

Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016



Summary of Last Lecture

Definition

the difference of two lists is denotes as $As \setminus Bs$ and called **difference list**

Example

```
number(I) —>
    digit(D0),
    digits(D),
    {number_codes(I, [D0|D])}.
```

```
digits([D|T]) —>
    digit(D), digits(T).
```

```
digits([]) —>
    [].
```

```
digit(D) —>
    [D],
    {code_type(D, digit)}.
```

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, database and recursive programming, termination, complexity

Incomplete Data Structures and Constraints

incomplete data structures, definite clause grammars, constraint logic programming, answer set programming

Full Prolog

semantics (revisited), correctness proofs, meta-logical predicates, cuts non-deterministic programming, efficient programs, complexity

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, database and recursive programming, termination, complexity

Incomplete Data Structures and Constraints

incomplete data structures, **definite clause grammars**, constraint logic programming, answer set programming

Full Prolog

semantics (revisited), correctness proofs, meta-logical predicates, cuts non-deterministic programming, efficient programs, complexity

Example (Definite Clause Grammars)

`sentence(sentence(N,V)) → noun_phrase(N), verb_phrase(V).`

`noun_phrase(np(D,N)) → determiner(D), noun_phrase2(N).`

`noun_phrase(np(N)) → noun_phrase2(N).`

`noun_phrase2(np2(A,N)) → adjective(A), noun_phrase2(N).`

`noun_phrase2(np2(N)) → noun(N).`

`verb_phrase(vp(V,N)) → verb(V), noun_phrase(N).`

`verb_phrase(vp(V)) → verb(V). determiner → [the].`

`determiner → [a].`

`noun → [pie-plate].`

`noun → [surprise].`

`adjective → [decorated].`

`verb → [contains].`

`sentence(PT) $\overset{*}{\Rightarrow}$ "the decorated pie-plate contains a surprise"`

Example

sentence(PT) \Rightarrow^* ‘‘the decorated pie-plate contains a surprise’’

sentence(PT) \Rightarrow^* ‘‘the decorated pie-plates contain a surprise’’

Example

```
sentence(PT)  $\Rightarrow^*$  ‘‘the decorated pie-plate contains a surprise’’
sentence(PT)  $\Rightarrow^*$  ‘‘the decorated pie-plates contain a surprise’’
```

Example

```
determiner(det(the))  $\rightarrow$  [the].
determiner(det(a))  $\rightarrow$  [a].
noun(noun(pie-plate))  $\rightarrow$  [pie-plate].
noun(noun(pie-plates))  $\rightarrow$  [pie-plates].
noun(noun(surprise))  $\rightarrow$  [surprise].
noun(noun(surprises))  $\rightarrow$  [surprises].
adjective(adj(decorated))  $\rightarrow$  [decorated].
verb(verb(contains))  $\rightarrow$  [contains].
verb(verb(contain))  $\rightarrow$  [contain].

sentence(PT)  $\Rightarrow^*$  ‘‘the decorated pie-plates contain a surprise’’
```

Extension: Number Agreement

Example

```
sentence(sentence(NP,VP),Num) →  
    noun_phrase(N,Num), verb_phrase(V,Num).
```

```
⋮
```

```
determiner(det(the),Num) → [the].
```

```
determiner(det(a),singular) → [a].
```

```
noun(noun(pie-plate),singular) → [pie-plate].
```

```
noun(noun(pie-plates),plural) → [pie-plates].
```

```
noun(noun(surprise),singular) → [surprise].
```

```
noun(noun(surprises),plural) → [surprises].
```

```
adjective(adj(decorated)) → [decorated].
```

```
verb(verb(contains),singular) → [contains].
```

```
verb(verb(contain),plural) → [contain].
```

```
sentence(PT)  $\Rightarrow^*$  ‘‘the decorated pie-plates contain a surprise’’
```


Example

```
sentence —>  
    subject ,  
    predicate .
```

```
subject —>  
    [the], [big], [bear].
```

```
subject —>  
    "the", "little", "lion".
```

```
predicate —>  
    [roars].
```

```
predicate —>  
    [is, happy].
```

```
predicate —>  
    [lives, in, the, golden, city].
```

Example

```
sentence —>  
    subject ,  
    predicate .
```

```
subject —>  
    [the], [big], [bear].
```

```
subject —>  
    "the", "little", "lion".
```

```
predicate —>  
    [roars].
```

```
predicate —>  
    [is, happy].
```

```
predicate —>  
    [lives, in, the, golden, city].
```

```
:- phrase(sentence, Text), Text = [the, big, bear, roars].
```

```
:- phrase(sentence, Text), Text = [116, 104, 101, 108, 105|_].
```

Regular Predicate from Within a DCG

Task

write a DCG for `number(N)` that recognised numbers in English:

```
?- phrase(number(N),"onehundredandseventyfive").  
N = 175 ;
```

Regular Predicate from Within a DCG

Task

write a DCG for `number(N)` that recognised numbers in English:

```
?- phrase(number(N), "onehundredandseventyfive").  
N = 175 ;
```

Definition

Prolog provides an arithmetical interface

Value is Expression

Regular Predicate from Within a DCG

Task

write a DCG for `number(N)` that recognised numbers in English:

```
?- phrase(number(N), "onehundredandseventyfive").
N = 175 ;
```

Definition

Prolog provides an arithmetical interface

Value is Expression

Example

`X is 3+5`

`8 is 3+5`

`N is N+1`

`X ↦ 8`

`true`

nonsensical

Arithmetic Operations

- + - * // (integer division) / (float division)

Arithmetic Operations

- + - * // (integer division) / (float division)
- ...

Arithmetic Operations

- + - * // (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- < =< > >=

Arithmetic Operations

- + - * // (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- < =< > >=

?- 3 > 2.

true

Arithmetic Operations

- + - * // (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- < =< > >=

?- 3 > X.

ERROR: >/2: Arguments are not sufficiently instantiated

Arithmetic Operations

- + - * // (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- < =< > >=
- == (equality)

Arithmetic Operations

- `+` `-` `*` `//` (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `:=` (equality)
 ?- `1+2 = 3.`
 `false`

Arithmetic Operations

- `+` `-` `*` `//` (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `==` (equality)
 ?- `1+2 == 3.`
 true

Arithmetic Operations

- `+` `-` `*` `//` (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `:=` (equality)
- `\=` (disequality)

Arithmetic Operations

- `+` `-` `*` `//` (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `:=` (equality)
- `=\=` (disequality)
 - ?- `1+2 =\= 3.`
 - `false`

Arithmetic Operations

- `+` `-` `*` `//` (integer division) / (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `:=` (equality)
- `\=` (disequality)
 - ?- `1+2 \= 2+1.`
 - `false`

Arithmetic Operations

- `+` `-` `*` `//` (integer division) / (float division)
- `...`

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `:=` (equality)
- `=\=` (disequality)
 - ?- `1+2 =\= 2+1.`
 - `false`

Non Standard Predicates

- `between(Low, High, Value)` is true when
 - 1 `Value` is an integer, and $Low \leq Value \leq High$
 - 2 `Value` is a variable, and $Value \in [Low, High]$

Arithmetic Operations

- `+` `-` `*` `//` (integer division) / (float division)
- `...`

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `:=` (equality)
- `=\=` (disequality)
 - ?- `1+2 =\= 2+1.`
 - `false`

Non Standard Predicates

- `between(Low, High, Value)` is true when
 - 1 `Value` is an integer, and $Low \leq Value \leq High$
 - 2 `Value` is a variable, and $Value \in [Low, High]$
- `succ(Int1, Int2) ...`

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
factorial(0,1).
```

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
  
factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
```

```
fibonacci(1,1).
```

```
fibonacci(N,X) :-
```

```
    N > 1,
```

```
    fibonacci(N-1,Y),
```

```
    fibonacci(N-2,Z),
```

```
    X = Y+Z.
```

```
?- fibonacci(3,X).
```

```
false
```

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
  
factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
```

```
fibonacci(1,1).
```

```
fibonacci(N,X) :-
```

```
    N > 1,
```

```
    fibonacci(N-1,Y),
```

```
    fibonacci(N-2,Z),
```

```
    X = Y+Z.
```

```
?- fibonacci(3,X).
```

```
false
```

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
  
factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
```

```
fibonacci(1,1).
```

```
fibonacci(N,X) :-
```

```
    N > 1,
```

```
    N1 is N-1, fibonacci(N1,Y),
```

```
    N2 is N-2, fibonacci(N2,Z),
```

```
    X = Y+Z.
```

```
?- fibonacci(3,X).
```

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
  
factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
```

```
fibonacci(1,1).
```

```
fibonacci(N,X) :-
```

```
    N > 1,
```

```
    N1 is N-1, fibonacci(N1,Y),
```

```
    N2 is N-2, fibonacci(N2,Z),
```

```
    X = Y+Z.
```

```
?- fibonacci(3,X).
```

```
X ↦ 1+1
```

```
true
```


Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
  
factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
```

```
fibonacci(1,1).
```

```
fibonacci(N,X) :-
```

```
    N > 1,
```

```
    N1 is N-1, fibonacci(N1,Y),
```

```
    N2 is N-2, fibonacci(N2,Z),
```

```
    X = Y+Z.
```

```
?- fibonacci(3,X).
```

```
X ↦ 1+1
```

```
true
```

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
  
factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
```

```
fibonacci(1,1).
```

```
fibonacci(N,X) :-
```

```
    N > 1,
```

```
    N1 is N-1, fibonacci(N1,Y),
```

```
    N2 is N-2, fibonacci(N2,Z),
```

```
    X is Y+Z.
```

```
?- fibonacci(3,X).
```

Example (Factorials)

```
factorial(0,s(0)).
```

```
factorial(s(N),F) ←  
    factorial(N,F1),  
    times(s(N),F1,F).
```

```
factorial(N,F) ←  
    N>0, N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
  
factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
```

```
fibonacci(1,1).
```

```
fibonacci(N,X) :-
```

```
    N > 1,
```

```
    N1 is N-1, fibonacci(N1,Y),
```

```
    N2 is N-2, fibonacci(N2,Z),
```

```
    X is Y+Z.
```

```
?- fibonacci(3,X).
```

```
X ↦ 2
```

```
true
```

Solution

`number(0) —> "zero".`

`number(N) —> xxx(N).`

`xxx(N) —> digit(D), "hundred", rest_xxx(N1),
 {N is D * 100 + N1}.`

`rest_xxx(0) —> "".`

`rest_xxx(N) —> "and", xx(N).`

`xx(N) —> digit(N).`

`xx(N) —> teen(N).`

`xx(N) —> tens(T), rest_xx(N1), {N is T + N1}.`

`rest_xx(0) —> "".`

`rest_xx(N) —> digit(N).`

`digit(1) —> "one".`

`digit(2) —> "two".`

`teen(10) —> "ten".`

`tens(20) —> "twenty".`

Example

```
deriv_from_(X,X,1) —>
```

```
    [D],
```

```
    {atom_char(X,D)}.
```

```
deriv_from_(N,X,0) —>
```

```
    number(N).
```

```
...
```

```
deriv_from_(A+B,X,DerA+DerB) —>
```

```
    "(", deriv_from_(A,X,DerA),
```

```
    "+",
```

```
    deriv_from_(B,X,DerB), ")" .
```

```
deriv_from_(A*B,X,DerA*B+A*DerB) —>
```

```
    "(", deriv_from_(A,X,DerA),
```

```
    "*",
```

```
    deriv_from_(B,X,DerB), ")" .
```

```
deriv_from_(A-B,X,DerA-DerB) —>
```

```
    "(", deriv_from_(A,X,DerA),
```

```
    "-",
```

```
    deriv_from_(B,X,DerB), ")" .
```

```
:- phrase(deriv_from_(A,x,ADer), '(((x*x)+y)-(3*(x+1)))').
```

Once Again: Difference Lists

Example

```
:- listen_zusammen ([[1,2],[4,5]],[1,2,4,5]).
```

```
listen_zusammen(Xss,Xs) :-  
    phrase(seqq(Xss),Xs).
```

Once Again: Difference Lists

Example

```
:- listen_zusammen ([[1,2],[],[4,5]],[1,2,4,5]).
```

```
listen_zusammen(Xss,Xs) :-  
    phrase(seqq(Xss),Xs).
```

```
seqq([]) —>  
    [].
```

```
seqq([Xs|Xss]) —>  
    seq(Xs),  
    seqq(Xss).
```

Once Again: Difference Lists

Example

```
:- listen_zusammen ([[1,2],[4,5]],[1,2,4,5]).
```

```
listen_zusammen(Xss,Xs) :-  
    phrase(seqq(Xss),Xs).
```

```
seqq([]) —>  
    [].
```

```
seqq([Xs|Xss]) —>  
    seq(Xs),  
    seqq(Xss).
```

```
seq([]) —>  
    [].
```

```
seq([C|Cs]) —>  
    [C],  
    seq(Cs).
```


Example (reverse revisited)

```
reverse_dl(Xs,Ys) :-
    reverse_dl(Xs,Ys,[]).
```

```
reverse_dl([],Ys,Ys).
reverse_dl([X|Xs],Ys0,Ys) :-
    reverse_dl(Xs,Ys0,[X|Ys]).
```

```
:- reverse_dl(Xs,[b,a]), Xs = [a, b].
```

```
reverse_dcg(List,Reversed) :-
    phrase(reverse(List),Reversed).
```

```
reverse([]) —>
    [].
reverse([X|Xs]) —>
    reverse(Xs),
    [X].
```

```
:- reverse_dcg(Xs,[b,a]), Xs = [a, b].
```

Generate and Test

Theorem (Four Colour Theorem)

no more than four colours are required to colour the regions of a map so that no two adjacent regions have the same color

Generate and Test

Theorem (Four Colour Theorem)

no more than four colours are required to colour the regions of a map so that no two adjacent regions have the same color

Example



Auxiliary Predicates

Example

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

```
member(X,[X|_Xs]).
member(X,[_Y|Xs]) :-
    member(X,Xs).
```

```
:- subset_of([b,d],[a,b,c,d]).
```

```
subset_of([],_Ys).
subset_of([X|Xs],Ys) :-
    member(X,Ys),
    subset_of(Xs,Ys).
```

Generate and Test

Example

```
is_map([region(a,A,[B,C,D]), region(b,B,[A,C,E]),  
        region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),  
        region(e,E,[B,C,F]), region(f,F,[C,D,E]))).
```

Generate and Test

Example

```
is_map([region(a,A,[B,C,D]), region(b,B,[A,C,E]),  
        region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),  
        region(e,E,[B,C,F]), region(f,F,[C,D,E]))).  
  
coloured_map([Region|Regions], Colours) :-  
    coloured_region(Region,Colours),  
    coloured_map(Rregions,Colours).  
coloured_map([],Colours).
```

Generate and Test

Example

```
is_map([region(a,A,[B,C,D]), region(b,B,[A,C,E]),
        region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
        region(e,E,[B,C,F]), region(f,F,[C,D,E]))).
```

```
coloured_map([Region|Regions], Colours) :-
    coloured_region(Region,Colours),
    coloured_map(Regions,Colours).
coloured_map([],Colours).
```

```
coloured_region(region(Name,Colour,Neighbours), Colours) :-
    select(Colour,Colours,Colours1),
    subset_of(Neighbours,Colours1).
```

Generate and Test

Example

```
is_map([region(a,A,[B,C,D]), region(b,B,[A,C,E]),
        region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
        region(e,E,[B,C,F]), region(f,F,[C,D,E]))).
```

```
coloured_map([Region|Regions], Colours) :-
    coloured_region(Region,Colours),
    coloured_map(Regions,Colours).
coloured_map([],Colours).
```

```
coloured_region(region(Name,Colour,Neighbours), Colours) :-
    select(Colour,Colours,Colours1),
    subset_of(Neighbours,Colours1).
```

```
test_colour(Map) :-
    is_map(Map),
    is_colours(Colours),
    coloured_map(Map,Colours).
```


Nondeterministic Programming

Example

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

Nondeterministic Programming

Example

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

Definition

A **NFA** is quintuple $(Q, \Sigma, \Delta, I, F)$ such that

- 1 Q is a set of states
- 2 Σ is an alphabet
- 3 Δ is relation on $(Q \times \Sigma) \times Q$
- 4 I are the initial states
- 5 F are the final states

Example

```
accept(S) :-  
    initial(Q),  
    accept(Q,S).  
  
accept(Q, [X|Xs]) :-  
    delta(Q,X,Q1),  
    accept(Q1,Xs).  
  
accept(Q, []) :-  
    final(Q).
```

Example

```
accept(S) :-  
    initial(Q),  
    accept(Q,S).  
  
accept(Q, [X|Xs]) :-  
    delta(Q,X,Q1),  
    accept(Q1,Xs).  
  
accept(Q, []) :-  
    final(Q).  
  
initial(q0).  
final(q2).
```

Example

```
accept(S) :-  
    initial(Q),  
    accept(Q,S).  
  
accept(Q, [X|Xs]) :-  
    delta(Q,X,Q1),  
    accept(Q1,Xs).  
  
accept(Q, []) :-  
    final(Q).  
  
initial(q0).  
final(q2).  
  
delta(q0,0,q0).  
delta(q0,0,q1).  
delta(q0,1,q0).  
delta(q1,1,q2).
```

Example

```
accept(S) :-  
    initial(Q),  
    accept(Q,S).  
  
accept(Q, [X|Xs]) :-  
    delta(Q,X,Q1),  
    accept(Q1,Xs).  
  
accept(Q, []) :-  
    final(Q).  
  
initial(q0).  
final(q2).  
  
delta(q0,0,q0).  
delta(q0,0,q1).  
delta(q0,1,q0).  
delta(q1,1,q2).  
  
:- accept([0,0,0,1,0,1]).
```

Type Predicates

Recall

type predicates are unary relations concerning the type of a term

Type Predicates

Recall

type predicates are unary relations concerning the type of a term

Definition

- `is_list`: type check for a list

Type Predicates

Recall

type predicates are unary relations concerning the type of a term

Definition

- `is_list`: type check for a list
- **integer**: type check for an **integer**
- **atom**: type check for an **atom**
- **compound**: type check for a **compound** term

Type Predicates

Recall

type predicates are unary relations concerning the type of a term

Definition

- `is_list`: type check for a list
- **integer**: type check for an **integer**
- **atom**: type check for an **atom**
- **compound**: type check for a **compound** term

Example

```
constant(X) :-  
    integer(X).  
constant(X) :-  
    atom(X).
```

Example

```
:- flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).
```

Example

```
:- flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).
```

```
:- \+ listen_zusammen([[a],[b,[c,d]],e],[a,b,c,d,e]).
```

Example

```
:- flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).  
:- \+ listen_zusammen([[a],[b,[c,d]],e],[a,b,c,d,e]).  
  
flatten([X|Xs],Ys) :-  
    is_list(X), flatten(X,Ys1),  
    flatten(Xs,Ys2), append(Ys1,Ys2,Ys).  
flatten(X,[X]) :- constant(X), X  $\neq$  [].  
flatten([],[]).
```

Example

```
:- flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).
:- \+ listen_zusammen([[a],[b,[c,d]],e],[a,b,c,d,e]).

flatten([X|Xs],Ys) :-
    is_list(X), flatten(X,Ys1),
    flatten(Xs,Ys2), append(Ys1,Ys2,Ys).
flatten(X,[X]) :- constant(X), X  $\neq$  [].
flatten([],[]).
```

Example

```
flatten(Xs,Ys) :- flatten(Xs,[],Ys).
```

Example

```
:- flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).
:- \+ listen_zusammen([[a],[b,[c,d]],e],[a,b,c,d,e]).

flatten([X|Xs],Ys) :-
    is_list(X), flatten(X,Ys1),
    flatten(Xs,Ys2), append(Ys1,Ys2,Ys).
flatten(X,[X]) :- constant(X), X  $\neq$  [].
flatten([],[]).
```

Example

```
flatten(Xs,Ys) :- flatten(Xs,[],Ys).
flatten([X|Xs],Stack,Ys) :-
    is_list(X), flatten(X,[Xs|Stack],Ys).
```

Example

```
:- flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).
:- \+ listen_zusammen([[a],[b,[c,d]],e],[a,b,c,d,e]).

flatten([X|Xs],Ys) :-
    is_list(X), flatten(X,Ys1),
    flatten(Xs,Ys2), append(Ys1,Ys2,Ys).
flatten(X,[X]) :- constant(X), X  $\neq$  [].
flatten([],[]).
```

Example

```
flatten(Xs,Ys) :- flatten(Xs,[],Ys).
flatten([X|Xs],Stack,Ys) :-
    is_list(X), flatten(X,[Xs|Stack],Ys).
flatten([X|Xs],Stack,[X|Ys]) :-
    constant(X), X  $\neq$  [], flatten(Xs,Stack,Ys).
```


Example

```

:- flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).
:- \+ listen_zusammen([[a],[b,[c,d]],e],[a,b,c,d,e]).

flatten([X|Xs],Ys) :-
    is_list(X), flatten(X,Ys1),
    flatten(Xs,Ys2), append(Ys1,Ys2,Ys).
flatten(X,[X]) :- constant(X), X  $\neq$  [].
flatten([],[]).
```

Example

```

flatten(Xs,Ys) :- flatten(Xs,[],Ys).
flatten([X|Xs],Stack,Ys) :-
    is_list(X), flatten(X,[Xs|Stack],Ys).
flatten([X|Xs],Stack,[X|Ys]) :-
    constant(X), X  $\neq$  [], flatten(Xs,Stack,Ys).
flatten([], [X|Stack],Ys) :- flatten(X,Stack,Ys).
flatten([],[],[]).
```