Logical programming languages implementation

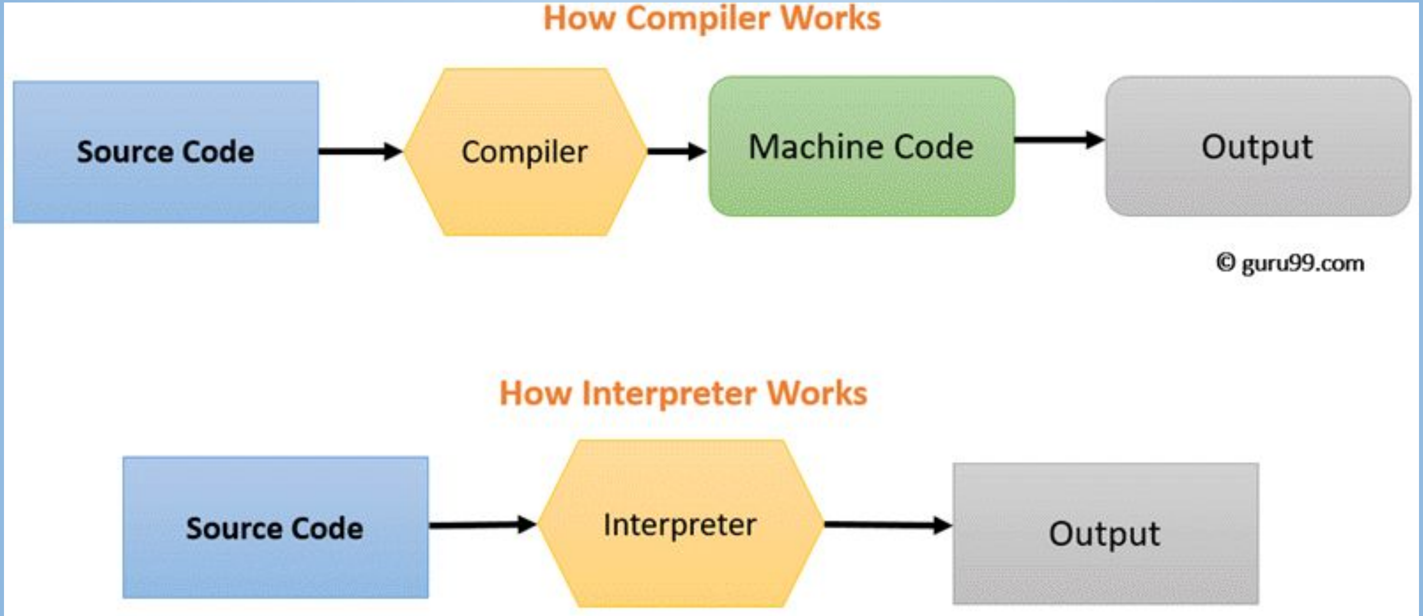
Speakers: Joel Aumedes and Marc Cervera

ESCOLA POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA

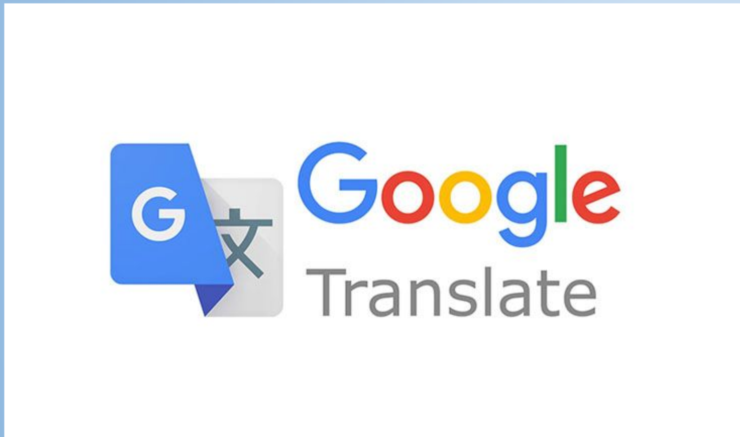
Index of contents

Introduction

- Implementation of a programming language → is which provides a way to execute a program in a concrete combination of software and hardware.
- There exist two ways of implement a programming language → compilation, interpretation.
- Focus → PROLOG
- PROLOG → logic programming language associated with AI and computational linguistics.
- PROLOG → roots in CP1, and unlike many other programming languages, PROLOG is intended as a declarative programming language.
- Logical programming → programming paradigm which is based on formal logic.


How Compiler Works
Source Code
Compiler
Machine Code
Output
© guru99.com
How Interpreter Works
Source Code
Interpreter
Output

PROLOG, compiled or interpreted?



Compiler

Interpreter

PROLOG's interpreter program

SWI Prolog

PROLOG's implementation: ISO PROLOG

ISO = International organization of standarization

The ISO PROLOG has two parts	
ISO/IEC 13211-1	ISO/IEC 13211-2

PROLOG's implementation: Compilation

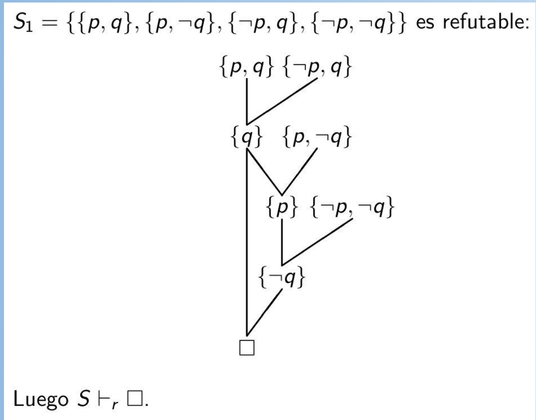
Abstract machine code

- Some implementations employ abstract interpretation to derive type and mode information of predicates at compile time, or compile to real machine code for high performance.
- Devising efficient implementation methods for PROLOG code is a field of active research in the logic programming community, and various other execution methods are employed in some implementations.

PROLOG's implementation: Tail recursion

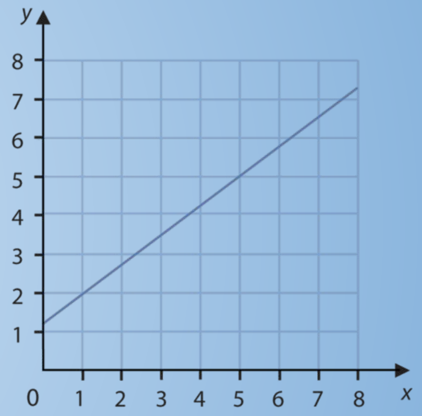
- PROLOG systems typically implement a well-known optimization method called tail call optimization for deterministic predicates exhibiting tail recursion or, more generally, tail calls: A clause's stack frame is discarded before performing a call in a tail position.
    - Tail calls can be implemented without adding a new stack frame to the call stack.
    - Most of the frame of the current procedure is no longer needed, and can be replaced by the frame of the tail call, modified as appropriate.
    - The program can then jump to the called subroutine.
    - Producing such code instead of a standard sequence is called tail call optimization.
- Therefore, deterministic tail-recursive predicates are executed with constant stack space, like loops in other languages.

PROLOG's implementation: Term indexing



Find clauses that can be unified with a term in a query

The query has a linear time



- Term indexing uses a data structure that allows sublinear time searches.
- Indexing only affects program performance, not semantics.
- Most PROLOGs only use indexing in the first term, as the indexing of all terms is expensive, but techniques based on field-encoded words or overlapping code words provide quick indexing to the entire query and head.

PROLOG's implementation: Hashing

Hashing implementation to help to handle big data sets more efficiently

Performance increase

PROLOG's implementation: Tabling

Tabling memorization system

Tabling = spacetime compensation

PROLOG's implementation: Tabling

Given the following CP1 logic program P (remember, all the variables are assumed to be universally quantified):

$$P = \begin{cases} 1.\ q(a) \\ 2.\ p(X) \leftarrow q(X) \end{cases}$$

Then, for the question:

$$P \models p(a) \ \equiv \ P \cup \{\neg p(a)\} \text{ is UNSAT}$$

a SLD resolution proof is as follows:

$$\begin{array}{l} \leftarrow p(a) \\ \quad \mid \; 2.\{a/X_1\} \\ \leftarrow q(a) \\ \quad \mid \; 1.\{\} \\ \quad \square \end{array}$$

Steps in the SLD resolution proof

1. From the top goal $\{\neg p(a)\}$ and the clause $\{p(X_1), \neg q(X_1)\}$ we obtain the resolvent (new goal) $\{\neg q(a)\}$ using the mgu $\theta = \{a/X_1\}$.
2. Then, from the new goal $\{\neg q(a)\}$ and the clause $\{q(a)\}$ we obtain the resolvent $\{\}$ (empty clause) using the mgu $\theta = \{\}$.

PROLOG's implementation: Implementation in hardware

- During the fifth generation computer systems project, there attempt to implement PROLOG in hardware with the aim of achieving faster execution with dedicated architectures.
- Furthermore, PROLOG has a number of properties that may allow speed-up through parallel execution.
- A more recent approach has been to compile restricted PROLOG programs to a field programmable gate array (FPGA).
  - FPGA is an integrated circuit designed to be configured by a customer or a designer after manufacturing.
- However, rapid progress in general-purpose hardware has consistently overtaken more specialised architectures.

Limitations

- PROLOG and other logic programming languages have not had a significant impact on the computer industry in general.
- Most apps are small by industrial standards.
- “Programming in the large” is considered to be complicated because not all PROLOG compilers support modules (compatibility problems).
- Portability of PROLOG code across implementations
- Software developed in PROLOG has been criticised for having a high performance penalty.
- PROLOG is not pure declarative.

Extensions

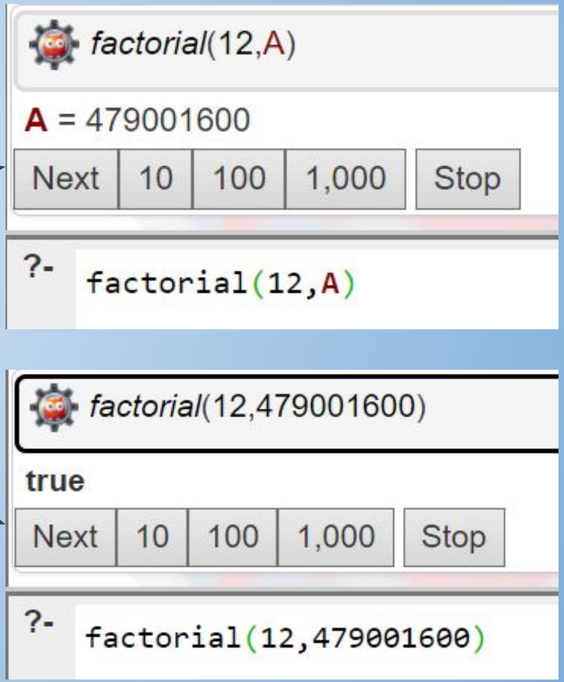
- Various implementations have been developed from PROLOG to extend logic programming capabilities in numerous directions.
  - The extensions are:
    - Types.
    - Modes.
    - Constraints.
    - Object-orientation.
    - Graphics.
    - Concurrency.
    - Web programming.

PROLOG examples

```
main() :- write('Hello world\n').
```

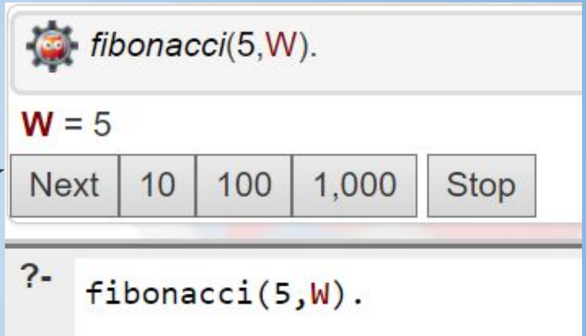
PROLOG examples

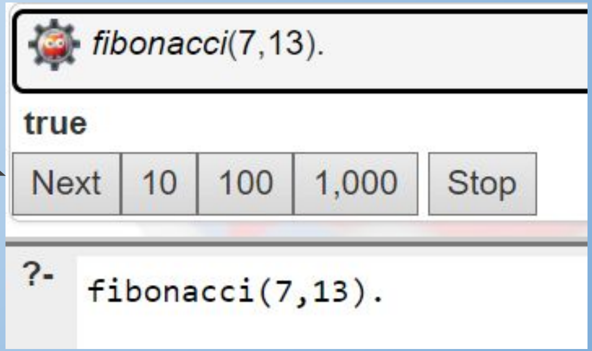
```
factorial(0,1).
factorial(N,F) :-
    N>0,
    N1 is N-1,
    factorial(N1,F1),
    F is N * F1.
```



PROLOG examples

```
fibonacci(0,0).
fibonacci(1,1).
fibonacci(2,1).
fibonacci(N,F) :-
    N>2,
    N1 is N-1,
    N2 is N-2,
    fibonacci(N1, F1),
    fibonacci(N2, F2),
    F is F1+F2.
```





Related languages

VISUAL PROLOG

datalog 5

MERCURY

Graphtalk

PLANNER

ERLANG