University of Malta
BSc(Hons.) IT Year II

Problem Solving Techniques in Artificial Intelligence

Lecture 1

Prolog Review

- Basic Data Types
- Syntax
- Execution Part 1

Department of Computer Science and AI
1998/9

Prolog Review

- Basic Data Types
- Syntactic Issues
- Variables and Unification
- Prolog Semantics
- Input/Output
- Expressing Repetition
- Efficiency
- Style

Basic Data Types

Every data item that can exist in Prolog has a written representation (like Lisp, unlike Pascal).

Prolog data items are called *terms* and they come in three flavours: *variables*, *atomic terms* (numbers and atoms), and *compound terms* (structures).

Atoms normally begin with a lowercase letter and contain alphanumeric characters and “_”. However, *any* series of characters enclosed in single quotes is an atom.

An atom consisting entirely of special characters from the set

```
+ - * /  \ ^ < = > ~ : . ? @ # $ &
```

does not need quotes.

Exercise

(a) Which of the following are Prolog atoms?

```
abc123    list_all  list-all
----->    --def-->   32
'ab cd'   "ab  cd"   ?-
'23'      '2+2'      (2+2)
```

(b) Which of the above are atomic terms

(c) Why are there quotes in write('One') but not in write(one)?

(d) Are abc and 'abc' the same atom. How would you find out?

Internal Representation of Atoms

Atoms are not character strings. They are locations in a symbol table: only one copy of an atom is stored no matter how many times it occurs in a program.

During program execution, the comparison

asdfgghjjhshs = asdfgghjjhshs

takes no more time than

a=b

since the pointers get compared rather than the character strings.

Compound Terms (structures)

A structure consists of a *functor* with one or more arguments such as f(a,b,c).

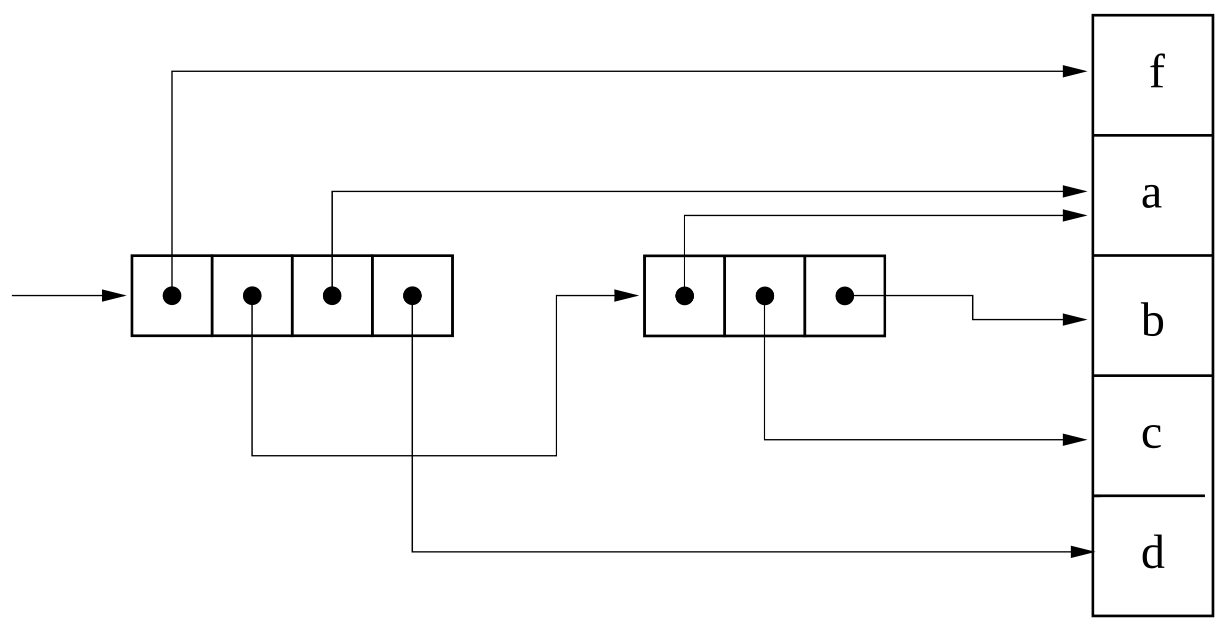
Here the arguments are atoms, but in general they can be terms of any type including structures.

The functor is an atom with a specific arity (i.e. number of arguments), so f of arity 1,2 and 3 (referred to as f/1, f/2, and f/3 respectively) are distinct functors.

Exercise

Suppose you are creating a Prolog knowledge base about a family. Would it make sense to use both parent/2 and parent/1 in the same program? What would each of these mean?

Internal Representation of Structures



Exercise

- For the above structure write down (a) the predicate-argument structure and (b) the tree diagram.
- Which structure takes more memory and why?

  f(a,b,c,d) or f(abcdefg,abcdefg)

Lists

A distinguished functor "./2", is used to encode a list.

```
PREDICATE/ARGUMENT     TREE STRUCTURE
                                 .
                                / \
                               /   \
                              a     .
                                   / \
.(a,.(b,.(c.[])))                 /   \
                                 b     .
                                      / \
                                     /   \
                                    c    [ ]
```

- The first argument to the "." functor is an element.
- The second argument is always a list
- the special atom "[ ]" (pronounced "nil") stands for the empty list.

List Notation

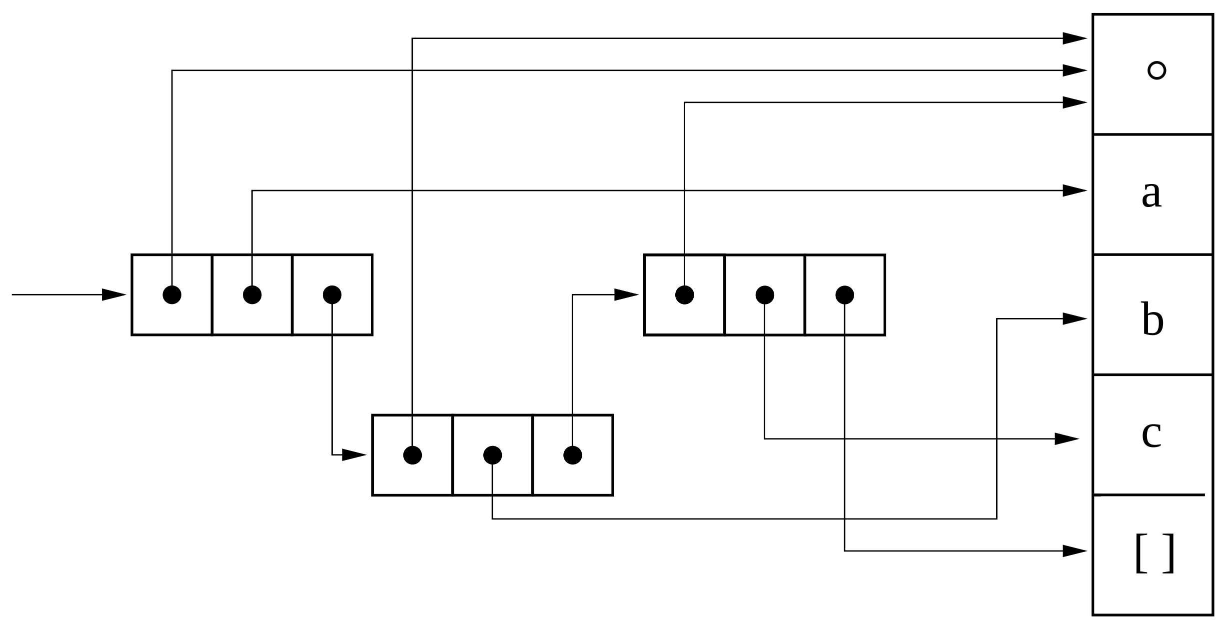
The main advantage of a list over a multiargument structure is that it can be processed without knowing how many elements it has. Hence a nonempty list of any length will match against the structure

```
       .
      / \
     /   \
    X     Y
```

Lists are so useful and so common that there is a special shorthand notation:

```
.(a,[])                 [a] or [a|[]]
.(a,.(b,.(c.[])))       [a,b,c] [a|[b|[c|[]]]]
.(a.b)                  [a|b]
.(a,.(b,.(c,d)))        [a,b,c|d] [a|[b|[c|d]]]
```

Implementation of Lists



Exercise

Write down the list structure implemented above using list notation.

Strings

A string such as "abc" is simply another way of writing a list of the ASCII codes of the characters, in this case [97,98,99].

The built-in name/2 interconverts atoms and strings:

```
?- name(abc,What).
What= [97,98,99].

?- name(What,"abc")
What = abc
```

Exercise

Define a predicate a2n(A,N) which converts between a single character and its ASCII code.

Syntactic Issues: Operators

Operators are nothing more than functors whose arguments need not be enclosed in parentheses. This provides notational advantages. For example if we wish Prolog programs to manipulate other progams as data, we must represent items like rules as terms. For instance the rule `a :- b, c, d.` can be represented as the following term:

```
.
 \
  \
  :-
  / \
 /   \
a     ,
     / \
    /   \
   b     ,
        / \
       /   \
      c     d
```

By providing appropriate operators, the mere reading of the rule results in the construction of the corresponding term.

Operator Declarations

Information about operators is conveyed through *declarations* which include:

1. The character string that defines the operator. Often, but not always, we make use the special characters to form operators (e.g. `'-->'` or `'<='`).

2. The number of arguments it takes (either 1 or 2), and its associativity. This is indicated using the following codes

Code	**Args**	**Type**	**Assoc**	**e.g.**
fx	1	prefix	no	'?-'
fy	1	prefix	yes	'-'
xf	1	postfix	no	'.'
yf	1	postfix	yes	mouseclick
xfx	2	infix	no	':-'
yfx	2	infix	left	'*'
xfy	2	infix	left	','

3. The precedence (an integer)

Associativity and Precedence

Associativity

Intuitively, an operator is associative (whether unary or binary) if it can "nest".

(1 + (2 + (3 + 4)))	right associative (xfy)
(((1 + 2) + 3) + 4)	left associative (yfx)
p :- q :- r	non assocative (illegal)

Precedence

Precedence is an integer which indicates how weakly an operator binds its arguments. Roughly, the *higher* the precedence, the *weaker* the operator.

Thus expression a + b * c normally means (a + (b * c)) because + has *higher* precedence than *.

Declaring Operators

To declare an operator use one or more operator declarations at the top of your progrem. Note that the ':-' actually appears in the file.

```
:- op(510,xfy,'-').
:- op(500,xfy,',').
```

Exercises

Draw the tree-diagram for `a-functor(arg1,other-arguments)` assuming that the above declarations are in force. What happens if the precendences are inverted?

- If you want to see the structure of any term in ordinary predicate-argument notation, the predicate display/1 does it for you.
- The man page giving the operators predeclared for SWI Prolog are to be found at `http://www.cs.um.edu.mt/~mros/SWI-Manual/sec-B.3.html`

Variables Unification

- Prolog variables begin with an uppercase letter or '_'.
- Like named variables within the same term are considered to be the same variable.
- The names themselves have no significance, and are often displayed as integers.
- Each occurrence of the anonymous variable '_' is a unique variable (so it can never bind with any other variable).

Which of the following are equivalent to each other?

```
f(X,Y,Y)    f(_3,_q,_q)   f(X,Y,_)
```

Unification

Two terms are *unifiable* if they are alike or can be made alike by instantiating variables. Instantiating a variable means either giving it a value (i.e. another term), or making it equivalent to another variable.

T1	T2	T1 Unified with T2
f(X,Y)	f(a,b)	X = a, Y = b
f(X,Y)	f(Z,Z)	X,Y,Z become the same var
f(X,X)	f(a,v)	fail

Unification is order-independent.

Unification Exercise

What is the result of unifying the following pairs of terms?

`f(X,Y)`	`f(a,b)`
`f(X,Y)`	`g(X,Y)`
`f(a,b)`	`f(a,X)`
`f(a,b)`	`f('a ',X)`
`f(a,b)`	`f(((('a'))),X)`
`[a,b,c,d]`	`[Q|_234]`
`[a,b,c,d]`	`[X,Y|Z]`
`[a,X,b,Y]`	`[Z,a,B,b]`
`[a,X,Y,Y]`	`[Z,a,b,c]`
`[a,b,c,d]`	`[X,Y,Z]`
`(2+3)+X`	`Q+R`
`(2+3)+X`	`2+(3+4)`

Structure of a Prolog Program

A Prolog program file contains **clauses** and/or **queries**.

Clauses express information to be stored in the database and are either facts or rules.

Facts are atoms or structures whose principle functor is not ":-" or "?-".

```
cheese(c).
moon(m).
made_of(m,c).
```

Rules consist of a term, the symbol ":-", then one or more terms linked by commas.

```
green(X) :- cheese(X), moon(Y), made_of(Y,X).
```

The **head** of a rule is the term to the left of ":-". The remaining clauses constitute the **body**.

Indexing of Facts and Rules

A fact is indexed by its principle functor (e.g. moon/1), whilst a rule is indexed by the principle functor of the head (eg green/1).

A **predicate** is defined by a set of one or more clauses with the same index.

Queries

A query is a request for the computer to do something. A query can either be typed at the keyboard (i.e. after “?- “), or else witten into a file preceded by “:-” (and nothing else).

```
:- op(500,xfx,'**').
:- write('loading program...").
:- green(cheese), write('gottit - 1st!'), nl.
green(cheese).
:- green(cheese), write('gottit - 2nd!'), nl.
```

Execution of Queries (Goals)

If the query is a sequence of terms are joined by commas, the individual terms are treated as sub-queries (subgoals) and executed in order. So given

```
?- green(X), write(X).
```

the system executes green(X) and then write(X). Note that because the entire query is one term, the two Xs are the same variable.

If the principle functor is a built in predicate, the system just carries it out.

Otherwise, the system tries to find a clause whose head unifies with that of the query.

Execution of Queries

If the clause is a fact, the query succeeds, and no other action is performed (except to transmit the variable bindings to the caller).

If the clause is a rule, its body is treated as a new query *within the binding environment establised by the match*. So the query a(xxx) is transformed by the rule

a(X) :- b(X), c(X).

into the subgoals

b(zzz), c(zzz).

Exercise

Show how the query "?- amphibian(kermit)" is transformed into other queries and solved in the following database. Do the same for "?- amphibian(Who)".

```
amphibian(X) :-
 latin_name(X,Genus,Species),
 class('Amphibia',Genus).

latin_name(kermit,'Rana',catesbiana).

class('Amphibia','Rana').
```

Backtracking

If at any point there is no clause to match the current query, it is said to fail.

Execution then backs up to the next backtrack point (untried alternative) undoing any unifications that took place up to that point.

Execution then proceeds forward along an alternative computation path.

Alternatives exist because more than one clause can match the same query.

Whenever execution enters a clause and there are other untried ones remaining, the system records a backtrack point.

Exercise

Given the knowledge base

```
a(X) :- b(X).
a(X) :- c(X).
b(Z) :- d(Z).
b(Z) :- e(Z).
c(Y) :- h(Y).
e(f).
```

show exactly what goals are tried, in what order, to solve the query "?- a(f)". Which backtrack point are left untried?