What we have discussed so far are common to many of the programming paradigms.

Most of our discussion was around imperative/procedural. Especially statements, expressions, procedure passing… So we are not gonna talk about it anymore.

We are gonna talk about:

* OO
* Functional
* Logic
* …

PROLOG

Prolog:

* “Programming in Logic” (PROgrammation en LOgique)

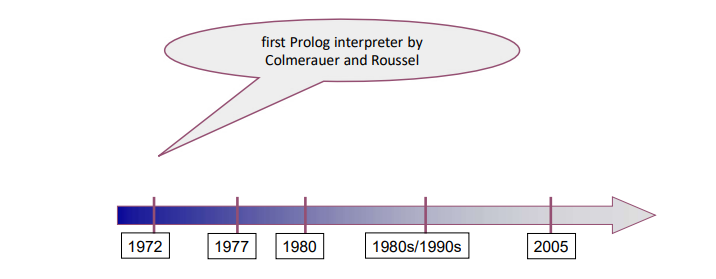
One (and maybe the only one) successful logic programming languages

Useful in AI applications, expert systems, natural language processing, database query languages

Declarative instead of procedural: “What” instead of “How”

* Procedural
  + You define the actions that needs to be taken and effects that needs to be created step by step. Then define these steps as procedures. 🡪 do this, do that, do this, … and you achieve the result.
  + You have data and program.
    - Program is composed of individual steps. Control is quite sequential. It can go from units to units, doesn’t matter but it manipulates data as part of the programming task.
  + Idea in procedural/imperative language is to process data and make changes through program steps.
* Declarative
  + Go the other way around. You say: “I want to declare my axioms instead of writing the steps” and then programming is just writing those axioms and you’ll have program running in that case just as simple proof of whether your query is explained by your facts or not.
  + You are gonna declare your facts.
  + There is no data. Data becomes your program, your facts.
  + Programming is about writing these facts:
    - Today is sunny 🡪 fact
    - When it is sunny, no rain 🡪 another fact
  + When I say a query to this program or run of this program could be “is today raining”. I am gonna say since today is sunny, and when it is sunny, no rain; answer should be there is no rain today.
  + Prolog is gonna do this (how to declare these certain axioms and also how to prove a given query based on these axioms) in a specific way.

History of Prolog



Diagram, timeline

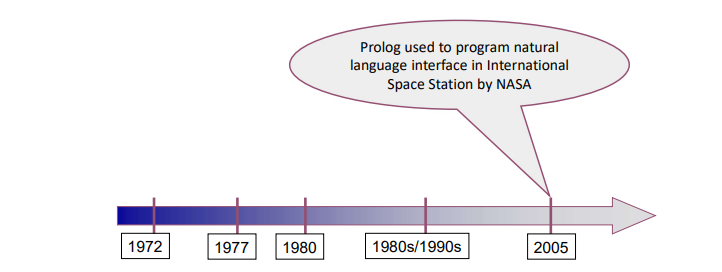
Description automatically generated with medium confidence

Diagram, timeline

Description automatically generated

Timeline

Description automatically generated



Logic Programming

Program

* Axioms (facts): true statements
  + Program is all your axioms. You declare your axioms (that’s why it is called declarative programming). These are the true statements of your problem domain. Whatever you are solving (you are solving a problem) and in your problem domain you have facts or axioms or true statements. You write them down and compiler or interpreter gets your query (in terms of inputting the program means), based on the true statements you have, your program will prove or disprove this particular query.

Input to Program

* query (goal): statement true (theorems) or false?

Thus ----> Logic programming systems = deductive databases

* datalog

Example:

Axioms:

* 0 is a natural number. (Facts)
* For all x, if x is a natural number, then so is the successor of x.

Query (goal):

* Is 2 natural number? (can be proved by facts)
* Is -1 a natural number? (cannot be proved)

Example:

Axioms:

* The factorial of 0 is 1. (Facts)
* If m is the factorial of n - 1, then n \* m is the factorial of n.

Query:

* The factorial of 2 is 3?

These are in natural language, how can I declare my facts in a formal language? We need to describe syntax and semantic.

First-Order Predicate Calculus (formal mechanism to declare facts)

Logic used in logic programming:

* First-order predicate calculus
* First-order predicate logic
* Predicate logic
* First-order logic
* ---
* x (x x+1)

Second-order logic

* S x (xS V xS)

First-Order Logic

First-order logic (FOL) models the world in terms of

* Objects, which are things with individual identities (STUDENTS, LECTURES, COMPANIES, CARS…)
* Properties of objects that distinguish them from other objects (COLOR, SHAPE…)
* Relations that hold among sets of objects (SMN IS BROTHER OF SMN, SMN VISITS SMN, SMN OWNS STH, STH OCCUR BEFORE OR AFTER STH…)
* Functions, which are a subset of relations where there is only one “value” for any given “input”

Examples:

* Objects: Students, lectures, companies, cars ...
* Relations: Brother-of, bigger-than, outside, part-of, has-color, occurs-after, owns, visits, precedes, ...
* Properties: blue, oval, even, large, ...
* Functions: father-of, best-friend, second-half, one-more-than ...

User Provides

Constant symbols, which represent individuals in the world

* Mary
* 3
* Green

Function symbols, which map individuals to individuals

* father-of(Mary) = John
* color-of(Sky) = Blue

Predicate symbols, which map individuals to truth values

* greater(5,3)
* green(Grass)
* color(Grass, Green)

FOL Provides

Variable symbols

* E.g., x, y, foo

Connectives

* Same as in predicate logic: not (), and (), or (V), implies (🡪), if and only if (biconditional )

Quantifiers

* Universal x or (Ax)
* Existential x or (Ex)

Sentences built from Terms and Atoms

A term (denoting a real-world individual) is a constant symbol, a variable symbol, or an n-place function of n terms.

* x and f(x1 , ..., xn ) are terms, where each xi is a term.
* A term with no variables is a ground term

An atomic sentence (which has value true or false) is an n-place predicate of n terms

A complex sentence is formed from atomic sentences connected by the logical connectives:

* ¬P, P∨Q, P∧Q, P→Q, P↔Q where P and Q are sentences

A quantified sentence adds quantifiers ∀ and ∃

A well-formed formula (wff) is a sentence containing no “free” variables. That is, all variables are “bound” by universal or existential quantifiers. You can have no variables too.

* (∀x)P(x,y) has x bound as a universally quantified variable, but y is free.

Quantifiers

Universal quantification

* (∀x)P(x) means that P holds for all values of x in the domain associated with that variable
* E.g., (∀x) dolphin(x) → mammal(x) ------> quantified complex sentence

Existential quantification

* (∃ x)P(x) means that P holds for some value of x in the domain associated with that variable
* E.g., (∃ x) mammal(x) ∧ lays-eggs(x)
* Permits one to make a statement about some object without naming it

Translating English to FOL

Text

Description automatically generated

First 🡪 x: variable, Sun:entity/object, gardener and likes: predicates, x is universally quantified 🡪 (WFF)

First-Order Predicate Calculus: Example

Text

Description automatically generated with medium confidence

successor is function, it generates another object

natural is predicate

0 is object

natural(0) 🡪 fact, WFF

example queries:

* natural(5)?
* natural(x)? ----> x is a variable. For this to be true, I need to have at least one instance of x that should be true. You can already say that natural(0) is true so if I replace x with 0, then I will have an answer to this one and I will say this query is true when x is 0. What if I try 1? You can use second rule, if 0 is a natural number, 1 is supposed to be a natural number as well so there will be answer for this 1 as well. 0, 1, 2, … are all answers.
  + This particular query needs to give me as many answers as possible because it’s existantially quantified. So I don’t say it is just 1. If the successor function goes to infinity, if you can have x replaced by an infinite number of entities, then this query will go till the end. So this gives all the numbers possible in this domain -> infinitely many. My compiler when answering this question running off my query will fail.

Text

Description automatically generated

father(bill, jill) 🡪 bill is jill’s father  
mother(jill, sam) 🡪 jill is sam’s mother

There are 4 people in this domain of universe that contains these 4 objects.

Program has data and relationships or facts about those data.

example queries:

* parent(jill, sam)?
* parent(x, sam)? ----> I am asking who are the parents of sam using x (variable). This quary should give me 2 answers. If sam has a parent in my domain then x should be at least existentially quantified. One or more people being sam’s parent would give me a true answer for this query. If there are no people that is in my domain that can be designated as sam’s parent, then this query will give me an answer “no”.

We can use quaries to find out, at least in the query perspective, an existential quantification of that variable will give us a “true” answer. Existential quantifier says there is at least one or more (maybe two, three, four, a lot, infinite) answer to this question.



Background pattern

Description automatically generated with low confidence

factorial(0,1) 🡪 means 0’s factorial is 1

factorial is predicate, relation that has 2 inputs

example quaries:

* factorial(3, 6)? --> Is 6 3’s factorial?
* factorial(3, x)? --> There has to be at least one x that this query should be true. If you find that x making this true at least one, report it to me. Maybe 2, report these 2 to me. Maybe 3, … report them all to me. This is the way that you can use predicate calculus to answer quaries.

Program 🡪 set of axioms  
Query 🡪 running of a program

First-Order Predicate Calculus: Statements

Symbols in statements:

Constants (a.k.a. atoms)

* numbers (e.g., 0) or names (e.g., bill).

Predicates

* Boolean functions (true/false) . Can have arguments. (e.g. parent(X,Y)).

Functions

* non-Boolean functions (successor(X) ).
* map constant to another constant

Variables

* e.g., X.

Connectives (operations)

* and, or, not
* implication (🡪):a🡪b (b or not a)
* equivalence (⬄) : a ⬄b (a🡪b and b🡪a)

Quantifiers

* universal quantifier "for all“ ∀
* existential quantifier "there exists" ∃
* bound variable (a variable introduced by a quantifier)
* free variable

Punctuation symbols

* parentheses (for changing associativity and precedence.)
* comma
* period

Arguments to predicates and functions can only be terms:

* Contain constants, variables, and functions.
* Cannot have predicates, qualifiers, or connectives.

Problem Solving

Program = Data + Algorithms 🡪 Imperative programming / Procedural programming

* Data and algorithms are described by the language
* Process data with algorithms

Program = Object.Message(Object) 🡪 OOP

Program = Functions of Functions 🡪 Functional

Algorithm = Logic + Control

* We describe logic, system proves or disproves it

Programmers: facts/axioms/statements Logic programming systems: prove goals from axioms

We specify the logic itself, the system proves.

* Not totally realized by logic programming languages. Programmers must be aware of how the system proves, in order to write efficient, or even correct programs.

Prove goals from facts:

* Resolution and Unification

Proving Things

A proof is a sequence of sentences, where each sentence is either a premise (axiom/fact) or a sentence derived from earlier sentences in the proof by one of the rules of inference.

The last sentence is the theorem (also called goal or query) that we want to prove.

Horn Clause

First-order logic too complicated for an effective logic programming system.

Horn Clause: a fragment of first-order logic

Graphical user interface, application, Word

Description automatically generated

Variables in head: universally quantified

Variables in body only: existentially quantified

Need “or” in head? Multiple clauses

If body is true, head is true as well.

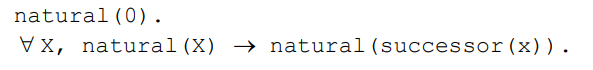
fact: head without body

query: body without head

When body is true, head must be true.

Horn Clause: Example

First-Order Logic:



Horn Clause:

A screenshot of a computer

Description automatically generated with low confidence

natural(0) : horn clause with just head, it is a fact, always true

natural(0) OR natural(successor(x)) 🡨 natural(X).

good(X). : all x is good, everybody is good

if successor of x is natural, then x is natural.

Horn Clause: Example

First-Order Logic:



Horn Clause:



Horn Clause: Example

Horn Clause:

Text, letter

Description automatically generated

Horn Clause: Example

First-Order Logic:



Horn Clause:

A picture containing text

Description automatically generated

**PROLOG SYNTAX**

:- for 🡨

, for and

Text

Description automatically generated

Prolog BNF Grammar

Graphical user interface, text, application

Description automatically generated

Start symbol: <program>

natural(0).

natural(successor(x)) :- natural(x).

?- natural(2). -----> query

* I can say true or false
* Try to see if this is already existing in my list of basic facts, match it with head.
* Look first one (natural(0)). predicate matches but argument doesn’t match.
* We need 2 things:
  + successor(x) part is the part I need to match with 2.
  + natural(x) part (body part) needs to be true.
* Eğer bu ikisi olmazsa kanıtlayamayız.
* Associate successor(x) to 2 with a universal quantification, I have to be careful. For this we have 2 things:
  + Resolution
  + Unification

?- natural(x).

Resolution and Unification

Resolution

**Resolution**: Using a clause, replace its head in the second clause by its body, if they “match”.

Chart, scatter chart

Description automatically generated

***Problem solving in logic programming systems:***

Program:

* Statements/Facts (clauses).

Goals:

* Headless clauses, with a list of subgoals.

Problem solving by resolution:

* Matching subgoals with the heads in the facts, and replacing the subgoals by the corresponding bodies.
* Cancelling matching statements.
* Recursively do this, till we eliminate all goals. (Thus original goals proved.)

Example:  
Graphical user interface, application

Description automatically generated

Goal is query.

Is goal matching with anything in head?

Example:

Graphical user interface, application

Description automatically generated

legs(horse, 4). query match with legs(X, 4) head.

I have a variable X. For this we are gonna use unification. We take horse as X:

* mammal(horse), arms(horse, 0).
  + mammal(horse). is one of my facts so I can prove it
  + arms(horse, 0). is one of my facts so I can prove it

This is unification and resolution.

* I unify my variable with horse
* I can do this because head says it is for any X. horse is one of the Xs.

Unification

**Unification**: Pattern matching to make statements identical (when there are variables).

Set variables equal to patterns: instantiated.

In previous example:

* legs(X,4) and legs(horse,4) are unified.
* (X is instantiated with horse.)

Once this is done, you can do the resolution to replace that predicate with the body part.

Example: Euclid’s algorithm for GCD

Graphical user interface

Description automatically generated with low confidence

Text

Description automatically generated with medium confidence

existentially quantified

true when X is 5.

Things unspecified

The order to resolve subgoals.

The order to use clauses to resolve subgoals.

Possible to implement systems that don’t depend on the order, but too inefficient.

Thus programmers must know the orders used by the language implementations. (Search Strategies)

Example:

Graphical user interface, text, application

Description automatically generated

Prolog Search Strategy

Applies resolution in strictly linear fashion

* Replacing goals left to right
* Considering clauses top to bottom order
* A depth-first search on a tree of possible choices…

Text

Description automatically generated with medium confidence

Prolog Loops and Controls

A picture containing graphical user interface

Description automatically generated

Diagram

Description automatically generated