LOGIC PROGRAMMING

- Chapter 9 of the book

Paradigm

- Declarative programming paradigm
  - The programmer declares the goals of the computation rather than the detailed algorithm by which these goals can be achieved.
- Logic programming is based on:
  - unification (Robinson, 1965) and
  - resolution (Robinson, 1965)
- Two important features of logic programming are:
  - non-determinism and
  - backtracking
- Popular in artificial intelligence
- Applications:
  - Natural language processing
  - Theorem proving
  - Databases
  - Expert systems
- PROLOG is a logic programming language (Colmerauer, 1972)

Logic

- **Propositional Logic**
  - Propositions:
    - true and false are propositions.
    - Propositional variables are propositions.
    - If p and q are propositions, then:
      - $p \wedge q$, $p \vee q$, $p \rightarrow q$, $p \leftrightarrow q$ and $\neg p$

      are propositions.
    - Precedence on connectives: $\neg > \wedge > \vee > \rightarrow > \leftrightarrow$
    - Examples: How do you formalize the following English sentences?
      - Provided that Marvin stays Nancy leaves.
      - Marvin stays but Nancy leaves.
      - Marvin stays although Nancy leaves.
  - Each proposition can be interpreted as either *true* or *false*.
    - Semantics methods
    - Syntactic methods
  - Propositional logic is decidable.

- **Predicate Logic - First order predicate calculus**
  - A **predicate** "is a quantified proposition with variables".
  - Quantifiers are $\forall$ (for all) and $\exists$ (exists).
  - A predicate is **satisfiable** if for some particular assignment of values to its variables the predicate is true.
  - A predicate is **valid** if for all assignment of values to its variables the predicate is true.
  - Examples:
    - $parentof(x, y)$ is the same as $\forall x, \forall y, parentof(x, y)$
    - $fatherof(x, y)$
    - $speaks(x, y)$
    - $prime(n)$
    - $\forall x(speaks(x, Japanese))$
    - $\exists x(speaks(x, Japanese))$
    - $\forall x \exists y(speaks(x, y))$
    - "Not every child can read" is equivalent to "There is at least one child who cannot read."
  - The Incompleteness Theorem of Goedel, proven in 1930, demonstrates that first-order logic is in general undecidable.

Normal Forms

- Equivalences can often be used to *simplify* formulas, to obtain equivalent formulas of a certain syntactic form, also called **normal forms**.
- An advantage of normal forms is that certain questions can often be easier answered. **Conjunctive** and **disjunctive forms** are especially useful in this sense.
- A propositional formula is said to be in **conjunctive form** if
  1. it contains only the logical connectives $\neg$, $\wedge$ and $\vee$,
  2. no logical connective occurs inside of a negation.
  3. no conjunction occurs inside of a disjunction.
- We speak of a **disjunctive form** if the last condition is replaced by the condition that *no disjunction occur inside any conjunction.*
- For example, $\neg(p \wedge q)$ is neither in disjunctive nor conjunctive form, whereas $p \vee (q \wedge r)$ is in disjunctive form, but not in conjunctive form.

Clauses

- A **literal** is either a predicate or the negation of a predicate.
- Disjunctions of literals, $L_1 \vee \cdots \vee L_n$, are also called **clauses**.
- Since a conjunction $\alpha_1 \wedge \cdots \wedge \alpha_n$ is true under a given truth assignment if, and only if, each formula α_i is true, and each formula is equivalent to a **conjunctive (normal) form**, we may conclude that each formula can be represented in logically equivalent form as a collection of clauses.
  - For example, $p \iff q$ can be represented by the two clauses, $\neg p \vee q$ and $p \vee \neg q$.
- If a clause contains *at most* one positive literal, then it is called a **Horn clause**.
  - For example, $\neg p \vee \neg q$ and $\neg p \vee \neg q \vee r$ are Horn clauses, but $p \vee q$ is not a Horn clause.
- An interesting aspect of Horn clauses is that they can be interpreted as program rules and used for computation, as is done in **logic programming**.

Logic Programming

- A **Horn clause** $\neg p_1 \vee \cdots \vee \neg p_n \vee q$ is logically equivalent to the implication $(p_1 \wedge \cdots \wedge p_n) \rightarrow q$.
- If the implication is known to be true, and one wishes to prove q, then it sufficient to show that $p_1, \ldots, p_n$ are all true; an observation that provides the logical basis for logic programming.
- A **logic program** is a set of Horn clauses, each containing exactly one positive literal (and zero or more negative literals). Such Horn clauses are usually written as backward implications

$$q \leftarrow p_1, \ldots, p_n$$

  and called **program rules**. More specifically, q is called the **head** of the rule, and the sequence $p_1, \ldots, p_n$ the **body** of the rule. (Each rule must have a head, but the body may be empty and in that case the rule is called a **fact**. For instance $q \leftarrow$ is a fact.)

Notations

- A Horn clause is written as:

$$q \leftarrow p_1, \ldots, p_n$$

  It means the same as:

$$\neg p_1 \vee \cdots \vee \neg p_n \vee q$$

- If $n = 0$, the clause is: $q \leftarrow$.
  $q \leftarrow$ is the same as q.
- $\leftarrow p$ is the same as $\neg p$.

Unification

- **Unification** is a pattern-matching process that determines what particular instantiation can be made to variables to make two predicates equal. This instantiation is called a **substitution**.
- Examples:
  - How to make $brotherof(John, x)$ and $brotherof(y, Bill)$ equal?

    With the substitution: $x \mapsto Bill$, $y \mapsto John$
  - How to make b and b equal?

    With the substitution: id (identity)

Unification algorithm

See the handout.

Logic program

Propositional case

$e \leftarrow$
$f \leftarrow$
$b \leftarrow$
$c \leftarrow a, b$
$a \leftarrow e, f$

- is a propositional logic program of five rules. The first three rules have an empty body and represent **facts**.
- In addition to the program rules one needs to specify a **goal** (or a list of goals) that we want to prove.

  Example: If we want to prove c, the goal is c.
- A computation with a logic program represents an attempt to derive the goal from the program rules (in an indirect way by deriving a **contradiction** in the form of the "empty clause " (represented by $\Box$) from the **negation** of the goal).
- The logical inference rule underlying such computations is called **resolution**.

Logic program

With variables

P(Edward VII, George V) $\leftarrow$
P(Victoria, Edward VII)$\leftarrow$
P(Alexandra, George V)$\leftarrow$
P(George VI, Elizabeth II)$\leftarrow$
P(George V, George VI)$\leftarrow$
$G(x, y) \leftarrow P(x, z), P(z, y)$

- is a logic program of six rules. The first five rules have an empty body and represent facts (about the British royal family).
- The last rule defines the *grandparent relation* in terms of the *parent relation*: a person x is a grandparent of y if there is a third person z, such that x is the parent of z, and z the parent of y.
- The use of variables, such as x, y, and z, which denote individuals goes beyond the scope of propositional logic, but is crucial for the usefulness of logic programming.
- Informally, the rule $G(x, y) \leftarrow P(x, z), P(z, y)$ may be thought of as a schema representing all clauses

obtained by substituting specific values for the variables, e.g.,

$$G(Vict, G.V) \leftarrow P(Vict, E.VII), P(E.VII, G.V)$$

x = Vict, y = E. VII, z = G. V

- In addition to the program rules one needs to specify a **goal** (or a list of goals) that we want to prove.

  Example: If we want to prove that the grandfather of George V is Victoria then the goal is G(Victoria, George V).

- A computation with a logic program represents an attempt to derive the goal from the program rules (in an indirect way by deriving a **contradiction** in the form of the "empty clause" (□) from the **negation** of the goal).

- The logical inference rule underlying such computations is called **resolution**.

Resolution

Propositional case

- The propositional version of resolution for Horn clauses is:

  From $\leftarrow p_1, \ldots, p_n$ *and* $p_1 \leftarrow q_1, \ldots, q_k$
  derive $\leftarrow q_1, \ldots, q_k, p_2, \ldots, p_n$.

$$\frac{\begin{array}{l}\leftarrow p_1, \ldots, p_n \\ p_1 \leftarrow q_1, \ldots, q_k\end{array}}{\therefore \leftarrow q_1, \ldots, q_k, p_2, \ldots, p_n}$$

  - What is the rule if $n = 1$ and $k = 2$?

$$\frac{\begin{array}{l}\leftarrow p_1 \\ p_1 \leftarrow q_1, q_2\end{array}}{\therefore \leftarrow q_1, q_2}$$

  - What is the rule if $n = 1$ and $k = 0$?

$$\frac{\begin{array}{l}\leftarrow p_1 \\ p_1 \leftarrow\end{array}}{\therefore \Box}$$

- **Example:** Assume we want to prove c.
  - The negation of the goal c is written as a negative clause

    $$\leftarrow c.$$

  - We have also seen that c is the head of a rule $(c \leftarrow a, b)$.
  - This indicates that the given goal may be *reduced* to subgoals (by the resolution rule)

    $$\leftarrow a, b.$$

  - We have also seen that a is the head of a rule $(a \leftarrow e, f)$.
  - This indicates that the given goal may be *reduced* to subgoals (by the resolution rule)

    $$\leftarrow e, f, b.$$

    where a is replaced by e, f.

- The three subgoals are present as facts and hence can be deleted, which results in the empty clause (□).

- We conclude that the original goal logically follows from the program clauses.

- But much of the power of logic programming derives from the fact that resolution can be generalized to effectively handle clauses with variables.

Resolution

With variables

- Assume we want to prove that Victoria is the grandmother of George.

- The negation of the above goal is written as a negative clause

  $$\leftarrow G(Victoria, George V).$$

- We have also seen that suitable values may be substituted for the variables in the last program rule, so that the head is G(Victoria, George V) (x=Vict and y = G. V).

- This indicates that the given goal may be *reduced* to subgoals

  $$\leftarrow P(Victoria, Edward VII), P(Edward VII, George V).$$

- Both subgoals are present as facts and hence can be deleted, which results in the empty clause (□).

- We conclude that the original goal logically follows from the program clauses.

- Goals with variables are also possible.

  Example: If one specifies the goal

  $$\leftarrow G(Victoria, x)$$

  the result of the computation will be a list of all grandchildren of Victoria. A discussion of these aspects of logical programming is beyond the scope of this course.

  - In PROLOG:
    * A variable begins with a capital letter in PROLOG.
    * A predicate is written in lower cases.
    * Underscore characters are taken as variables.
    * All facts, rules and queries end with a period.

PROLOG

- SWI prolog.
- On matrix:
  - Save your PROLOG programs in files.
    Example: Let's consider the *likes.pl* file.

    ```
    likes(john,mary).
    likes(mary,sue).
    likes(mary,tom).
    ```

    We just defined 3 facts in the *likes.pl* file.
  - To run PROLOG type: *pl*, then
  - To load the *likes.pl* file, type: *[likes].* or *consult(likes).*.
  - You can now play with prolog:
    Who are the people Mary likes?

    ```
    likes(mary,X).
    ```

    X is a variable and must be written using a capital letter.
    To have all the solutions to the $likes(mary, X)$ goal, type n (for next) after each solution.

Examples of programs

- Explicit definition 1:

  ```
  f(x) = if x=0 then 1 else 5

  PROLOG:
  f(0,1).
  f(X,5) :- X>0.
  ```

- Explicit definition 2:

  ```
  f(x) = 2*x

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

- Example:

```
PROLOG:
speaks(allen,russian).
speaks(bob,english).
speaks(mary,russian).
speaks(mary,wnglish).
talkswith(Person1,Person2):-speaks(Person1,L),
speaks(Person2,L), Person1 \= Person2.
```

How to know who talks with who?

- Recursive definition 1:

```
fact(n) = if n=0 then 1 else n*fact(n-1)

PROLOG:
factorial(0,1).
factorial(N,Result) :- N>0, M is N-1,
factorial(M,SubResult), Result is N*SubResult.
```

- Recursive definition 2:

```
fib(n) = if n=0 then 1 else if n=1 then 1
else fib(n-1)+fib(n-2)

PROLOG:
fib(0,1).
fib(1,1).
fib(N,R) :- N>1, N1 is N-1, N2 is N-2, fib(N1,R1),
fib(N2,R2), R is R1+R2.
```

Tracing in PROLOG

- To trace a particular function f use:

```
trace(f/2).
```

- Example:

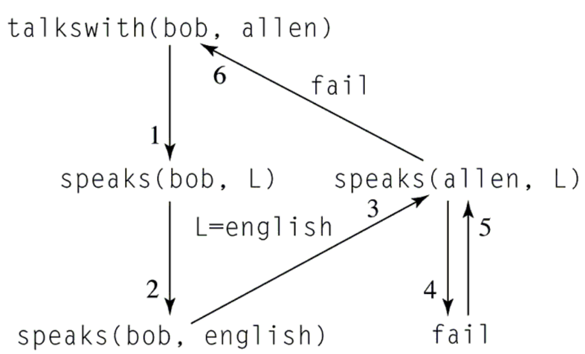
```
trace(factorial/2).
```

```
?- factorial(4, X).               N  M   P     Result
Call:  (  7) factorial(4, _G173)  4  3  _G173  4*P
Call:  (  8) factorial(3, _L131)  3  2  _L131  3*P
Call:  (  9) factorial(2, _L144)  2  1  _L144  2*P
Call:  ( 10) factorial(1, _L157)  1  0  _L157  1*P
Call:  ( 11) factorial(0, _L170)  0     _L170
Exit:  ( 11) factorial(0, 1)                   1
Exit:  ( 10) factorial(1, 1)                   1*1 = 1
Exit:  (  9) factorial(2, 2)                   2*1 = 2
Exit:  (  8) factorial(3, 6)                   3*2 = 6
Exit:  (  7) factorial(4, 24)                  4*6 = 24
```

Unification, Evaluation, Backtracking

Goal without variables

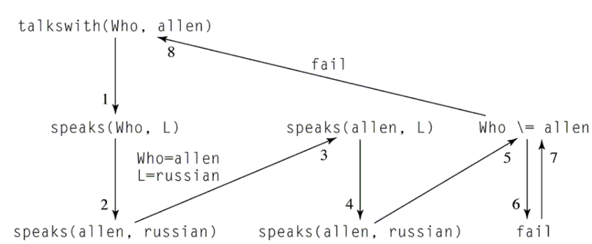
```
talkswith(bob,allen).
```



Unification, Evaluation, Backtracking

Goal with variables

```
talkswith(Who,allen).
```



Lists in PROLOG

- The basic data structure in PROLOG is the *list*.
  - [] is the empty list
  - $[X, Y]$ is a list with 2 elements
  - $[\_, \_, Y]$ is a list with 3 elements
  - $[X|Y]$ denotes a list with head X and tail Y.
- Some built-in functions on lists:
  - $append(?List1, ?List2, ?List3)$
  - $length(?List1, ?Int)$
  - $reverse(+List1, -List2)$
  - $member(?Elem, ?List)$
  - $sort(+List, -Sorted)$ (to sort a list – it removes the duplicates)
- Definition of functions on lists:
  - member:

```
member1(X,[X|_]).
member1(X,[_|Y]) :- member1(X,Y).
```

  - append:

```
append1([],X,X).
append1([H|T],Y,[H|Z]) :- append1(T,Y,Z).
```