Programming Paradigms

Unit 6 — Prolog Basics

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

1. Basics of Prolog

2. Backtracking

3. Rules

Outline

Logic Programming

- Logic programming is a programming paradigm based on formal logic
- Programming languages for logic programming are very different to those encountered so far
- They are declarative languages
  - Programmer defines "what" to do, program will figure out "how" to do it
  - In other words, you express the logic of a computation without describing its control flow
  - In imperative and object-oriented languages, the programmer has to do both
- A program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain

Prolog

- One of the most well-known logic programming languages is Prolog
    - Stands for PROgramming in LOGic
    - Developed by Alain Colmerauer and colleagues in Marseille in the early 1970s
- Prolog is very useful in some problem areas
    - such as artificial intelligence, natural language processing, databases, . . .
- But pretty useless in others
    - such as for instance graphics or numerical algorithms
- Major logic programming language families include Prolog, Answer set programming (ASP) and Datalog.

Hello World!

- Start a Prolog interpreter, e.g., `gprolog` for GNU Prolog or `swipl` for SWI Prolog
- The Prolog Interpreter shows a prompt and is ready to accept programs

  ```
  ?-
  ```
- Let's have a look at a very simple program: Hello World!

  ```
  ?- write('Hello World!'), nl.
  ```
- This will output (may also say `yes` instead of `true`):

  ```
  Hello World!
  true
  ```
- Although this works, it's a very untypical example of a Prolog program

Language Basics: Data and Queries

- Prolog has two parts
  - One to represent the data
  - Another to query the data
- Data is represented in the form of facts and logical rules
  - Facts: a fact is a basic assertion about some world
    - Mary is a student
    - Students like books
  - Rules: a rule is an inference about facts in that world
    - A person likes books if she/he is a student
- Query: a query is a question about that world
  - Does Mary like books?

Knowledge Base

- Facts and rules go into a knowledge base
  - Prolog allows you to express the contents of a knowledge base
  - Usually a compiler turns this base into a form efficient for querying
- Querying links together facts and rules to tell you something about the world modeled in the knowledge base

Facts

- Facts are basic assertions/statements about objects in the world

- Consider the animation series "Wallace and Gromit"
  - Wallace is a good, eccentric cheese loving inventor
  - Gromit is a silent yet intelligent anthropomorphic dog
  - ...

- This is represented in a little Prolog knowledge base of five facts

```
likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).
```

- The facts can be read as the following assertions about the world
  - "Wallace likes toast"
  - "Wallace likes cheese"
  - ...

Atoms

- Atoms refer to individual things/objects
  - e.g., `wallace`, `gromit`, `wendolene`, `toast`, `cheese`, `cake`, and `sheep` in the facts on the previous slide are atoms
- Atoms always
  - begin with a lower-case character, e.g., `wallace`
  - or they have to be quoted, e.g., 'Wallace'
- An atom is a fixed value, similar to a Ruby symbol

Predicates

- Predicates represent relationships between objects in the world
- In the facts on the previous example, `likes` is a predicate
    - e.g., `likes(gromit, cheese)`
- The predicate `likes` has two parameters
- The order of the parameters is important

Compiling/Loading the Knowledge Base

- We write the facts in a file "wallace.pl" and load it into the interpreter
  - Checks for syntax errors
  - Compiles the knowledge base into form that is efficient for querying

```
?- ['wallace.pl']
% wallace.pl compiled 0.00 sec, 8 clauses
true.
```

- Now we are ready to ask some questions
- The most basic ones are questions about facts

Queries

- After compilation we can query the Prolog knowledge base
- Prolog tries to match a query to known facts

```
?- likes(gromit,cheese).
true
```

```
?- likes(wallace,sheep).
false
```

Knowledge base

```
likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).
```

- This confirms that gromit likes cheese, but wallace doesn't like sheep
  - `true` means that Prolog is able to prove this statement given the actual knowledge base
  - `false` means that Prolog cannot prove this statement given the actual knowledge base

Variables and Instantiation/1

- Variables make queries more exciting
  - Variables begin with an uppercase letter or with an underscore '_x'
- We can ask Prolog to find values for variables
  - e.g., Who likes cheese?

```
?- likes(Who,cheese).
```

Knowledge base

```
likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).
```

- Who is an uninstantiated variable, i.e., it has no assigned value
- Prolog searches the KB from the beginning, trying to find a matching fact
- The first matching fact is likes(wallace,cheese), so Who is instantiated with wallace

```
?- likes(Who,cheese).
Who = wallace
```

- At this point Prolog outputs Who = wallace and asks what to do next

Variables and Instantiation/2

- We can
  - stop searching by just hitting the return key
  - or continue searching by entering ";"
- If we continue, Prolog
  - forgets the value `wallace` for the variable `Who`
  - and continues at the position it previously stopped (using a placemarker)
- Continuing will output `Who = gromit`; then the query terminates as there will be no further solutions

```
?- likes(Who,cheese).
Who = wallace ;
Who = gromit.
?-
```

Anonymous Variables

- Sometimes we want to use a variable but don't care about its value
  - We don't want to use the variable anywhere else
- e.g., is there anyone who likes cheese (but we don't need to know who)

```
?- likes(_,cheese).
true ?
yes
```

- We use an underscore _ for the anonymous variable
- Several occurrence of _ in the same clause do not need to be given consistent interpretations

Outline

Goals/1

- By submitting a query, we ask Prolog to try to satisfy a goal
- We can ask Prolog to satisfy the conjunction of two goals using the "," operator (pronounced "and")

```
?- likes(wallace,toast), likes(gromit,toast).
no
```

- We can combine conjunctions with variables to make queries more interesting

Goals/2

- Now that we found out that at least one of them does not like toast ...
- ... is there something both of them like?

```
?- likes(wallace,What), likes(gromit,What).
What = cheese;
no
```

- How does Prolog process this query (conceptually)?
- It uses backtracking to try to satisfy the first goal and then the second goal

Backtracking/1

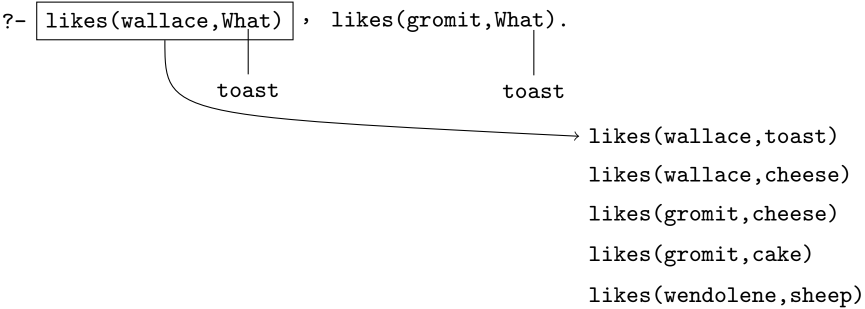
- Process the following query consisting of two goals

```
?- likes(wallace,What) , likes(gromit,What).
       first goal            second goal
```

```
likes(wallace,toast)
likes(wallace,cheese)
likes(gromit,cheese)
likes(gromit,cake)
likes(wendolene,sheep)
```

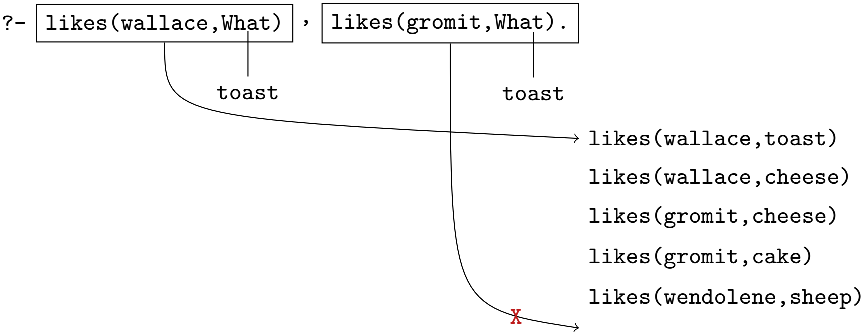
- Attempt to satisfy the first goal

Backtracking/2



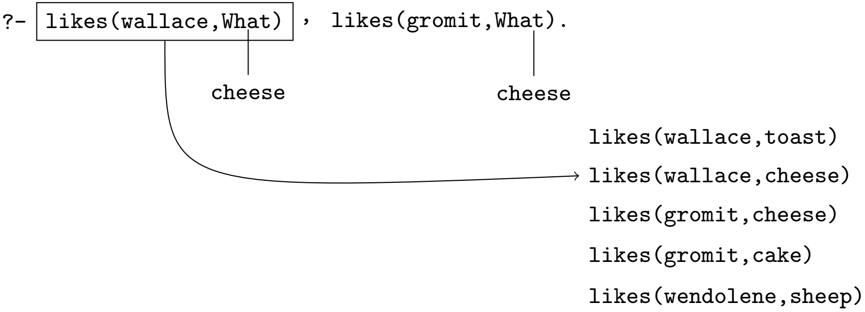
- The first goal succeeds, instantiating `What = toast`
- Next, attempt to satisfy the second goal

Backtracking/3



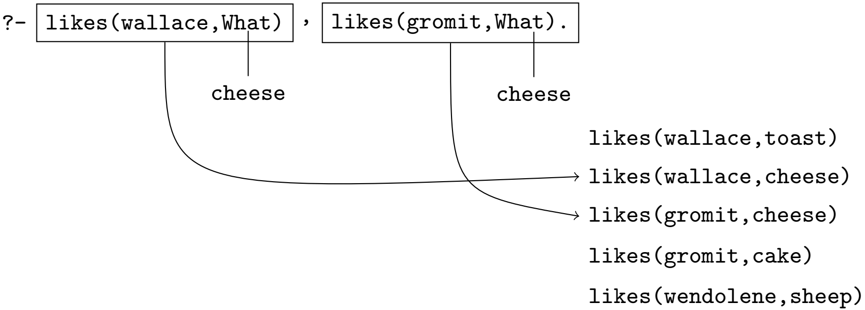
- The (fully instantiated) second goal `likes(gromit,toast)` fails
- Next, backtracking starts: forget the instantiation `What = toast` and attempt to re-satisfy the first goal

Backtracking/4



- The first goal succeeds again, instantiating `What = cheese`
- Next attempt to satisfy the second goal

Backtracking/5



- The second goal succeeds
- Prolog notifies you of success with `What = cheese` and waits for a reply

Outline

Rules/1

- Suppose we want to state that *Wallace likes all people*
- Could be done by many facts

```
likes(wallace, gromit).
likes(wallace, tom).
...
```

- ... but this becomes tedious and long
- Another way to state the same would be:
  Wallace likes any object provided it is a person
- This fact is in the form of a more compact rule

Rules/2

- A rule is a general statement about objects and their relationships
- A rule for being a sister of someone in plain English could be:

  X is a sister of Y if:
  X is female and
  X and Y have the same parents.

- Important: a variable stands for the same object wherever it occurs in a rule

Rules/3

- Rules in Prolog consist of a head and a body connected by the symbol ":-" (pronounced if)

  head :- *body*

- The head is a predicate that describes what the rule is intended to define
- The body is a conjunction of goals that must be satisfied for the head to be true
- In other words: to prove the head, the body needs to be proven

Rules Example

- If we want to express that *Wallace is a friend of anyone who likes cheese*, we could formulate it like this:

```
friend(wallace,X) :- likes(X,cheese).
```

- `X` is a variable and can match any object

Matching Rules/1

- Prolog tries to satisfy the goal by matching it with rules and/or facts in the knowledge base

```
?- friend(wallace,X).
```

X

```
friend(wallace,X) :- likes(X,cheese).

likes(wallace,toast).
likes(wallace,cheese).
likes(gromit,cheese).
likes(gromit,cake).
likes(wendolene,sheep).
```

- After matching the head of the rule, the body of the rule needs to be proven

Matching Rules/2

- Prove the body of the rule

```
?- likes(X,cheese).

   wallace

friend(wallace,X) :- likes(X,cheese).

likes(wallace,toast).
likes(wallace,cheese).
likes(gromit,cheese).
likes(gromit,cake).
likes(wendolene,sheep).
```

- The body of the rule is proven with `X = wallace`, so the head is proven, too, i.e., Wallace is a friend of Wallace
- Backtracking produces the second result, i.e., `X = gromit`.

Rules Example Revisited

- Running the query `friend(wallace,X)` will produce two results:
  - `gromit` is ok
  - `wallace` not really ok
- We can exclude `wallace` by saying that `X` shouldn't be `wallace`

```
friend(wallace,X) :-
    likes(X,cheese),
    \+(X=wallace).
```

- \+ is the negation/logical not of a subgoal (`not` can also be used)
- This only lists friends of `wallace` (those persons who like cheese)

A Generalization of the Rule Example/1

- A generalization of the rule on the previous slides would be: *if X and Y like the same thing Z, and X and Y are different, then X and Y are friends*
- This makes our knowledge base more interesting

```
likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).

friend(X,Y) :-
        likes(X,Z),
        likes(Y,Z),
        \+(X=Y).
```

A Generalization of the Rule Example/2

- Let's try it out with constants in the query

```
?- friend(gromit,wallace).
yes
?- friend(wallace,gromit).
yes
?- friend(wallace,wallace).
no
?- friend(wallace,wendolene).
no
```

A Generalization of the Rule Example/3

- We can also use variables instead of atoms in the query.
- Now let's ask who is a friend of Wallace:

```
?- friend(wallace,Who).
Who = gromit ?  ;
no
```

- Or let's find all pairs of friends:

```
?- friend(Who1,Who2).
Who1 = wallace
Who2 = gromit ?  ;
Who1 = gromit
Who2 = wallace ?  ;
no
```

Queen Victoria's Family/1

- Just using facts, rules, and variables we can already do some interesting things, e.g., model Queen Victoria's family
- We use the predicate `parents(X,Y,Z)` to represent *the parents of X are Y and Z*
- We also use `male(X)` and `female(X)` in the obvious way

```
male(albert).
male(edward).

female(alice).
female(victoria).

parents(edward, victoria, albert).
parents(alice, victoria, albert).
```

Queen Victoria's Family/2

- Now we add a rule for sister: *X is a sister of Y if X is female and they both have the same parents*

```
sister_of(X, Y) :-
        female(X),
        parents(X, M, F),
        parents(Y, M, F).
```

- Now you can query:

```
?- sister_of(alice, edward).
true
```

```
?- sister_of(alice, X).
X = edward
```

Map Coloring/1

- A slightly more complex example
- Assume we want to color a map, such that two regions with a common border don't have the same color

Map Coloring/2

- In order to simplify things, we'll only look at regions 3, 4, 5, and 6 and use the colors red, green, and blue
- Now all we have to do is describe this to Prolog
  - We tell Prolog the different colors to use for pairs of neighoring regions
  - and the neighoring regions

```
border(red,green).
border(red,blue).
border(green,red).
border(green,blue).
border(blue,red).
border(blue,green).

coloring(L,TAA,V,FVG) :-
    border(L,TAA),
    border(L,V),
    border(TAA,V),
    border(V,FVG).
```

Map Coloring/3

- Querying `coloring(L,TAA,V,FVG)` will now provide all the answers:

```
?- coloring(L,TAA,V,FVG).
FVG = r
L = r
TAA = g
V = b ?   ;
FVG = g
L = r
TAA = g
V = b ?
...
```

Where's the Program?

- In Prolog you don't have to write a program
  - You express the logic of a problem in facts and rules
  - And then let the computer do the work in figuring out a solution
- Solving the map coloring problem with a language like Java or Ruby would be (much) harder to do
- Here is the Ruby code

```
def mapcoloring
  colors = [:red, :green, :blue]
  colors.each do |l|
    colors.each do |taa|
      colors.each do |v|
        colors.each do |fvg|
          if l != taa && l != v && taa != v && v != fvg then
            puts "L = #{l}, TAA = #{taa}, V = #{v}, FVG = #{fvg}"
          end
        end
      end
    end
  end
end
```

Predicates Revisited

- Predicates can be defined by a combination of facts and rules
- We use also the term clause of a predicate to refer either to a fact or a rule defining the predicate