

Introduction to Logic Programming in Prolog

Outline

Programming paradigms

Logic programming basics

- Introduction to Prolog

- Predicates, queries, and rules

Understanding the query engine

- Goal search and unification

- Structuring recursive rules

Complex terms, numbers, and lists

Cuts and negation

What is a programming paradigm?

Paradigm

adapted from Oxford American

A conceptual model underlying the theories and practice of **a scientific subject**

scientific subject = programming

Programming paradigm

A conceptual model underlying the theories and practice of **programming**

Common programming paradigms

Paradigm	View of computation
imperative	sequence of state transformations
object-oriented	simulation of interacting objects
functional	function mapping input to output
logic	queries over logical relations

Outline

Programming paradigms

Logic programming basics

Introduction to Prolog

Predicates, queries, and rules

Understanding the query engine

Goal search and unification

Structuring recursive rules

Complex terms, numbers, and lists

Cuts and negation

What is Prolog?



SWI Prolog

- an **untyped logic** programming language
- programs are **rules** that define **relations** on values
- run a program by formulating a **goal** or **query**
- result of a program: a true/false answer and a **binding of free variables**

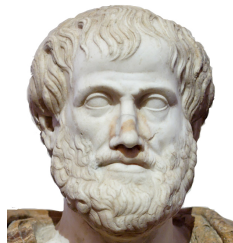
Logic: a tool for reasoning

Syllogism (logical argument) – Aristotle, 350 BCE

Every human is mortal.

Socrates is human.

Therefore, Socrates is mortal.



First-order logic – Gottlob Frege, 1879 *Begriffsschrift*

$\forall x. \text{Human}(x) \rightarrow \text{Mortal}(x)$

$\text{Human}(\text{Socrates})$

$\therefore \text{Mortal}(\text{Socrates})$

Logic and programming

rule $\forall x. \text{Human}(x) \rightarrow \text{Mortal}(x)$

fact $\text{Human}(\text{Socrates})$

goal/query $\therefore \text{Mortal}(\text{Socrates})$

} logic program

} logic program execution

Prolog program

```
mortal(X) :- human(X).  
human(socrates).
```

Prolog query (interactive)

```
?- mortal(socrates).  
true.
```


Outline

Programming paradigms

Logic programming basics

Introduction to Prolog

Predicates, queries, and rules

Understanding the query engine

Goal search and unification

Structuring recursive rules

Complex terms, numbers, and lists

Cuts and negation

SWI-Prolog logistics

Predicate	Description
<code>[myfile].</code>	load definitions from “myfile.pl”
<code>listing(P).</code>	lists facts and rules related to predicate P
<code>trace.</code>	turn on tracing
<code>nodebug.</code>	turn off tracing
<code>help.</code>	view documentation
<code>halt.</code>	quit

Atoms

An **atom** is just a primitive value

- string of characters, numbers, underscores starting with a **lowercase letter**:
 - **hello, socrates, sUp3r_At0m**
- any single quoted string of characters:
 - **'Hello world!', 'Socrates'**
- numeric literals: **123, -345**
- empty list: **[]**

Variables

A **variable** can be used in rules and queries

- string of characters, numbers, underscores starting with an **uppercase letter** or an **underscore**
 - **X**, **SomeHuman**, **_g_123**
- special variable: **_** (just an underscore)
 - unifies with anything – “don’t care”

Predicates

Basic entity in Prolog is a **predicate** \cong **relation** \cong **set**

Unary predicate

```
hobbit(bilbo).  
hobbit(frodo).  
hobbit(sam).
```

hobbit = {bilbo, frodo, sam}

Binary predicate

```
likes(bilbo, frodo).  
likes(frodo, bilbo).  
likes(sam, frodo).  
likes(frodo, ring).
```

**likes = { (bilbo, frodo), (frodo, bilbo)
(sam, frodo), (frodo, ring) }**

Simple goals and queries

Predicates are:

- **defined** in a file *the program*
- **queried** in the REPL *running the program*

Response to a query is a **true/false** answer

when **true**, provides a **binding** for each variable in the query

Is **sam** a hobbit?

```
?- hobbit(sam).  
true.
```

Is **gimli** a hobbit?

```
?- hobbit(gimli).  
false.
```

Who is a hobbit?

```
?- hobbit(X).  
X = bilbo ;  
X = frodo ;  
X = sam .
```

Type ; after each response to search for another

Querying relations

You can query **any argument** of a predicate

- this is fundamentally different from passing arguments to functions!

Definition

```
likes(bilbo, frodo).  
likes(frodo, bilbo).  
likes(sam, frodo).  
likes(frodo, ring).
```

```
?- likes(frodo,Y).  
Y = bilbo ;  
Y = ring .
```

```
?- likes(X,frodo).  
X = bilbo ;  
X = sam .
```

```
?- likes(X,Y).  
X = bilbo,  
Y = frodo ;  
X = frodo,  
Y = bilbo ;  
X = sam,  
Y = frodo ;  
X = frodo,  
Y = ring .
```

Overloading predicate names

Predicates with the **same name** but **different arities** are **different predicates**!

hobbit/1

```
hobbit(bilbo).  
hobbit(frodo).  
hobbit(sam).
```

```
?- hobbit(X).  
X = bilbo ;  
X = frodo ;  
X = sam.
```

hobbit/2

```
hobbit(bilbo, rivendell).  
hobbit(frodo, hobbiton).  
hobbit(sam, hobbiton).  
hobbit(merry, buckland).  
hobbit(pippin, tookland).
```

```
?- hobbit(X,_).  
...  
X = merry ;  
X = pippin .
```


Conjunction

Comma (,) denotes **logical and** of two predicates

Do **sam** and **frodo** like each other?

```
?- likes(sam,frodo), likes(frodo,sam).  
true.
```

Do **merry** and **pippin** live in the same place?

```
?- hobbit(merry,X), hobbit(pippin,X).  
false.
```

Do any hobbits live in the same place?

```
?- hobbit(H1,X), hobbit(H2,X), H1 \= H2.  
H1 = frodo, X = hobbiton, H2 = sam.
```

```
likes(frodo, sam).  
likes(sam, frodo).  
likes(frodo, ring).
```

```
hobbit(frodo, hobbiton).  
hobbit(sam, hobbiton).  
hobbit(merry, buckland).  
hobbit(pippin, tookland).
```

H1 and H2 must be different!

Rules

Rule: **head** :- **body**

The **head** is true **if** the **body** is true

Examples

```
likes(X,beer)  :- hobbit(X,_).  
likes(X,boats) :- hobbit(X,buckland).  
  
danger(X) :- likes(X,ring).  
danger(X) :- likes(X,boats), likes(X,beer).
```

Note that **disjunction** is described by multiple rules

Outline

Programming paradigms

Logic programming basics

Introduction to Prolog

Predicates, queries, and rules

Understanding the query engine

Goal search and unification

Structuring recursive rules

Complex terms, numbers, and lists

Cuts and negation

How does Prolog solve queries?

Basic algorithm for solving a (sub)goal

1. Linearly **search** database for candidate facts/rules
2. Attempt to **unify** candidate with goal
If unification is **successful**:
 - if a **fact** – we're done with this goal!
 - if a **rule** – add body of rule as new subgoalsIf unification is **unsuccessful**: keep searching
3. Backtrack if we reach the end of the database

1. Search the database for candidate matches

What is a candidate fact/rule?

- **fact**: predicate matches the goal
- **rule**: predicate of its **head** matches the goal

Example goal: `likes(merry,Y)`

Candidates

```
likes(sam,frodo).  
likes(merry,pippin).  
likes(X,beer) :- hobbit(X).
```

Not candidates

```
hobbit(merry,buckland).  
danger(X) :- likes(X,ring).  
likes(merry,pippin,mushrooms).
```

2. Attempt to unify candidate and goal

Unification

Find an **assignment of variables** that makes its arguments **syntactically equal**

Prolog: $A = B$ means attempt to **unify** A and B

```
?- likes(merry,Y) = likes(sam,frodo).  
false.
```

```
?- likes(merry,Y) = likes(merry,pippin).  
Y = pippin .
```

```
?- likes(merry,Y) = likes(X,beer).  
X = merry ; Y = beer .
```

2a. if **fail**, try next candidate

2b. if **success**, add new subgoals

Tracking subgoals

Deriving solutions through rules

1. Maintain a list of goals that need to be solved
 - when this list is empty, we're done!
2. If current goal unifies with a rule **head**, add **body** as subgoals to the list
3. After each unification, **substitute variables** in all goals in the list!

Database

```
1 lt(one,two).  
2 lt(two,three).  
3 lt(three,four).  
4 lt(X,Z) :- lt(X,Y), lt(Y,Z).
```

Sequence of goals for `lt(one,four)`

	<code>lt(one,four)</code>
4: <code>X=one,Z=four</code>	<code>lt(one,Y1), lt(Y1,four)</code>
1: <code>Y1=two</code>	<code>lt(two,four)</code>
4: <code>X=two,Z=four</code>	<code>lt(two,Y2), lt(Y2,four)</code>
2: <code>Y2=three</code>	<code>lt(three,four)</code>
3: <code>true</code>	done!

3. Backtracking

For each subgoal, Prolog maintains:

- the **search state** (goals + assignments) before it was produced
- a **pointer** to the rule that produced it

When a **subgoal fails**:

- **restore** the previous state
- **resume** search for previous goal from the pointer

When the **initial goal fails**: return **false**

Outline

Programming paradigms

Logic programming basics

Introduction to Prolog

Predicates, queries, and rules

Understanding the query engine

Goal search and unification

Structuring recursive rules

Complex terms, numbers, and lists

Cuts and negation

Potential for infinite search

Why care about how goal search works?

One reason: to write **recursive rules** that don't loop!

Bad example: symmetry

```
married(abe,mona).  
married(clancy,jackie).  
married(homer,marge).  
married(X,Y) :- married(Y,X).
```

```
?- married(jackie,abe).
```

ERROR: Out of local stack

Bad example: transitivity

```
lt(one,two).  
lt(two,three).  
lt(three,four).  
lt(X,Z) :- lt(X,Y), lt(Y,Z).
```

```
?- lt(three,one).
```

ERROR: Out of local stack

Strategies for writing recursive rules

How to avoid infinite search

1. Always list **non-recursive cases first** (in database and rule bodies)
2. Use helper predicates to **enforce progress** during search

Example: symmetry

```
married_(abe,mona).  
married_(clancy,jackie).  
married_(homer,marge).  
married(X,Y) :- married_(X,Y).  
married(X,Y) :- married_(Y,X).
```

```
?- married(jackie,abe).  
false.
```

Example 2: transitivity

```
lt_(one,two).  
lt_(two,three).  
lt_(three,four).  
lt(X,Y) :- lt_(X,Y).  
lt(X,Z) :- lt_(X,Y), lt(Y,Z).
```

```
?- lt(three,one).  
false.
```

Outline

Programming paradigms

Logic programming basics

Introduction to Prolog

Predicates, queries, and rules

Understanding the query engine

Goal search and unification

Structuring recursive rules

Complex terms, numbers, and lists

Cuts and negation

Representing structured data

Can represent structured data by **nested predicates**

Example database

```
drives(bart, skateboard(green)).  
drives(bart, bike(blue)).  
drives(lisa, bike(pink)).  
drives(homer, car(pink)).
```

```
?- drives(lisa, X).  
X = bike(pink) .
```

```
?- drives(X, bike(Y)).  
X = bart, Y = blue ;  
X = lisa, Y = pink .
```

Variables can't be used for predicates:

```
?- drives(X, Y(pink)). ← illegal!
```

Relationship to Haskell data types

Haskell data type

```
data Expr = Lit Int
          | Neg Expr
          | Add Expr Expr
          | Mul Expr Expr
```

```
Add (Neg (Lit 3))
     (Mul (Lit 4) (Lit 5))
```

- build values w/ data constructors
- data types statically define valid combinations

Prolog predicate

```
expr(N)      :- number(N).
expr(neg(E)) :- expr(E).
expr(add(L,R)) :- expr(L), expr(R).
expr(mul(L,R)) :- expr(L), expr(R).
```

```
add(neg(3),mul(4,5))
```

- build values w/ predicates
- use rules to dynamically identify or enumerate valid combinations

Lists in Prolog

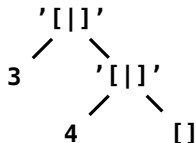
Lists are structured data with special syntax

- similar basic structure to Haskell
- but can be heterogeneous

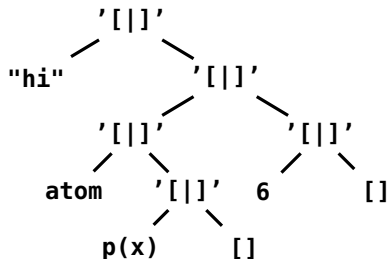
`[3,4] ≡ '[]'(3, '[]'(4, []))`

cons ↗

nil ↗



`["hi", [atom, p(x)], 6]`



List patterns

[X|Y]

head
tail



Database

```
story([3,little,pigs]).
```

```
?- story([X|Y]).  
X = 3,  
Y = [little, pigs].
```

```
?- story([X,Y|Z]).  
X = 3,  
Y = little,  
Z = [pigs].
```

```
?- story([X,Y,Z|V]).  
X = 3,  
Y = little,  
Z = pigs,  
V = [].
```

```
?- story([X,Y,Z]).  
X = 3,  
Y = little,  
Z = pigs.
```

```
?- story([X,Y,Z,V]).  
false.
```


Arithmetic in Prolog

Arithmetic expressions are also structured data (nested predicates)

- special syntax: can be written infix, standard operator precedence
- can be evaluated:

`X is expr` evaluate `expr` and bind to `X`

`expr ::= expr` evaluate expressions and check if equal

`3*4+5*6 ≡ +(*(3, 4), *(5, 6))`

`?- X is 3*4+5*6.`

`X = 42.`

`?- 8 is X*2.`

**ERROR: is/2: Arguments are not
sufficiently instantiated**

Arithmetic operations

`+` `-` `*` `/` `mod`

Comparison operations

`<` `>` `=<` `>=` `::=` `=\=`

Using arithmetic in rules

Example database

```
fac(1,1).  
fac(N,M) :- K is N-1, fac(K,L), M is L*N.
```

```
?- fac(5,M).  
M = 120.
```

```
?- fac(N,6).  
ERROR: fac/2: Arguments are not sufficiently instantiated
```

Unification vs. arithmetic equality

Unification: $A = B$

Find an **assignment of variables** that makes its arguments **syntactically equal**

Arithmetic equality: $A ::= B$

Evaluate terms as **arithmetic expressions** and check if **numerically equal**

```
?- X = 3*5.  
X = 3*5.
```

```
?- 8 = X*2.  
false.
```

```
?- 4*2 = X*2.  
X = 4.
```

```
?- X is 3*5.  
X = 15.
```

```
?- 8 is X*2.  
ERROR: is/2: Arguments are not  
sufficiently instantiated
```

Outline

Programming paradigms

Logic programming basics

Introduction to Prolog

Predicates, queries, and rules

Understanding the query engine

Goal search and unification

Structuring recursive rules

Complex terms, numbers, and lists

Cuts and negation

How cut works

Cut is a special atom ! used to prevent backtracking

When encountered as a subgoal it:

- always succeeds
- commits the current goal search to the matches and assignments made so far

Database without cut

```
foo(1). foo(2).  
bar(X,Y) :- foo(X), foo(Y).  
bar(3,3).
```

```
?- bar(A,B).  
A = 1, B = 1 ;    A = 1, B = 2 ;  
A = 2, B = 1 ;    A = 2, B = 2 ;  
A = 3, B = 3.
```

Database with cut

```
foo(1). foo(2).  
bar(X,Y) :- foo(X), !, foo(Y).  
bar(3,3).
```

```
?- bar(A,B).  
A = 1, B = 1 ;  
A = 1, B = 2.
```

Green cuts vs. red cuts

A **green cut**: doesn't affect the members of a predicate

- only cuts paths that would have failed anyway
- the cut is used purely for efficiency

```
max(X,Y,Y) :- X < Y, !.  
max(X,Y,X) :- X >= Y.
```

A **red cut**: any cut that isn't green

- if removed, meaning of the predicate changes
- the cut is part of the “logic” of the predicate

```
find(X,[X|_]) :- !.  
find(X,[_|L]) :- find(X,L).
```

Negation as failure

Negation predicate

```
not(P) :- P, !, fail.  
not(P).
```

if **P** is true, commit that **not(P)** is false
otherwise, **not(P)** is true

Database

```
hobbit(frodo).  
hobbit(bilbo).  
likes(X,beer)  
:- hobbit(X).
```

```
?- not(likes(frodo,beer)).  
false.
```

“frodo doesn’t like beer”

```
?- not(likes(gimli,beer)).  
true.
```

“gimli doesn’t like beer”

```
?- not(likes(bilbo,X)).  
false.
```

“bilbo doesn’t like anything”

```
?- not(likes(X,pizza)).  
true.
```

“nobody likes pizza”