Logik für Informatiker: PROLOG
Part 4: Accumulators, Programming Methodology & Second Order Predicates

Andreas Karwath

&

Wolfram Burgard

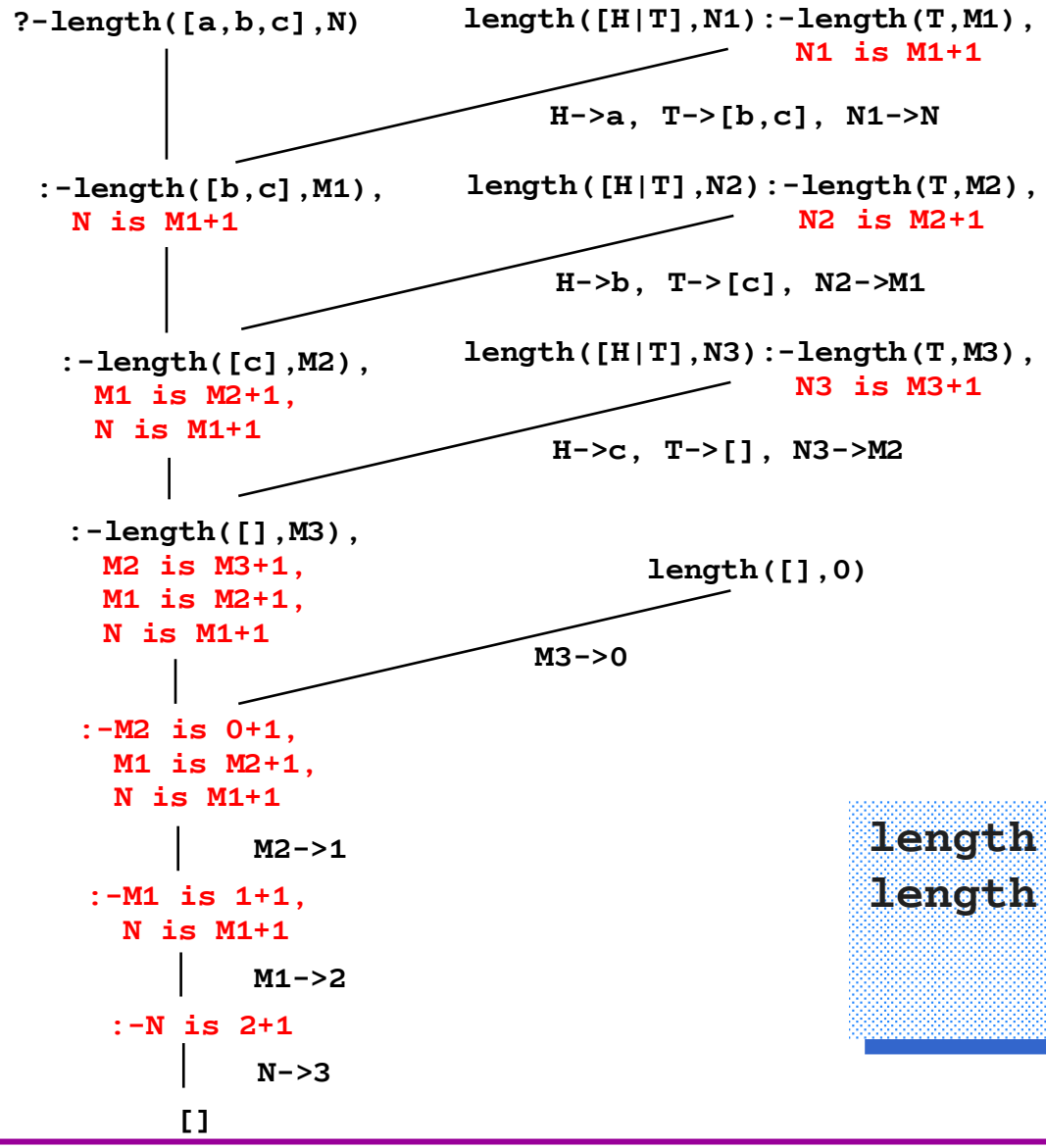
(original slides by Peter Flach)

- In Prolog, variables cannot be changed (logical variables)
- *Accumulator passing:*
  a function obtains the current value of a variable as an argument and returns the new value in a different variable
- Input argument: *accumulator*
- Output argument: *result*
- One „conventional“ variable → two Prolog arguments called the accumulator pair
- Very commonly used, very efficient
- „Recursive versions“ of predicates often waste memory
- „Accumulator versions“ only need constant *stack space*

Accumulators

```
length ([], 0).
length ([H|T], N) :-
     length(T, M),
     N is M + 1.
```


Example: Length of a list (recursive version)

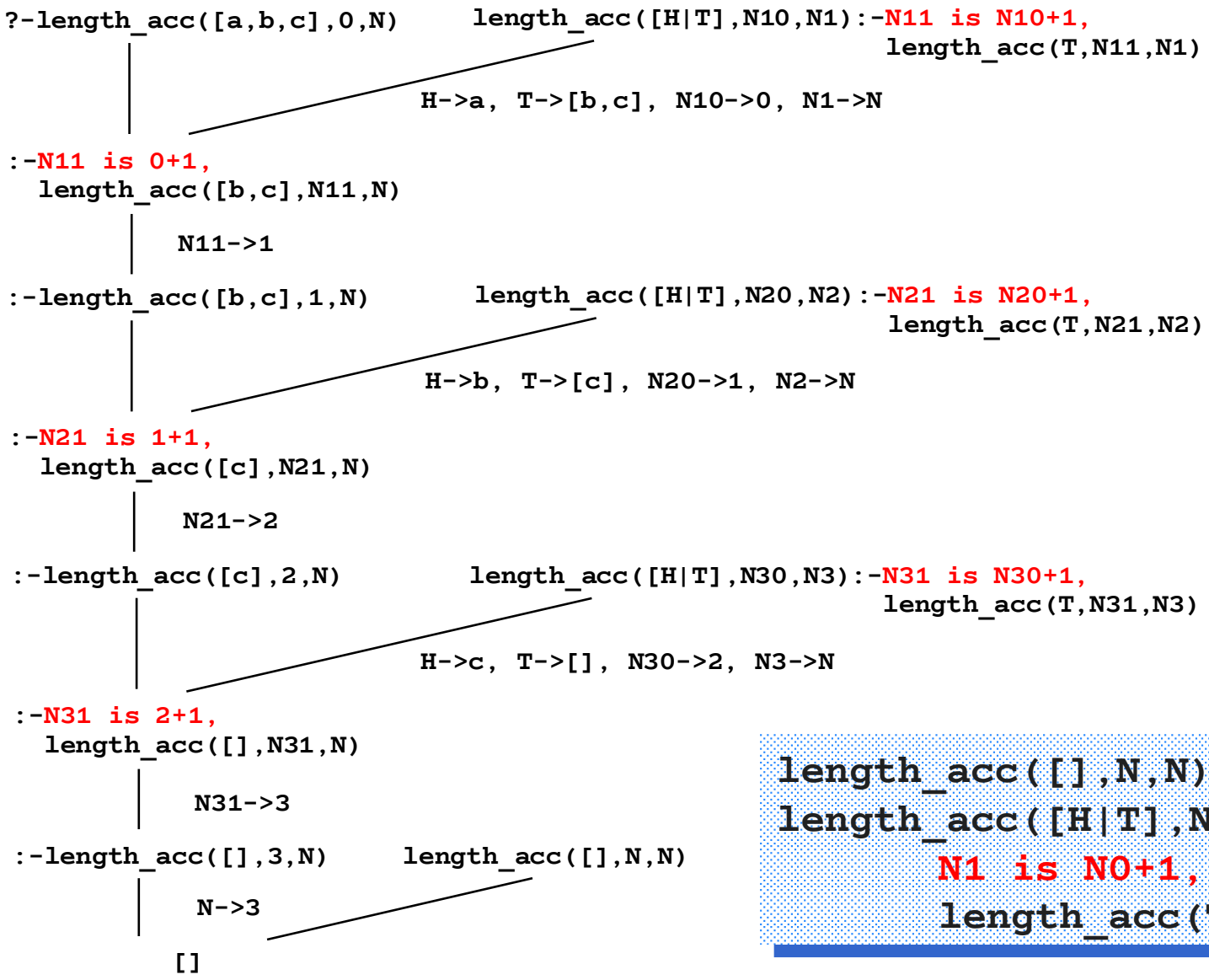



```
length([],0).
length([H|T],N):-
        length(T,M),
        N is M+1.
```


Recursive version of `length`

```
length(L, N) :- length(L, 0, N).

length ([], N, N).
length ([H|T], N0, N) :-
      N1 is N0 + 1,
      length (T, N1, N).
```


Example: Length of a list with accumulators




```
length_acc([],N,N).
length_acc([H|T],N0,N):-
        N1 is N0+1,
        length_acc(T,N1,N).
```


Accumulator version of `length`

```
reverse([], []).
reverse([H|T], R) :-
      reverse(T,R1),
      append(R1,[H],R).



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

reverse([], Y, Y).
reverse([H|T], Y0, Y) :-
      reverse(T, [H|Y0], Y).
```


Example: Reversing a list

Logic Programming Methodology

Problem: Given a number N, define a predicate `partition/4`, that divides a list of numbers into two lists, one smaller than N and the other one containing the rest.

- Write down declarative specification

```
% partition(L,N,Littles,Bigs) <- Littles contains numbers
%                                 in L smaller than N,
%                                 Bigs contains the rest
```

- Identify recursion and 'output' arguments

- Write down skeleton

```
partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
  /* do something with Head */
  partition(Tail,N,Littles,Bigs).
```

- Complete bodies

```
partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
          Head < N,
          partition(Tail,N,Littles,Bigs),
          ?Littles = [Head|Littles],?Bigs = Bigs.
partition([Head|Tail],N,?Littles,?Bigs):-
          Head >= N,
          partition(Tail,N,Littles,Bigs),
          ?Littles = Littles,?Bigs = [Head|Bigs].
```

- Fill in 'output' arguments

```
partition([],N,[],[]).
partition([Head|Tail],N,[Head|Littles],Bigs):-
          Head < N,
          partition(Tail,N,Littles,Bigs).
partition([Head|Tail],N,Littles,[Head|Bigs]):-
          Head >= N,
          partition(Tail,N,Littles,Bigs).
```

- Write down declarative specification

```
% sort(L,S) <- S is a sorted permutation of list L
```

- Write down skeleton

```
sort([],[]).
sort([Head|Tail],?Sorted):-
   /* do something with Head */
   sort(Tail,Sorted).
```

- Complete body (auxiliary predicate needed)

```
sort([],[]).
sort([Head|Tail],WholeSorted):-
   sort(Tail,Sorted),
   insert(Head,Sorted,WholeSorted).
```


Writing a sort predicate

- Write down declarative specification

```
% insert(X,In,Out) <- In is a sorted list, Out is In
%                      with X inserted in the proper place
```

- Write down skeleton

```
insert(X,[],?Inserted).
insert(X,[Head|Tail],?Inserted):-
  /* do something with Head */
  insert(X,Tail,Inserted).
```


Writing an insert predicate

- Complete bodies

```
insert(X,[],?Inserted):-?Inserted=[X].
insert(X,[Head|Tail],?Inserted):-
        X > Head,
        insert(X,Tail,Inserted),
        ?Inserted = [Head|Inserted].
insert(X,[Head|Tail],?Inserted):-
        X =< Head,
        ?Inserted = [X,Head|Tail].
```

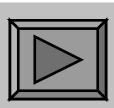
- Fill in 'output' arguments

```
insert(X,[],[X]).
insert(X,[Head|Tail],[X,Head|Tail]):-
        X =< Head.
insert(X,[Head|Tail],[Head|Inserted]):-
        X > Head,
        insert(X,Tail,Inserted).
```


Writing an insert predicate

```
sort([],[]).
sort([Head|Tail],WholeSorted):-
  sort(Tail,Sorted),
  insert(Head,Sorted,WholeSorted).


insert(X,[],[X]).
insert(X,[Head|Tail],[X,Head|Tail]):-
        X =< Head.
insert(X,[Head|Tail],[Head|Inserted]):-
        X > Head,
        insert(X,Tail,Inserted).
```


Complete sort predicate



Second-Order Predicates

```
parent(john,peter).
parent(john,paul).
parent(john,mary).
parent(mick,davy).
parent(mick,dee).
parent(mick,dozy).
```

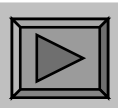
```
?-findall(C,parent(john,C),L).

  L = [peter,paul,mary]
```

```
?-findall(C,parent(P,C),L).

  L = [peter,paul,mary,davy,dee,dozy]
```

```
?-bagof(C,parent(P,C),L).

  P = john
  L = [peter,paul,mary];

  P = mick
  L = [davy,dee,dozy]
```

```
?-bagof(C,P^parent(P,C),L).

  L = [peter,paul,mary,davy,dee,dozy]
```


Second-order predicates: all solutions

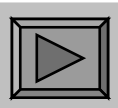


```
parent(john,peter).
parent(john,paul).
parent(john,mary).
parent(mick,davy).
parent(mick,dee).
parent(mick,dozy).
```

```
?-findall(P,parent(P, C),L).

  L = [john, john, john, mick, mick, mick]
```

```
?-setof(P,C^parent(P, C),L).

  L = [john, mick]
```


Second-order predicates: all solutions



Adding/retracting clauses from the program:

```
assert/1

retract/1
```

Have to be declared as *dynamic* before.

Example:

```
assert(student_of(peter, maria)).

assert((likes(X, Y) :- student_of(X, Y))).

retract ((likes(X, Y) :- student_of(X, Y))).
```


Second-order predicates: modifying the program

Constructing/decomposing an atom:

```
Atom =.. [Predicate,Arg1,...,ArgN].
```

Example:

```
p(X,c) =.. [p,X,c].

p(X,c) =.. [P|Args].

A =.. [p,X,c].
```


Second-order predicates:modifying an atom

- `constant(Term)`: succeeds when `Term` is a constant
- `var(Term)`: succeeds when `Term` is a variable
- `ground(Term)`: succeeds when `Term` has no variable (i.e. is a ground term)
- `arg(N,Term,Arg)`: succeeds when `Arg` is the `N`th argument of `Term`
- `functor(Term,F,N)`: succeeds when the `Term` starts with the functor `F` of arity `N`.

Further second-order predicates

Proving a goal:

```
call/1
```

Example:

```
call(likes(X, Y)).
```

Searching for clauses with a given head:

```
clause/2
```

Examples:

```
clause(likes(X,Y), Body).
```


Second-order predicates: `call` and `clause`