Programming Paradigms CSI2120 – Winter 2018

Jochen Lang
EECS, University of Ottawa
Canada

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

Arithmetic Expressions and I/O

- **Arithmetic Expressions**
  - Built-in operators
  - Unification with numbers
  - Recursive calculations
  - Looping with repeat
  - Generator
- **Input and output: Streams**
  - Reading and writing to console
  - Reading and writing to file
  - Character i/o

Numbers in Prolog

- **Prolog recognizes numbers**
  - integers and floating point
- **Number constants**

```
5 1.75 0 1.345e-10 -27 -3.4 42
```

- **Rules about arithmetic expressions use**
  - number constants
  - arithmetic operators
  - arithmetic variables

Arithmetic Expressions

- **Prolog supports common operators as built-ins including**

```
X+Y
X-Y
X*Y
X/Y
X // Y %integer division
X mod Y
```

- **Mathematical functions, e.g.,**

```
abs(X)
ln(X)
sqrt(X)
```

Evaluating Arithmetic Expressions

- **Special predicate "is" in order to treat variables and operators as relating to mathematical operations**

```
?- 1+2 = 1+2.
true.
?- 3 = 1+2.
false.
?- 1+2 = 2+1.
false.
?- 3 is 1+2.
true.
?- X is 1+2, X is 2+1.
X = 3.
```

Unification with Arithmetic Expressions

- **Careful with expressions and unification**
  - Unification of 1+2 and 3 fails.
    - 3 is a number, while 1+2 is a term.
  - Evaluation of arithmetic expression is not part of the regular unification algorithm and does not happen automatically

Infix Comparison Operators

- **Comparisons**

```
X =:= Y   % X equals Y
X =\= Y   % X not equals Y
X < Y
X =< Y
X > Y
X >= Y
```

- **The operators are applied after calculations, e.g.,**

```
?- 1+2 =:= 2+1.
true.
```

Example: Min Predicate

```
min(X,Y,X) :- X<Y.
min(X,Y,Y) :- X>=Y.
```

- **What queries can we ask?**

```
?- min(5,7,X).
?- min(5,X,7). % false
?- min(X,5,7). % false
?- min(X,Y,7). % error - why?
```

Predicates using Recursion: power

- **Positive Powers**
  - boundary case for power to 1

```
pow( X, 1, X ).
```

  - recursion to calculate the product

```
pow( X, Y, Z ) :- Y > 1,
                  Y1 is Y - 1,
                  pow( X, Y1, Z1 ),
                  Z is X * Z1.
```

Predicates using Recursion: gcd

- **Greatest common divisor**
  - Boundary condition
    - gcd of 0 and any number is the number itself

```
gcd(U,0,U).
```

  - Recursive clause based on Euclid's algorithm
    - modulo divisions until remainder is 0 at which point we found a divisor for all intermediate divisors and the original number

```
gcd(U,V,W) :-  V>0, R is U mod V,
               gcd(V,R,W).
```

- **Alternative implementation of Euclid's algorithm**

```
gcd(A,A,A).
gcd(A,B,GCD):- A<B, NB is B-A, gcd(A,NB,GCD).
gcd(A,B,GCD):- A>B, NA is A-B, gcd(NA,B,GCD).
```

Predicates using Recursion: fibonacci

- **Fibonacci numbers**
  - a series of numbers 1 1 2 3 5 8 13 21 …
  - Recursive clause based on Fibonacci's algorithm
    - fib(N) = fib(N-1) + fib(N-2)

```
fib(N,F):- N>1,
           N1 is N-1,
           N2 is N-2,
           fib(N1,F1),
           fib(N2,F2),
           F is F1+F2.
```

  - Two boundary conditions are needed.

```
fib(0,1).
fib(1,1).
```

Example with Crossed Recursions

Predicate to test if a positive number is even

```
even(0).
odd(N)  :-  N>0,
            M is N-1,
            even(M).
even(N):-   N>0,
            M is N-1,
            odd(M).
```

A Last Example

- **Interval test to see if X is in the interval between L and H**

```
intervalTest(X,L,H):- X>=L, X=<H.
```

Simple but cannot generate numbers between L and H, i.e.,

```
?- intervalTest(X,1,5).
```

will produce an error.

- **Generative predicate (or Generator)**

```
interval(X,X,H):- X=<H.
interval(X,L,H):- L<H,
                  L1 is L+1,
                  interval(X,L1,H).
```

  - Now we can ask

```
?- interval(X,1,5).
```

Input-Output

- **Write to the screen or to a file**
- **Read from the keyboard or from a file**
- **Writing terms with the built-in predicate `write/1`**
  - `write(X).` adds the value of `X` to the currently active output stream (by default the console).
  - Example:
    - `write(1+2)` outputs `1+2`
  - `nl` is the new line command, i.e.,
    - `writeln(X) :- write(X), nl.`
  - `tab(N)` outputs `N` spaces

More Output Commands

- **write/1 vs. display/1**
  - Both, write and display output to the current streams
  - write displays operators as operators
  - displays ignores all operator definitions
  - Example:

```
write(3+4), nl, display(3+4), nl.
```

    - Output:

```
3+4
+(3,4)
YES
```

Input

- **Reading terms:**
- **`read/1` is for input from the currently open stream.**
  - The term has to be followed by a . (dot) and return at which point the read goal will succeed and X will be instantiated to the entered characters.
  - The prompt is system dependent, e.g., a `:` (colon).
    - Example :

```
?- read(X).
|: a(1,2).
X = a(1,2)
```

Interactive Example

```
age(X, Y) :-
  write('Give the age of
'),
  write(X), write(': '),
  read(Y).
?- age(teddy, Z).
Give the age of teddy:
22.
Z = 22
Yes
```

```
?- age(teddy, 22).
Give the age of
teddy: 23.
No
?- read(abc).
:23.
No
?- read(X + Y).
:2 + 3.
X = 2
Y = 3
Yes
```

Calculator Example:

- **Read an arithmetic expression from a stream**
- **Calculate result**
- **Exit on end**

```
calculator :- repeat, % loop forever
        read(X),      % read expression
        eval(X,Y),    % our evaluation
        write(Y), nl, % output result
        Y = end, !.   % stopping condition

eval(end, end) :- !. % end evaluates to itself
eval(X, Y) :- Y is X. % otherwise calculate
```

Control of Backtracking in Calculator

- **The built-in predicate repeat is a way to generate multiple solution through backtracking.**
- **Definition**

```
repeat.
repeat :- repeat.
```

- **Calculator**
  - if end test fails, we backtrack until repeat succeeds again
- **The "Cut " ! stops backtracking across it**
  - More details in the next lecture
- **Calculator**
  - if end succeeds, we don't backtrack across it to find more solutions

Repeat Again

```
test :- repeat,
        write('Answer to everything? (num)'),
        read(X),
        (X=:=42).
```

Opening and Closing a File

- **Predicate `open/3`**
  - argument 1: Filename
  - argument 2: File mode: `write`, `append` or `read`
  - argument 3: Instantiated with the name of the stream (file handle) that must be used to manipulate the stream status (close, set_input, etc.)
- **Modes for writing:**
  - `write` mode opens the file and puts the stream marker at the beginning of the file.
    - existing content is overwritten
  - `append` mode puts the stream marker at the end of the file
- **Predicate `close/1`**
  - takes a file handle and closes the stream

Reading and Writing

- **The current input and output stream can be set, affecting all input and output commands (e.g., read, write, etc.)**
  - `set_input(X)`
    - `user_input` is the keyboard
    - Query with `current_input(X)`
  - `set_output(X)`
    - `user_output` is the console
    - Query with `current_output(X)`
- **All the read and write predicates can take an extra parameter for the file handle**
  - `write(X, Y).` X is the file handle (as above)
  - `read(X,Y)` `get(X, Y)` `get0(X,Y)`

Example: Write to File

Write X to file

```
writeFile(X):- open('test.txt', append, F),
               write(F, X), nl(F),
               close(F).
```

Default Input and Output Stream

- **Alternative (simpler) ways to set the current input and output stream**
  - `see(Filename)`. Filename becomes the current output stream; opens file in write mode
  - `seen`. Closes current output stream and reverts back to the console.
  - `tell(Filename)`. Filename becomes the current input stream
  - `told`. Closes current input stream and reverts back to the keyboard.

Character Input and Output

- **`put_char(Character)` puts a character code into the current stream**
  - character can either be an integer (e.g., ASCII_Code) or a character,e.g., ‘a’
  - `put(ASCII_Code)` also exists as a non-ISO primitive
- **`get_char(Character)` gets a character into the current stream**
  - *Non-iso primitives*
    - `get0(X)` unifies the variable X with the ASCII code character entered .
    - `get(X)` is the same as get0 but skips spaces.

Example: Province.pl

```
capital(ontario,toronto).
capital(quebec,quebec).

start :- write('The capitals of Canada'),nl,
  askP.

askP :- write('Province? '), read(Province),
  answer(Province).

answer(stop) :- write('Exiting'),nl.
answer(Province) :- capital(Province,City),
  write(City),write(' is the capital of '),
  write(Province),nl,nl,
  askP.
```

Summary

- **Arithmetic Expressions**
  - Built-in operators
  - Unification with numbers
- **Recursive calculations**
    - power, factorial, gcd, fibonacci
    - crossed recursion
- **Generator**
- **Looping with Repeat**
- **Input and output: Streams**
  - Reading and writing to console
  - Reading and writing to file
  - Character i/o