Logic Programming

Using Data Structures
Part 1

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria
kutsia@risc.jku.at

Contents

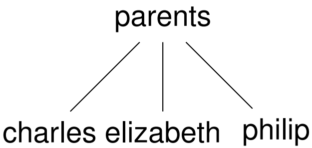
Representing Structures as Trees

Structures can be represented as trees:

- Each functor — a node.
- Each component — a branch.

Example

parents(charles, elizabeth, philip).

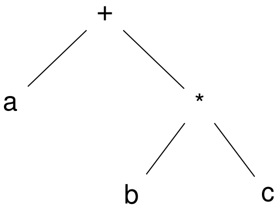


Representing Structures as Trees

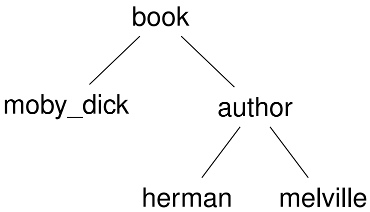
Branch may point to another structure: nested structures.

Example

a+b*c.



book(moby_dick, author(herman, melville)).



Parsing

Represent a syntax of an English sentence as a structure.

Simplified view:

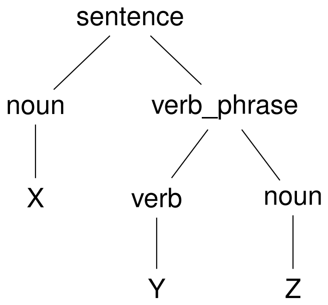
- Sentence: noun, verb phrase.
- Verb phrase: verb, noun.

Parsing

Structure:

sentence(noun(X), verb_phrase(verb(Y), noun(Z))).

Tree representation:

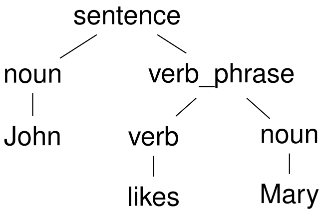


Parsing

Example

John likes Mary.

sentence(noun(John), verb_phrase(verb(likes), noun(Mary))).



Lists

Very common data structure in nonnumeric programming.

Ordered sequence of elements that can have any length.

- Ordered: the order of elements in the sequence matters.
- Elements: any terms — constants, variables, structures — including other lists.

Can represent practically any kind of structure used in symbolic computation.

The only data structures in LISP — lists and constants. In

PROLOG — just one particular data structure.

Lists

A list in PROLOG is either

- the empty list $[\,]$, or
- a structure $.(h, t)$ where h is any term and t is a list.
  h is called the head and t is called the tail of the list $.(h, t)$.

Example

- $[]$.
- $.(a, [])$.
- $.(a, .(b, []))$.
- $.(a, .(a, .(1, [])))$.
- $.(.(f(a, X), []), .(X, []))$.
- $.([], [])$.

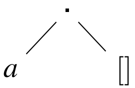
NB. $.(a, b)$ is a PROLOG term, but not a list!

Lists as Trees

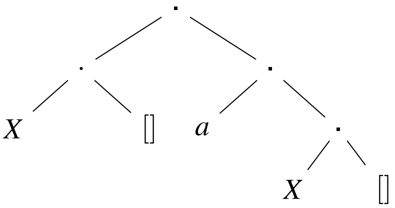
Lists can be represented as a special kind of tree.

Example

$.(a, [])$



$.(.(X, []), .(a, .(X, [])))$



List Notation

Syntactic sugar:

- Elements separated by comma.
- Whole list enclosed in square brackets.

Example

$$
\begin{array}{ll}
.(a, []) & [a] \\
.(.(X, []), .(a, .(X, []))) & [[X], a, X] \\
.([], []) & [[]]
\end{array}
$$

List Manipulation

Splitting a list L into head and tail:

- Head of L — the first element of L.
- Tail of L — the list that consists of all elements of L except the first.

Special notation for splitting lists into head and tail:

- $[X|Y]$, where X is head and Y is the tail.

NB. $[a|b]$ is a PROLOG term that corresponds to $.(a, b)$.

It is not a list!

Head and Tail

Example

List	Head	Tail
$[a, b, c, d]$	a	$[b, c, d]$
$[a]$	a	$[]$
$[]$	(none)	(none)
$[[the, cat], sat]$	$[the, cat]$	$[sat]$
$[X + Y, x + y]$	$X + Y$	$[x + y]$

Unifying Lists

Unifying Lists

$[X, Y, Z]$	$=$	$[john, likes, fish]$	$X = john$, $Y = likes$, $Z = fish$

Unifying Lists

$[X,Y,Z]$	$=$	$[john, likes, fish]$	$X = john$, $Y = likes$, $Z = fish$
$[cat]$	$=$	$[X\|Y]$	$X = cat$, $Y = []$

Unifying Lists

$[X, Y, Z]$	$=$	$[john, likes, fish]$	$X = john$, $Y = likes$, $Z = fish$
$[cat]$	$=$	$[X \mid Y]$	$X = cat$, $Y = []$
$[X, Y \mid Z]$	$=$	$[mary, likes, wine]$	$X = mary$, $Y = likes$, $Z = [wine]$

Unifying Lists

$[X, Y, Z]$	$=$	$[john, likes, fish]$	$X = john$, $Y = likes$, $Z = fish$
$[cat]$	$=$	$[X \mid Y]$	$X = cat$, $Y = []$
$[X, Y \mid Z]$	$=$	$[mary, likes, wine]$	$X = mary$, $Y = likes$, $Z = [wine]$
$[[the, Y], Z]$	$=$	$[[X, hare], [is, here]]$	$X = the$, $Y = hare$, $Z = [is, here]$

Unifying Lists

$[X, Y, Z]$	$=$	$[john, likes, fish]$	$X = john$, $Y = likes$, $Z = fish$
$[cat]$	$=$	$[X \mid Y]$	$X = cat$, $Y = []$
$[X, Y \mid Z]$	$=$	$[mary, likes, wine]$	$X = mary$, $Y = likes$, $Z = [wine]$
$[[the, Y], Z]$	$=$	$[[X, hare], [is, here]]$	$X = the$, $Y = hare$, $Z = [is, here]$
$[[the, Y] \mid Z]$	$=$	$[[X, hare], [is, here]]$	$X = the$, $Y = hare$, $Z = [[is, here]]$

Unifying Lists

$[X, Y, Z]$	$=$	$[john, likes, fish]$	$X = john$, $Y = likes$, $Z = fish$
$[cat]$	$=$	$[X\|Y]$	$X = cat$, $Y = []$
$[X, Y\|Z]$	$=$	$[mary, likes, wine]$	$X = mary$, $Y = likes$, $Z = [wine]$
$[[the, Y], Z]$	$=$	$[[X, hare], [is, here]]$	$X = the$, $Y = hare$, $Z = [is, here]$
$[[the, Y]\|Z]$	$=$	$[[X, hare], [is, here]]$	$X = the$, $Y = hare$, $Z = [[is, here]]$
$[golden\|T]$	$=$	$[golden, norfolk]$	$T = [norfolk]$

Unifying Lists

$[X, Y, Z]$	$=$	$[john, likes, fish]$	$X = john$, $Y = likes$, $Z = fish$
$[cat]$	$=$	$[X \mid Y]$	$X = cat$, $Y = []$
$[X, Y \mid Z]$	$=$	$[mary, likes, wine]$	$X = mary$, $Y = likes$, $Z = [wine]$
$[[the, Y], Z]$	$=$	$[[X, hare], [is, here]]$	$X = the$, $Y = hare$, $Z = [is, here]$
$[[the, Y] \mid Z]$	$=$	$[[X, hare], [is, here]]$	$X = the$, $Y = hare$, $Z = [[is, here]]$
$[golden \mid T]$	$=$	$[golden, norfolk]$	$T = [norfolk]$
$[vale, horse]$	$=$	$[horse, X]$	(none)

Unifying Lists

$[X, Y, Z]$	$=$	$[john, likes, fish]$	$X = john,\ Y = likes,$ $Z = fish$
$[cat]$	$=$	$[X\|Y]$	$X = cat,\ Y = []$
$[X, Y\|Z]$	$=$	$[mary, likes, wine]$	$X = mary,\ Y = likes,$ $Z = [wine]$
$[[the, Y], Z]$	$=$	$[[X, hare], [is, here]]$	$X = the,\ Y = hare,$ $Z = [is, here]$
$[[the, Y]\|Z]$	$=$	$[[X, hare], [is, here]]$	$X = the,\ Y = hare,$ $Z = [[is, here]]$
$[golden\|T]$	$=$	$[golden, norfolk]$	$T = [norfolk]$
$[vale, horse]$	$=$	$[horse, X]$	(none)
$[white\|Q]$	$=$	$[P\|horse]$	$P = white,\ Q = horse$

Strings are Lists

PROLOG strings — character string enclosed in double quotes.

Examples: “This is a string”, “abc”, “123”, etc.

Represented as lists of integers that represent the characters (ASCII codes).

For instance, the string “system” is represented as $[115, 121, 115, 116, 101, 109]$.

Membership in a List

`member(X,Y)` is true when X is a member of the list Y.

One of two conditions:

1. X is a member of the list if X is the same as the head of the list

   ```
   member(X,[X|_]).
   ```

2. X is a member of the list if X is a member of the tail of the list

   ```
   member(X,[_|Y]) :- member(X,Y).
   ```

Recursion

The first condition is the *boundary condition*.

(A hidden boundary condition is when the list is the empty list, which fails.)

The second condition is the *recursive case*.

In each recursion the list that is being checked is getting smaller until the predicate is satisfied or the empty list is reached.

Member Success

Step-by-step, using `trace`.

```
?- member(a, [a,b,c]).
   Call: (8) member(a,[a,b,c]) ?
   Exit: (8) member(a,[a,b,c]) ?
true

?- member(b, [a,b,c]).
   Call: (8) member(b,[a,b,c]) ?
   Call: (9) member(b,[b,c]) ?
   Exit: (9) member(b,[b,c]) ?
   Exit: (8) member(b,[a,b,c]) ?
true
```

Member Failure

Step-by-step, using `trace`.

```
?- member(d,[a,b,c]).
   Call: (8) member(d,[a,b,c]) ?
   Call: (9) member(d,[b,c]) ?
   Call: (10) member(d,[c]) ?
   Call: (11) member(d,[]) ?
   Fail: (11) member(d,[]) ?
   Fail: (10) member(d,[c]) ?
   Fail: (9) member(b,[b,c]) ?
   Fail: (8) member(b,[a,b,c]) ?
false
```

Member. Questions

What happens if you ask PROLOG the following questions:

```
?- member(X, [a,b,c]).
?- member(a, X).
?- member(X, Y).
?- member(X, _).
?- member(_, Y).
?- member(_, _).
```

Recursion. Termination Problems

Avoid circular definitions. The following program will loop on any goal involving `parent` or `child`:

```
parent(X, Y) :- child(Y, X).
child(X, Y) :- parent(Y, X).
```

Use left recursion carefully. The following program will loop on `?- person(X)`:

```
person(X) :- person(Y), mother(X, Y).
person(adam).
```

Recursion. Termination Problems

Rule order matters.

General heuristics: Put facts before rules whenever possible.

Sometimes putting rules in a certain order works fine for goals of one form but not if goals of another form are generated:

```
islist([_|B]) :- islist(B).
islist([]).
```

works for goals like `islist([1,2,3])`, `islist([])`, `islist(f(1,2))` but loops for `islist(X)`.

What will happen if you change the order of `islist` clauses?

Weaker Version of `islist`

Weak version of `islist`.

```
weak_islist([]).
weak_islist([_|_]).
```

Can it loop?

Does it always give the correct answer?

Mapping?

Map a given structure to another structure given a set of rules:

1. Traverse the old structure component by component
2. Construct the new structure with transformed components.

Mapping a Sentence to Another

Example

you are a computer maps to a reply i am not a computer.
do you speak french maps to a reply no i speak german.

Procedure:

1. Accept a sentence.
2. Change you to i.
3. Change are to am not.
4. Change french to german.
5. Change do to no.
6. Leave the other words unchanged.

Mapping a Sentence. PROLOG Program

Example

```
change(you, i).
change(are, [am,not]).
change(french, german).
change(do, no).
change(X, X).

alter([], []).
alter([H|T], [X|Y]) :-
    change(H, X),
    alter(T, Y).
```

Boundary Conditions

Termination:

```
alter([],[]).
```

Catch all (If none of the other conditions were satisfied, then just return the same):

```
change(X,X).
```