ys]) :-
    X =< Y,
    ordered([Y|Ys]).
```

orderedInsert/3

`ordered_insert(z, xs, ys)`: success iff `ys` is a `xs` with element `z` inserted such that all elements to the left are smaller.

```
ordered_insert(X, [], [X]).
ordered_insert(X, [Y|Ys], [Y|Zs]) :-
    X > Y,
    ordered_insert(X, Ys, Zs).
ordered_insert(X, [Y|Ys], [X, Y|Ys]) :-
    X =< Y.
```

partition/4

`partition(xs, z, ls, rs)`: success iff `xs` is partitioned into `ls` of elements smaller or equal than `z` and `rs` of elements larger than `z`.

```
partition([X|Xs], Z, [X|Ls], Rs) :-
    X =< Z,
    partition(Xs, Z, Ls, Rs).
partition([X|Xs], Z, Ls, [X|Rs]) :-
    X > Z,
    partition(Xs, Z, Ls, Rs).
partition([], Z, [], []).
```

Sorting

permutationSort/2

`permutationSort(xs, ys)`: success iff `ys` sorted version of `xs`.

```
permutationSort(Xs, Ys) :-
    permutation(Xs, Ys),
    ordered(Ys).
```

insertionSort/2

`insertionSort(xs, ys)`: success iff `ys` sorted version of `xs`.

```
insertion_sort([X|Xs], Ys) :-
    insertion_sort(Xs, Zs),
    ordered_insert(X, Zs, Ys).
insertion_sort([], []).
```

quickSort/2

`quickSort(xs, ys)`: success iff `ys` sorted version of `xs`.

```
quickSort([X|Xs], Ys) :-
    partition(Xs, X, Ls, Rs),
    quickSort(Ls, LSs),
    quickSort(Rs, RSs),
    append(LSs, [X|RSs], Ys).
quickSort([], []).
```

Other

substitute/4

`substitute(e, f, ys, zs)`: success iff `zs` is a list in which all occurances of `e` in `ys` have been substituted with `f`.

```
substitute(E, F, [E|Ys], [F|Zs]) :-
    substitute(E, F, Ys, Zs).
substitute(E, F, [Y|Ys], [Y|Zs]) :-
    E \= Y,
    substitute(E, F, Ys, Zs).
substitute(E, F, [], []).
```

set/2

`set(xs, ys)`: success iff `ys` is a set of `xs` .

```
set([], []).
set([X|Xs], Ys) :-
    member(X, Xs),
    !,
    set(Xs, Ys).
set([X|Xs], [X|Ys]) :-
    set(Xs, Ys).
```

split/4

`split(xs, k, ls, rs)`: success iff `xs` is split into `ls=xs[1..k]` and `rs = xs[k+1..n]`

```
split(Xs, 0, [], Xs).
split([X|Xs], N, [X|Ls], Rs) :-
    Nless is N - 1,
    split(Xs, Nless, Ls, Rs).
```

splitHalf/3

`splitHalf(xs, ls, rs)`: success iff `xs` is split into half onto `ls` and `rs`.

```
splitHalf(Xs, Ls, Rs) :-
    len3(Xs, N),
    Half is N//2,
    split(Xs, Half, Ls, Rs).
```

merge/3

`merge(ls, rs, xs)`: success iff `xs` is `ls` merged with `rs` .

```
merge(Ls, [], Ls).
merge([], Rs, Rs).
merge([L|Ls], [R|Rs], [L|Xs]) :-
    L =< R,
    merge(Ls, [R|Rs], Xs).
merge([L|Ls], [R|Rs], [R|Xs]) :-
    L > R,
    merge([L|Ls], Rs, Xs).
```

mergeSort

`mergeSort(xs, ys)`: success iff `ys` . TODO: need to cut, zwraca za duzo

```
mergeSort([], []).
mergeSort([X], [X]).
mergeSort(Xs, Ys) :-
    splitHalf(Xs, Ls, Rs),
    mergeSort(Ls, SLs),
    mergeSort(Rs, SRs),
    merge(SLs, SRs, Ys).
```

kth_largest

: success iff `ys` .

: success iff `ys` .

Binary Trees

We define a binary tree with functor `tree(Element, Left, Right)` and `void` representing empty tree.

binary_tree/1

`binary_tree(t)`: success iff `t` is a binary tree.

```
binary_tree(void).
binary_tree(tree(E, Left, Right)) :-
    binary_tree(Left),
    binary_tree(Right).
```

tree_member/2

`tree_member(x, t)`: success iff `x` is an element of a tree `t` .

```
tree_member(X, tree(X, Left, Right)).
tree_member(X, tree(Z, Left, Right)) :-
    tree_member(X, Left).
tree_member(X, tree(Z, Left, Right)) :-
    tree_member(X, Right).
```

isotree/2

`isotree(t1, t2)`: success iff `t1` and `t2` are isomorphic (same up to branch reordering).

```
isotree(void, void).
isotree(tree(X, Left1, Right1), tree(X, Left2, Right2)) :-
    isotree(Left1, Left2),
    isotree(Right1, Right2).
isotree(tree(X, Left1, Right1), tree(X, Left2, Right2)) :-
    isotree(Left1, Right2),
    isotree(Right1, Left2).
```

substitute/4

: success iff `ys` .

```
substitute(void, Xold, Xnew, void).
substitute(tree(Elem1, Left1, Right1), Xold, Xnew, tree(Elem2, Left2, Right2)) :-
    replace(Xold, Xnew, Elem1, Elem2),
    substitute(Left1, Xold, Xnew, Left2),
    substitute(Right1, Xold, Xnew, Right2).
```

replace/4

`replace(x, y, e1, e2)`: success iff `ys` .

```
replace(Xold, Xnew, Xold, Xnew).
replace(Xold, Xnew, Elem, Elem) :-
    Xold \= Elem.
```

preorder/2

`preorder(t, xs)`: success iff `xs` is preorder traversal of `t`.

```
preorder(tree(X, Lt, Rt), Xs) :-
    preorder(Lt, Ls),
    preorder(Rt, Rs),
    append([X|Ls], Rs, Xs).
preorder(void, []).
```

inorder/2

`inorder(t, xs)`: success iff `xs` is inorder traversal of `t`.

```
inorder(tree(X, Lt, Rt), Xs) :-
    inorder(Lt, Ls),
    inorder(Rt, Rs),
    append(Ls, [X|Rs], Xs).
inorder(void, []).
```

postorder/2

`postorder(t, xs)`: success iff `xs` is inorder traversal of `t`.

```
inorder(tree(X, Lt, Rt), Xs) :-
    inorder(Lt, Ls),
    inorder(Rt, Rs),
    append(Ls, [X|Rs], Xs).
inorder(void, []).
```

heapify/2

`heapify(t, h)`: success iff `h` is a heapified `t` .

```
% greater(X, t) == X is greater than the root of t
greater(X, void).
greater(X, tree(X1, Lt, Rt)) :-
    X >= X1.
```

```
% empty tree is heap
heapify(void, void).
% recursively get heaps and then just heapify-down the root if
% necessary ("adjust")
heapify(tree(X, Lt, Rt), Heap) :-
    heapify(Lt, Lheap),
    heapify(Rt, Rheap),
    adjust(X, Lheap, Rheap, Heap).
% root is greater than both child's roots
adjust(X, Lheap, Rheap, tree(X, Lheap, Rheap)) :-
    greater(X, Lheap),
    greater(X, Rheap).
% left child root is greater than original root and right root
% left root becomes main root, original root goes down to the
% left
adjust(X, tree(Z, Lt, Rt), Rheap, tree(Z, Lheap, Rheap)) :-
    X < Z,
    greater(Z, Rheap),
    adjust(X, Lt, Rt, Lheap).
% symmetrically
adjust(X, Lheap, tree(Z, Lt, Rt), tree(Z, Lheap, Rheap)) :-
    X < Z,
    greater(Z, Lheap),
    adjust(X, Lt, Rt, Rheap).
```

subtree/2

`subtree(t1, t2)`: success iff `t1` is a subtree of `t2`.

```
subtree(T, T).
subtree(T, tree(X, Lt, Rt)) :-
    subtree(T, Lt).
subtree(T, tree(X, Lt, Rt)) :-
    subtree(T, Rt).
```

sum_tree/2

`sum_tree(t, n)`: success iff `n` is a sum of node elements of integer tree `t`.

```
sum_tree(tree(X, Lt, Rt), N) :-
    sum_tree(Lt, Ln),
    sum_tree(Rt, Rn),
    N is X+Ln+Rn.
sum_tree(void, 0).
```

height/2

`height(t, h)`: success iff `h` is the height of the `t`.

```
height(tree(X, Lt, Rt), N) :-
    height(Lt, Lh),
```

```
        height(Rt, Rh),
        N is max(Lh, Rh) + 1.
    height(void, -1).
```

tree_insert/3

`tree_insert(x, t, tx)`: success iff `tx` is a bst of bst `t` with `x` inserted.

```
tree_insert(X, tree(Y, Lt, Rt), tree(Y, Lxt, Rt )) :-
    X < Y,
    tree_insert(X, Lt, Lxt).
tree_insert(X, tree(Y, Lt, Rt), tree(Y, Lt, Rxt)) :-
    X > Y,
    tree_insert(X, Rt, Rxt).
tree_insert(X, tree(X, Lt, Rt), tree(X, Lt, Rt)).
tree_insert(X, void, void, tree(X, void, void)).
```

flatten

: success iff `ys` .

```
flatten(Xs, Zs) :-
    flatten(Xs, [], Zs).
flatten([X|Xs], Acc, Zs) :-
    flatten(Xs, Acc, Ys),
    flatten(X, Ys, Zs).
flatten(X, Acc, [X|Acc]) :-
    integer(X).
flatten([], Zs, Zs).
```

path/3

`path(x, t, path)`: success iff list `path` is a path from root to `x` in `t` .

```
path(X, tree(Y, Lt, Rt), [Y|Ps]) :-
    path(X, Lt, Ps).
path(X, tree(Y, Lt, Rt), [Y|Ps]) :-
    path(X, Rt, Ps).
path(X, tree(X, Lt, Rt), [X]).
```

count_leaves/2

`count_leaves(t, n)`: success iff `n` is the number of leaves in `t` .

```
% count_leaves
count_leaves(void, 0).
count_leaves(tree(E, void, void), 1) :- !.
count_leaves(tree(E, Lt, Rt), N) :-
    count_leaves(Lt, N1),
    count_leaves(Rt, N2),
    N is N1 + N2.
```

Graphs

We will use two graph representations:

- clause form, f.e.:

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

- term form, f.e.:

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

connect/2

`connect(u, v)`: success iff there is a path between `u` and `v` in clause predefined DAG

```
% clause representation, DAG
connect(A, B) :-
    edge(A, X),
    connect(X, B).
connect(A, B) :-
    edge(A, B).
```

connect/3

`connect(graph, u, v)`: success iff there is a path between `u` and `v` in DAG `graph`

```
% term representation, DAG
connect([e(A, B) | Gs], A, B).
connect([e(A, Y) | Gs], A, B) :-
    connect(Gs, Y, B).
connect([e(X, Y) | Gs], A, B) :-
    connect(Gs, A, B).
```

path_dag/3

`path_dag(a, b, ps)`: success iff `ps` is a path (list of nodes) from `a` to `b` in clause-predefined dag

```
% clause representation, path in DAG
path_dag(A, B, [A|Ps]) :-
    edge(A, X),
    path_dag(X, B, Ps).
path_dag(A, B, [A, B]) :-
    edge(A, B).
```

path/3

`path(a, b, ps)`: success iff `ps` is a path (list of nodes) from `a` to `b` in (potentially cyclical) graph

```
path(A, B, P) :-
    path(A, B, [A] , P).
path(A, B, V, [A, B]) :-
    edge(A, B).
path(A, B, V, [A|P]) :-
    edge(A, X),
    \+ member(X, V),
    path(X, B, [X |V], P).
```

: success iff

: success iff

: success iff

: success iff

: success iff

: success iff

: success iff

: success iff

: success iff

: success iff

: success iff