Logic Programming is a paradigm for programming by declaring facts and rules. Programs are executed by querying whether a particular fact can be deduced from the facts and rules in the program. This topic briefly introduces the core of Prolog, the most popular logic programming language.
Background

Logic programming originated from AI representations of knowledge. This knowledge was then used to deduce new knowledge using formal logic, often called *theorem proving*.

- John McCarthy first suggested programming via logic in 1958.
- [He is also known for inventing LISP and functional programming, and also the term “artificial intelligence”.]

A key difference between theorem proving and logic programming is that the order that each fact or rule will be used is predictable.

Prolog was the first logic programming language to achieve popularity.

- It was originally designed by Colmerauer and Kowalski in 1972.
- It answers *queries* roughly by treating them like procedure calls.
- This allows declarative programming with some control over how queries are solved.

Logic programming has some similarities to functional programming, but additionally includes *unification* and *backtracking search*. 
Basic elements of prolog

- A prolog program consists of a set of rules that specify when certain relationships hold between objects.

- An object is just a name (not starting with a capital), possibly applied to some other objects.
  - E.g., these 3 are objects: a apple truck_year(mazda, 1986)
  - There are also special objects for numbers, lists, etc.

- A relationship similarly is a name (not starting with a capital) possibly applied to some objects.
  - E.g., father_child(fred, joe)
  - A relationship is something that might be known like “Fred is Joe’s father”
  - We say that a relationship holds when it is known, or can be deduced.

- A rule declares that a relationship holds whenever some other set of relationships holds. This is written (using rel etc, for relationships).
  
  rel :- rel1, ..., reln.

- Rules may contain variables, which start with a capital letter.
  - The rule holds for every possible way of replacing the variable by objects.

- E.g., here’s a rule for “X and Y are siblings if they have the same parent”:

  sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).
Example Prolog program

The relationships after the :- are called subgoals.
- When there are no subgoals, the :- is left out (such rules are facts).

Example – here is a program with 4 facts and 3 rules with subgoals.

```prolog
mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).
pARENT_CHILD(X, Y) :- mother_child(X, Y).
pARENT_CHILD(X, Y) :- father_child(X, Y).
sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).
```

Programs are run by making queries, which are relationships to check.

Executing a query attempts to use the rules to show the query holds.

A query containing variables requires finding some objects that make the query hold - the object chosen for each variable is printed.
- There may be many solutions: each is printed in turn.

```
sibling(sally, mike)       Result: no
sibling(sally, erica)      Result: yes
sibling(eric, W)           Result: W=sally or W=erica
sibling(V, W)              Result: V=sally W=sally or V=sally W=erica
or ...                     
```
Execution of Prolog programs

Execution proceeds by trying to show that goals hold, with the initial goal being the query, and subgoals arising when rules are used.

Solving a goal involves trying to unify with the LHS of each rule, in order.

- **Unification** means replacing the variables in the LHS and the goal with objects in a way that makes the LHS equal to the goal.
  - More precisely, unification only replaces as necessary to achieve LHS=goal
  - Note that the objects chosen may themselves include variables.

- Each time a variable is used, the object chosen for it is retrieved.
  - This includes variables in subgoals, and after a rule succeeds and “returns”.
  - If the retrieved object includes variables, these might be further unified.

- Unification allows variables to be used as both inputs and outputs.
  - This is how solutions to queries are generated.
  - More generally it is how data flows through a logic program.

When unifying with a LHS succeeds, each of the sub-goals on the RHS are solved. If all subgoals succeed, then the original goal succeeds.

- On success, the objects chosen for variables can be used by the caller.

After returning, if a failure later occurs, more solutions will be requested.

- At this point we trying to unify with the remaining rules.
- This is called *backtracking*, and is a distinctive feature of LP.
Logic variables

Variables in logic programming are quite different from those in functional programming and imperative programming.

The essential differences between these three paradigms can be viewed in terms of how their variables behave.

- An imperative variable contains a value, but this value can be modified. Thus, over time the variable may refer to many different values.
- Functional variables are just names for values, and cannot be modified.
- A logic variable represents an object that we do not yet know.
  - As the program executes the object may become known because a rule is used that requires it to be a particular object.
  - It can also become partially known, when the variable is unified with objects that include other variables (which may later be known).
  - During backtracking, alternative solutions to goals may include different objects for each variable, only some of which ultimately lead to success.
  - It is not until the top-level query succeeds that we “know” what object a logic variable represents.
    (And, it may represent different objects each time the query succeeds.)
List Append in Prolog

- Prolog uses \([x_1, x_2, x_3]\) for a list containing \(x_1\), \(x_2\) and \(x_3\).
- Also \([X|XS]\) for a list with head \(X\) and tail \(XS\).

Appending two lists in Prolog.

\[
\text{append([], YS, YS).}\\
\text{append([X|XS], YS, [X|ZS]) :- append(XS, YS, ZS).}
\]

- Compare this to the F# code for appending from topic 5.

\[
\text{let rec append xxs ys =}\\
\text{match xxs with}\\
\text{| [] -> ys}\\
\text{| x::xs -> x :: append xs ys}
\]

- The Prolog version works just like the F# version when the first argument is an input, i.e., it contains no variables.
  - Both match the 1\(^{st}\) argument, returning the 2\(^{nd}\) when the 1\(^{st}\) is empty.
  - Otherwise they use recursion on the tail, and add the head to the result.
- However, as we’ll see, the Prolog version also works when the last argument is the input, and the first is a variable.
Execution of append

Query:
?- Append([1, 2], [3, 4], RS).

\[ RS = [1, 2, 3, 4] \]

The main steps in the executing of this query are:

- The goal matches second rule for append (but not the first):
  \[ \text{append}([X_1|XS_1], YS_1, [X_1|ZS_1]) : \text{append}(XS_1, YS_1, ZS_1). \]
  Unifying the goal with the LHS of the rule yields:
  \[ X_1=1 \quad XS_1=[2] \quad YS_1=[3,4] \quad RS=[1|ZS_1] \]

- The subgoal \( \text{append}([2], [3,4], ZS_1) \) then matches the second rule:
  \[ \text{append}([X_2|XS_2], YS_2, [X_2|ZS_2]) : \text{append}(XS_2, YS_2, ZS_2). \]
  \( X_2=2 \quad XS_2=[] \quad YS_2=[3,4] \quad ZS_1=[2|ZS_2] \)

- The subgoal \( \text{append}([], [3,4], ZS_2) \) then matches the first rule:
  \[ \text{append}([], YS_3, YS_3). \]
  \( YS_3=[3,4] \quad ZS_2=[3,4] \)

- There are no outstanding subgoals so query succeeds with \( RS=[1, 2, 3, 4] \)
  - Note that: \( RS=[1|ZS_1]=[1|[2|ZS_2]]=[1|[2|[3,4]]]=[1,2,3,4] \)
Execution with the last argument as input

Query:

\[
\text{Append}(\text{XS}, \text{YS}, [1,2]).
\]

This query succeeds three times, corresponding to the three different ways of breaking \([1,2]\) into two parts.

The main steps in executing this query are:

- The goal matches the 1\textsuperscript{st} rule for append, unified to obtain:
  \[
  \text{append}([], [1,2], [1,2]). \quad \text{Succeeds: } \text{XS}=[] \quad \text{YS}=[1,2]
  \]

- \textit{Backtrack} – instead match the goal with the 2\textsuperscript{nd} (unified) rule:
  \[
  \text{append}([1|\text{XS1}], \text{YS1}, [1|[2]]) : \text{- append}(\text{XS1}, \text{YS1}, [2]).
  \]

- The subgoal \texttt{append(XS1, YS1, [2])} matches the 1\textsuperscript{st} rule:
  \[
  \text{append}([], [2], [2]). \quad \text{Succeeds: } \text{XS}=[1] \quad \text{YS}=[2]
  \]

- \textit{Backtrack} – instead match \texttt{append(XS1, YS1, [2])} with the 2\textsuperscript{nd} rule:
  \[
  \text{append}([2|\text{XS2}], \text{YS2}, [2|[\text{]}]) : \text{- append}(\text{XS2}, \text{YS2}, []).
  \]

- The subgoal \texttt{append(XS2, YS2, [])} matches the 1\textsuperscript{st} rule (only):
  \[
  \text{append}([], [], []). \quad \text{Succeeds: } \text{XS}=[1,2] \quad \text{YS}=[]
  \]
Example: Reverse

Reversing a list can be implemented as follows.

reverse([], []).  
reverse([A|B], Z) :- reverse(B, Brev), append(Brev, [A], Z).

However, this has the same efficiency issues as the simple version of reverse in F# in topics 5 and 6.

This can be solved by adding an accumulator, just like in F#.

revAppend([], X, X).  
revAppend([X|Y], Z, W) :- revAppend(Y, [X|Z], W).

reverse(A, R) :- revAppend(A, [], R).

Note that neither version works well when the second argument is the input and the first is the output.
Logic Programming in F#

- F# is a multi-paradigm language, but it has no direct support for logic programming.
- However, the combination of higher-order functions and effects allows us to build a library that supports logic programming.
- This is an embedding of the core of Prolog into F# - it will allow us write F# programs that appear quite similar to Prolog.
- The key elements of this embedding are:
  - Logic computations which take a success continuation as an extra argument. Each time they succeed, they call this continuation.
  - When a logic computation has no more solutions, it returns. If it was called as a continuation, this allows the caller to backtrack and try alternative rules.
  - Substitution is implemented by placing reference cells in logic variables that are updated as the variable becomes known. (And reverted during backtracking.)
**An Embedding of LP in F#**

```fsharp
type cons = string

type term = Var of term option ref | Cons of cons * term list

member this.vget = match this with Var ({contents=Some tm} as r) -> tm.vget
    | _ -> this

static member (^|) (tm1, tm2) = Cons("|", [tm1; tm2])  // List constructor

// An LP computation takes a success continuation k:unit->unit, and then () to delay the computation.
// Note that applying the computation to a continuation yields another continuation : unit -> unit
// Hence two lpComp's can be composed - lp1 << lp2 succeeds only if lp1 succeeds and then lp2.

type lpComp = (unit -> unit) -> unit -> unit

let rec unify : term -> term -> lpComp = fun tm1 tm2 k () ->
    match tm1.vget,tm2.vget with
    | (Var r1, Var r2) when obj.ReferenceEquals (r1, r2) -> k ()  // succeeds - calls k
    | (Var r, tm) | (tm, Var r) -> r:=Some tm; k (); r:=None  // succeeds - calls k
    | Cons(c1,tms1), Cons(c2,tms2) -> if c1<>c2 then ()  // fails - returns
    else (List.foldBack2 unify tms1 tms2) k ()

let (^=) = unify
let (~(%%)) n = Cons (string n, [])
let nil = Cons ("[]", [])

let newvar () = Var (ref None) and newvar2 () = newvar(), newvar() and ...

let rec append(XS, YS, ZS) k () = newvar3() |> fun(XX, XXS, ZZS) ->
    XS=nil << YS=ZS << |k|()
    XS=(XX|XXS) << ZS=(XX|ZZS) << append(XXS, YS, ZZS) <|k|()
An Improved Embedding of LP in F#

// The previous embedding requires unifying the arguments separately.
// The following is closer to Prolog – there’s some extra “noise” on the first line,
// but the remaining lines are structured just like in Prolog.

let unify2 (X,Y) (XX,YY) = X^=XX << Y^=YY
let unify3 (X,Y,Z) (XX,YY,ZZ) = X^=XX << Y^=YY << Z^=ZZ

let rec append2 args k () = (newvar4(), unify3 args) |> fun((X, XS, YS, ZS), uniArgs) ->
    uniArgs(nil, YS, YS) <|k<|()
    uniArgs((X|x|XS),YS,(X|x|ZS)) <| append2(XS, YS, ZS) <|k<|()

// This constructs a top level continuation that prints the values of some variables on success
let kEnd vars s = printfn "\nture:"
    for (v, tm) in vars do printfn "%s = %s" v (tm.ToString())

let lpRun vnames lpGen = let vars = List.map newvar vnames
    lpGen vars (kEnd (List.zip vnames vars)) ()

lpRun ["ZS"] (fun [ZS] -> append2((%%1 ^| %%2 ^| nil), (%%3 ^| %%4 ^| nil), ZS))
// Output:
// true
// ZS = 1 | 2 | 3 | 4 | []