Recursion in Prolog. Compound terms

Alex Muscar

Software Engineering Department
Faculty of Automation, Computers and Electronics
University of Craiova

March 20, 2011

Compound terms

Definition

Compound terms are composed of an atom called *functor* and a sequence of one or more terms called *arguments*.

The functor acts as the name of the compound term.
Note: The arguments can be themselves compound terms.

Examples of compound terms

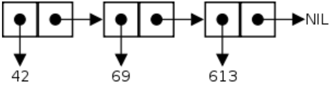
Example of a structure defining a person and of the predicates to extract the information.

Example

```
% A compound term defining a person with a name and
% a location
person('Alex', location(craiova, romania)).
% Name extractor
name(person(Name, _), Name).
% Location extractor
location(person(_, Location), Location).
% City extractor
city(person(_, location(City, _)), City).
```

Examples of compound terms (cont'd)

Compound terms can be used to define complex data structures such as linked lists.

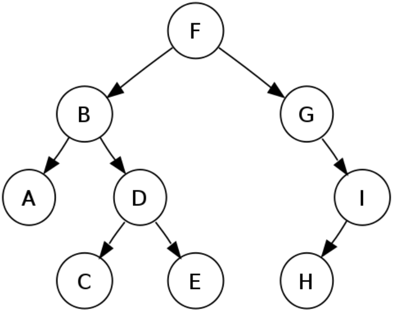


Example

```
cons(42, cons(69, cons(613, nil))).
```

Examples of compound terms (cont'd)

Compound terms can be used to define complex data structures such as trees.



Examples of compound terms (cont'd)

Example

```
node(f,
    node(b,
        node(a, leaf, leaf),
        node(d,
            node(c, leaf, leaf),
            node(e, leaf, leaf))),
    node(g,
        leaf,
        node(i,
            node(h, leaf, leaf),
            leaf)))
```

Recursion

While defining some rules it is sometimes necessary to refer to the rule we are defining.
Such rules are called *recursive* predicates.
Where you would use a loop (e.g. *for*, *while*) in an imperative language, in Prolog you use a recursive predicate.

Recursion scheme

A recursive predicate usually has:

base case acts as a termination condition

Recursion scheme

A recursive predicate usually has:

- base case acts as a termination condition
- recursive case the main part of the predicate

Note: in some cases the base case is not mandatory since the termination condition can be checked in other ways.

Recursion scheme

The basic shape of a recursive predicate in Prolog:

Example

```
predicate($t_1$, ..., $t_n$).
predicate($t_1$, ..., $t_n$) :-
    $g_1$, ..., $g_n$.
```

Recursion example

Assuming we have a knowledge base with the appropriate predicates and we wanted to find the ancestor of a person X we could (naively) go about it like this:

Example

```
ancestor(X, Y) :-  parent(X, Y).
ancestor(X, Y) :-
    parent(X, A1), parent(A1, Y).
ancestor(X, Y) :-
    parent(X, A1), parent(A1, A2), parent(A2, Y).
...
ancestor(X, Y) :-
    parent(X, A1), ..., parent(AN, Y).
```

Recursion example (cont'd)

We could define the ancestor relation as:

Definition

1. X is the ancestor of Y if X is a direct parent of Y

Recursion example (cont'd)

We could define the ancestor relation as:

Definition

1. X is the ancestor of Y if X is a direct parent of Y
2. X is the ancestor of Y if X is a direct parent of Z and Z is the ancestor of Y

Recursion example (cont'd)

Example

```
ancestor(X, Y) :-  parent(X, Y).
ancestor(X, Y) :-
    parent(X, Z), ancestor(Z, Y).
```

Tail recursion

Often recursive implementations are not as efficient as iterative ones.

This happens because recursion wastes stack space.

The solution is to use *tail recursion* implemented by *tail calls*.

Tail calls

Definition

A tail call is a call in the tail position of the recursive rule.

Tail calls allow for efficient implementations of recursive predicates.
Note: the *ancestor* predicate we defined earlier is tail recursive.
Note: sometimes in order to make a predicate tail recursive we can use accumulators.

Tail calls

The basic shape of a tail recursive predicate in Prolog:

Example

```
predicate($t_1$, ..., $t_n$).
predicate($t_1$, ..., $t_n$) :-
    $g_1$, ..., $g_n$, predicate($u_1$, ..., $u_n$).
```

Resources

- http://www.doc.gold.ac.uk/~mas02gw/prolog_tutorial/prologpages/recursion.html
- http://www.cse.unsw.edu.au/~billw/cs9414/notes/write-recursive-proc.html