What is $ECL^{PS}\text{e}$? 

"$ECL^{PS}\text{e}$ is designed for solving combinatorial optimization problems, for the development of new constraint solver technology and their hybrids, and for the teaching of modelling, solving and search techniques."

(http://sourceforge.net/projects/eclipse-clp/)

$ECL^{PS}\text{e}$:

1. is a declarative language
2. is a Constraint Programming toolkit
3. is available open source (Mozilla Public License)
4. has been applied to areas of planning, scheduling, resource allocation, timetabling, transport...
5. provides bindings for Python, C/C++, Tcl/Tk, Java (Eclipse IDE plugin), SQL, ...
6. can be interfaced to remotely (RPC)
Declarative programming with logic

The TPK algorithm in ECLiPS© / Prolog:

```prolog
f(T, Y) :-
Y is sqrt(abs(T)) + 5*T^3.
main :-
read(As),
length(As, N), reverse(As, Rs),
foreach(Ai, Rs, for(I, N-1, 0, -1) do
  Bi is f(Ai),
  ( Bi > 400 ->
    printf("\%w TOO LARGE\n", I)
  ;
    printf("\%w \%w\n", [I, Bi])
  )).
```

(Dymchenko & Mykhailova, 2014)

Declarative programming with functions

The TPK algorithm in python 3:

```python
>>> def f(x): return abs(x)**0.5 + 5*x**3
>>> print(', '.join('x:%s' % (x, v if v <=400 else ' TOO LARGE!') for x,v in ((y, f(float(y))) for y in input( 'numbers : ').strip().split()[:11][::-1])))
```

(http://rosettacode.org/wiki/Trabb_Pardo%E2%80%93Knuth_algorithm#Python)

Is the logical Prolog version "easier/better" than the functional Python version?

Python binding for ECLiPS©

The “Search Solution” example of pyclp:

```python
from pyclop import *
init() # Init ECLiPSe engine
Compound("Lib",Atom("ic")).post_goal() # Load ic library
A_var=Var() # Create variable A
B_var=Var() # Create variable B
# [A,B]#::1..10
Compound(#::,PList([A_var,B_var]),
  Compound(#:<,1,10)).post_goal()
Compound(#:<,[A_var,B_var]).post_goal() # A<B
Compound(#:<,[A_var,B_var]).post_goal() # A>B
# labeling([A,B])
Compound("labeling",
  PList([A_var,B_var])).post_goal()
# Loop on all solution and print them.
while (resume()[0]==SUCCEED):
  print(B_var)
  # backtracking over solutions
  Atom("fail").post_goal()
  cleanup() # Shutdown ECLiPSe engine
```

(Dymchenko & Mykhailova, 2014)
So again, why ECLiPS\textsuperscript{e}?

⇒ Because it can be considered more elegant and even easier to (only) declare a problem logically and then use all the power of standard reasoning and search algorithms to find a solution. ECLiPS\textsuperscript{e} contains all major CLP algorithms as libraries and provides the full power of the Prolog language for problem modeling.

2 Problem modeling

- Issues
- CLP and ECLiPS\textsuperscript{e}

Suitable problems for ECLiPS\textsuperscript{e}

Characteristics of problems suitable for ECLiPS\textsuperscript{e}:

1. There are no general methods or algorithms
   - NP-completeness
   - Different strategies and heuristics have to be tested
2. Requirements are quickly changing
   - Programs should be flexible enough to adapt
3. Decision support required
   - Co-operate with user
   - Friendly interfaces

Logic programming

Constraint Programming can be characterized by two pseudo-equations:

\textit{Solution} = \textit{Logic} + \textit{Control} \hspace{1cm} (1)
\textit{Control} = \textit{Reasoning} + \textit{Search} \hspace{1cm} (2)

Equation (1): a solution can be found by:

1. a logical, declarative description of the problem, and
2. control information for the computer to deduce it.

Equation (2): control is a combination of:

1. reasoning to (efficiently) limit the search space, and
2. subsequent (inefficient) search through that space

Problem modeling deals with the Logic part of Equation (1).
Issues in Problem Modeling

A good formalism should fulfill the following criteria:

1. Expressive power:
   formal model of real world problem possible?
2. Clarity for humans:
   ease of use of formalism (read, write, understand, modify)
3. Solvability for computers:
   Good methods available to solve problem?

Higher-level models
+ closer to the user and the problem
+ easier to understand and trust, to debug and modify, but
- difficult to see how they can be solved

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 15 / 100

Modeling with CLP and \( ECL^iPS^e \)

Built-in language constructs used in modeling:

- **Build-in constraints**: \( X \ #> \ Y \)
- **Abstraction**:
  \( \text{bef}(\text{task}(S_i,D_i),\text{task}(S_j,D_j)) := S_i + D_j \leq S_j \).
- **Conjunction**:
  \( \text{betw}(X,Y,Z) := X \ #< \ Y, Y \ #< \ Z \).
- **Disjunction**:
  \( \text{neighb}(X,Y) := (X \ #= Y + 1 ; Y \ #= X + 1) \).
- **Iteration**:
  \( \text{not_among}(X,L) := (\text{foreach}(Y,L, \text{param}(X)) \ \text{do} \ X \ #\ = \ Y) \).
- **Recursion**:
  \( \text{not_among}(X,[\ ]). \)
  not_among(X,[Y|Ys]) := X \ #\ = \ Y, \text{not_among}(X,Ys).

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 17 / 100

CLP and \( ECL^iPS^e \)

An example constraint network (Cheadle et al., 2014):

But, of course, one problem can be modeled in multiple ways..
**Motivation**

Problem modeling

Issues

CLP and ECL

ECLiPSe

Prolog & ECLiPSe

Programming in Prolog

Interval Constraints Library

Example

Summary

Literature

---

**Modeling rules**

Both models work fine, but involve different variables and constraints.

⇒ “Finding good models [...] requires substantial expertise and experience.” (Cheadle et al., 2014)

Declarative model is constraint setup code ⇒ should be deterministic and terminating, so **general rules**:

- **Careful with disjunctions**: Don’t leave choice points (i.e., alternatives for backtracking); should be deferred until search phase
- **Use only simple conditionals**: Conditions \((\ldots \rightarrow \ldots ; \ldots)\) must be true or false at modeling time!
- **Use only structural recursion and loops**: Termination conditions must be known at modeling time!

---

**Preliminary summary**

**ECLiPSe**:

- is a declarative constraint programming framework
- interfaces with many programming languages
- is based on the paradigm: Solution = Logic + Control
- uses Prolog for problem modeling

⇒ Next, introduction to Prolog..

---

**3 Prolog & ECLiPSe**

- Terms and their data types
- Predicates, Goals, and Queries
- Conjunctions & Disjunctions
- Unification and Logical Variables
Introduction: Terms and their data types

Prolog data (terms) and programs are built from the following data types:

- **Numbers**
- **Strings**
- **Atoms**
- **Lists**
- **Structures**

They are introduced next..

**Motivation**

**Problem modeling**

**Prolog & ECLiPS**

Terms and their data types

Numbers

Integers can be as large as fits into memory, e.g.:

-123 0 -27 393423874981724

Floating point number (repr. as IEEE double floats), e.g.:

0.0 3.141592653589793 6.02e23 -35e-12 -1.0Inf

Also available: rationals and bounded reals

Beware: Performing arithmetic requires **is/2** predicate:

```
?- X is 3 + 4.
X = 7
Yes
```

Predicate **=\(/2** constructs term corresp. to arithmetic expression:

```
?- X = 3 + 4.
X = 3 + 4
Yes
```

Strings

Strings represent arbitrary sequences of bytes:

```
"I am a string!"
"string with a newline \n and a null \000 character"
```

Strings versus Atoms:

- **Strings**
- **Atoms**

Internally, the data types are quite different:

- **Strings** stored as character sequence
- **Atoms** mapped into internal constant via dictionary table

→ Copying and comparing:

- **Atoms** in unit time
- **Strings** in time proportional to string length

→ However, recollection of freed dictionary memory needs garbage collection

Beware: Atoms should always be preferred when they are involved in unification and matching.
Atoms

Atoms are simple symbolic constants:
1. similar to enumeration type constants in other languages
2. no special meaning attached to them by the language
3. syntactically:
   - all words starting with a lower case letter are atoms
   - sequences of symbols are atoms
   - anything in single quotes is an atom
4. E.g.: atom quark is --?? ??? ’Atom’ ‘an atom’

Lists

Lists are:
1. ordered sequences of (any number of) elements, each of which is itself a term
2. delimited by square brackets ([ ] ) and its elements comma separated

Examples: [1,2,3], [berlin, tokyo, freiburg], ["csp1415", 42, [1,2,3], freiburg]

More notation:
1. empty list: []
2. head and tail: [Head|Tail], with
   - Head a single element
   - Tail a (possibly empty) list
→ Equivalent lists: [1,2,3], [1|[2,3]], [1|[2|[3|[]]]]

Operator syntax (structures)

Prefix, infix, and postfix notation:
1. Unary structures also possible in prefix or postfix notation, e.g., old berta. the same as old(berta).
2. Binary structures also possible in prefix or infix notation, e.g., 1 plus 5. the same as plus(1, 5).
3. these notations need to be declared with
   :- op(+Precedence, +Associativity, ++Name)
4. if in doubt, use display/1 to check parsing of term:
   [eclipse]: display(a+b*c).
   +(a, *(b, c))
   yes.
Summary of data types

1. **Numbers**: ECL/PS® has integers, floats, rationals, and bounded reals
2. **Strings**: character sequences in double quotes
3. **Atoms**: symbolic constants, usually lower case or in single quotes
4. **Lists**: constructed from cells that have an arbitrary head and a tail, which is again a (possibly empty) list
5. **Structures**: have a name and a certain number (arity) of arbitrary arguments, this characteristic is called the functor, and written name/arity

Goals and queries

Predicate examples for integer/1:

- `integer(123)` is true
- `integer(atom)` is false
- `integer([1,2])` is false

These predicate calls are goals.
If supplied by a user as a starting goal, the goal becomes a query, e.g.:

- `?- integer(123).` Yes.
- `?- integer(atom).` No.

Queries always return either Yes. or No.

Conjunctions

Goals are often combined to form **conjunctions** (AND) or **disjunctions** (OR).

Conjunctions:

- are built using commas
- are only true if all conjuncts are true

Examples:

- `?- integer(5), integer(7), integer(9).` Yes.
- `?- integer(5), integer(hello).` No.

Predicates

Other programming languages have procedures and functions

⇒ Prolog and ECL/PS® have predicates:

- a predicate is something that has a truth value
- a predicate definition defines what is true
- a predicate invocation or call checks its truth value
Disjunctions

Disjunctions:
1. are built using semicolons
2. are true if at least one disjunct is true

Examples:
?- (integer(5); integer(hello); integer(world)).
Yes.
?- (integer(hello); integer(world)).
No.

Use parentheses with disjunctions to clarify the structure.

Symbolic Equality

Equality in Prolog:
→ structural equality by pattern matching
  - two terms only equal, if they have exactly same structure
  - No evaluation of any kind involved

Examples:
1. ?- 3 = 3.
  2. Yes.
  3. ?- 3 = 4.
  4. No.
  5. ?- foo(a,2) = foo(a,2).
  6. Yes.
  7. ?- foo(a,2) = foo(b,2).
  8. No.
  9. ?- +(3,4) = 7.
  10. No.
  11. ?- 3 = 4 = 7.
  12. No.

Logical Variables

Logical variables:
1. are variables in the mathematical sense (not in the usual programming language sense)
2. are placeholders for values which are not yet known, not labels for storage locations
3. are aliases for logical values and refer to terms
4. keep their value once assigned to them
5. are written beginning with an upper-case letter or an underscore, e.g.: X Var Quark _123 R2D2 Wumpus
6. same identifier multiple times in input term denotes the same variable
Unification

With logical variables equality test become much more interesting: **Unification**

**Unification:**

1. is an extension of pattern matching of two terms
2. also causes binding (instantiation, aliasing) of variables in the two terms
3. Idea: instantiate variables such that terms become equal

**Examples:**

\[
\begin{align*}
X = 7 & \quad \text{is true with } X \text{ instantiated to } 7 \\
X = Y & \quad \text{is true with } X \text{ aliased to } Y \text{ (or vice versa)} \\
\text{foo}(X) = \text{foo}(7) & \quad \text{is true with } X \text{ instantiated to } 7 \\
\text{foo}(X,Y) = \text{foo}(3,4) & \quad \text{is true with } X \text{ instantiated to } 3 \text{ and } Y \text{ to } 4 \\
\text{foo}(X,Y) = \text{foo}(3,Y) & \quad \text{is true with } X \text{ instantiated to } 3 \text{ and } Y \text{ to } 4 \\
\text{foo}(X) = \text{foo}(Y) & \quad \text{is true with } X \text{ aliased to } Y \text{ (or vice versa)} \\
\text{foo}(X,X) = \text{foo}(3,4) & \quad \text{is false because no possible value for } X \\
\text{foo}(X,Y) = \text{foo}(3,X) & \quad \text{is false because no possible value for } X
\end{align*}
\]

January 25, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 40 / 100

Summary: Basic Terminology

- **Predicate:** Something that is true or false, depending on its definition and its arguments. Defines a relationship between its arguments.
- **Goal:** A logical formula whose truth value we want to know. A goal can be a conjunction or disjunction of other (sub-)goals.
- **Query:** The initial Goal given to a computation.
- **Unification:** An extension of pattern matching which can bind logical variables (placeholders) in the matched terms to make them equal.
- **Clause:** One alternative definition for when a predicate is true. A clause is logically an implication rule.

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 41 / 100

Comments

Comments can be:

1. **Block comments** enclosed between /\* and */
2. **Line comments** anything following % in a line (unless '%' character is part of a quoted atom or string)

(Nothing more to say.)

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 44 / 100
Clauses and facts

In Prolog a program is a collection of predicates, and a predicate is a collection of clauses. So, what is a clause:

1. defines that something is true
2. simplest form is a fact, which syntactically is a structure or an atom terminated by a full stop, e.g.:
   - capital(london, england).
   - brother(fred, jane).
3. General form of a clause:
   - Head :- Body
   - Where Head is a structure (or atom) and Body is a Goal, e.g.:
     uncle(X,Z) :- brother(X,Y), parent(Y,Z).

Clauses and logical implication

The example clause:

\[ \text{uncle}(X,Z) \leftarrow \text{brother}(X,Y), \text{parent}(Y,Z). \]

is equivalent to the following reverse implication:

\[ \text{uncle}(X,Z) \rightarrow \text{brother}(X,Y) \land \text{parent}(Y,Z) \]

or more precisely:

\[ \forall X \forall Z : \text{uncle}(X,Z) \leftarrow \exists Y : \text{brother}(X,Y) \land \text{parent}(Y,Z) \]

A fact is equivalent to a clause with its Body being true:

\[ \text{brother}(fred, jane) :- \text{true}. \]

Clauses and predicates

One or multiple clauses with the same head functor define(s) a predicate, e.g. (with facts):

1. parent(abe, homer).
2. parent(abe, herbert).
3. parent(homer, bart).
4. parent(marge, bart).

Logically, multiple clauses:

- are read as disjunctions
- define multiple alternative ways a predicate can be true

Another example, the ancestor/2 predicate:

1. ancestor(X,Y) :- parent(X,Y).
2. ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).

Execution scheme ⇒ resolution

Resolution:

- **Given**: set of facts and rules as a program
- **Starting point**: a query as an initial goal to be resolved
- **Resolvent**: set of goals that still have to be resolved
Execution mechanism:

1. Pick one goal from the resolvent. If resolvent is empty, stop.
2. Find all clauses whose head successfully unifies with goal. If no such clause, go to step 6.
3. Select first of these clauses. If more exist, remember remaining ones. (choice point)
4. Unify goal with head of the selected clause. (may instantiate variables both in the goal and in the clause’s body).
5. Prefix this clause body to the resolvent and go to 1.
6. Backtrack: Reset whole computation state to how it was when the most recent choice point was created. Take the clauses remembered in this choice point and go to 3.

Execution scheme example

1. \textbf{ancestor}(X, Y) :- \textbf{parent}(X, Y). \quad \% \text{ clause 1}
2. \textbf{ancestor}(X, Y) :- \textbf{parent}(Z, Y), \textbf{ancestor}(X, Z). \quad \% \text{ clause 2}
3. \textbf{parent}(abe, homer). \quad \% \text{ clause 3}
4. \textbf{parent}(abe, herbert). \quad \% \text{ clause 4}
5. \textbf{parent}(homer, bart). \quad \% \text{ clause 5}
6. \textbf{parent}(marge, bart). \quad \% \text{ clause 6}

With query `?- ancestor(X, bart).` we get:

1. both \textbf{ancestor}/2 predicates can unify the goal
2. but the textually first clause (1) is selected first:
   - Goal (Query): \textbf{ancestor}(X, bart)
   - Selected: \textbf{clause 1}
   - Unifying: \textbf{ancestor}(X, bart) = \textbf{ancestor}(X1, Y1)
   - results in: X=X1, Y1=bart
   - New resolvent: parent(X, bart)
   - More choices: \textbf{clause 2}

3. \textbf{empty resolvent} \Rightarrow \text{execution completes successfully, found first solution } X = \textbf{homer}
4. \textbf{ECLiPS}® returns solution and asks if more solutions wanted
5. If \textbf{yes, backtrack} to most recent \textbf{choice point}
6. any \textbf{variable bindings} done after \textbf{choice point} are \textbf{undone}, here binding of \textbf{X} to \textbf{homer} is undone
   - Goal: \textbf{parent}(X, bart)
   - Selected: \textbf{clause 6}
   - Unifying: \textbf{parent}(X, bart) = \textbf{parent}(marge, bart)
   - results in: X = marge
   - New resolvent: \textbf{parent}(X, bart)
   - More choices: \textbf{clause 2}
Execution scheme example

1. \texttt{ancestor(X,Y) :- parent(X,Y).} % clause 1
2. \texttt{ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).} % clause 2
3. \texttt{parent(abe, homer).} % clause 3
4. \texttt{parent(abe, herbert).} % clause 4
5. \texttt{parent(homer, bart).} % clause 5
6. \texttt{parent(marge, bart).} % clause 6

4. empty resolvent $\Rightarrow$ execution completes successfully, found second solution $X = \text{marge}$.

4. If still more solutions wanted, backtrack to most recent choice point.

4. no further alternatives for parent/2 $\Rightarrow$ check ancestor/2.

Goal: \texttt{ancestor(X,bart)}
Selected: clause 2
Unifying: \texttt{ancestor(X,bart) = ancestor(X1,Y1)}
results in: $Y1 = \text{bart}, X1 = X$
New resolvent: \texttt{parent(Z1, bart), ancestor(X1, Z1)}
More choices: clause 5

5. new resolvent contains two goals: \texttt{parent(Z1, bart)}, \texttt{ancestor(X1, Z1)}

5. Check leftmost first, \texttt{parent(Z1, bart)} $\Rightarrow$ new choice point.

5. Select clause 5 first
Goal: \texttt{parent(Z1, bart)}
Selected: clause 5
Unifying: \texttt{parent(Z1, bart) = parent(homer, bart)}
results in: $Z1 = \text{homer}$
New resolvent: \texttt{ancestor(X1, homer)}
More choices: clause 6

6. Via finding the ancestor of \texttt{homer} a few steps later:
\texttt{ancestor(X,Y) :- parent(X,Y).} $\Rightarrow \text{X = abe}$
\texttt{ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).} $\Rightarrow \text{Z1 would be bound to \texttt{marge} for \texttt{homer}}$
\texttt{X = \text{abe}.}$
\texttt{More? ();}$

6. Finally, \texttt{Z1} would be bound to \texttt{marge} for whom no ancestors are found $\Rightarrow$ False.

Here, the execution terminates.
More control structures

- Call:
  - in Prolog: data == programs
  - both are represented as terms
  - use predicate call to treat terms as goals
  - call(X): at runtime X has to be instantiated, but not at compile time!
  - possible definition of disjunction (;):
    1. \( X ; Y :- \text{call}(X). \)
    2. \( X ; Y :- \text{call}(Y). \)

- All solutions:
  - Alternative to one-by-one solution computation:
    1. \(?- \text{findall}(X, \text{weekday}(X), \text{List}). \)
    2. \( \text{List} = [\text{mo}, \text{tu}, \text{we}, \text{th}, \text{fr}, \text{sa}, \text{su}] \)
    3. Yes
  - See also setof/3 and bagof/3 predicates

Using cut: Overview

Use cut (!) to prune away part of the Prolog search-space.
- Powerful mechanism to improve program performance
- Suppresses unwanted solutions
- BUT: easily mis- or overused!
- Cut does two things:
  1. commit: disregard later clauses for a predicate
  2. prune: Throw away alternative solutions to the goal to the left of the cut

Commit to current clause

Consider the following encoding of the “minimum” predicate:

1. \( \text{min}(X,Y, \text{Min}) :- X < Y, \text{Min} = X. \)
2. \( \text{min}(X,Y, \text{Min}) :- Y =< X, \text{Min} = Y. \)

Problems:
- logically correct, but non-optimal performance
- with :- \( \text{min}(2,3,\text{M}) \). Prolog leaves an open choice point
  ⇒ during backtracking another minimum would be searched for unnecessarily!
- unnecessary choice point:
  1. consumes memory
  2. costs execution time

Solution, use cut:

1. \( \text{min}(X,Y, \text{Min}) :- X < Y, !, \text{Min} = X. \)
2. \( \text{min}(X,Y, Y). \)

Prune alternative solutions

A cut may occur anywhere where a goal may occur:

1. \( \text{first_prime}(X, P) :- \)
2. \( \text{prime}(X, P), !. \)

\text{first_prime}/2:
- returns the first prime number smaller than \( X \)
- calls predicate \text{prime}/2, which generates prime numbers smaller than \( X \) in descending order
- ! (cut) prunes away all remaining solutions
  ⇒ on backtracking no alternatives are tried
5 ECL\textsuperscript{Ps}\textsuperscript{e} Programming

- Control & data structures
- Input and Output
- More functions
- Modules

Structure Notation

Properties of declared structure notation:
- the arguments can be written in any order
- “dummy” arguments with anonymous variables do not need to be written
- the arity of the structure is not implied
- the of-syntax can be used to return index of argument, e.g.:
  \texttt{arg(year of book, B, Y)} is equiv. to \texttt{arg(3, B, Y)}

Loops

To reduce the need for auxiliary predicates $\Rightarrow$ iteration construct:

\begin{verbatim}
( IterationSpecs do Goals )
\end{verbatim}

For example, iteration over a list:

\begin{verbatim}
?- ( foreach (X,[1,2,3]) do writeln(X) )
1 2
3
Yes (0.00s cpu)
\end{verbatim}

If a parameter remains constant across all loop iterations $\Rightarrow$ must be specified explicitly (via \texttt{param}):

\begin{verbatim}
?- Array = [4,3,6,7,8],
  ( for(I,1,5),
    fromto(0,In,Out,Sum),
    param(Array)
  do
    Out is In + Array[I]
  ).
\end{verbatim}
Loops

Possible Iteration Specs:
- `fromto(First, In, Out, Last)` iterate Goals starting with `In = First` until `Out = Last`.
- `foreach(X, List)` iterate Goals with `X` ranging over all elements of `List`.
- `foreacharg(X, StructOrArray)` iterate Goals with `X` ranging over all arguments of `StructOrArray`.
- `foreacharg(X, StructOrArray, Idx)` same as before, but `Idx` is set to the argument position of `X` in `StructOrArray`.
- `for(I, MinExpr, MaxExpr, [Increment])` iterate Goals with `I` ranging over integers from `MinExpr` to `MaxExpr` with optional increment.
- (see http://eclipseclp.org/doc/tutorial/tutorial025.html)

Arrays

Arrays can be of any dimension, indices start at 1, and they are declared with the `dim/2` predicate:

```
?- dim(M,[3,4]).
M = [[_131, _132, _133, _134],
     [_126, _127, _128, _129],
     [_121, _122, _123, _124]]
yes.
```

To query dimensions:

```
?- dim(M,[3,4]), dim(M,D).
D = [3, 4]
yes.
```

To access specific elements, specify its index:

```
?- M = [[_131, _132, _133, _134],
       [_126, _127, _128, _129],
       [_121, _122, _123, _124]], X is M[1,2] + M[2,3].
X = 10
M = [[_131, _132, _133, _134],
     [_126, _127, _128, _129],
     [_121, _122, _123, _124]]
yes.
```

Printing

Printing `ECLiPSe` terms:
- `write(+Stream, ?Term)` write one term in a default format
- `write_term(+Stream, ?Term, +Options)` write one term, format options can be selected
- `printf(+Stream, +Format, +ArgList)` write a string with embedded terms, according to a format string
- `writeq(+Stream, ?Term)`, `write_canonical(+Stream, ?Term)` write one term so that it can be read back
- `put(+Stream, +Char)` write one character

```
   | eclipse 1: | read(X).
   | [3,X,foo(bar),Y].
   | X = [3, X, foo(bar), Y]
   | yes.
```

Reading

Reading `ECLiPSe` terms:
- `read(+Stream, -Term, [Options])` read one fullstop-terminated `ECLiPSe` term.
- `get(+Stream, -Char)` read one character
- `read_string(+Stream, +Terminator, -Length, -String)` read a string up to a certain terminator character
- `read_token(+Stream, -Token, -Class)` read one syntactic token (e.g. a number, an atom, a bracket, etc)

Example:
```
```

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 66 / 100
Matching (one-way unification)

Clauses can use matching (or one-way unification) instead of head unification:

- written with ?- functor instead of :-
- No variables in the caller will be bound

1 [eclipse 1]: [user].
2 p(f(a,X)) ?- writeln(X).
3 ?- p(F).
4 Query failed: ?- p(F)
5 ?- p(f(A,B)).
6 Query failed: ?- p(f(A, B))
7 ?- p(f(a, b)).
8 b

List processing

**ECL/PS®** provides `append/3`, `length/2`, `member/2`, and `sort/2`:

- `append/3`: append or split lists, e.g.
  1 ?- append([1, 2], [3, 4], L).
  2 L = [1, 2, 3, 4]
  3 ?- append(A, [3, 4], [1, 2, 3, 4]).
  4 A = [1, 2]
  5 ?- append([1, 2], B, [1, 2, 3, 4]).
  6 B = [3, 4]

- `length/2`: compute the length of a list or construct list of given length, e.g.
  1 ?- length([1, 2, 3, 4], N).
  2 N = 4
  3 ?- length(List, 4).
  4 List = [1, 2, 3, 4]

List processing

**ECL/PS®** provides `append/3`, `length/2`, `member/2`, and `sort/2`:

- `member/2`: check membership in a list (but `memberchk/2` should be preferred), or backtrack over all list members, e.g.
  1 ?- memberchk(2, [1, 2, 3]).
  2 Yes (0.00 s cpu)
  3 ?- member(X, [1, 2, 3]).
  4 X = 1
  5 More (0.00 s cpu)
  6 X = 2
  7 More (0.01 s cpu)
  8 X = 3
  9 Yes (0.01 s cpu)

- `sort/2`: sort any list and remove duplicates, e.g.
  1 ?- sort([5, 3, 4, 3, 2], Sorted).
  2 Sorted = [2, 3, 4, 5]

Term processing

Generic built-in predicates:

- `=..`: converts structures into lists and vice versa
- `arg/3`: extracts an argument from a structure
- `functor/3`: extracts functor name and arity from structured term
- `term_variables/2`: extracts all variables from arbitrarily complex terms
- `copy_term/2`: creates a copy of a term with fresh variables
Making a module

With `module` directive a new module is declared, e.g.:

1. `:- module(greeting).`
2. `:- export hello/0.`
3. `hello :-`
4. ` who(X),`
5. ` printf("Hello %w!%n", [X]).`
6. `who(world).`
7. `who(friend).`

and with `export` a predicate is exported.

One can now import the module and call its exported predicate, e.g.:

1. `:- module(main).`
2. `:- import greeting. % or 'import hello/0 from greeting.'`
3. `main :-`
4. ` hello. % or (without 'import') 'greeting:hello.'`

Exporting items other than predicates

Most commonly exported items (apart from predicates) are structure and operator declarations:

1. `:- module(data).`
2. `:- export struct(employee(name,age,salary)).`
3. `:- export op(500, xfx, reports_to).`
4. `...`

Import declarations by importing module (`import data.`).

6 Interval Constraints Library

- Constraints
- Global Constraints

Interval Constraints Library

The IC (Interval Constraints) library provides a general interval propagation solver which can be used to solve problems over both integer and real variables.

Load the IC library using either of the following:

1. `:- lib(ic).`
2. `:- use_module(library(ic)).`

The typical top-level structure of a constraint program:

1. `solve(Variables) :-`
2. ` read_data(Data),`
3. ` setup_constraints(Data, Variables),`
4. ` labeling(Variables).`

The `labeling/2` predicate is the search part of the program and part of the IC library.
Built-in constraints

The `ic` library provides the following predicates:

- **Vars :: Domain**
  - constrains `Vars` to take only integer or real values from the domain specified by `Domain`
  - `Domain` can be simple range `Lo .. Hi`, or a list of subranges and/or individual elements (integer variables)
  - type of bounds determines type of the variable
  - also allowed: symbolic bound values `inf`, `+inf` and `-inf`
- **Vars #:: Domain**
  - like `::/2`, but for declaring real variables (never imposes integrality, regardless of the types of bounds)
- **Vars #: Domain**
  - like `::/2`, but for declaring integer variables
- **reals(Vars)** and **integer(Vars)**
  - declares that variables are IC variables / constraints variables to integer values only

Built-in constraints: examples

```
?- X :: -10 .. 10. % integer values from -10 to 10
X = X{-10 .. 10}
Yes
?- X :: -10.0 .. 10.0. % real values from -10 to 10
X = X{-10.0 .. 10.0}
Yes
?- X #:: -10 .. 10. % explicitly declared integer values
X = X{-10 .. 10}
Yes
?- X ::= -10 .. 10. % explicitly declared real values
X = X{-10.0 .. 10.0}
Yes
?- X :: 0 .. 1.0Inf. % integers from zero to infinity
X = X{0 .. 1.0Inf}
Yes
?- X :: 0.0 .. 1.0Inf. % reals from zero to infinity
X = X{0.0 .. 1.0Inf}
Yes
?- X :: [1, 4 .. 6, 9, 10]. % subranges (integers only)
X = X{[1, 4 .. 6, 9, 10]}
Yes
```

Arithmetic constraints

Arithmetic constraints in integral (#) and non-integral ($) version:

- `#=` & `$=`: equality of integer (#) or general ($) expressions
- `#>=` & `$>=`: greater than or equal
- `#<=` & `$=<=`: less than or equal
- `#>` & `$>`: greater than
- `#<` & `$<`: less than
- `#\=` & `$\=`: not equal
- **ac_eq(X,Y,C)**: arc-consistent implementation of `X # Y + C. X` and `Y` are constrained to be integer variables and to have “reasonable” bounds. `C` must be an integer.
Arithmetic constraints

Further notes:
- \( \frac{X}{2} + \frac{Y}{2} \neq 1 \) and \( X + Y \neq 2 \) are different constraints \( \Rightarrow \) the first constraints \( X \) and \( Y \) to be even
- Except for disequality and \( \text{ac_eq}/3 \), all basic constraints propagate bound information (performing interval reasoning), e.g.:
  1. ?- [X, Y] :: 0 .. 10, X #>= Y + 2.
  2. X = X(2 .. 10).
  3. Y = Y(0 .. 8).
  4. There is 1 delayed goal.
  5. Yes
- delayed goal indicates that still some combinations of values for \( X \) and \( Y \) violate the constraint
- constraint remains until no such violation is possible anymore

January 26, 2015 Becker-Asano, Nebel and Wißhof – Constraint Satisfaction Problems 83 / 100

Constraint Expression Connectives

Once it is known that \( X \#=< Y - 6 \) cannot be true, the constraint \( X \#>= Y + 6 \) is enforced.

Example:
  1. ?- [X, Y] :: 0 .. 10, X #>= Y + 6 or X #=< Y - 6, X #>= 5.
  2. Y = Y(0 .. 4).
  3. X = X(6 .. 10).
  4. There are 3 delayed goals.
  5. Yes

January 26, 2015 Becker-Asano, Nebel and Wißhof – Constraint Satisfaction Problems 84 / 100

Global constraints

The IC (Interval Constraints) has optional components, which provide global constraints \( \Rightarrow \) high-level constraints that provide more global reasoning than the ones in main IC library.
- contained in \( \text{ic_global}, \text{ic_cumulative}, \text{ic_edge_finder} \), and \( \text{ic_edge_finder3} \)
- To use, e.g., \( \text{ic_global} \):
  1. \(:= \text{lib(ic_global).} \)
  2. \(:= \text{use_module(library(ic_global)).} \)
- Note: some predicates appear in more than one library, e.g., \( \text{ic:alldifferent/1} \) and \( \text{ic_global:alldifferent/1} \)

January 26, 2015 Becker-Asano, Nebel and Wißhof – Constraint Satisfaction Problems 85 / 100
Different strength of propagation

Compare the `ic:alldifferent/1` version:
1 ?- [X1, X2] :: 1 .. 2, [X3, X4] :: 1 .. 4,
2  ic:alldifferent([X1, X2, X3, X4]).
3 X1 = X1([1, 2])
4 X2 = X2([1, 2])
5 X3 = X3([1 .. 4])
6 X4 = X4([1 .. 4])
7 There are 4 delayed goals.
8 Yes

with the `ic_global:alldifferent/1` version:
1 ?- [X1, X2] :: 1 .. 2, [X3, X4] :: 1 .. 4,
2  ic_global:alldifferent([X1, X2, X3, X4]).
3 X1 = X1([1, 2])
4 X2 = X2([1, 2])
5 X3 = X3([3, 4])
6 X4 = X4([3, 4])
7 There are 2 delayed goals.
8 Yes

Trade-off: longer propagation time for `ic_global` version

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 87 / 100

N-Queens problem

N-Queens
Place N queens on an \( N \times N \) chessboard so that no queen attacks another. A queen attacks all cells in horizontal, vertical and diagonal direction.

- certain solutions for all sizes can be constructed
- not a hard problem
- long history in AI and CP papers

N-Queens problem: Column based Model

- A 1..N variable for each column, stating position of queen in the column
- ...because exactly one queen needed for each column
- \( N \) variables \( \Rightarrow N^2/2 \) binary constraints

\[
\begin{align*}
\forall 1 \leq i \leq N : & \quad X_i \in 1..N \\
\forall 1 \leq i < j \leq N : & \quad X_i \neq X_j \\
\forall 1 \leq i < j \leq N : & \quad X_i \neq X_j + i - j \\
\forall 1 \leq i < j \leq N : & \quad X_i \neq X_j + j - i
\end{align*}
\]

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 91 / 100
Possible program

One possible ECL/PS\textsuperscript{8} -program is:

```
1 :- module(nqueen).
2 :- export(top/0).
3 :- lib(ic).
4 top:-
5 nqueen(B,L), writeln(L).
6 nqueen(N,L):-
7 length(L,N),
8 L :: 1..N,
9 alldifferent(L),
10 noattack(L),
11 labeling(L).
```

(see: http://4c.ucc.ie/~hsimonis/ELearning/nqueen/slides.pdf)

- noattack/2 defines binary constraints

Search improvement

The search predicate:

```
1 [..]
2 top:-
3 nqueen(B,L), writeln(L).
4 nqueen(N,L):-
5 length(L,N),
6 L :: 1..N,
7 alldifferent(L),
8 noattack(L),
9 labeling(L), % <= change this line
```

- last line can be changed to:

```
1 search(L,0,first_fail,indomain,complete,[])
```

- packaged search library in ic constraint solver
- provides many alternative search methods
- just select the right keywords as arguments

Binary constraints

Definition of noattack/2:

```
1 noattack([]).
2 noattack([H|T]):-
3 noattack1(H,T ,1) ,
4 noattack (T).
5 noattack1(_,[],_).
6 noattack1 (X ,[Y|R],N):-
7 X #\= Y+N,
8 Y #\= X+N,
9 N1 is N+1 ,
10 noattack1 (X,R,N1).
```

(see: http://4c.ucc.ie/~hsimonis/ELearning/nqueen/slides.pdf)

Search predicate parameters

```
search(+L, ++Arg, ++Select, +Choice, ++Method, +Option):
```

- L: List / collection of terms / domain variables
- Arg: 0 if List L is list of domain variables, greater than 0 if L consists of terms with arity at least Arg
- Select: name of variable selection method (input_order, first_fail, smallest, largest,...)
- Choice: name of value choice method (indomain, indomain_min, indomain_max, indomain_middle,...)
- Method: tree search method (complete, bbs(Steps:integer), lds(Disc:integer),...)
- Option: list of option terms

```
search(L,0,first_fail,indomain,complete,[]):
```

```
search(L,1,input_order,indomain,complete,[]):
```

```
search(L,2,first_fail,indomain,complete,[]):
```

```
search(L,3,first_fail,indomain,complete,[]):
```

January 26, 2015 Becker-Asano, Nebel and Wölfl – Constraint Satisfaction Problems 92 / 100
search improvements

With first_fail strategy:

```prolog
search([L, O, first_fail, indomain, complete, []])
```

- solution is different from naive input_order approach
- more solutions can be found

Further improvements:
- start from the middle of the board
- use most_constraint instead of first_fail (tie break based on number of constraints in which variable occurs)
- try **multiple strategies** in parallel (multi-core CPUs)
- limit number of backtracks for search attempt and use randomization (of value and/or variable choice)
- ...

Summary

**ECLiPSe**:  
- interfaces with many programming languages  
- provides a stand-alone and a command line user interface  
- consists of a collection of libraries  
- is intended for problem solving in a declarative fashion enabling rapid prototyping based on the CLP paradigm

Last but not least:
- It is actively developed,  
- but already mature enough,  
- and very well documented.

We only scratched its surface here.

Check out [http://eclipseclp.org/](http://eclipseclp.org/), next...

Literature

- Sergii Dymchenko & Mariia Mykhailova. Declaratively solving tricky Google Code Jam problems with Prolog-based ECLiPSe CLP system.  
- Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen, Mark G. Wallace. ECLiPSe: A Tutorial Introduction.  
- Helmut Simonis. ECLiPSE ELearning Website  
  [http://4c.ucc.ie/~hsimonis/ELearning/index.htm](http://4c.ucc.ie/~hsimonis/ELearning/index.htm)