Recursion, Structures, and Lists

Artificial Intelligence Programming in Prolog
Lecturer: Tim Smith
Lecture 4
04/10/04
The central ideas of Prolog

- **SUCCESS/FAILURE**
  - any computation can "succeed" or "fail", and this is used as a 'test' mechanism.

- **MATCHING**
  - any two data items can be compared for similarity, and values can be bound to variables in order to allow a match to succeed.

- **SEARCHING**
  - the whole activity of the Prolog system is to search through various options to find a combination that succeeds.
    - Main search tools are **backtracking** and **recursion**

- **BACKTRACKING**
  - when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.
Likes program

1) drinks(alan,beer).
2) likes(alan,coffee).
3) likes(heather,coffee).

4) likes(Person,Drink):-
   drinks(Person,Drink).

5) likes(Person,Somebody):-
   likes(Person,Drink),
   likes(Somebody,Drink).
Representing Proof using Trees

- To help us understand Prolog’s proof strategy we can represent its behaviour using **AND/OR trees**.

1. **Query** is the top-most point (node) of the tree.
2. **Tree** grows downwards (looks more like roots!).
3. **Each branch** denotes a subgoal.
   1. The branch is labelled with the number of the matching clause and
   2. any variables instantiated when matching the clause head.
4. **Each branch ends with either:**
   1. A successful match \( \bigcirc \),
   2. A failed match \( \bigotimes \), or
   3. Another subgoal.

---

\[ \text{?- likes(alan,X).} \]

2 \[ X/\text{coffee} \]

\[ X = \text{coffee} \]

1st solution

= “Alan likes coffee.”
Representing Proof using Trees (2)

- Using the tree we can see what happens when we ask for another match (;

\[?-\;\text{likes(alan,}X).\]

Using the tree we can see what happens when we ask for another match (;

\[X/\text{coffee}\]

\[X = \text{coffee}\]

2nd solution

= “Alan likes beer because Alan drinks beer.”

1st match is failed and forgotten

Backtracking

\[\text{drinks(alan,}X).\]

\[X/\text{beer}\]

\[X = \text{beer}\]
Recursion using Trees

- When a predicate calls itself within its body we say the clause is **recursing**
Recursion using Trees (2)

- When a predicate calls itself within its body we say the clause is **recursing**

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

```
<table>
<thead>
<tr>
<th>X = coffee</th>
</tr>
</thead>
</table>

```

```
<table>
<thead>
<tr>
<th>likes(alan,coffee)</th>
</tr>
</thead>
</table>

```

```
<table>
<thead>
<tr>
<th>drinks(alan,X).</th>
</tr>
</thead>
</table>

```

```
<table>
<thead>
<tr>
<th>X = coffee</th>
</tr>
</thead>
</table>

```

```
<table>
<thead>
<tr>
<th>likes(Somebody,coffee)</th>
</tr>
</thead>
</table>

```

```
<table>
<thead>
<tr>
<th>Somebody = alan</th>
</tr>
</thead>
</table>

```

```
<table>
<thead>
<tr>
<th>X = coffee</th>
</tr>
</thead>
</table>
```

3rd solution = “Alan likes Alan because Alan likes coffee.”
Recursion using Trees (3)

- When a predicate calls itself within its body we say the clause is **recursing**

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

<table>
<thead>
<tr>
<th>2</th>
<th>X/coffee</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>X = coffee</td>
</tr>
<tr>
<td>1</td>
<td>X/beer</td>
</tr>
</tbody>
</table>

```
drinks(alan,X).
likes(alan,coffee)
likes(Somebody,coffee)
```

4th solution = “Alan likes Heather because Heather likes coffee.”
Infinitely Recursive Loop

- If a recursive clause is called with an incorrect goal it will loop as it can neither prove it nor disprove it.

```
likes(Someb, coffee)

Somebody = alan

Somebody = heather

likes(coffee, coffee)
likes(coffee, X)
likes(coffee, X2)
likes(coffee, X3)
likes(X, X2)
likes(X2, X3)
likes(X3, X3)
```

`likes/2` is a left recursive clause.
Why use recursion?

- It allows us to define very **clear** and **elegant code**.
  - Why repeat code when you can reuse existing code.
- Relationships may be recursive
  e.g. “X is my ancestor if X is my Ancestor’s ancestor.”
- Data is represented recursively and best processed iteratively.
  - Grammatical structures can contain themselves
  - E.g. NP → (Det) N (PP), PP → P (NP)
  - Ordered data: each element requires the same processing
- Allows Prolog to perform complex search of a problem space without any dedicated algorithms.
Prolog Data Objects (*Terms*)

- **Simple objects**
  - **Constants**
    - **Atoms**
      - a
      - bob
      - l8r_2day
    - **Integers**
      - -6
      - 987
    - **Symbols**
      - a
      - 'a'
      - 'Bob'
      - 'L8r 2day'
  - **Variables**
    - X
    - A_var
    - _Var
    - A_var
    - person(bob, 48)
    - [a, b, g]
    - [[a], [b]]
    - [bit(a, d), a, 'Bob']
  - **Signs**
    - <---
    - ==>
    - -->
    - ...

- **Structured Objects**
  - **Structures**
    - date(4, 10, 04)
    - person(bob, 48)
  - **Lists**
    - []
    - [a, b, g]
    - [[a], [b]]
    - [bit(a, d), a, 'Bob']
Structures

- To create a single data element from a collection of related terms we use a *structure*.
- A structure is constructed from a *functor* (a constant symbol) and one or more *components*.
- The components can be of any type: atoms, integers, variables, or structures.
- As functors are treated as data objects just like constants they can be unified with variables.

```
|?- X = date(04,10,04).
X = date(04,10,04).
yes
```
Structure unification

• 2 structures will unify if
  – the functors are the same,
  – they have the same number of components,
  – and all the components unify.

| ?- person(Nm,london,Age) = person(bob,london,48). |
Nm = bob,  
Age = 48?  
yes

| ?- person(Someone,_,45) = person(harry,dundee,45). |
Someone = harry ?  
yes

• (A plain underscore ‘_’ is not bound to any value. By using it you are telling Prolog to ignore this argument and not report it.)
Structure unification (2)

- A structure may also have another structure as a component.

```
?- addr(flat(4),street(‘Home Str.’),postcode(eh8_9lw))
   = addr(flat(Z),Yy,postcode(Xxx)).
Z = 4, 
Yy = street(‘Home Str.’), 
Xxx = eh8_9lw ?
yes
```

- Unification of nested structures works recursively:
  - first it unifies the entire structure,
  - then it tries to unify the nested structures.
Structures = facts?

• The syntax of structures and facts is identical but:
  – Structures are not facts as they are not stored in the database as being true (followed by a period ‘.’);
  – Structures are generally just used to group data;
  – Functors do not have to match predicate names.

• However predicates can be stored as structures

command(X):-
    X.

| ?- X = write('Passing a command'), command(X).
  Passing a command
  X = write('Passing a command') ?
  yes

By instantiating a variable with a structure which is also a predicate you can pass commands.
Lists

- A collection of ordered data.
- Has zero or more elements enclosed by square brackets (‘[ ]’) and separated by commas (‘,’).

```prolog
[a] ← a list with one element
[] ← an empty list
[34, tom, [2, 3]] ← a list with 3 elements where the 3rd element is a list of 2 elements.

Like any object, a list can be unified with a variable
```

```prolog
?- [Any, list, ‘of elements’] = X.
X = [Any, list, ‘of elements’]? yes
```
List Unification

- Two lists unify if they are the same length and all their elements unify.

| ?- [a, B, c, D] == [A, b, C, d]. | ?- [(a + X), (Y + b)] == [(W + c), (d + b)]. |
| A = a, | W = a, |
| B = b, | X = c, |
| C = c, | Y = d? |
| D = d ? | yes |

| yes |
| ?- [[X, a]] == [b, Y]. | ?- [[a], [B, c], []] == [X, [b, c], Y]. |
| no | B = b, |
| Length 1 | X = [a], |
| Length 2 | Y = [] ? |
| yes |
Definition of a List

- Lists are *recursively defined* structures.
  
  "An empty list, [], is a list.
  A structure of the form [X, …] is a list if X is a term and […] is a list, possibly empty."

- This recursiveness is made explicit by the bar notation
  - [Head|Tail] (‘|’ = bottom left PC keyboard character)

- **Head** must unify with a single term.
- **Tail** unifies with a list of any length, including an empty list, [].
  - the bar notation turns everything after the Head into a list and unifies it with Tail.
Head and Tail

|?- [a, b, c, d] = [Head | Tail]. |
Head = a,  
Tail = [b, c, d]?  
yes

|?- [a] = [H | T]. |
H = a,  
T = [];  
yes

|?- [] = [H | T]. |
no

|?- [a, b, c, d] = [X | [Y | Z]]. |
X = a,  
Y = b,  
Z = [c, d];  
yes

|?- [a, b, c] = [W | [X | [Y | Z]]]. |
W = a,  
X = b,  
Y = c,  
Z = []?  
yes

|?- [a | [b | [c | []]]] = List. |
List = [a, b, c]?  
yes
Summary

• Prolog’s proof strategy can be represented using AND/OR trees.
• Tree representations allow us trace Prolog’s search for multiple matches to a query.
• They also highlight the strengths and weaknesses of recursion (e.g. economical code vs. infinite looping).
• Recursive data structures can be represented as structures or lists.
• Structures can be unified with variables then used as commands.
• Lists can store ordered data and allow its sequential processing through recursion.