Prolog 3

Junli Tao

jtao076@aucklanduni.ac.nz

Rm 321-723, Tamaki Campus
(Wed. 1:00pm-2:00pm)

Based on tutorial notes from Mike
Outline

• Arithmetic Operations in Prolog

• Input/Output Streams

• Prolog Control Flow
Arithmetic Operations in Prolog

• Prolog arithmetic can be a little surprising
  
  ?- X = 3 + 4.
  
  X = 3+4 ;

• 3 + 4 is just a term whose functor is + and arguments are 3 and 4
Arithmetic Operators

A special predefined operator, is, is provided to circumvent this problem.

The is operator will force evaluation. So the right way to invoke arithmetic is:
?- X is 3+4.

Now the answer will be: X=7

The addition here was carried out by a special procedure that is associated with the operator +.

We call such procedures built-in procedures
Arithmetic Operators

- \( X+Y \), the sum of \( X \) and \( Y \)
- \( X-Y \), the difference of \( X \) and \( Y \)
- \( X*Y \), the product of \( X \) and \( Y \)
- \( X/Y \), the quotient of \( X \) and \( Y \)
- \( X^Y \), \( X \) to the power of \( Y \)
- \(-X\), the negative of \( X \)
- \( \text{abs}(X) \), the absolute value of \( X \)
- \( \sin(X) \), the sine of \( X \) (for \( X \) measured in degrees)
- \( \cos(X) \), the cosine of \( X \) (for \( X \) measured in degrees)
- \( \text{max}(X,Y) \), the larger of \( X \) and \( Y \)
- \( \text{sqrt}(X) \), the square root of \( X \)
- \( \ll \gg \) Left and right shift
Define a Predicate

• Define \textit{add\_3\_and\_double}
  \[
  \text{add\_3\_and\_double}(X,Y) :- Y \text{ is } (X+3)\times 2.
  \]

• It works:
  \[
  \text{- add}\_3\_and\_double(1,X). \\
  \hspace{1cm}X = 8 \\
  \]
  \[
  \text{- add}\_3\_and\_double(2,X). \\
  \hspace{1cm}X = 10 \\
  \]
  \[
  \text{- add}\_3\_and\_double(X,12). \\
  \hspace{1cm}\text{ERROR}
  \]
Arithmetic Operators

• Relational Operators
The infix operators:

\[ =:= \]
\[ =\backslash= \]
\[ > \]
\[ \geq \]
\[ < \]
\[ \leq \]

are special type known as relational operators
Equality operators

- Arithmetic Expression Equality

E1 =:= E2 succeeds if the arithmetic expressions E1 and E2 evaluate to the same value.

?- 6+4 =:= 6*3-8.
yes

?- sqrt(36)+4 =:= 5*11-45.
yes
Equality operators

- **Arithmetic Expression Inequality \( \neq \)**

  \( E_1 \neq E_2 \) succeeds if the arithmetic expressions \( E_1 \) and \( E_2 \) do not evaluate to the same value

?- 10\( \neq \)8+3.

Yes
Equality Operators

• TERM IDENTICAL ==

Both arguments of the infix operator == must be terms. The goal Term1==Term2 succeeds if and only if Term1 is identical to Term2. Any variables used in the terms may or may not already be bound, but no variables are bound as a result of evaluating the goal.

?- likes(X,prolog)==likes(X,prolog).
X = _

?- likes(X,prolog)==likes(Y,prolog).
no
(X and Y are different variables)

?- X is 10,pred1(X)==pred1(10).
X = 10

?- X==0.
no

?- 6+4==3+7.
no
Equality Operators

• Terms Not Identical `\==`
  Term1`\==`Term2 tests whether Term1 is not identical to Term2. The goal succeeds if Term1`==`Term2 fails. Otherwise it fails.

?- pred1(X)`\==`pred1(Y).
  X = _ ,
  Y = _

(The output signifies that both X and Y are unbound and are different variables.)
Equality Operators

- **Terms Identical With Unification =**
  The term equality operator = is similar to == with one vital (and often very useful) difference. The goal Term1=Term2 succeeds if terms Term1 and Term2 unify, i.e. there is some way of binding variables to values which would make the terms identical. If the goal succeeds, such binding actually takes place.

  ?- pred1(X)=pred1(10).
  X = 10
  (Variable X is bound to 10, which makes the two terms identical.)

  ?- likes(X,prolog)=likes(john,Y).
  X = john ,
  Y = prolog
  (Binding X to the atom john and Y to the atom prolog makes the two terms identical.)
Equality Operators

• \( ?- \ X=0,X=:=0. \)
  \( \mathbf{X} = 0 \)
  (\( X=0 \) causes \( X \) to be bound to 0. The goal \( X=:=0 \) succeeds, which confirms that
  \( X \) now has the value zero.)

\( ?- \ 6+4=3+7. \)
\( \text{no} \)

• \(- \ \text{likes}(X,\text{prolog})=\text{likes}(Y,\text{prolog}). \)
  \( \mathbf{X} = \mathbf{Y} = _{\_} \)
  (Binding \( X \) and \( Y \) makes the terms identical.)

\( ?- \ \text{likes}(X,\text{prolog})=\text{likes}(Y,\text{ada}). \)
\( \text{no} \)
(No unification can make the atoms \text{prolog} and \text{ada} identical.)
Equality Operators

• Non-Unification Between Two Terms $\neq$
  The goal Term1$\neq$Term2 succeeds if Term1=Term2 fails, i.e. the two terms cannot be unified. Otherwise it fails.

?- 6+4$\neq$3+7.
Yes
Operator Precedence

When there is more than one operator in an arithmetic expression, e.g. A+B*C-D, Prolog needs a means of deciding the order in which the operators will be applied.

Prolog achieves this by giving each operator a numerical precedence value.

Operators with relatively high precedence such as * and / are applied before those with lower precedence such as + and -.

Operators with the same precedence (e.g. + and -, * and /) are applied from left to right.

The effect is to give an expression such as A+B*C-D the meaning that a user who is familiar with algebra would expect it to have, i.e. A+(B*C)-D.
## Relational Operator Notations

<table>
<thead>
<tr>
<th>Condition</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &lt; y$</td>
<td>$X &lt; Y$</td>
</tr>
<tr>
<td>$x \leq y$</td>
<td>$X =&lt; Y$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>$X =:= Y$</td>
</tr>
<tr>
<td>$x \neq y$</td>
<td>$X =\= Y$</td>
</tr>
<tr>
<td>$x \geq y$</td>
<td>$X =\geq Y$</td>
</tr>
<tr>
<td>$x &gt; y$</td>
<td>$X &gt; Y$</td>
</tr>
</tbody>
</table>

### Examples

- ?- $4 = \\leq 5$.  
  yes
- ?- $4 = \\leq 4$.  
  no
- ?- $2+1 < 3+2$.  
  yes

Note the difference between $=$  $=:= =$

- ?- $4 = 4$.  
  yes
- ?- $2+2 = 4$.  
  no
- ?- $2+2 =:= 4$.  
  yes
- ?- $a = a$.  
  yes
- ?- $2+2 == 4$.  
  no
Summary for =, ==, =:=

• = The unification predicate. Succeeds if it can unify its arguments, fails otherwise.
• \= The negation of the unification predicate. Succeeds if = fails, and vice-versa.
• == The identity predicate. Succeeds if its arguments are identical, fails otherwise.
• \== The negation of the identity predicate. Succeeds if == fails, and vice-versa.
• =:= The arithmetic equality predicate. Succeeds if its arguments evaluate to the same integer.
• =\= The arithmetic inequality predicate. Succeeds if its arguments evaluate to different integers.
Input/Output Streams

• A Prolog program can read data from input streams, and write data to output streams.
• Streams are associated with files.
• Data coming from the user's terminal is treated as just another input stream. Data output to the terminal is treated as another output stream. Both of these "pseudo-files" are referred to by the name user.
• The names of other files can be chosen by the programmer according to the rules of the OS.
File-based I/O: `write/1`

- This prints a term to the current output stream.
  
  ```prolog
  ?- write(c), write(u_l), write(8), write(r).
  cu_l8r ← writes to terminal by default.
  yes
  ?- write([a,b,c,d]).
  [a,b,c,d]
  yes
  
  It will only accept Prolog terms so strings must be enclosed within single quotation marks.
  
  ?- write(Hello World) ?- write('Hello World').
  syntax error
  Hello World
  yes```
Formatting Output

• We can use built-in predicates to format the output:
  - `nl/0` = write a new line to the current output stream.
  - `tab/1` = write a specified number of white spaces to the current output stream.

  • *this prints single spaces not actual tabs!*

|?− write(a), tab(3), write(b), nl, tab(1), write(c), tab(1), write(d), nl, tab(2), write(e).

  a   b
  c   d
  e
  yes

• We can add syntax by writing string fragments.

|?− Day=04, Mth=11, Year=04, write(Day), write('/'), write(Mth), write('/'), write(Year).

  4/11/4   Day=4, Mth=11, Year=4, yes
Reading input: \texttt{read/1}

- Now that we know how to control our output we need to do the same for our input.
- The default input stream is the user terminal.
- We can read terms from the terminal using the command \texttt{read/1}.
  - this displays a prompt ‘|:’ and waits for the user to input a term followed by a full-stop.

\begin{verbatim}
| ?- write('What is your name?'), nl, read(X), write('Greetings '), write(X).

What is your name?
|: tim.
Greetings tim
X = tim ?
yes
\end{verbatim}
Reading input

• `read/1` can only recognise *Prolog terms* finished with a *full-stop*.

```
|?- read(X).
|: hello

← Waits for full-stop to finish term.

. ← Finds full-stop and succeeds.
```

```
X = hello?
yes
```

• Therefore, strings of text must be enclosed in single quotes.

```
|?- read(X).
|: Hi there!.
syntax error
```

```
|?- read(X).
|: ‘Hi there!’.
```

```
X = ‘Hi there!’?
yes
```
Usage of Quotes

• When we are reading strings there are two ways we can input them:
  – if we enclose them in single quotes (e.g. ‘Hi Bob!’) the string is read verbatim.

```
| ?- read(X).
| : 'Hi bob!'.
X = 'Hi bob!' ?
yes
```

  – if we enclose them in double quotes (e.g. “Hi Bob!”) the string is interpreted into the corresponding list of ASCII codes.

```
| ?- read(X).
| : "Hi bob!".
X = [72,105,32,98,111,98,33] ?
yes
```

• It is important to use the right quotes as otherwise you won’t be able to process the input correctly.
Example compute cube of number

- % Compute the cube of a number
  :- module(cube,[cube/0]).
cube :-
  write('Input number or stop (end with .): '),
  read(X), process(X).
process(stop) :- !.
process(N) :-
  Result is N*N*N,
  write('The cube of '), write(N),
  write(' is '),
  write (Result),nl,cube.
% Given a list of lists, display the elements of each list in one line.
displaylist([ ]).  
displaylist([L | LL]) :- doline(L), nl, displaylist(LL).
doline([ ]).  
doline([H | T]) :- write(H), tab(2), doline(T).
?- displaylist([[a,b,c], [d,e,f],[g,h]]).
a b c
d e f
g h
Yes
Prolog Procedural Semantics

• Prolog’s control flow different from languages like Java.

• In Prolog no distinction between statements & method calls - all goals recursively matched against KB (system defined relations (write/1) have own definitions).

• Goal parameters can be unbound variables.

• Goals succeed or they fail.
Prolog Control Flow Example

• **KB**

• 1. \( a(X) :- b(X) \).

• 2. \( b(2) \).

• Execution trace of query

\([a(2)]\) (goal stack)

1. \( X = 2 \)

\([b(2)]\) (which clause matched & bindings)

2. [ ]
Prolog Control Flow Example

• Given the query $a(2)$, Prolog scans the KB for the first clause head that matches the first query goal.
• If it finds a match then it recurses on the goals in the clause body until it bottoms out.
• Query succeeded here.
• Notice that after the query goal matched the clause head, the query goal ($a(2)$) is replaced by the first clause body goal ($b(2)$) in the next step of the trace with its variable bound.
Prolog Control Flow Example

• **KB**
  1. \( a(X) :- b(X). \)
  2. \( b(2). \)

• **Execution trace of query**
  
  \[ a(3) \]
  1. \( X = 3 \)
  
  \[ b(3) \]
  \( \times \)
Prolog Control Flow Example

• \( a(3) \), Prolog scans the KB for first clause head matching first query goal.
• If it fails to find a match then it tries to backtrack its matches.
• There are no other possible matches.
• Query fails.

• Ques: What if we have another rule with value as \( a(3) \) in the same Example : **BACKTRACKING**
Flow of Control with Conjunctions

- **Query:**
  
  \( a(X) \).

- **KB:**
  
  1. \( a(X) :- b(X), c(X) \).
  2. \( b(2) \).
  3. \( b(3) \).
  4. \( c(3) \).

- **Response:**

  \( X = 3 \) ?
Flow of Control with Conjunctions

\[ a(X) \]

1. \( X = X' \)

\[ b(X'), c(X') \]

2. \( X' = 2 \)

3. \( X' = 3 \)

\[ c(2) \]

\[ c(3) \]

\times \quad 4.

\[ \]
Prolog Procedural Semantics II

- When goal fails, control undoes any bindings done at that step & backs up to previous step.
- It continues scan of KB for clause heads matching current goal.
- If none found then continues to backtrack.
Flow of Control with Conjunctions

• **Query:**
  \( a(X) \).

• **KB:**
  1. \( a(X) : - b(X), c(X) \).
  2. \( b(Y) : - d(Z), e(Y,Z) \).
  3. \( b(3) \)
  4. \( c(3) \).
  5. \( d(5) \).
  6. \( e(3,5) \).

• **Response:**
  \( X = 3 ? \)
Flow of Control with Conjunctions

\[
\begin{align*}
[a(X)] & \\
1.\ X &= X' \\
[b(X'), c(X')] & \\
2.\ X' &= Y \\
d(Z), e(Y,Z), c(Y) & \\
5.\ Z &= 5 \\
e(Y,5), c(Y) & \\
6.\ Y &= 3 \\
c(3) & \\
4. & \\
\end{align*}
\]