Prolog

- logic programming language
- developed in 70s in Europe
- used for AI and computability theory
- original interpreter was written in Fortran, by Colmerauer in France, but developed extensively in Edinburgh, Scotland.

Model of Computation

- declarative, based on logical proof
- procedural, based on constraint satisfaction

Constraint problem: system of linear equations

e.g. $2y = x + 5$

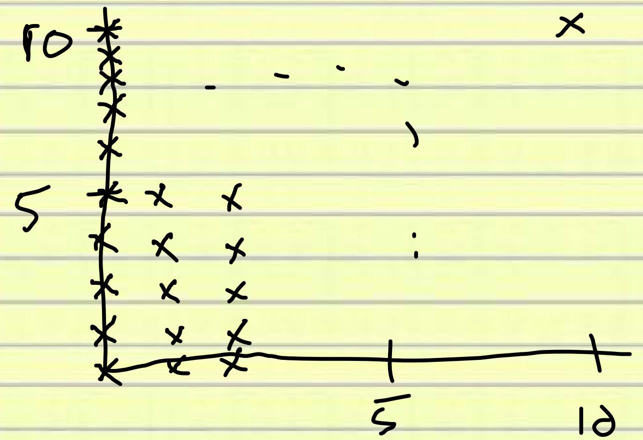
$y = 2x - 2$

Can we solve this system, using algebra and arithmetic?

e.g. multiply eq. 2 by 2 and subtract

$$2y = x + 5$$
$$2y = 4x - 4$$
$$0 = -3x + 9$$
$$3x = 9$$
$$x = 3$$
$$2y = 3 + 5$$
$$2y = 8$$
$$y = 4$$

solution $x = 3, y = 4$



Now we can find a solution by searching — choose a value for x in the solution, then choose a value for y. If the chosen pt. is a soln, stop, else continue the search by choosing different values for x, y until the search space (finite) is exhausted, in which case there is no. soln.

Prolog works by exhaustively searching a finite space for a solution to a system of constraints.

calculate from $2y = x + 5$

y	x	$2x-2$
$\emptyset$	–	–
1	–	–
2	–	–
3	1	0
4	3	4
5	5	8
6	7	–
7	9	–
8	–	–
9	–	–
10	–	–

— soln. we are looking for

the finite search guarantees a soln. if one exists.

Prolog will therefore find it.

We will write a Prolog program to solve those equations.

* without using arithmetic

First thing we need is a way to represent numbers.

Prolog has atoms, just like Scheme.

ø is represented by nø
1 " " " n1
.
.
10 " " " n1ø

Next comes basic properties of numbers. We will define a relationship between 2 numbers, called inc.

```
inc(nø, n1).
inc(n1, n2).
inc(n2, n3).
inc(n3, n4).
...
inc(n9, n1ø).
```

set of facts about integers from ø to 1ø.

Next step is to represent the inverse of inc, called dec.

Could write a set of facts:

```
dec(n10, n9).
dec(n9, n8).
   ⋮
dec(n1, n0).
```

However we will collect ~~all~~ of these facts into a rule — relate inc to dec.

```
dec(X,Y) :- inc(Y,X).
```

'if'

For all X, Y if the increment of Y is X then the decrement of X is Y

Prolog is an interpreter, the prompt is ?-

?- inc(n0, n1).

Prolog searches the set of facts for inc and finds a match, so it answers 'yes'

?- inc(n3, n7).

no

?- dec(n5, n4).

This requires use of the rule. The variables X, Y are bound to the atoms in the query. X = n5, Y = n4. This is true if inc(n4, n5) is true. Again Prolog searches the facts and finds a match. ∴ dec(n5, n4) is true, and Prolog answers yes.

?- dec(n5, n3).

no

Prolog matches a query with either a fact, if there is one, or with the LHS (the 'head') of a rule. For a rule, the RHS (the 'body') becomes a new query, with any variable bindings substituted.

So Prolog has a semantics rather like a 'clever' database. Matching is done in a search loop, looking at all facts and or rules.

Prolog can also answer queries with variables:

?- inc(X, n3).

i.e. Whose inc is n3?

Again a search of facts is done, and a match with inc(n2,n3) is found with X = n2, which is therefore the answer.

Prolog matching is called unification and is quite sophisticated in full Prolog. Unification can match any two terms, with or without variables. A variable in one term matches the corresponding part of the other term.

Now we need a way to define addition. We need a strategy (an algorithm) It's going to be recursive. (Prolog has no regular control structures).

We will use the fact the $X + Y = (X+1) + (Y-1)$

When $Y-1$ reaches zero, the other one will be the answer.

'and'

```
add(X, nø, X).
add(X, Y, Z) :- inc(X, X1), dec(Y, Y1),
                add(X1, Y1, Z).
```

e.g. $3+2 = 4+1 = 5+0$

?- add(n1, n2, X).

This query does not the fact add(X, n0, X)
But it does match the head of the rule

query : add(n1, n2, X)

LHS : add(X, Y, Z)

We can unify each term, so the head is matched.

with $X_1 = n1$
$Y_1 = n2$
$Z_1 = X$ ← unifying two variables as aliases

The RHS now becomes inc(n1, X1), dec(n2, Y1), add(X1, Y1, X)

We proceed from L to R, attempting to match each new query.