

Logic programming with Prolog

Dr. C. Constantinides

Department of Computer Science and Software Engineering
Concordia University Montreal, Canada

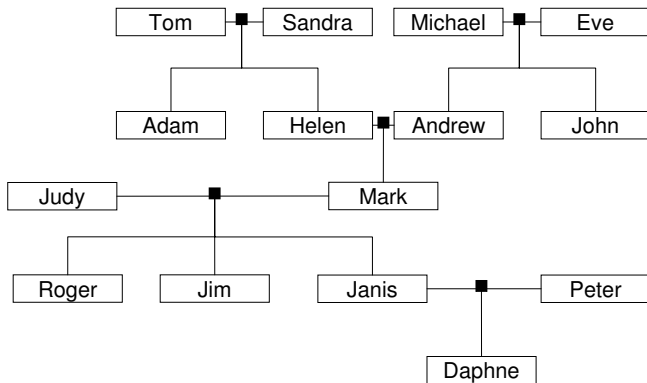
August 14, 2013

Clauses (ch. 2)

Clauses

- ▶ Predicate logic can be used to represent and reason about knowledge.
- ▶ We will adopt the Prolog programming language to model and process clauses.
- ▶ In this discussion we will use a running example to express the meaning and constraints of data as well as to construct queries over their representation in order to obtain information.

An example genealogy tree



Facts

- ▶ A Prolog program consists of *assertions* (*clauses*).
- ▶ These are divided into *facts* and *rules*. Facts are propositions which are assumed to be true. (We discuss rules later.)

Procedures

- ▶ A *procedure* consists of one or more clauses where each clause defines a certain relation between its arguments.
- ▶ A Prolog program consists of a collection of *procedures*. For example, `parent(tom, adam)` defines procedure `parent` specifying a relationship between its two arguments.

Procedure parent

```
parent(tom, adam).  
parent(tom, helen).  
parent(sandra, adam).  
parent(sandra, helen).  
parent(michael, andrew).  
parent(michael, john).  
parent(eve, andrew).  
parent(eve, john).  
parent(helen, mark).  
parent(andrew, mark).  
parent(judy, roger).  
parent(judy, jim).  
parent(judy, janis).  
parent(mark, roger).  
parent(mark, jim).  
parent(mark, janis).  
parent(janis, daphne).  
parent(peter, daphne).
```

- ▶ Consider the example family genealogy tree. The clause `parent(peter, daphne) :- true.` can be simplified to `parent(peter, daphne).` and can read as “Peter is a parent of Daphne.”
- ▶ The proposition can be regarded as an instance of the binary predicate *parent*(*X*, *Y*) and is obtained by substituting *Peter* for *X* and *Daphne* for *Y*.

Queries

- ▶ Given the fact `parent(peter, daphne)`, we can ask “Is Peter a parent of Daphne?”
- ▶ Questions are codified into *queries*.
- ▶ We can build a Prolog query to represent this question as follows:

```
?- parent(peter, daphne).
```
- ▶ Prolog will respond Yes implying that it has been successful in obtaining a fact which satisfies the query.

Genealogy database: facts

```
man(tom).  
man(michael).  
man(adam).  
man(andrew).  
man(john).  
man(mark).  
man(roger).  
man(jim).  
man(peter).  
woman(sandra).  
woman(eve).  
woman(helen).  
woman(judy).  
woman(janis).  
woman(daphne).
```

Genealogy database: facts /cont.

```
parent(tom, adam).  
parent(tom, helen).  
parent(sandra, adam).  
parent(sandra, helen).  
parent(michael, andrew).  
parent(michael, john).  
parent(eve, andrew).  
parent(eve, john).  
parent(helen, mark).  
parent(andrew, mark).  
parent(judy, roger).  
parent(judy, jim).  
parent(judy, janis).  
parent(mark, roger).  
parent(mark, jim).  
parent(mark, janis).  
parent(janis, daphne).  
parent(peter, daphne).
```

- ▶ Variables can be used in queries (must always start with a capital letter) to find all values which can be substituted in order to make the clause true.
- ▶ A different query can be formed: “Who is a parent of Daphne?”

`?- parent(X, daphne).`

`X = janis`

- ▶ This is a correct answer but we know that it is not complete, since Daphne has two parents.

- ▶ Prolog allows an interaction during a query. We can now ask “Are there more matches?”
- ▶ With the semicolon symbol (;) we instruct the Prolog system to continue its search.

```
?- parent(X, daphne).
```

```
X = janis ;
```

```
X = peter
```

“Are there still more matches?”

```
?- parent(X, daphne).
```

```
X = janis ;
```

```
X = peter ;
```

```
No
```

- ▶ In a similar fashion to the semicolon symbol during an interaction with the interpreter, a period symbol (.) indicates our intention to stop the search.

Rules

- ▶ A rule is a clause described in the general form

$$\textit{head} : - \textit{body}$$

which reads “*The head (of the rule) is true, if the body is true.*”, or alternatively “*The head of the rule can succeed if the body of the rule can succeed.*”

- ▶ The body consists of predicates, which are called the *goals* of the rule. The predicates in the body of a rule can be combined by conjunction (logical *and*, denoted by comma), disjunction (logical *or*, denoted by semicolon), or combinations of them.

Rules /cont.

- ▶ In $H :- P_1, P_2, \dots, P_n$. in order to prove (or show) H , we need to prove (or show) P_1 , and P_2 , and ..., and P_n .

Extending our database with rules

- ▶ Let us extend the database with a new relation: Suppose we let p stand for the *isParentOf* relation and let g stand for the *isGrandParentOf* relation.
- ▶ Then we can define g in terms of p by the following formula we will call G :

$$G = \forall x \forall y \forall z ((p(x, z) \wedge p(z, y)) \rightarrow g(x, y))$$



$$G = \forall x \forall y \forall z ((p(x, z) \wedge p(z, y)) \rightarrow g(x, y))$$

- ▶ In other words, if x is a parent of z and z is a parent of y , then we conclude that x is a grandparent of y . We can represent this in Prolog with the rule below.
- ▶ We use variables to express the feature that a grandparent is a parent whose child is itself a parent.
- ▶ The rule below is a compound proposition comprised by two goals.

`grandparent(X, Y) :- parent(X, Z), parent(Z, Y).`

- ▶ We can now pose the following question: “Is Judy a grandparent of Daphne?” The question can be codified into the following query:

```
?- grandparent(judy, daphne).
```

Yes

- ▶ Consider the question: “Is Roger a grandparent of Daphne?” The query is as follows:

```
?- grandparent(roger, daphne).
```

No

- ▶ Consider the question: “Who are the grandparents of Daphne?” The query is as follows:

```
?- grandparent(X, daphne).
```

```
X = judy ;
```

```
X = mark ;
```

```
No
```

- ▶ Consider the question: “Who is Helen a grandparent of?” The query is as follows:

```
?- grandparent(helen, X).
```

```
X = roger ;
```

```
X = jim ;
```

```
X = janis ;
```

```
No
```

- ▶ We can now further extend the database: Suppose we let p stand for the *isParentOf* relation and let a stand for the *isAncestorOf* relation. Then we can define a in terms of p by the following formula we will call A :

$$A = \forall x \forall y (p(x, y) \rightarrow a(x, y))$$

$$A = \forall x \forall y \forall z ((p(x, z) \text{ and } a(z, y)) \rightarrow a(x, y))$$

- ▶ In other words, x is an ancestor of y if either x is a parent of y , or x is a parent of an ancestor of y . We can represent this in Prolog with the rules below:

```
ancestor(X, Y) :- parent(X, Y).
```

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

- ▶ Consider the question: “Is Tom an ancestor of Daphne?” The query is as follows:

```
?- ancestor(tom, daphne).
```

Yes

- ▶ Consider the question: “Is Tom an ancestor of Peter?” The query is as follows:

```
?- ancestor(tom, peter).
```

No

- Consider the question: “Who are the ancestors of Janis?”
The query is as follows:

```
?- ancestor(X, janis).
```

```
X = judy ;
```

```
X = mark ;
```

```
X = tom ;
```

```
X = sandra ;
```

```
X = michael ;
```

```
X = eve ;
```

```
X = helen ;
```

```
X = andrew ;
```

```
No
```

- ▶ Consider the question: “Who are the ancestors of Peter?”
The query is as follows:

`?- ancestor(X, peter).`

No

- ▶ Note that Prolog finds no ancestors not because Peter has no ancestors (all humans have ancestors), but because no such facts exist in our database which define any.

- Consider the question: “Who is Eve an ancestor of?” The query is as follows:

```
?- ancestor(eve, X).
```

```
X = andrew ;
```

```
X = john ;
```

```
X = mark ;
```

```
X = roger ;
```

```
X = jim ;
```

```
X = janis ;
```

```
X = daphne ;
```

```
No
```


- ▶ Suppose we let a stand for the *isAncestorOf* relation and let d stand for the *isDescendantOf* relation.
- ▶ Then we can define d in terms of a by the following formula we will call D :

$$D = \forall x \forall y (a(x, y) \rightarrow d(y, x))$$

- ▶ In other words, if x is an ancestor of y then we can conclude that y is a descendant of x . We can represent this in Prolog with the rule below:

`descendant(X, Y) :- ancestor(Y, X).`

- ▶ Consider the question: “Is Jim a descendant of Michael?”
The query is as follows:

```
?- descendant(jim, michael).
```

Yes

- ▶ Consider the question: “Is Peter a descendant of Michael?”
The query is as follows:

```
?- descendant(peter, michael).
```

No

- ▶ We can further extend the database by adding more rules. Suppose we let m stand for the *isMan* relation, p stand for *isParentOf* relation and let f stand for the *isFatherOf* relation.
- ▶ Then we can define f in terms of m and p by the following formula we will call F :

$$F = \forall x \forall y ((m(x) \wedge p(x, y)) \rightarrow f(x, y))$$

- ▶ In other words, if x is a man and x is a parent of y , then we conclude that x is the father of y .



$$F = \forall x \forall y ((m(x) \wedge p(x, y)) \rightarrow f(x, y))$$

- ▶ We can represent this in Prolog with the first rule below.
- ▶ We use variables to express the feature that every man which is a parent of any child is also his/her father. A similar reasoning can be applied to build a rule for the *isMotherOf* relation:

```
father(X, Y) :- man(X), parent(X, Y).  
mother(X, Y) :- woman(X), parent(X, Y).
```

- ▶ We can now pose even more different types of questions in our system, such as: “Who is the father of Helen?” The query is as follows:

```
?- father(X, helen).  
X = tom
```

- ▶ We can hit ; to instruct Prolog to continue its search for more possible matches:

```
?- father(X, helen).  
X = tom ;  
No
```

- ▶ There is no other match, as was expected.

- Consider the question: “Who is Sandra the mother of?” The query is as follows:

```
?- mother(sandra, X).
```

```
X = adam ;
```

```
X = helen ;
```

```
No
```

- ▶ Suppose we let m stand for the *isMan* relation, let p stand for *isParentOf* relation and let s stand for the *isSonOf* relation.
- ▶ Then we can define s in terms of m and p by the following formula we will call S :

$$S = \forall x \forall y ((m(x) \wedge p(y, x)) \rightarrow s(x, y))$$

- ▶ In other words, if x is a man and y is a parent of x , then we conclude that x is the son of y .
- ▶ We can represent this in Prolog with the rules below.
- ▶ We use variables to express the feature that every man who has a parent, is also his/her parent's son. A similar reasoning can applied to build a rule for the *isDaughterOf* relation:

```
son(X, Y)      :- man(X), parent(Y, X).
daughter(X, Y) :- woman(X), parent(Y, X).
```

- ▶ Consider the question: “Is Adam the son of Tom?” The query is as follows:

```
?- son(adam, tom).
```

Yes

- ▶ Consider the question: “Who is Adam the son of?” The query is as follows:

```
?- son(adam, X).
```

```
X = tom ;
```

```
X = sandra ;
```

No

Anonymous variables

- ▶ If any parameter of a relation is not important, we can replace it with an *anonymous variable* (denoted by the underscore character `_`) as follows:

```
is_father(X) :- father(X, _).
```

```
is_mother(X) :- mother(X, _).
```

- ▶ We can now pose more queries such as “Is Tom a father?” To answer this type of question, it does not matter who Tom is the father of, as long as Tom is found as the first term in a `father(X, _)` fact. The query is as follows:

```
?- is_father(tom).
```

Yes

Ground queries

- ▶ Ground queries consist only of value identifiers as parameters to the predicate (e.g. `parent(peter, daphne)`).
- ▶ The answer to a ground query is of the form Yes/No.
- ▶ The answer “Yes” means that the system has proved that the goal was true under the given database of facts and relationships (rules).
- ▶ The answer “No” means that either the goal was proved false or the system was unable to prove it.

Non-ground queries

- ▶ Non-ground queries contain variables as parameters (e.g. `parent(X, daphne)`).
- ▶ A non-ground query is satisfiable relative to the program if there is a substitution for its variable which makes the query true.

The inferencing process

- ▶ To prove that a goal is true, the inferencing process must find a chain of *inference rules* and/or *inference facts* in the database that connect the goal to one or more facts in the database.
- ▶ Given a goal Q , then either Q must be found as a fact in the database or the inferencing process must find a fact P_1 and a sequence of propositions $P_2, \dots P_n$ such that:

$$P_2 : -P_1$$

$$P_3 : -P_2$$

...

$$Q : -P_n$$

- ▶ The process of proving a goal is called *matching*.

Unification and resolution

- ▶ The mechanisms of *unification* and *resolution* are vital to query evaluation:
- ▶ **Unification** The process of taking two atoms (one from the query and the other being a fact or the head of a rule) and determining if there is a substitution which makes them the same.
- ▶ **Resolution** When an atom from the query has been unified with the head of a rule (or a fact), resolution replaces the atom with the body of the rule (or nothing, if a fact) and then applies the substitution to the new query.

Unification and resolution /cont.

- ▶ Given a query, Prolog searches the database of clauses from top to bottom:
- ▶ If it finds a fact, it tries to unify the query with the fact. If successful, one solution has been found. If not successful, it tries the next clause in the program.
- ▶ If it finds a rule, it tries to unify the query with the head of the rule. If successful, the goals of the body of the clause are treated as those queries which must be satisfied in order for the initial query to be satisfied. If not successful, it tries the next clause in the program.

Example: Query evaluation

- ▶ Consider the evaluation of the query `grandparent(judy, daphne)`.
- ▶ Prolog will search the database from top to bottom, trying to unify the query with one of the clauses of the database. It will unify the query with the head of the rule:

```
grandparent(X, Y) :- parent(X, Z),  
                      parent(Z, Y).
```

instantiating `X` to `judy` and `Y` to `daphne`, and apply the substitution as follows:

```
grandparent(judy, daphne) :- parent(judy, Z),  
                              parent(Z, daphne).
```

Example: Query evaluation /cont.

- ▶ For the head of the new query to be true, both goals of the body of the clause must be evaluated to true.
- ▶ To evaluate the two goals, Prolog will consider the two new queries

`parent(judy, Z), parent(Z, daphne).`

and it will perform a new search of the database to unify each one, looking for an instantiation which can satisfy them both.

- ▶ Variable Z can be instantiated to `janis` thus making the original query true.

Qualifiers

- ▶ What if we now wanted to pose a different type of question: “Are all men parents?” We can do this with a *qualifier*:
`qualify(X) :- forall(man(X), parent(X, _)).`
- ▶ The body of the rule will be true only if *each* instantiation of `man(X)` appears as a first term in a `parent(X, _)` clause.
`?- qualify(X).`
No

Arithmetic operators

- ▶ We can evaluate the truth value of an arithmetic expression. The operators `+`, `-`, `*` and `/` denote their respective *arithmetic operations* and `mod` denotes the remainder operation.

```
?- (7 is 6 + 1).
```

```
Yes
```

- ▶ The keyword `is` is a built-in arithmetic evaluator. It takes an arithmetic expression as its right-hand side (RHS) operand and a variable as its left-hand side (LHS) operand.
- ▶ All variables on the RHS must already be instantiated.

Arithmetic operators /cont.

- ▶ Alternatively we can query under what conditions a given expression can be evaluated:

```
?- (X is 6 + 1).
```

```
X = 7 ;
```

```
No
```

```
?- (X is 7 mod 2).
```

```
X = 1 ;
```

```
No
```

Arithmetic operators /cont.

- ▶ We can use arithmetic operators in the definition of rules, for example:

```
double(X, Y) :- Y is X * 2.
```

```
?- double(2, 4).
```

Yes

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

```
X = 6 ;
```

No

Arithmetic operators /cont.

- In general, a function that takes n arguments will be represented in Prolog as a relation that takes $n + 1$ arguments, the last one being used to hold the result, as shown in the example above.

```
?- double(X, 16).
```

```
ERROR
```

Relational and logical operators

- ▶ We can use the Prolog interpreter to evaluate *relational operators*:

?- 1 < 3.

Yes

?- (1 < 3).

Yes

- ▶ Other relational operators are <=, >, >=, and ==.
- ▶ We can also have *logical operators*. The comma operator (,) denotes a logical AND:

?- (1 < 3), (4 < 2).

No

whereas the semicolon operator (;) denotes a logical OR:

?- (1 < 3); (4 < 2).

Yes

Example: Relational operators

- ▶ Consider function `max` to return the maximum between two numbers:

`max(X, Y, X) :- X > Y.`

- ▶ The rule reads “The maximum of `X` and `Y` is `X`, provided that the body of the rule can be proved true.”

`?- max(9, 5, X).`

`X = 9 ;`

No

Lists I (ch. 3)

Lists I

- ▶ A list is a finite ordered sequence of zero or more elements that can be repeated.
- ▶ We can only access two things in a list: the first element of the list (*head*) and the list made up of all except the head, called the *tail* of the list.
- ▶ The number of elements in a list is called the *length* of the list.
- ▶ For example, the list $L = \langle a, b, c, d \rangle$ has length 4, its head is a and its tail is $\langle b, c, d \rangle$.
- ▶ We will use the notation $head(L)$ and $tail(L)$ to denote the head of L and the tail of L .
- ▶ The empty list, denoted by $\langle \rangle$, does not have a head or tail.

Lists I /cont.

- ▶ Note that $a \neq \langle a \rangle \neq \langle \langle a \rangle \rangle$.
- ▶ The elements of a list can be any kind of objects, including lists themselves in which case a list is said to be *nested* (as opposed to being *flat*).
- ▶ Consider the following examples:

L	$head(L)$	$tail(L)$
$\langle a, \langle b \rangle \rangle$	a	$\langle \langle b \rangle \rangle$
$\langle \langle a \rangle, \langle b, c \rangle \rangle$	$\langle a \rangle$	$\langle \langle b, c \rangle \rangle$
$\langle a \rangle$	a	$\langle \rangle$

Clauses and lists

- ▶ Syntactically, a Prolog list is represented by square brackets [...].
- ▶ The empty list is represented as [].
- ▶ Every non-empty list can be represented in two parts: head and tail.
- ▶ Consider the list $L = [a, b, c, d, e]$: The notation $[H|T]$ is used to represent a list whose head is H and its tail is T .

Clauses and lists /cont.

- The list L can be represented as

$$\begin{aligned} L &= [a, b, c, d, e] \\ &= [a \mid [b, c, d, e]] \\ &= [a \mid [b \mid [c, d, e]]] \\ &= [a \mid [b \mid [c \mid [d, e]]]] \\ &= [a \mid [b \mid [c \mid [d \mid [e]]]]] \\ &= [a \mid [b \mid [c \mid [d \mid [e \mid []]]]]] \end{aligned}$$

Example 1

- ▶ In this example we want to define a clause `first` which succeeds if an element is the head of a list. The rule below, `first(F, [F|_])`, reads “Clause `first` succeeds if an element `F` is found to be the first element of a given list, represented as `[F|_]`, since we are not really interested in the contents of the tail.”

Example 1 /cont.

- ▶ The query below reads “Is element a the head of the list [a b c]?” to which Prolog responds with a Yes.

```
?- first(a, [a, b, c]).
```

Yes

Example 1 /cont.

- ▶ Let us now rephrase the question. We ask “Under what conditions an element is the head of the list [a b c]?” The condition is that an element must be equal to a. Let us translate the question in Prolog and see its response:

```
?- first(F, [a, b, c]).
```

$F = a$

which means that the condition under which the statement can be true is when $F = a$.

Example 2

- In this example, we want to define a clause c which succeeds if a list can be broken down into a head and a tail. The rule below,

$c([H|T], H, T).$

reads “Clause c succeeds if a list, represented as $[H|T]$, can be broken down into a head H and a tail T .”

Example 2 /cont.

- ▶ The query below reads “Given the list [a b c d], is a the head and [b c d] the tail?” to which Prolog responds with a Yes. (What type of question is this?)

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

Yes

Example 2 /cont.

- ▶ The query below reads “Given the list [a b c d], what are the head and tail, if any?” (What type of question is this?)
`?- c([a, b, c, d], H, T).`

`H = a`

`T = [b, c, d]`

to which Prolog will respond by providing the conditions under which the statement can be true.

Example 2 /cont.

- ▶ The query below reads “Given the list [], what are the head and tail, if any?” (What type of question is this?)
?- c([], H, T).

No

which means that there are no conditions under which the empty list can have a head or tail.

Example 3

- ▶ Consider a procedure to define a predicate `member(X,L)` which is true if `X` is an element of the list `L`.
- ▶ An element `X` is a member of the list `L` if `X` is the head of `L` (regardless of what its tail is):
`member(X, [X|_]) .`
- ▶ Additionally, `X` can be a member of `L` if `X` is a member of the tail of `L` (regardless of what its head is).
`member(X, [_|T]) :- member(X, T) .`

Example 3 /cont.

- ▶ Let us execute some queries:

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

Yes

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

Yes

```
?- member(X, [a,b]).
```

X = a ;

X = b ;

No

Example 3 /cont.

- ▶ Let us trace the call to `member(e, [a, b, c, d, e])`:

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

```
    member(e, [b,c,d,e]).
```

```
    member(e, [c,d,e]).
```

```
    member(e, [d,e]).
```

```
    member(e, [e]).
```

Yes

Example 4

- ▶ In this example, we want to define a clause `add` which succeeds if a new list can be created by placing an element as the head of some other list. The rule below,
`add(X, L, [X|L]).`
reads “Clause `add` succeeds if a new list, represented as `[X|L]`, can be created whose head is an element `X` and whose tail is a list `L`.”

Example 4 /cont.

- ▶ The query below reads “Is the list [a, b, c] created when placing element a as the head and list [b c] as the tail?” to which Prolog responds with a Yes.

```
?- add(a, [b, c], [a, b, c]).
```

Yes

Example 4 /cont.

- ▶ The query below reads “Is the list [a, b, c] created when placing element b as the head and list [b c] as the tail?” to which Prolog responds with a *No*.

```
?- add(b, [b, c], [a, b, c]).
```

No

Example 4 /cont.

- ▶ The query below reads “Under what conditions, if any, can a list be comprised with a as its head and with the list [b c] as its tail?” to which Prolog provides the condition as the list [a b c].

```
?- add(a, [b, c, d], NewList).
```

```
NewList = [a, b, c, d]
```

Example 4 /cont.

- ▶ The query below reads “Under what conditions, if any, an element a can be added to a list creating the list [a b c d e]?”

`?- add(a, L, [a, b, c, d, e]).`

`L = [b, c, d, e]`

which means that the condition under which the statement can be true is when the list is [b c d e].

Example 5

- ▶ In this example, we would like to define a rule `last` which succeeds if an element is the last element of a given non-empty list. We can identify two cases for this:
- ▶ The list has one element.
- ▶ The list has more than one element.

Example 5 /cont.

- ▶ **Case 1: The list has only one element.** In this case, the last element is the only existing element of the list. Let us translate this into Prolog. The following rule,
`last(L, [L]).`
reads “Rule `last` succeeds if an element `L` is found to be the only element of a given list.”

Example 5 /cont.

- ▶ The query below reads “Is element a the last element of the list [a]?” to which Prolog responds with a Yes.

```
?- last(a, [a]).
```

Yes

Example 5 /cont.

- ▶ The query below reads “Under what conditions, if any, is an element the last element of the list [a]?”

`?- last(L, [a]).`

`L = a`

which means that the condition under which the statement can be true is when `L = a`.

Example 5 /cont.

- **Case 2: The list has more than one element.** In this case, we need to reduce the problem to the one that can be handled by **case 1**. In other words, the clause will succeed once it chops off all elements, one by one, until it ends up with one element. The following rule,

`last(L, [H|T]) :- last(L, T).`

reads “Rule `last` can be proven true for a list whose head is `H` and whose tail is `T`, if it can be proven true for a new list which is the tail `T` of the original list.

Example 5 /cont.

- ▶ In other words, let us get rid of the first element and see if we end up with only one element in which case the rule of **case 1** will determine that this remaining element is indeed the last element.
- ▶ However, if after getting rid of the first element we end up with something which has a non-empty tail (i.e. there is still more than one element in the list), we must repeat this chopping off the head of the list, until we end up with a list which has only one element and subsequently handled by the first rule (of **case 1**).
- ▶ The query below reads “Is element c the last element of the list [a b c]?” to which Prolog responds with a Yes.

```
?- last(c, [a, b, c]).
```

Yes

Example 5 /cont.

- ▶ The query below reads “Under what conditions, if any, is an element the last element of the list [a b c]?”.

`?- last(L, [a, b, c]).`

`L = c`

which means that the condition under which the statement can be true is when `L = c`.

Example 6

- ▶ Consider a Prolog rule `is-contained` to determine if an atom, provided as the first argument, is contained in a list provided as second argument.

```
is-contained(X, [X|_]).
```

```
is-contained(X, [_|T]) :- is-contained(X,T).
```

Example 6 /cont.

- We can execute queries as follows:

```
?- is-contained(a, [a, b, c]).
```

Yes

```
?- is-contained(a, [c, b, a]).
```

Yes

```
?- is-contained(a, []).
```

No

Example 7

- Consider the following Prolog program, a2b, which behaves as follows:

?- a2b([a,a,a,a],[b,b,b]).

No

?- a2b([a,a,a,a],[b,b]).

Yes

?- a2b([a,a,a,a,a,a],[b,b,b]).

Yes

?- a2b([a,a,a,a],[b]).

No

?- a2b([a,d,f,a,a],[b,b]).

No

?- a2b([a,a],[b]).

Yes

?- a2b([a,a],[b,b,b,b]).

No

?- a2b([a,a,a,a,a,r],[b,b,b]).

No

?a2b([a,a],[b,b]).

No

?- a2b([a,a,a,a],[b,d]).

No

Example 7

- ▶ Our task is to describe what the program does, and write a Prolog program to perform this task.
- ▶ The program takes two lists as arguments, and succeeds if the first argument is a list of a's, and the second argument is a list of b's where the list of a's is twice the size of the list of b's.
- ▶ The Prolog program is as follows:

```
;; shortest possible list is the empty list  
a2b([], []).
```

```
;; need to have two a's for one b  
a2b([a,a|Ta],[b|Tb]) :- a2b(Ta, Tb).
```

Finite state machines (ch. 5)

Introduction to finite state machines

- ▶ An *automaton* is an abstract machine that can perform a task according to a specified set of instructions.
- ▶ It can read a string as input, perform operations and produce output.
- ▶ Each step in a computation is called a state.

Introduction to finite state machines /cont.

- ▶ In its initial state, the machine reads the first symbol of an input string, followed by the next symbol etc. until there are no more input symbols.
- ▶ After each reading, the machine changes state and possibly produces some output.
- ▶ After having read the entire string, the machine stops.

Finite state machines

Formally, a *finite automaton* (also called a *finite state machine* or FSM) is defined as a 5-tuple (read: “quintuple”) as follows:

$$(Q, q_0, F, \Sigma, \delta)$$

where

1. Q is a finite, non-empty set of *states*.
2. $q_0 \in Q$ is the *initial state* (or *start state*).
3. $F \subset Q$ is a set of *final states*.
4. Σ is a finite, non-empty set of symbols, called the *input alphabet*.
5. δ is a *state transition function*: $\delta : Q \times \Sigma \rightarrow q \in Q$. This function defines a *deterministic finite state machine* as opposed to a *nondeterministic finite state machine* whose state transition function returns a set of states.

Acceptors and Transducers

- ▶ Acceptors (also: *recognizers*, or *sequence detectors*) process all input and then produce a binary output (Yes/No) based on whether or not the input is accepted, i.e. if the current state is an accepting state.
- ▶ On the other hand, *transducers* generate output based on a given input and/or a state using actions.
- ▶ In the *Moore machine*, the output depends solely on the state, whereas in the *Mealy machine*, the output depends on both input and state.

Example 1

- ▶ Consider a machine that can accept any binary string that ends in ab , such as ab , bab , $baab$, $bbbaab$, etc.
- ▶ What does it mean “to execute an acceptor FSM over an input alphabet Σ ”?
- ▶ Given an FSM and a string $w \in \text{Power}(\Sigma)$, the FSM accepts each one of the letters of w as input (from left to right) following a path starting from the start state.

Example 1 /cont.

- ▶ Each letter causes a state transition from the start state to the next and so forth.
- ▶ If this path eventually ends in the final state, then we say that the FSM accepts w .
- ▶ Otherwise we say that the FSM rejects w .
- ▶ The *language* of an FSM is the set of all strings that it accepts.
- ▶ An automaton can be represented by a *state-transition table*, or by a directed graph, called a *state diagram*.

Example 1 /cont.

A state transition table for the FSM of this example is shown below.

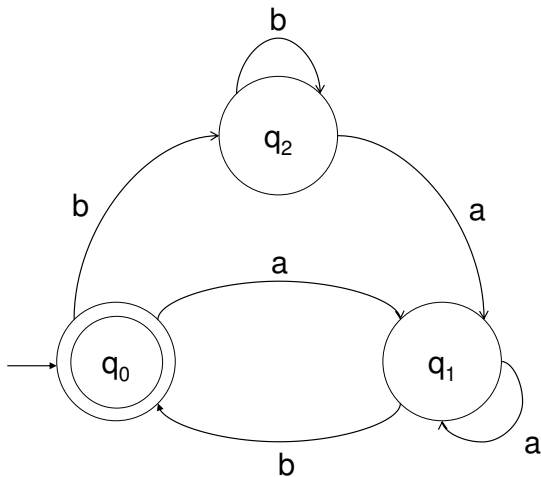
		a	b
initial/final state	q_0	q_1	q_2
	q_1	q_1	q_0
	q_2	q_1	q_2

Example 1 /cont.

- ▶ In this example, it just so happens that this is also the final state.
- ▶ Each bit input causes a transition either to the same state or to another state.
- ▶ For example, for the input string ab , we start at state q_0 and read the first bit, a , which will cause a transition to state q_1 . While at state q_1 the FSM will then read b and perform a transition to state q_0 .
- ▶ As the input string has been read and the FSM is currently at its final state, we say that the input string ab has been accepted by the FSM.

Example 1 /cont.

The state diagram of the FSM that can accept any binary string that ends in ab is shown below:



Example 1 /cont.

- ▶ Consider an input string *abb*.
- ▶ From the initial state q_0 the FSM will read *a* and perform a transition to state q_1 .
- ▶ While at state q_1 it will read *b* and perform a transition to state q_0 .
- ▶ While at state q_0 it will then read *b* which is the final symbol in the input string and perform a transition to state q_2 .
- ▶ As the input string has been read and the FSM is currently not at its final state, we say that the input string *abb* has been rejected by the FSM.

Example 1 /cont.

- ▶ The behavior of the FSM can also be represented by a state diagram.
- ▶ A state diagram is a directed graph, where the set of nodes, denoted as circles, defines the set of states (including initial and final states, where the latter is denoted as a double circle), and the set of edges defines the set of transitions.
- ▶ The arrow without a source node points to the initial state (in the example: q_0). An FSM can model the behavior of any type of entity in terms of its life-cycle and this behavior is captured by a traversal of the state diagram.

Regular expressions

- ▶ Consider a (possibly large) set of strings. Listing all its elements may be impractical or costly.
- ▶ We would like to provide a concise description of such a set.
- ▶ A *regular expression* is a pattern that describes a set of strings, making use of certain operations (see next).

Regular expressions /cont.

- ▶ A vertical bar represents the disjunction operator separating alternatives.
- ▶ Parentheses can define the scope of operators.
- ▶ For example, the regular expression $h(e|a)llo$ describes strings that start with an h followed by either an e or an a , and ending with a llo , thus capturing both *hello* and *hallo*.

- ▶ A quantifier after an element (a character or a group of characters) specifies how often that preceding element is allowed to occur.
- ▶ The question mark (?) indicates that there is zero or one of the preceding element, the asterisk (*) indicates that there are zero or more of the preceding element and the plus sign (+) indicates that there is one or more of the preceding element.

Example 2

ab^+c is a regular expression that denotes the set of strings that start with an a , followed by one or more b 's and finally followed by a c , such as $abc, abbc, abbbc, abbbbc, \dots$ etc.

Example 3

$a?(b^*|c)$ is a regular expression that denotes the set of strings that start with zero or one a , and followed either by zero or more b 's, or a c , such as $\Lambda, c, b, ac, ab, abb, abbb, \dots$ etc., where Λ is the empty string.

Example 4

Suppose we need to build a deterministic FSM to recognize strings that end in *aab* such as *aab*, *aaaab*, *babaab*, *bbabaab*, *aababaab*, etc.

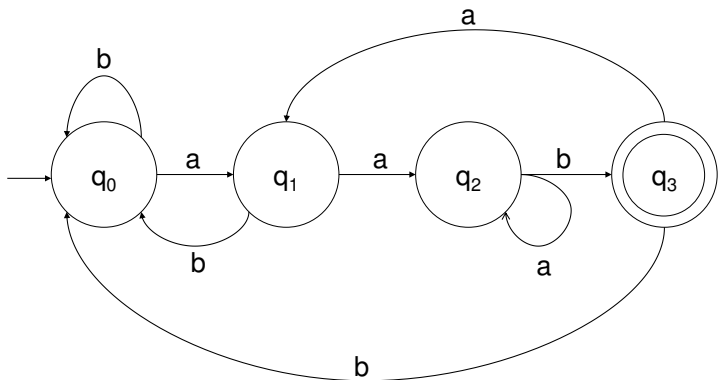
Example 4 /cont.

The state-transition table is shown below:

		a	b
initial state	q_0	q_1	q_0
	q_1	q_2	q_0
	q_2	q_2	q_3
final state	q_3	q_1	q_0

Example 4 /cont.

The state diagram of the FSM to recognize strings that end in *aab* is shown below:



A logic program interpreter for deterministic FSMs

- ▶ In problems of this kind, we need two pieces of information:
 1. The representation of an FSM by a sequence of facts, and
 2. An interpreter to recognize a language. The interpreter is made up of a sequence of rules and the language that it is meant to recognize is expressed as a regular expression.

FSM representation

- ▶ We can represent an FSM by facts of the the following form:
start(state).
transition(currentState, condition, nextState).
end(state).

FSM representation /cont.

- ▶ The start and final states can be taken directly from the figure, whereas the transitions can be more easily taken from the transition table.

```
start(q0).
```

```
final(q3).
```

```
transition(q0, a, q1).
```

```
transition(q0, b, q0).
```

```
transition(q1, a, q2).
```

```
transition(q1, b, q0).
```

```
transition(q2, a, q2).
```

```
transition(q2, b, q3).
```

```
transition(q3, a, q1).
```

```
transition(q3, b, q0).
```

Building an interpreter

- ▶ Given a set of facts as above, we need to build rules to determine whether or not a given string can be accepted by the FSM.
- ▶ A string w is accepted by an FSM if its reading from left to right (i.e. each symbol in turn is taken as a condition which determines some transition) causes a path from the start state to the final state.

Building an interpreter /cont.

- Consider a predicate `accept(Xs)`, where `Xs` is a an input string, represented by a list. A parsing is only valid if initiated from the start state.

`accept(Xs) :- start(Q), path(Q, Xs).`

Building an interpreter /cont.

- ▶ The second goal above needs to be defined as a new rule.
While at the start state Q , a string Xs will be accepted if its head causes a transition to a new state $Q1$ as well as if starting from $Q1$ the tail of Xs is accepted.
`path(Q, [X|Xs]) :- transition(Q, X, Q1), path(Q1, Xs).`

Building an interpreter /cont.

- ▶ If our input string is valid, we will eventually reach the final state, having exhausted all symbols in the string, i.e. once we reach the final state and we have an empty string.

```
path(Q, [ ]) :- final(Q).
```

Building an interpreter /cont.

- ▶ Putting everything together, we can provide the full listing of our interpreter program for the FSM as follows:

```
start(q0).  
final(q3).  
transition(q0, a, q1).  
transition(q0, b, q0).  
transition(q1, a, q2).  
transition(q1, b, q0).  
transition(q2, a, q2).  
transition(q2, b, q3).  
transition(q3, a, q1).  
transition(q3, b, q0).  
accept(Xs) :- start(Q), path(Q, Xs).  
path(Q, [X|Xs]) :- transition(Q, X, Q1), path(Q1, Xs).  
path(Q, [ ]) :- final(Q).
```

Executing the interpreter

`?- accept([a,a,b]).`

Yes

`?- accept([a,a,b,a,b,a,a,b]).`

Yes

`?- accept([]).`

No

`?- accept([b,a,a]).`

No

`?- accept([b,b,b,b,b,a,a,a]).`

No

`?- accept([a,a,b,a]).`

No

Boolean algebra and digital gates (ch. 6)

Boolean algebra and digital gates

- ▶ We will deploy clauses to model and simulate Boolean expressions and digital circuits.
- ▶ We have already seen that a proposition is a sentence that is either true or false (but not both).
- ▶ Many statements can be constructed by combining one or more propositions.
- ▶ New propositions, called *compound propositions*, can be formed from existing propositions using *logical operations* which are expressed as functions, called *truth functions*.
- ▶ Logical operators that are used to form new propositions from two or more existing propositions are called *connectives*.

Boolean operations

- ▶ Commonly used logical operators include:
- ▶ Conjunction (*and* connective) constructs a new proposition whose truth value is *true* if both of its operands are true, otherwise is *false*. It is denoted by \times , \wedge , or \cdot , e.g. $p \times q$. Many authors prefer to omit the conjunction symbol and simply write pq instead of $p \times q$.
- ▶ Disjunction (*or* connective) constructs a new proposition whose truth value is *true* if either or both of its operands are true, otherwise is *false*. It is denoted by $+$, or \vee , e.g. $p + q$.
- ▶ Inverse (*not* connective) constructs a new proposition whose truth value is the reverse truth value of its operand. It is denoted by $'$, \sim , or \neg . Some authors use \bar{q} to denote the inverse of proposition q .

Boolean operations /cont.

- ▶ The relationships between the truth values of the above compound propositions can be displayed in a *truth table* as follows:

x	y	x'	$x \times y$	$x + y$
1	1	0	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	0

Defining procedures to represent Boolean operators

- ▶ We can define procedures to represent logical operators in Boolean algebra and consequently digital gates which are the building blocks of digital circuits.
- ▶ In defining clauses, we will follow the convention *operation(in, out)* to denote an operation whose input is *in* and whose output is *out*.
- ▶ For example, the Boolean operation $'$ (*inverse*) is a unary operation whose procedure `inv` will include the clause `inv(0, 1)`.
which reads “The inverse of 0 is 1.”

Defining procedures to represent Boolean operators /cont.

- ▶ The Boolean operation *or* is a *binary* operation whose procedure `or` will include the clause
`or(0, 1, 1)`.
which reads “The disjunction of 0 and 1 is 1.”

Defining procedures to represent Boolean operators /cont.

- ▶ Knowing the truth-table definitions for Boolean operations, we can define the corresponding procedures as follows:

`and(1, 0, 0).`

`and(0, 1, 0).`

`and(0, 0, 0).`

`and(1, 1, 1).`

`or(1, 0, 1).`

`or(0, 1, 1).`

`or(0, 0, 0).`

`or(1, 1, 1).`

`inv(0, 1).`

`inv(1, 0).`

Defining procedures to represent Boolean operators /cont.

- ▶ The above would be enough to be able to represent any Boolean expression.
- ▶ However, for convenience we can also define operations *nor* (not or), *xor* (exclusive or), and *nand* (not and) as follows:

`nand(1, 0, 1).`

`nand(0, 1, 1).`

`nand(0, 0, 1).`

`nand(1, 1, 0).`

`nor(1, 0, 0).`

`nor(0, 1, 0).`

`nor(0, 0, 1).`

`nor(1, 1, 0).`

`xor(1, 0, 1).`

`xor(0, 1, 1).`

`xor(0, 0, 0).`

`xor(1, 1, 0).`

Evaluating Boolean expressions

- ▶ We can build rules to represent Boolean expressions. Consider the expression $(x \times y') + y$ whose truth table is given below:

x	y	y'	$x \times y'$	$(x \times y') + y$
1	1	0	0	1
1	0	1	1	1
0	1	0	0	1
0	0	1	0	0

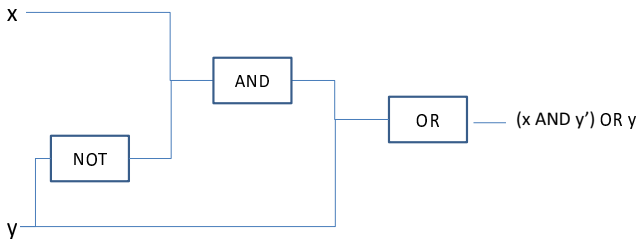
Evaluating Boolean expressions /cont.

- ▶ We can define a rule to represent the Boolean expression (and consequently the digital circuit) as follows:

```
circuit(X, Y, Out) :-  
    inv(Y, Tmp1),  
    and(X, Tmp1, Tmp2),  
    or(Tmp2, Y, Out).
```

Evaluating Boolean expressions /cont.

- The expression can be built as the digital circuit shown below:



Evaluating Boolean expressions /cont.

- ▶ We can now test the digital circuit by executing queries over particular input sequences as follows:

```
?- circuit(1, 1, Out).  
Out = 1 .
```

```
?- circuit(1, 0, Out).  
Out = 1 .
```

```
?- circuit(0, 1, Out).  
Out = 1 .
```

```
?- circuit(0, 0, Out).  
Out = 0 .
```

Evaluating Boolean expressions /cont.

- ▶ We can ask questions that correspond to ground and non-ground queries. For example, we can ask “Is it indeed the case that for $X = 1$ and for $Y = 1$, the output is 1?”, and the corresponding query is

```
?- circuit(1, 1, 1).  
true
```


Evaluating Boolean expressions /cont.

- We can also ask questions like “For what input values, if any, is the output 0?”

```
?- circuit(X, Y, 0).
```

```
X = 0,
```

```
Y = 0 ; %% Are there more values?
```

```
false.
```

Evaluating Boolean expressions /cont.

- ▶ We can also simulate the digital circuit by executing the program as follows:

```
?- circuit(X, Y, OUT).  
X = 0,  
Y = 0,  
OUT = 0 ;  
X = 1,  
Y = 0,  
OUT = 1 ;  
X = 1,  
Y = 1,  
OUT = 1 ;  
X = 0,  
Y = 1,  
OUT = 1 ;  
false.
```

Evaluating Boolean expressions /cont.

- ▶ It turns out that the Boolean expression of this example (and its corresponding digital circuit) can be simplified to a single logic gate or.
- ▶ How can we be sure? If we use simulation to investigate the behavior of the two circuits, then we see that for the same input, the output of the two circuits is the same.

?- or(X, Y, OUT).

X = 1,

Y = 0,

OUT = 1 ;

X = 0,

Y = 1,

OUT = 1 ;

X = 0,

Y = 0,

OUT = 0 ;

X = 1,

Y = 1,

OUT = 1.