In the Lecture Series Logic Programming and Constraints

Prologue to Prolog

Presented by Stéphane Bressan

Aims and Objectives

This course aims to discuss the basic aspects of constraint and logic programming.

It will focus on constraint logic programming and its realisation in Eclipse, a system that extends Prolog language by means of constraints.

The course will focus on problem modelling by means of constraints, and on logic programming techniques concerned with constraints.

Students will learn in detail a number of modules of the Eclipse system that aims to increase the versatility of programming by means of constraints. These include: fd (programming over finite domains), clp(R) (solving equations over reals), CHR (constraint handling rules).

Lecturers

- Logic Programming (weeks 1-6)

Stephane Bressan
COM1-03-44
6516 3543
steph@nus.edu.sg

- Constraint Logic Programming (weeks 7-12)

Joxan Jaffar
COM1-03-11
6516 4782
dcsjj@nus.edu.sg

Textbooks

- Logic Programming

Sterling and Shapiro

- Constraint Logic Programming

Marriott and Stuckey

Communication

- IVLE
  - Announcements
  - Lesson plan
  - Email
  - Forum
  - Workbin

Assessment

- final examination (50%)
- Quizzes (15%)
- Home assignments (15%)
- Project (20%)

Prolog Program

```
/* hello.pl */

:-writeln(“Hello World!”).
```

Prolog Program

```
/* royal.pl */

parent(X, Y):- father(X, Y).
parent(X, Y):- mother(X, Y).

grand_parent(X, Y):- parent(X, Z), parent(Z, Y).

% from  http://128.118.2.23/~saw/royal/

father("Louis XVIII, King of France", "Dauphin Louis").
father("Maria of Poland LECZINSKA", "Stanislaw LECZINSKI, King of Poland").
father("Dauphin Louis", "Louis XV, King of France").
father("Louis XV, King of France", "Louis, Duke of Burgundy").

mother("Dauphin Louis", "Maria of Poland LECZINSKA").
mother("Louis XVIII, King of France", "Marie-Josephe de Saxe").
mother("Louis XV, King of France", "Marie Adelaide of Savoy").

%
:-grand_parent("Louis XVIII, King of France", Y).
```

Intuitively: Facts

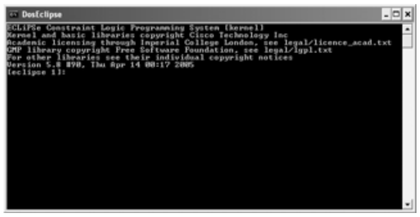
- Dauphin Louis is the father of Louis XVIII, King of France

```
father("Louis XVIII, King of France", "Dauphin Louis").
```

ECLiPSe Prolog

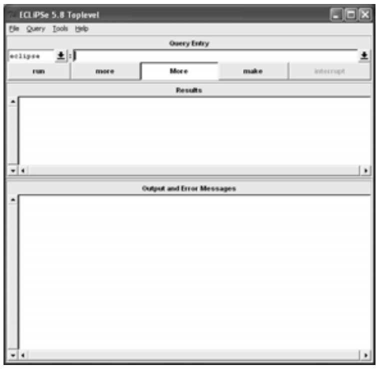
- ECLiPSe is a free and open-source Prolog system
- http://eclipse.crosscoreop.com/

Using ECLiPSe

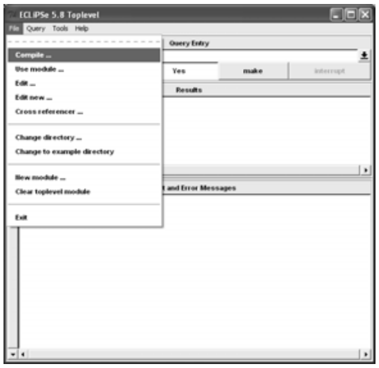


```
ECLiPSe Constraint Logic Programming System [kernel]
Kernel and basic libraries copyright Cisco Technology Inc
Academic licensing through Imperial College London, see legal/licence_acad.txt
GMP library copyright Free Software Foundation, see legal/lgpl.txt
For other libraries see their individual copyright notices
Version 5.8 #90, Thu Apr 14 00:17 2005
[eclipse 1]:
```

Using ECLiPSe



Using ECLiPSe



Intuitively: Goals

- is Dauphin Louis the father of Louis XVIII, King of France?

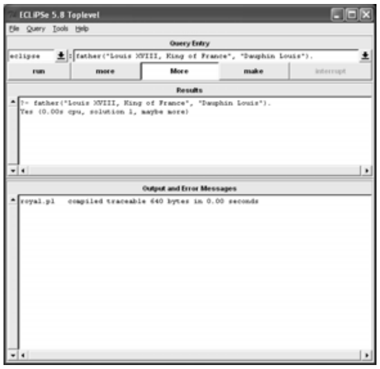
:- father("Louis XVIII, King of France", "Dauphin Louis").

Notice the message ‘solution 1, maybe more’.

What happens if we ask for more?

How does it compare to procedure in procedural languages?

Intuitively: Goals



```
eclipse : father("Louis XVIII, King of France", "Dauphin Louis").
```

```
?- father("Louis XVIII, King of France", "Dauphin Louis").
Yes (0.00s cpu, solution 1, maybe more)
```

```
royal.pl   compiled traceable 640 bytes in 0.00 seconds
```

Intuitively: Goals

- Who is the father of Louis XVIII, King of France?

:- father("Louis XVIII, King of France", Y).

Y = "Dauphin Louis"

We say that the variable Y is **unified** with the string "Dauphin Louis"

Intuitively: Variable and Unification vs Assignment

Y := "Dauphin Louis “; Y:= "Louis XVIII, King of France“;

:- Y = "Dauphin Louis ", Y= "Louis XVIII, King of France“.

Intuitively: Goals

- Who is the child of Dauphin Louis?

:-father(X, "Dauphin Louis").

Intuitively: Goals

- Who’s the father of who?

:-father(X, Y).

How does it compare to procedure call in procedural languages?

success/failure versus call/return

Intuitively: Rules

- Y is parent of X if Y is the father of X

parent(X, Y):- father(X, Y).

- Or, the mother

parent(X, Y):- mother(X, Y).

Intuitively: Rules

- if Y is the father of X
  then Y is parent of X

- Or if Y is mother of X and Y
  then Y is parent of X

(father(X, Y) ⇒ parent(X, Y)) ∨ (mother(X, Y) ⇒ parent(X, Y))

Intuitively: Rules

- If Z is parent of X and Y is parent of Z then Y is grand parent of Y

grand_parent(X, Y):- parent(X, Z), parent(Z, Y).

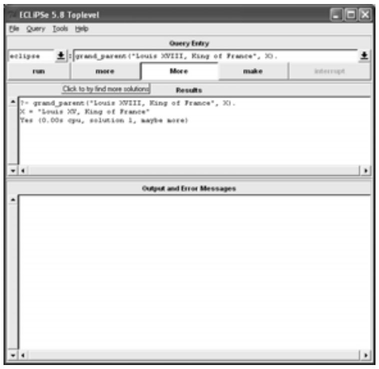
parent(X, Z) ∧ parent(Z, Y) ⇒ grand_parent(X, Y)

Intuitively: Goals

- Who is grand parent of Louis XV, King of France?

:-grand_parent("Louis XVIII, King of France", Y).

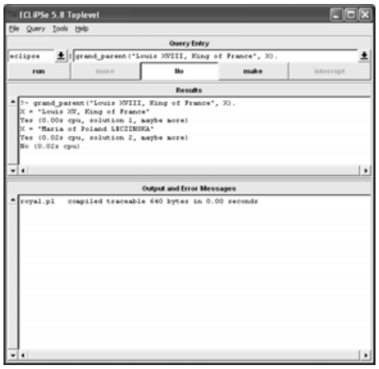
Using ECLiPSe



eclipse : grand_parent("Louis XVIII, King of France", X).

```
?- grand_parent("Louis XVIII, King of France", X).
X = "Louis XV, King of France"
Yes (0.00s cpu, solution 1, maybe more)
```

Using ECLiPSe

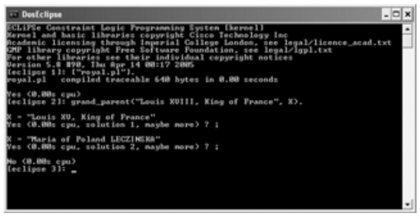


```
eclipse : grand_parent("Louis XVIII, King of France", X).
```

```
?- grand_parent("Louis XVIII, King of France", X).
X = "Louis XV, King of France"
Yes (0.00s cpu, solution 1, maybe more)
X = "Maria of Poland LECZINSKA"
Yes (0.02s cpu, solution 2, maybe more)
No (0.02s cpu)
```

```
royal.pl   compiled traceable 640 bytes in 0.00 seconds
```

Using ECLiPSe



```
ECLiPSe Constraint Logic Programming System [kernel]
Kernel and basic libraries copyright Cisco Technology Inc
Academic licensing through Imperial College London, see legal/licence_acad.txt
GMP library copyright Free Software Foundation, see legal/lgpl.txt
For other libraries see their individual copyright notices
Version 5.8 #70, Thu Apr 14 00:17 2005
[eclipse 1]: ["royal.pl"].
royal.pl   compiled traceable 640 bytes in 0.00 seconds

Yes (0.00s cpu)
[eclipse 2]: grand_parent("Louis XVIII, King of France", X).

X = "Louis XV, King of France"
Yes (0.00s cpu, solution 1, maybe more) ? ;

X = "Maria of Poland LECZINSKA"
Yes (0.00s cpu, solution 2, maybe more) ? ;

No (0.00s cpu)
[eclipse 3]: _
```

Prolog Program

- A prolog program consists of a list of ***clauses***

```
parent(X, Y):- father(X, Y).
parent(X, Y):- mother(X, Y).

grand_parent(X, Y):- parent(X, Z), parent(Z, Y).

father("Louis XVIII, King of France", "Dauphin Louis").
father("Maria of Poland LECZINSKA", "Stanislaw LECZINSKI, King of Poland").
father("Dauphin Louis", "Louis XV, King of France").
father("Louis XV, King of France", "Louis, Duke of Burgundy").

mother("Dauphin Louis", "Maria of Poland LECZINSKA").
mother("Louis XVIII, King of France", "Marie-Josephe de Saxe").
mother("Louis XV, King of France", "Marie Adelaide of Savoy").

:-grand_parent("Louis XVIII, King of France", Y).
```

Clauses

- A clause has a ***head*** and a ***body*** separated by the symbol ‘:-’ and ends with dot ‘.’

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

- If the head is empty the clause is a ***goal***

```
:-grand_parent("Louis XVIII, King of France", Y).
```

- If the body is empty the clause is a ***fact***

```
father("Louis XVIII, King of France", "Dauphin Louis").
```

- Otherwise it is sometimes referred to as a ***rule***

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

Clauses

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

- The head of a clause is formed of one ***literal***

```
grand_parent(X, Y)
```

- The body of clause is a list (a ***conjunction***) of zero or more ***literals*** separated by commas ‘,’

```
parent(X, Z), parent(Z, Y)
```

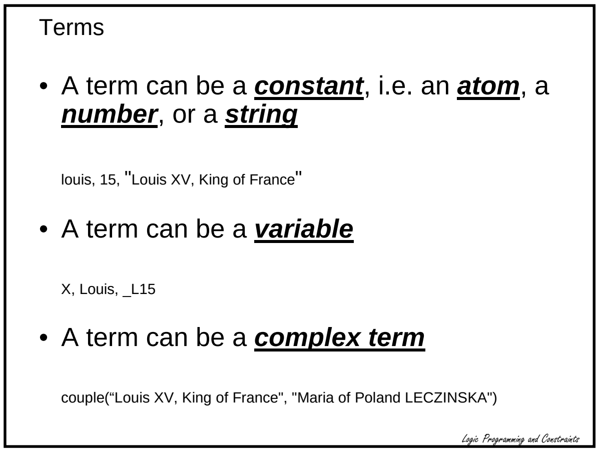
Literals

- A literal is formed of a ***predicate*** and its ***arguments***

```
grand_parent             (X, Y)
```

- Arguments of a predicate are ***terms***
- The number of arguments of a predicate is called its ***arity***

```
grand_parent/2
```



Terms

- A term can be a ***constant***, i.e. an ***atom***, a ***number***, or a ***string***

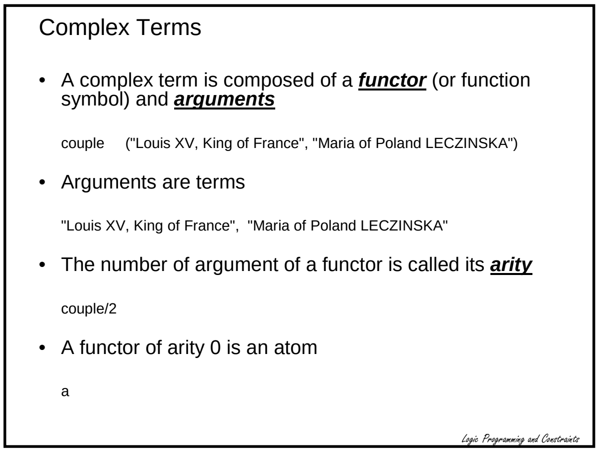
louis, 15, "Louis XV, King of France"

- A term can be a ***variable***

X, Louis, _L15

- A term can be a ***complex term***

couple(“Louis XV, King of France", "Maria of Poland LECZINSKA")



Complex Terms

- A complex term is composed of a ***functor*** (or function symbol) and ***arguments***

couple ("Louis XV, King of France", "Maria of Poland LECZINSKA")

- Arguments are terms

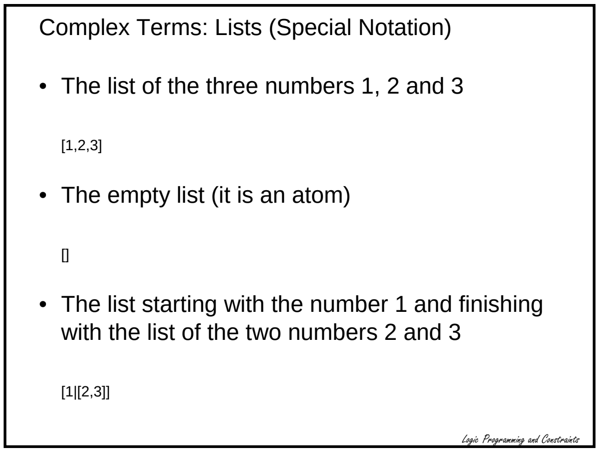
"Louis XV, King of France", "Maria of Poland LECZINSKA"

- The number of argument of a functor is called its ***arity***

couple/2

- A functor of arity 0 is an atom

a



Complex Terms: Lists (Special Notation)

- The list of the three numbers 1, 2 and 3

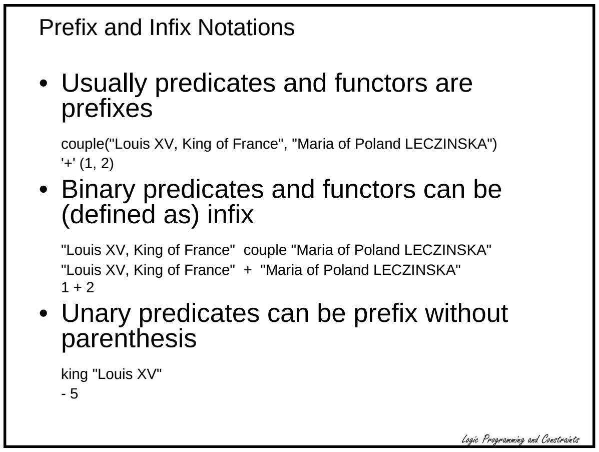
[1,2,3]

- The empty list (it is an atom)

[]

- The list starting with the number 1 and finishing with the list of the two numbers 2 and 3

[1|[2,3]]



Prefix and Infix Notations

- Usually predicates and functors are prefixes

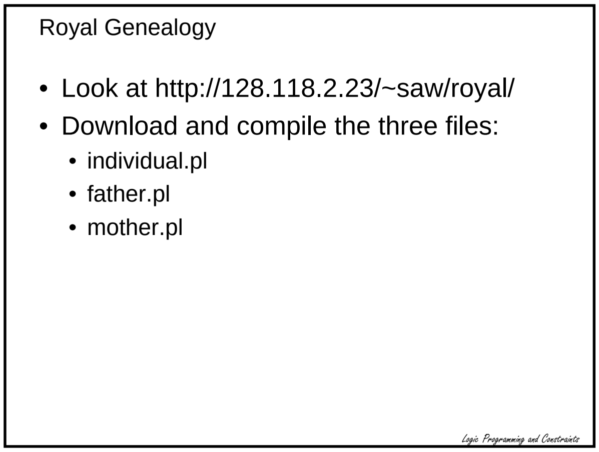
couple("Louis XV, King of France", "Maria of Poland LECZINSKA")
'+' (1, 2)

- Binary predicates and functors can be (defined as) infix

"Louis XV, King of France" couple "Maria of Poland LECZINSKA"
"Louis XV, King of France" + "Maria of Poland LECZINSKA"
1 + 2

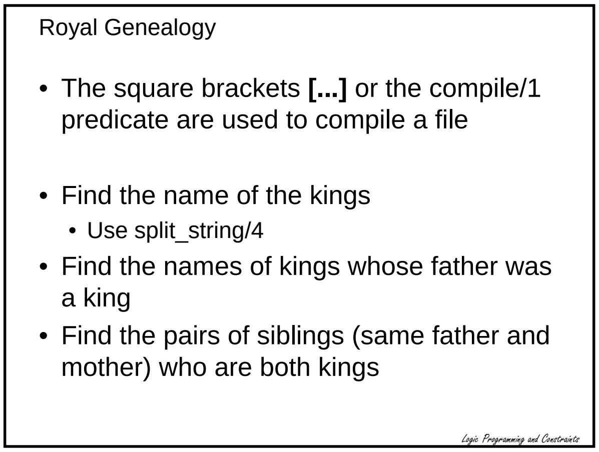
- Unary predicates can be prefix without parenthesis

king "Louis XV"
- 5



Royal Genealogy

- Look at http://128.118.2.23/~saw/royal/
- Download and compile the three files:
  - individual.pl
  - father.pl
  - mother.pl



Royal Genealogy

- The square brackets **[...]** or the compile/1 predicate are used to compile a file
- Find the name of the kings
  - Use split_string/4
- Find the names of kings whose father was a king
- Find the pairs of siblings (same father and mother) who are both kings

Built-in Arithmetic

- ECLiPSe has several numeric types:
  - Integers
  - Rationals
  - Floating Point Numbers
  - Bounded Real Numbers
- ECLiPSe has built-in arithmetic predicates/functions on numeric data

Arithmetic Predicates/Functions

- '+'(1, 1, 2).
- '+'(1, 1, 3).
- '+'(1, 1, X).
- '+'(1, X, 2).
- instantiation fault in +(1, X, 2)

Arithmetic Predicates/Functions

- plus(1, 1, 2).
- plus(1, 1, 3).
- plus(1, 1, X).
- plus(X, 1, 2).
- plus(1, X, 2).
- plus(X, Y, 2).
- See times/3

Arithmetic Expressions

- The predicate is/2 evaluates its second argument if it is an arithmetic expression and unifies it with the first argument.
- If the first and second arguments are not of the same type or the second is not an arithmetic expression it yields an error (but there is some type coercion)

is(X, 1 + 1).

X is 1 + 1.

2 is 1 + 1.

2.0 is 1 + 1.

2.0 is 1.0 + 1.

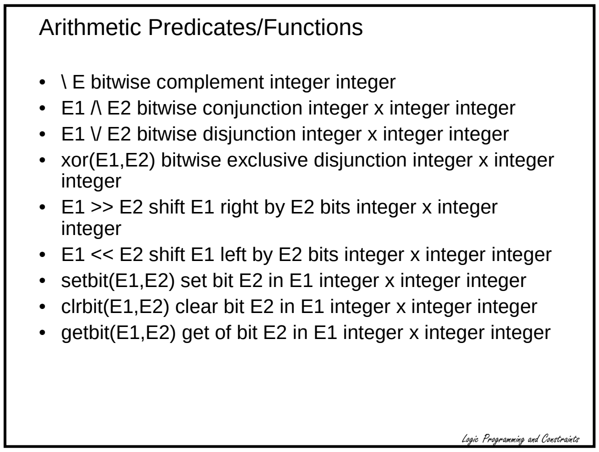
X is blabla. (notice the error message)

Arithmetic Expressions

- **Expr1 < Expr2**
  - succeeds if (after evaluation and type coercion) Expr1 is less than Expr2.
- **Expr1 >= Expr2**
  - succeeds if (after evaluation and type coercion) Expr1 is greater or equal to Expr2.
- **Expr1 > Expr2**
  - succeeds if (after evaluation and type coercion) Expr1 is greater than Expr2.
- **Expr1 =< Expr2**
  - succeeds if (after evaluation and type coercion) Expr1 is less or equal to Expr2.
- **Expr1 =:= Expr2**
  - succeeds if (after evaluation and type coercion) Expr1 is equal to Expr2.
- **Expr1 =\= Expr2**
  - succeeds if (after evaluation and type coercion) Expr1 is not equal to Expr2.

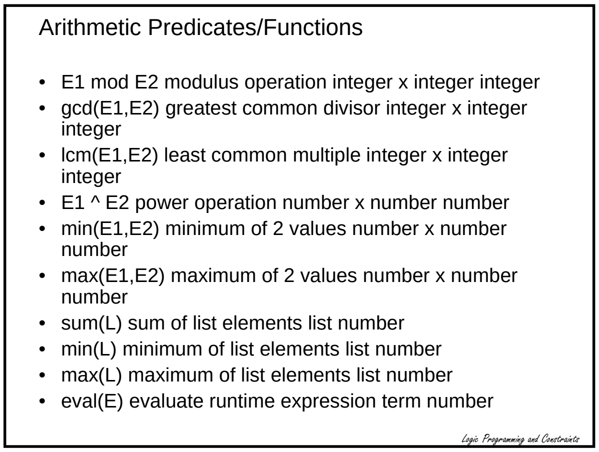
Arithmetic Predicates/Functions

- + E unary plus number number
- - E unary minus number number
- abs(E) absolute value number number
- sgn(E) sign value number integer
- floor(E) round down to integral value number number
- ceiling(E) round up to integral value number number
- round(E) round to nearest integral value number number
- E1 + E2 addition number x number number
- E1 - E2 subtraction number x number number
- E1 * E2 multiplication number x number number
- E1 / E2 division number x number see below
- E1 // E2 integer division integer x integer integer



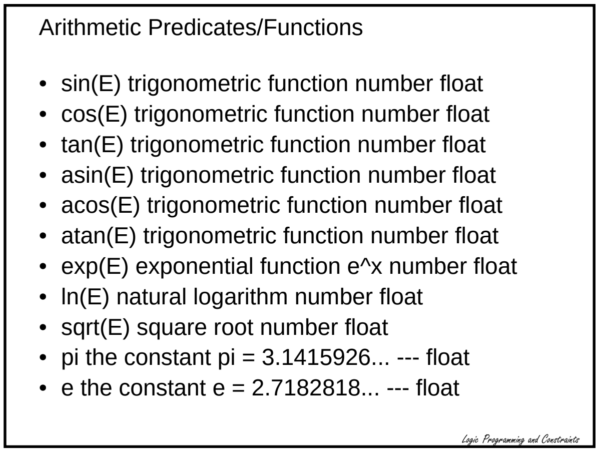
Arithmetic Predicates/Functions

- \ E bitwise complement integer integer
- E1 /\ E2 bitwise conjunction integer x integer integer
- E1 \/ E2 bitwise disjunction integer x integer integer
- xor(E1,E2) bitwise exclusive disjunction integer x integer integer
- E1 >> E2 shift E1 right by E2 bits integer x integer integer
- E1 << E2 shift E1 left by E2 bits integer x integer integer
- setbit(E1,E2) set bit E2 in E1 integer x integer integer
- clrbit(E1,E2) clear bit E2 in E1 integer x integer integer
- getbit(E1,E2) get of bit E2 in E1 integer x integer integer



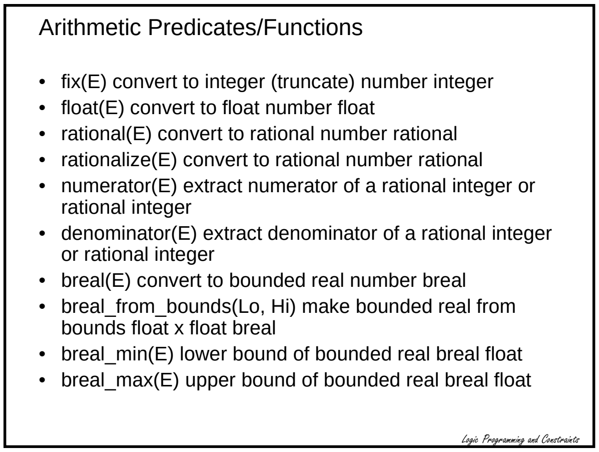
Arithmetic Predicates/Functions

- E1 mod E2 modulus operation integer x integer integer
- gcd(E1,E2) greatest common divisor integer x integer integer
- lcm(E1,E2) least common multiple integer x integer integer
- E1 ^ E2 power operation number x number number
- min(E1,E2) minimum of 2 values number x number number
- max(E1,E2) maximum of 2 values number x number number
- sum(L) sum of list elements list number
- min(L) minimum of list elements list number
- max(L) maximum of list elements list number
- eval(E) evaluate runtime expression term number



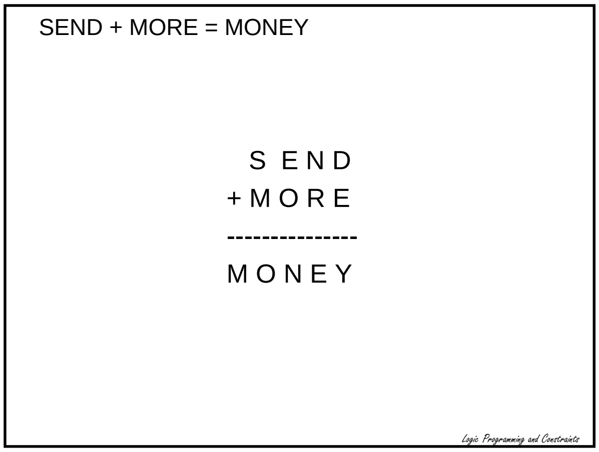
Arithmetic Predicates/Functions

- sin(E) trigonometric function number float
- cos(E) trigonometric function number float
- tan(E) trigonometric function number float
- asin(E) trigonometric function number float
- acos(E) trigonometric function number float
- atan(E) trigonometric function number float
- exp(E) exponential function e^x number float
- ln(E) natural logarithm number float
- sqrt(E) square root number float
- pi the constant pi = 3.1415926... --- float
- e the constant e = 2.7182818... --- float



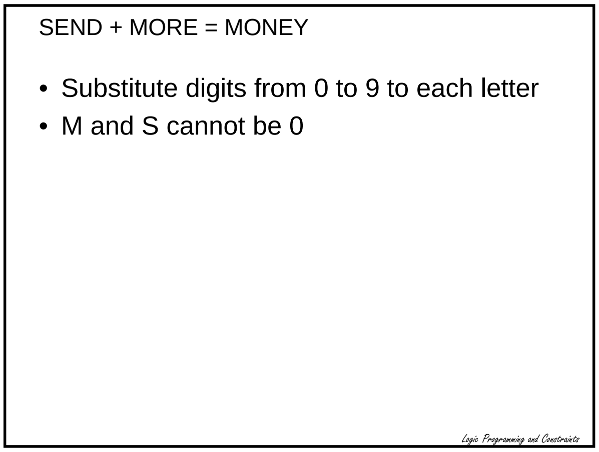
Arithmetic Predicates/Functions

- fix(E) convert to integer (truncate) number integer
- float(E) convert to float number float
- rational(E) convert to rational number rational
- rationalize(E) convert to rational number rational
- numerator(E) extract numerator of a rational integer or rational integer
- denominator(E) extract denominator of a rational integer or rational integer
- breal(E) convert to bounded real number breal
- breal_from_bounds(Lo, Hi) make bounded real from bounds float x float breal
- breal_min(E) lower bound of bounded real breal float
- breal_max(E) upper bound of bounded real breal float



SEND + MORE = MONEY

```
  SEND
+MORE
---------------
MONEY
```



SEND + MORE = MONEY

- Substitute digits from 0 to 9 to each letter
- M and S cannot be 0

SEND + MORE = MONEY

```
  9 0 0 0
+ 1 0 0 0
---------------
1 0 0 0 0
```

SEND + MORE = MONEY

```
smm :-
    d(D), e(E), m(M), n(N), o(O), r(R), s(S), y(Y),
    writeln([D, E, M, N, O, R, S, Y]),
            1000*S + 100*E + 10*N + D +
            1000*M + 100*O + 10*R + E =:=
      10000*M + 1000*O + 100*N + 10*E + Y,
        writeln("Solution> "), write([D, E, M, N, O, R, S, Y]).


d(0).
d(1).
d(2).
d(3).
d(4).
d(5).
d(6).
d(7).
d(8).
d(9).
```

SEND + MORE = MONEY

- Substitute digits from 0 to 9 to each letter
- M and S cannot be 0
- Each letter is a different digit

SEND + MORE = MONEY

```
  9 5 6 7
+ 1 0 8 5
---------------
1 0 6 5 2
```

SEND + MORE = MONEY

```
:-lib(fd).
smm :-
    d(D), e(E), m(M), n(N), o(O), r(R), s(S), y(Y),
    writeln([D, E, M, N, O, R, S, Y]),
            1000*S + 100*E + 10*N + D +
            1000*M + 100*O + 10*R + E =:=
      10000*M + 1000*O + 100*N + 10*E + Y,
    alldistinct([S,E,N,D,M,O,Y]),
        writeln("Solution> "), write([D, E, M, N, O, R, S, Y]).

d(0).
d(1).
d(2).
d(3).
d(4).
d(5).
d(6).
d(7).
d(8).
d(9).
```

Zebra Puzzle

There are five houses numbered 1 to 5. They are painted a different color. Their inhabitants are of different national extractions, own different pets, drink different beverages and smoke different brands of cigarettes

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately after the ivory house.
6. The Old Gold smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house.
10. The man who smokes Chesterfields lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

Now, who drinks water? Who owns the zebra?

Solution

- **House:** 1 2 3 4 5
- **Color:** yellow blue red ivory green
- **Nationality:** Norwegian Ukrainian Englishman Spaniard Japanese
- **Drink:** water teamilkorange juice coffee
- **Smoke:** Kools Chesterfield OldGold LuckyStrike Parliament
- **Pet:** fox horse snails dog zebra

Credits

Clipart and media are licensed from Microsoft Office Online Clipart and Media