Chapter 5

Logic Programming with PROLOG

Algorithm = Logic + Control

Fields of application:

- AI
- Computational linguistics

Literature: Bratko; Clocksin/Mellish

Syntax

PL1 / clause normal form	PROLOG	Description
$(\neg A_1 \vee \ldots \vee \neg A_m \vee B)$	`B :- A1, ..., Am.`	rule
$(A_1 \wedge \ldots \wedge A_m) \Rightarrow B$	`B :- A1, ..., Am.`	rule
A	`A.`	fact
$(\neg A_1 \vee \ldots \vee \neg A_m)$	`?- A1, ..., Am.`	query
$\neg(A_1 \wedge \ldots \wedge A_m)$	`?- A1, ..., Am.`	query

PROLOG systems and implementations

- GNU-PROLOG Diaz
- SWI-PROLOG

PROLOG systems interpret **Warren Abstract Machine code (WAM)**.

PROLOG source code is compiled into so-called WAM code, which is then interpreted by the WAM.

Performance: up to 10 million logical inferences per second (LIPS) on a 1 Gigahertz PC

Simple examples

```
1   child(oscar,karen,frank).
2   child(mary,karen,frank).
3   child(eve,anne,oscar).
4   child(henry,anne,oscar).
5   child(isolde,anne,oscar).
6   child(clyde,mary,oscarb).
7
8   child(X,Z,Y) :- child(X,Y,Z).
9
10  descendant(X,Y) :- child(X,Y,Z).
11  descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

PROLOG program with family relationships.

PROLOG interpreter

Loading and compiling:

```
?- [rel].


?- child(eve,oscar,anne).

Yes


?- descendant(X,Y).

X = oscar
Y = karen

Yes
```

```
?- descendant(clyde,Y).

Y = mary

Yes
```

```
?- descendant(clyde,karen).
```

is not answered.

Solution:

```
1   descendant(X,Y) :- child(X,Y,Z).
2   descendant(X,Y) :- child(X,Z,Y).
3   descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

But now the query `?- child(eve,oscar,anne).` is no longer correctly answered!

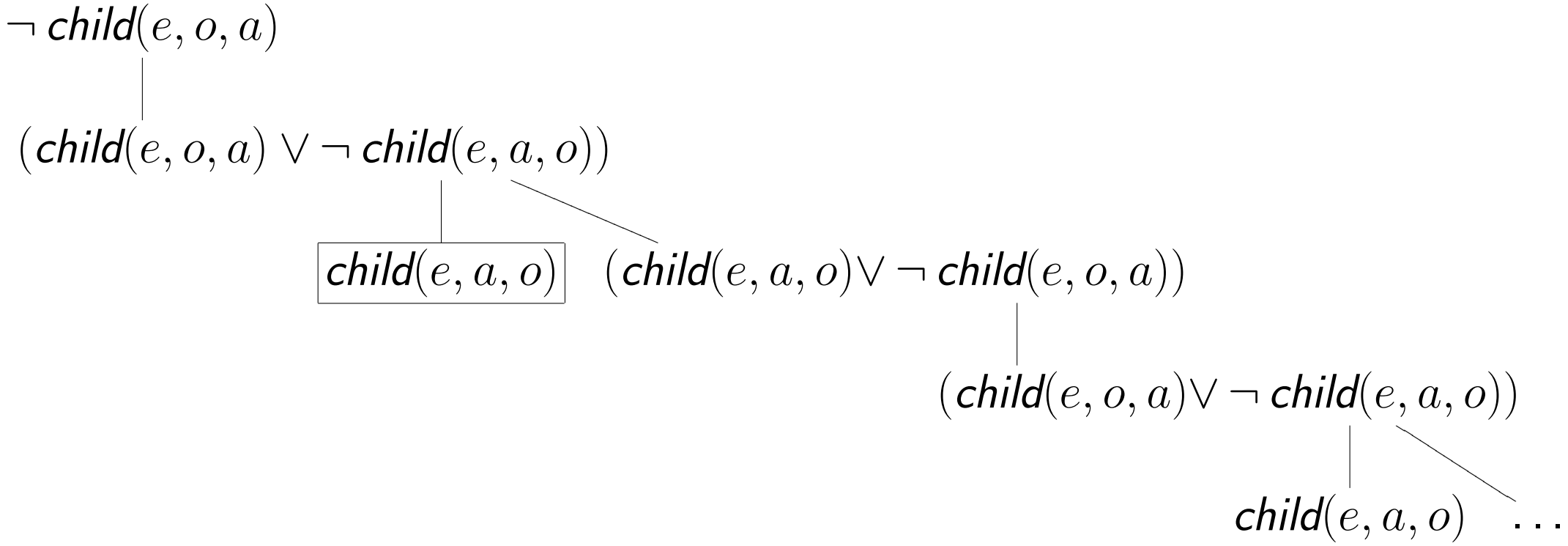
Solution:

```
1   child_fact(oscar,karen,franz).
2   child_fact(mary,karen,franz).
3   child_fact(eva,anne,oscar).
4   child_fact(henry,anne,oscar).
5   child_fact(isolde,anne,oscar).
6   child_fact(clyde,mary,oscarb).
7
8   child(X,Z,Y) :- child_fact(X,Y,Z).
9   child(X,Z,Y) :- child_fact(X,Z,Y).
10
11  descendant(X,Y) :- child(X,Y,Z).
12  descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

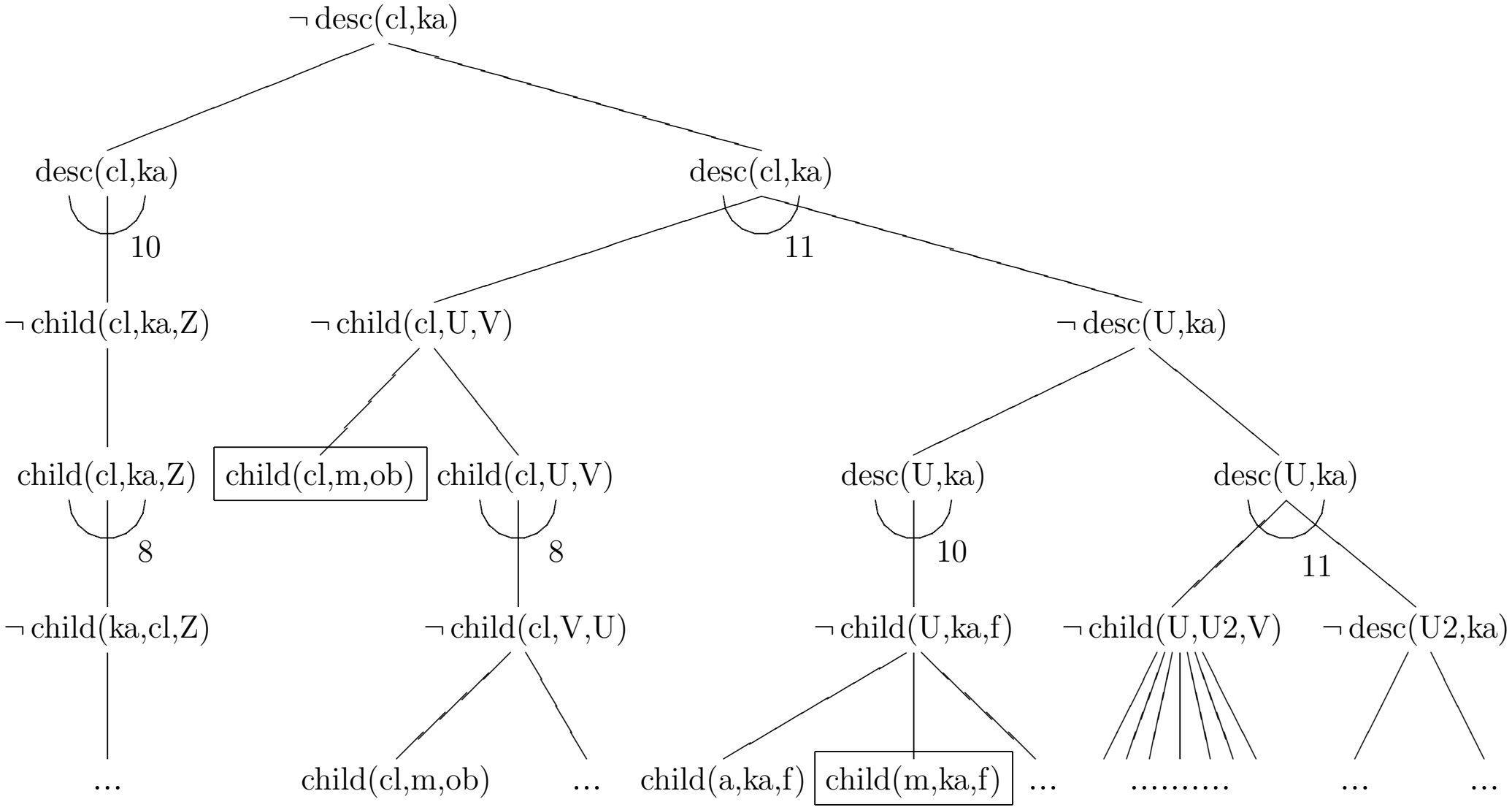
Semantics of PROLOG programmes

- **Declarative semantics:** logical interpretation of the Horn clauses
- **Procedural semantics:** processing of the PROLOG program

Search tree for `child(eve,oscar,anne)`



And-or tree for descendant(clyde,karen)



Execution Control and Procedural Elements

Avoiding unnecessary backtracking by Cut

```
1   max(X,Y,X) :- X >= Y.
2   max(X,Y,Y) :- X < Y.
```

Query:

```
?- max(2,3,Z), Z > 10.
```

Backtracking because of `Z = 3`

With Cut:

```
1   max(X,Y,X) :- X >= Y, !.
2   max(X,Y,Y).
```

The built-in predicate `fail`

Print out all children and their parents:

```
?- child_fact(X,Y,Z), write(X), write(' is a child of '),
write(Y), write(' und '), write(Z), write('.'), nl, fail.
```

Output:

```
oscar is a child of karen and frank.
mary is a child of karen and frank.
eve is a child of anne and oscar.
...

No.
```

What would be the output in the end without use of `fail`?

Negation as Failure

Query: `?- child_fact(ulla,X,Y).`

Answer: `No`

Answer is logical and correct!

Why?

The prover E

would here correctly answer: “`No proof found.`”

Restriction to Horn clauses

is important for the procedural processing using SLD resolution.

Counter example:

Russel's paradox (Example 3.5) contains the non-Horn clause

$\mathit{shaves}(\mathit{barber}, X) \vee \mathit{shaves}(X, X)$.

Lists

`[A,2,2,B,3,4,5]`

`[Head|Tail]` separates the first element (Head) from the rest (Tail) of the list.

WB:

```
list([A,2,2,B,3,4,5]).
```

Dialog:

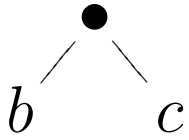
```
?- list([H|T]).

H = A
T = [2, 2, B, 3, 4, 5]

Yes
```

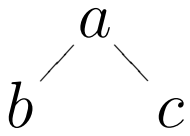
Trees as lists

`[b,c]`

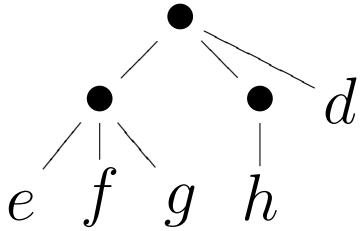


and

`[a,b,c]`

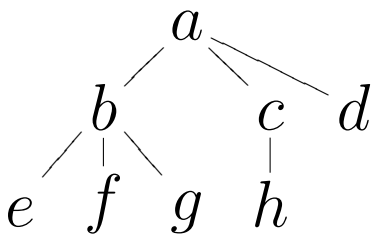


`[[e,f,g],[h],d]`



and

`[a,[b,e,f,g],[c,h],d]`



The predicate append(X,Y,Z)

```
1   append([],L,L).
2   append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

- Declarative (recursive) logical description and simultaneous processing

Examples:

```
?- append([a,b,c],[d,1,2],Z).

Z = [a, b, c, d, 1, 2]


?- append(X,[1,2,3],[4,5,6,1,2,3]).

X = [4, 5, 6]
```

Naive Reverse

```
1   nrev([],[]).
2   nrev([H|T],R) :- nrev(T,RT), append(RT,[H],R).
```

Better:

List	Accumulator
[a,b,c,d]	[]
[b,c,d]	[a]
[c,d]	[b,a]
[d]	[c,b,a]
[]	[d,c,b,a]

```
1   accrev([],A,A).
2   accrev([H|T],A,R) :- accrev(T,[H|A],R).
```

Self-modifying Programs

```
1   :- dynamic nachkomme/2.
2   nachkomme(X,Y) :- kind(X,Y,Z), asserta(nachkomme(X,Y)).
3   nachkomme(X,Y) :- kind(X,U,V), nachkomme(U,Y),
4                     asserta(nachkomme(X,Y)).
```

The query "`?- desc(clyde, karen).`" leads to the addition of:

```
desc(clyde, karen).
desc(mary, karen).
```

- Genetic programming
- Machine learning

A planning example

Example:

A farmer wants to bring a cabbage, a goat, and a wolf across a river, but his boat is so small that he can only take them across one at a time. The farmer thought it over and then said to himself: "If I first bring the wolf to the other side, then the goat will eat the cabbage. If I transport the cabbage first, then the goat will be eaten by the wolf. What should I do?"

```
start :-  action(state(left,left,left,left),
                 state(right,right,right,right)).

action(Start,Goal):-
        plan(Start,Goal,[Start],Path),
        nl,write('Solution:'),nl,
        write_path(Path).
%        write_path(Path), fail.   % all solutions output

plan(Start,Goal,Visited,Path):-
      go(Start,Next),
      safe(Next),
      \+ member(Next,Visited),       % not(member(...))
      plan(Next,Goal,[Next|Visited],Path).
plan(Goal,Goal,Path,Path).

go(state(X,X,Z,K),state(Y,Y,Z,K)):-across(X,Y). % farmer, wolf
go(state(X,W,X,K),state(Y,W,Y,K)):-across(X,Y). % farmer, goat
go(state(X,W,Z,X),state(Y,W,Z,Y)):-across(X,Y). % farmer, cabbage
go(state(X,W,Z,K),state(Y,W,Z,K)):-across(X,Y). % farmer

across(left,right).
across(right,left).

safe(state(B,W,Z,K)):- across(W,Z), across(Z,K).
safe(state(B,B,B,K)).
safe(state(B,W,B,B)).
```

Query: "?- start." Answer

```
Solution:
Farmer and goat from left to right
Farmer from right to left
Farmer and wolf from left to right
Farmer and goat from right to left
Farmer and cabbage from left to right
Farmer from right to left
Farmer and goat from left to right

Yes
```

Definition of `plan` in logic:

$$\forall z\ \mathit{plan}(z,z) \wedge \forall s\ \forall z\ \forall n\ [\mathit{go}(s,n) \wedge \mathit{safe}(n) \wedge \mathit{plan}(n,z) \Rightarrow \mathit{plan}(s,z)]$$

Situation calculus!

Constraint Logic Programming (CLP)

Problem with PROLOG: Frame axioms, see Tweety example:

`penguin(tweety)` does not exclude `raven(tweety)`!

- **CLP** = formulation of constraints for variables
- Interpreter monitors the execution of the program
- The programmer is relieved

Citation Freuder:

> "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."

Example: The secretary of Albert Einstein High School has to come up with a plan for allocating rooms for final exams. He has the following information: the four teachers Mayer, Hoover, Miller and Smith give tests for the subjects German, English, Math, and Physics in the ascendingly numbered rooms 1, 2, 3 and 4. Every teacher gives a test for exactly one subject in exactly one room. Besides that, he knows the following about the teachers and their subjects.

- Mr. Mayer never tests in room 4.
- Mr. Miller always tests German.
- Mr. Smith and Mr. Miller do not give tests in neighboring rooms.
- Mrs. Hoover tests Mathematics.
- Physics is always tested in room number 4.
- German and English are not tested in room 1.

Who gives a test in which room?

```
1  start :-
2      fd_domain([Maier, Huber, Mueller, Schmid],1,4),
3      fd_all_different([Maier, Mueller, Huber, Schmid]),
4
5      fd_domain([Deutsch, Englisch, Mathe, Physik],1,4),
6      fd_all_different([Deutsch, Englisch, Mathe, Physik]),
7
8      fd_labeling([Maier, Huber, Mueller, Schmid]),
9
10     Maier #\= 4,                      % Maier prüft nicht in Raum
11     Mueller #= Deutsch,               % Müller prüft Deutsch
12     dist(Mueller,Schmid) #>= 2,       % Abstand Müller/Schmid >= 2
13     Huber #= Mathe,                   % Huber prüft Mathematik
14     Physik #= 4,                      % Physik in Raum 4
15     Deutsch #\= 1,                    % Deutsch nicht in Raum 1
16     Englisch #\= 1,                   % Englisch nicht in Raum 1
17     nl,
18     write([Maier, Huber, Mueller, Schmid]), nl,
19     write([Deutsch, Englisch, Mathe, Physik]), nl.
```

Output:

```
[3,1,2,4]
[2,3,1,4]
```

Room plan:

Room num.	1	2	3	4
Teacher	Hoover	Miller	Mayer	Smith
Subject	Math	German	English	Physics

Finite domain constraint solver

Übung: Einstein puzzle

Summary

- Unification, lists, declarative programming
- Relationale view on procedures
- Parameters for input and output
- short programmes
- Tool for Rapid Prototyping
- CLP for optimization and planning tasks and logic puzzles
- PROLOG in Europe, LISP in USA

Literature: Bratko und Clocksin/Mellish, Handbooks: Wielemaker; Diaz, CLP: Bartak