1 Introduction

This course is about the principles of Logic Programming and its logical foundations. The course also gives an introduction to the logic programming language Prolog and discusses applications of logic programming, for example, to deductive databases and artificial intelligence.

The notes are self-contained and provide all information necessary for the course. The following books are recommended as further reading:


The first two books provide a thorough introduction into the foundations of logic programming and develop techniques for analyzing and verifying logic programs. The other books concentrate on programming techniques and applications to databases and artificial intelligence. They also contain many interesting and instructive examples.

1.1 Some introductory examples

In order to get a first idea what logic programming is, we look at some simple examples of logic programs and discuss their semantics (meaning, behavior) from an intuitive point of view.
1.1.1 Example

% Program SUMMER

\begin{verbatim}
summer. %1
warm :- summer. %2
warm :- sunny. %3
happy :- summer, warm. %4
\end{verbatim}

In this program the words \texttt{summer}, \texttt{warm}, \texttt{sunny}, \texttt{happy} should be viewed as statements. In Prolog such statements are called \texttt{atomic formulas} \footnote{In the literature atomic formulas are sometimes called ‘atoms’, which however collides with a different common use of the word ‘atom’ (cf. chapter 2.).}. For example the atomic formula \texttt{sunny} could represent the statement ‘it is sunny’, whereas \texttt{happy} might stand for ‘I am happy’.

Each line of this program is called a \texttt{clause}.

The first clause

\begin{verbatim}
summer.
\end{verbatim}

consisting of the atomic formula \texttt{summer} followed by a dot, is an example of a \texttt{unit clause} or \texttt{fact} and is to be understood as the assertion of a fact. In our case it asserts that it is summer.

The second clause

\begin{verbatim}
warm :- summer.
\end{verbatim}

asserts

\begin{quote}
‘if it is summer, then it is warm’
\end{quote}

or

\begin{quote}
‘it is warm provided it is summer’.
\end{quote}

Similarly, the third clause

\begin{verbatim}
warm :- sunny.
\end{verbatim}

asserts

In this program the words summer, warm, sunny, happy should be viewed as statements. In Prolog such statements are called \texttt{atomic formulas} \footnote{In the literature atomic formulas are sometimes called ‘atoms’, which however collides with a different common use of the word ‘atom’ (cf. chapter 2.).}. For example the atomic formula \texttt{sunny} could represent the statement ‘it is sunny’, whereas \texttt{happy} might stand for ‘I am happy’.

Each line of this program is called a \texttt{clause}.

The first clause

\begin{verbatim}
summer.
\end{verbatim}

consisting of the atomic formula \texttt{summer} followed by a dot, is an example of a \texttt{unit clause} or \texttt{fact} and is to be understood as the assertion of a fact. In our case it asserts that it is summer.

The second clause

\begin{verbatim}
warm :- summer.
\end{verbatim}

asserts

\begin{quote}
‘if it is summer, then it is warm’
\end{quote}

or

\begin{quote}
‘it is warm provided it is summer’.
\end{quote}

Similarly, the third clause

\begin{verbatim}
warm :- sunny.
\end{verbatim}

asserts
‘if it is sunny, then it is warm’.

Finally, the fourth clause

```prolog
happy :- summer, warm.
```

should be read as the assertion

‘if it is summer and warm, then I am happy’.

So, the ‘:-’ is like a reverse implication and the comma means ‘and’.

The second third and fourth clauses are called rules. The atomic formula to the left of ‘:-’ is the **head** of the rule and the atomic formulas to the right form its **body**. So, for example, the last clause has head happy and body summer, warm.

In the language of *formal logic* these clauses would be written as

```
summer
summer → warm
sunny → warm
summer ∧ warm → happy
```

We can use the program SUMMER by asking a **query** like

```
?- happy.
```

which could be read as the question ‘am I happy?’. Prolog will answer

**Yes**

The answer **Yes** means that from the program SUMMER the goal happy can be derived.

In order to understand how Prolog derives this answer it is useful to read the clauses of the program in a slightly different, more operational way. For example, we can read the clause

```prolog
happy :- summer, warm.
```

as
‘in order to derive happy it suffices to derive summer and warm’

or

‘the goal happy can be reduced to the goals summer and warm’

In general, Prolog derives a goal, or a list of goals, by reducing them using the given program clauses until no goals are left. We can visualize all attempts to derive a goal as a tree where each node contains a list of goals and has as children the goal lists that are obtained by one reduction step. Prolog always tries to reduce the leftmost goal first. A branch of the tree is successful if it ends with an empty list of goals. Otherwise it is failed.

**Exercise:** Draw the tree representing all attempts to derive the goal happy from the program SUMMER. Indicate which branches of the tree are successful and which fail.

Prolog answered Yes to the query happy because the tree above contains a successful branch (and Prolog has found it). If we however ask the query

?- sunny.

we will get the answer

No

which means that the goal sunny cannot be derived from the program SUMMER. How does the tree for sunny look like?
1.1.2 Example

%Program FLIGHTS,
%A data base for flight connections

direct(amsterdam, hongkong). %1
direct(amsterdam, cardiff). %2
direct(hongkong, sidney). %3
direct(sidney, brisbane). %4

collection(X, Y) :- direct(X, Y). %5
collection(X, Y) :- direct(X, Z), collection(Z, Y). %6

In this program the atomic formulas are more complicated than in the previous program. Take, for example,

direct(amsterdam, hongkong)

which can be read as

‘there is a direct flight from Amsterdam to Hong-Kong’

This atomic formula is built from a predicate direct and the constants amsterdam and hongkong. In general, constants denote objects and predicates denote relations between objects.

The meaning of the 5th clause in the program FLIGHTS is

‘If there is a direct flight from X to Y, then there is a connection from X to Y’

and clause 6 could be read

‘If there is a direct flight from X to Z and there is a connection from Z to Y, then there is a connection from X to Y’.

The capital letter X, Y, Z in these clauses are variables which stand for unspecified objects. They are implicitly universally quantified. Therefore e.g. clause 6 should be read more correctly

‘For all cities X, Y, Z: if there is a direct flight from X to Z and there is a connection from Z to Y, then there is a connection from X to Y’.
In the language of formal logic clauses 5 and 6 would be written as

\[
\forall X \forall Y (\text{direct}(X, Y) \rightarrow \text{connection}(X, Y)) \\
\forall X \forall Y \forall Z (\text{direct}(X, Z) \land \text{connection}(Z, Y) \rightarrow \text{connection}(X, Y))
\]

It is important to know that clause 6 is equivalent to

"For all cities X, Y; if there exists a city Z such that there is a direct flight from X to Z and there is a connection from Z to Y, then there is a connection from X to Y."

written formally

\[
\forall X \forall Y (\exists Z (\text{direct}(X, Z) \land \text{connection}(Z, Y)) \rightarrow \text{connection}(X, Y))
\]

In general, a formula of the form \( \forall Z (A(Z) \rightarrow B) \), where Z does not occur in B, is equivalent to \( \exists Z A(Z) \rightarrow B \).

Given the program FLIGHTS we may ask a query like

\(?- \text{connection}(\text{amsterdam}, \text{brisbane}).\)

which is to be read as the question

"is there a connection from Amsterdam to Brisbane?"

Prolog will return the answer

Yes

which means that Prolog has derived from the program FLIGHTS the information that there is indeed a connection from Amsterdam to Brisbane.

We may also ask the query

\(?- \text{connection}(\text{cardiff}, \text{amsterdam}).\)

to which Prolog will return the answer

No
because according to the program FLIGHTS there is no connection from Cardiff to Amsterdam (although there is a connection from Amsterdam to Cardiff). Note that there is no rule in the program expressing that whenever there is connection from X to Y, then there is also a connection from X to Y (so, we are considering one-way flights).

In order to understand how Prolog finds the answers to these queries we can draw trees similar to the ones for the program FLIGHTS. The main difference is that we now need to substitute, that is, replace, certain variables by suitable constants.

**Exercise:** Draw the tree for the goal `connection(amsterdam, brisbane)`. Indicate which substitutions took place at each step. Stop when a successful branch has been found.

---

**Exercise:** Draw the tree for the goal `connection(cardiff, amsterdam)`. Why does Prolog answer No?
Queries can not only be seen as questions to be answered Yes or No, they can also be used to compute results. For example, the query

?- connection(hongkong, X).

is to be read

‘where can one fly to from Hong-Kong?’

Prolog’s answer will be

X = sidney;
X = brisbane;
No

This means that from Hong-Kong one can fly to Sidney and Brisbane, but to no further place. The ‘;’ has to be typed by the user if they want to get a further answer.

Exercise: Confirm Prolog’s answers by drawing the full tree for the goal connection(hongkong, X).

A query may consist of more than one goal. For example, one could ask

?- connection(X, cardiff), connection(X, sidney).

which can be read
‘from where can one reach Cardiff and Sidney?’

Prolog’s answer is

\[
X = \text{amsterdam}; \\
\text{No}
\]

One could again draw a tree for this query. However for this and similar queries (see the next example) the trees quickly become rather big and tedious to draw. Later in the course we will study Prolog’s search method more systematically and in full generality.

Note that in the 6th clause the predicate \textit{connection} appears in the head and in the body, which means that \textit{connection} is defined \textbf{recursively}. Recursive definitions occur frequently in logic programs. They replace for- or while-loops.

\subsection*{1.1.3 Example}

\begin{verbatim}
\%Program MEMBER, \\
\%member(X,L) :<=> X is a member of the list L \\
member(X, [X|L]). %1 \\
member(X, [Y|L]) :- member(X,L). %2
\end{verbatim}

Here we used Prolog’s notation for constructing lists: \([X|L]\) consists of the list \(L\) with \(X\) added in front. For example, if \(L = [a,b,c]\), then \([d|L] = [d,a,b,c]\).

Note that in some Prolog implementations \textit{member} is a predefined predicate, in which case a different name for it should be chosen if one wants to carry out this example as an exercise.

The meaning of the two clauses of the program \textbf{MEMBER} is the following:

1. The first element of a list is a member of that list.
2. If \(X\) is a member of the list \(L\), then \(X\) is also a member of the list \([Y|L]\).

Again, this is a recursive definition.

We may test this program by asking the query

\[
?- \text{member(b, [a,b,c])}.
\]
Prolog answers

Yes

To the query

?- member(d,[a,b,c]).

Prolog answers

no

We can also write a query such that Prolog successively computes the common elements of the lists [a, b, c] and [b, d, c, e]:

?- member(X,[a,b,c]), member(X,[b,d,c,e]).

Prologs answers will be

X = b;
X = c;
No

Exercise. Write a Pascal program doing the same job.

Let us add to the program MEMBER a program for appending (or concatenating) lists. Let us write \( L@K \) for the concatenation of the lists \( L \) and \( K \) (this is not Prolog notation). So, for example, \([a, b]@[c, a, d] = [a, b, c, a, d] \). Concatenation can be defined recursively by (again the following is not Prolog):

1. \( []@L = L \) (\( [] \) denotes the empty list)
2. \( [X|K]@L = [X|K@L] \)

In order to obtain from this definition a Prolog program we define the relation

\[
\text{append}(K, L, M) \iff K@L = M
\]

Now the recursive definition of \( K@L \) may be recast in terms of the relation append as follows:

1. append([], L, L).
2. If \( \text{append}(K, L, M) \), then \( \text{append}([X|K], L, [X|M]) \).

which directly translates into the Prolog program

\[
\begin{align*}
\text{Program APPEND} & \quad \text{append}(K, L, M) \iff M \text{ is the result of concatenating } K \text{ and } L \\
\text{append}([], L, L). \\
\text{append}([X|K], L, [X|M]) & : \text{append}(K, L, M).
\end{align*}
\]

We can use this program to compute the concatenation of two lists:

?– \text{append}([a,b], [b,c,d], M).

\[
M = [a,b,b,c,d];
\]

no

but also to split a list in all possible ways into two parts:

?– \text{append}(K, L, [a,b,c]).

\[
\begin{align*}
K & = [] \\
L & = [a,b,c];
\end{align*}
\]

\[
\begin{align*}
K & = [a] \\
L & = [b,c];
\end{align*}
\]

\[
\begin{align*}
K & = [a,b] \\
L & = [c];
\end{align*}
\]

\[
\begin{align*}
K & = [a,b,c] \\
L & = [];\end{align*}
\]

no

Now we will use the programs MEMBER and APPEND to solve a typical (albeit simplified) problem in image recognition: Let us call a list \( L \) concave if it has two occurrences of the number 1 such that in between the number 0 occurs. In other words, \( L \) is of the form

\[
[\ldots, 1, \ldots, 0, \ldots, 1, \ldots]
\]
We wish to define a predicate \texttt{concave} such that \texttt{concave(L)} holds if and only if the list \texttt{L} is concave. Another way to describe the predicate \texttt{concave} is to say that there is an occurrence of 0 such that to the left and right of this occurrence the number 1 occurs (the parts denoted by \ldots are arbitrary, in particular they may be empty or contain 0 or 1 or other elements). In other words \texttt{L} can be split into three parts: a left part containing the number 1, a middle part consisting of the number 0, and a right part containing the number 1. This means that \texttt{L} is of the form

\[K@[0|M]\]

where \texttt{K} and \texttt{M} both contain the number 1. This immediately leads us to the clause

\texttt{concave(L) :- append(K,[0|M],L), member(1,K), member(1,M).}

\textbf{Exercise.} Write an alternative nonrecursive program for the predicate \texttt{member} using the predicate \texttt{append}.

\subsection{1.2 Procedural vs. Declarative Programming}

Let us summaries the main points we learned from these introductory examples:

\begin{itemize}
  \item A \textit{logic program} is a \textit{declaration}, i.e. a list of facts and rules describing a situation, database, world, problem, e.t.c.
  \item Queries are questions to the program to be entered by the user.
  \item Answers are \textbf{Yes} or \textbf{No}, or of the form \texttt{X = t}, where \texttt{t} is a \textit{term}, i.e. a syntactic expression denoting an object (a precise definition of terms will be given later).
  \item Answers are computed by the system automatically.
\end{itemize}

Logic programming is a typical example of \textit{declarative programming} which differs radically from the more traditional \textit{procedural programming style}:

\begin{itemize}
  \item In \textbf{procedural programming} (Basic, Pascal, C, \ldots) a program consists of \textbf{instructions} to the computer of \textbf{how} to solve a problem; the program does not explicitly express which problem it is supposed to solve.
  \item In \textbf{declarative programming} a program is a \textbf{description} of \textbf{what} the problem is; the program does not explicitly say how the problem is to be solved.
\end{itemize}
1.3 History of Logic Programming

1965 Alan Robinson – resolution principle.
1973 Alain Colmerauer and others – implementation of the first Prolog system.
1986+ International conferences on Logic Programming.

1.4 Aims and topics of the course

The aims of this course are

- to understand the basic principles of logic programming,
- to gain good skills in programming with Prolog,
- to be aware of the main fields of application of logic programming.

The following topics will be covered in the course:

- The proof techniques of logic programming: unification and resolution.
- Operative and declarative semantics of logic programs.
- Soundness and completeness of resolution.
- Search strategies.
- Control features of Prolog: backtracking and the cut.
- Data structures: terms, lists, trees and their manipulation in Prolog.
- Applications of Prolog including Deductive Databases and Artificial Intelligence.