**A Basic Constraint Language**

We will define the skeleton of a constraint language, without many interesting capabilities, but which will be enough to understand the principles of constraint programming without the burden of having to cope with unneeded details.

The basic components of our language are the following:

**Variables**

which hold values throughout the execution. Differently from other languages, variables do not need to be typed or declared anywhere, and so they are distinguished from other elements by their syntax. Variables will always be written starting with an uppercase character: *X*, *Y*, *Speed*.

**Constants**

which are immutable values. Usual languages can use only numbers as constants, or, at most, a set of predefined strings which make up an enumerated or cardinal type--in fact, this is just another way of assigning names to numbers. Constants are either numbers, including floating point numbers, or names starting with a lowercase character: 87, -45.87, *bogus*.

Underscores are allowed either in the names of variables or non-numerical constants to improve readability: $Second\_Task$, $a\_dog$.

**Atoms**

which will play a syntactic rôle similar to procedure definitions and procedure calls. Atoms have the form $p(X_1,\ldots,X_n)$, where *p* is the name of a *procedure* or, more strictly, a *predicate*. *X*1 to *Xn* are the *arguments* of the atom, and the number of arguments *n* is termed the *arity* of *p*. This is commonly written *p*/*n*. Examples of atoms are

*hates*(*dog*, *cat*)

$predates(big\_fish, small\_fish)$

**Constraints**

which allow writing equations relating variables and constants in the program are written. For now we will use only the constraint = of arity 2, which will denote syntactic equality. We will give examples of their use.

Although constraint languages include builtin atoms which can be used in programs to perform several tasks (e.g., opening and writing to files), this small language will not have them: all the atoms which appear in bodies must be defined by the user somewhere in the program--although they will not always appear explicitly defined in the examples. Conversely, some constraint languages allow the user to define and augment the constraints available, besides those already available in the system, but we will not allow that either at this point.

**Clauses**

A *clause* represents a way of achieving a goal. Clauses have the form

|  |  |
| --- | --- |
| \begin{displaymath} p \leftarrow b_1, \ldots, b_n. \end{displaymath} | (2.1) |

where *p* is an atom, as defined in the previous section, and ![$b_1,
\ldots, b_n$]()are either atoms or constraints. In this expression, *p*is commonly called the *head* of the clause, and ![$b_1,
\ldots, b_n$]()is called the *body*. The symbol $\leftarrow$(which, for typographical convenience, is often written as :-) is called the *neck*, for it connects the body and the head.

**Example 2.1**   The following are syntactically correct clauses, as usually written in a computer:

animal(X):- dog = X.   
likes(C, F):- C = cat, F = fish.   
bigger(M1, M2):- M1 = men, M2 = mice.

In this example, animal/1, dog/1, likes/2, and bigger/2 are atoms. X, C, F, M1, M2 are variables, and cat, fish, men, and mice are non-numerical constants. Note that variables and constants can be written on both sides of the equality symbol--it does not matter in which side they appear.

The program has no meaning in itself as it is written, in the same sense that writing x = 3 + y in a conventional language has no meaning other than a mathematical operation whose purpose in the program we do not know. The only *a priori* possible interpretation comes from the semantics of first order logic: a expression such as that in ([2.1](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node17.html#e:logical-clause)) is to be read as *for p to be true, ![$b_1,
\ldots, b_n$]()have to be true*. Then, under an interpretation directed by the names in the code, the example [2.1](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node17.html#ex:clauses-examp) can be interpreted as expressing the following:

*X is an animal if X equals ``dog''* , or   
*``dog'' is an animal*   
  
  
  
*``cat'' likes ``fish''*   
  
  
  
*M1 is bigger than M2 if M1 equals ``men'' and M2 equals ``mice''*, or   
*``men'' are bigger than ``mice''*

These clauses contained only *calls* to constraints. Clauses can also refer to other clauses written by the programmer (atoms). The variables in the clauses are used to pass arguments to the atoms in the body (and constants can be passed as well, of course).

**Example 2.2**   The following clauses have atoms defined by the user in the body:

eats(X, Y):- bigger(X, Y).   
pet(X):- animal(X), sound(X,Y), Y=bark.

Their reading depends on the interpretation of the user atoms, but a likely meaning of them is:

*The big eat the small*, or   
*If some X is bigger than some Y, then X eats Y*   
  
  
  
*For X to be a pet, it must be an animal and the sound it produces must be a bark*, or   
*If X is and animal and X barks, then X is a pet*, or   
*An animal which barks is a pet*

Of course, the final answer to the real meaning of this piece of code is what the programmer actually had in mind when writing animal/a, sound/2, and bigger/2.

**Implicit Equality**

Equality is a very common constraint in all domains, and so it is customary to write it in a shorter form: the clause

p(X):- X = something.

can also be written, with exactly the same meaning as

p(something).

i.e., every time a variable of a clause appears anywhere *within a clause*, the atom (or variable) this variable is equated to can replace every appearance of that variable.

**Example 2.3**   The clauses in Example [2.1](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node17.html#ex:clauses-examp) can also be written as follows:

bigger(men, mice).   
pet(X):- animal(X), sound(X, bark).

and their meaning and behavior is exactly the same as in the original example.

**Facts**

The previous section introduced a new type of clause, which is actually a shorthand expression for clauses we already know how to write: the expression

*p*.

where *p* is an atom, is called a fact. The first clause in Example [2.3](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node17.html#ex:clauses-examp-no-eq) is a fact, which appears because an equality constraint has been implicitly moved to the head of the clause.

**Example 2.4**   The first and second clauses in Example [2.1](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node17.html#ex:clauses-examp) can also be written as facts:

animal(dog).   
likes(cat, fish).

**Predicates**

A *predicate* is simply a collection of clauses which have the same head name and arity. Recall that the constraints and atoms in the body of a clause represent conditions to be fulfilled in order to achieve a goal--the head--, so they logically represent a conjunction of goals. Different clauses, in turn, represent a disjunction: alternative possibilities to accomplish a target. From a more logical point of view, different clauses of a predicate offer alternative possibilities for the predicate to be true.

**Example 2.5**   The following predicate expands our idea of what a pet can look like:

pet(X):- animal(X), sound(X, bark).   
pet(X):- animal(X), sound(X, bubbles).

What is the meaning of this example? In addition to the first, already known clause, which casted animals which bark into the category of pets, we are not including animals whose sound is *bubbles* (probably fishes) into the very same category. So, in a more colloquial form, the example above can be read as

*Animals which bark and animals which make bubbles are pets*

Note that when we describe the predicate in a goal-oriented form, the description must take a disjunctive form, closer to the logical meaning of the predicate, but less natural from the point of view of the human language:

*For something to be a pet, it must either be an animal and bark, or else be an animal and make bubbles.*

Note also that the same variable X appears in both clauses: the names of the variables in a clause are local to that clause, very much like local variables in procedural languages have an scope limited to the procedure/function they are defined in.

**Programs and Queries**

We are now ready to write programs in our constraint language. A program is simply a collection of predicates, much in the same way that a program in other languages is a collection of procedures or functions.

**Example 2.6**   The following code implements a program which has knowledge about what is a pet, and, using a database of facts defining some animals and characteristics, infers which animals are (to its knowledge), pets.

pet(X):- animal(X), sound(X, bark).

pet(X):- animal(X), sound(X, bubbles).

animal(spot).

animal(barry).

animal(hobbes).

sound(spot, bark).

sound(barry, bubbles).

sound(hobbes, roar).

Since most CLP systems provide an interactive shell for the interpreter / compiler, the user can usually issue commands to load the program, call predicates in it, change the program, and load it again. Calling a predicate from the interpreter yields the same results as calling it from inside a program.

A query issued by the user is just a conjunction of atoms, and has exactly the same form and meaning as the body of a clause. The answer to a query is a set of bindings for the variables which make the query true with respect to the program. Since some predicates may have several clauses which hold for a given query, multiple solutions are possible.

**Example 2.7**   We will give an example of a possible session with a CLP system. The prompt of the system will be shown as ?-. We will use the program in Example [2.6](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node17.html#ex:extended-pets).

Load the file where the program is stored

?- consult(pet).

Make queries!

?- sound(spot, X).

X = bark

?- sound(A, roar).

A = hobbes

?- animal(barry).

yes

?- animal(X).

X = spot ;

X = barry ;

X = hobbes

**Problem 2.1**   What will be the answer(s) to the query

?- sound(A, S)

**Searching**

The query

?- pet(X).

returns the following answers:

X = spot   
X = barry

How is this achieved? The CLP system performs a search using all the possibilities offered by having several clauses for the predicates. This is best depicted by a *search tree* which represents all possible paths in the program. Without entering into details, every time a predicate with more than a clause is called, a *choice point* is made at that execution point: this choice points keeps information about the state of the execution at that moment, so that, if more solutions are needed, the engine can backtrack up to that point, and resume the search with the next untried clause of that predicate.

 

|  |
| --- |
| **Figure 2.1:** A tree |
| \resizebox{\textwidth}{0.25\textheight}{\includegraphics{Figs/pet_search.eps}} |

The search process, automatically triggered by a failure in the resolution, allows logic programming based languages to return all possible solutions to a query: after having reached a solution, if the user requests for more answers, the toplevel just causes a failure and the backtracking process is (re)started[2.1](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/footnode.html" \l "foot2461). The order of backtracking is as follows:

* Clauses within a predicate are tried from top to bottom; backtracking on a predicate will cause the next untried clause to be executed. The order in which clauses are executed is defined by the *search rule*.
* Atoms within a clause body are executed from left to right, and so backtracking is attempted right to left. This is called the *selection rule*.

Other strategies to select which clause and which atom to try are possible, and those different search and selection rules give raise to different operational semantics for logic languages.

**Example 2.8**   The following query has been executed using the program in Example [2.6](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node17.html#ex:extended-pets):

?- pet(X), animal(Y).

X = spot, Y = spot ;

X = spot, Y = barry ;

X = spot, Y = hobbes ;

X = barry, Y = spot ;

X = barry, Y = barry ;

X = barry, Y = hobbes

Solutions for the clauses of animal/1 are generated first, in the order in which the clauses are written. After that, a new solution for pet/1 is generated, following the rules for atoms and clauses stated above.

**Logical Variables**

Variables in CLP languages are termed *logical variables*. The adjective *logical* stems from a unique character not present in other languages: these variables do not necessarily hold values--and yet they are completely legal, and run-time access exception errors are not generated by accessing them[2.2](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/footnode.html" \l "foot465)--, and they can be assigned (or, better, *bound*) to other uninitialized variables. The value of an uninitialized variable is not NULL or other esoteric, special value: that variable, simply, has no value at all yet.

Logical variable assignment is *monotonic*, which means that a logical variable cannot mutate its value within a search path.

**Example 2.9**   The variable X can take the value a:

?- X = a.

X = a

But it cannot take the value a and then change it to b

?- X = a, X = b.

no

**Problem 2.2**   Then, how is it possible that the following queries work perfectly?

?- X = a.

X = a

?- X = b.

X = b

**Hint:** the toplevel interpreter backtracks between goals, in order to recover the initial state.

The constraint =/2 we have introduced before not only assigns values to variables (or, better, binds variables to values), but it can also bind *free* variables, constraining them to have the same value.

**Example 2.10**   Variables can be bound one to each other, constraining them to take the same value, and this constraint is taken into account during the rest of the execution:

?- X = Y, X = a.

X = a, Y = a.

?- X = Y, pet(X).

X = spot, Y = spot ;

X = barry, Y = barry

**Problem 2.3**   Explain the following behavior: why the query has no solutions?

?- X = Y, pet(X), sound(Y, roar).   
no

**Problem 2.4**   Given the following program, which is intended to model kinship in a family:

father\_of(juan, pedro).

father\_of(juan, maria).

father\_of(pedro, miguel).

mother\_of(maria, david).

grandfather\_of(L,M):-

father\_of(L,N),

father\_of(N,M).

grandfather\_of(X,Y):-

father\_of(X,Z),

mother\_of(Z,Y).

answer the queries:

?- father\_of(juan, pedro).

?- father\_of(juan, david).

?- father\_of(juan, X).

?- grandfather\_of(X, miguel).

?- grandfather\_of(X, Y).

?- X = Y, grandfather\\_of(X, Y).

?- grandfather\_of(X, Y), X = Y.

**Problem 2.5**   Augment the code in Problem [2.4](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node19.html#ex:familiar-database) to contain rules for the relationship   
grandmother\_of(X, Y), following the spirit of the program.

**The Execution Mechanism**

Execution of CLP languages can be seen as a tree traversal, where the nodes of the tree are conjunctions of atoms to be proved (similar to bodies of clauses, which are also conjunctions of atoms). The root of the tree is the initial query posed by the user, and there might be one or several branches starting at every node, each branch corresponding to the clauses with matching heads for the first (leftmost) goal in the conjunction. The tree is explored by selecting the leftmost goal in a conjunction, and the leftmost untried branch (clause) for that goal. The tree can be explored partially or totally; in the latter case, all solutions to the initial query are returned.

Figure [2.2](http://clip.dia.fi.upm.es/%7Evocal/public_info/seminar_notes/node20.html#fig:exec-tree) shows how the execution tree is traversed for the following program and the query ?- grandparent(charles,X).

grandparent(C,G):-

parent(C,P),

parent(P,G).

parent(C,P):- father(C,P).

parent(C,P):- mother(C,P).

father(charles,philip).

father(ana,george).

mother(charles,ana).

Execution starts at the toplevel query grandparent(charles, X), which is equated to the first clause of the program. Variables in the body of the clause are substituted by the constants in the query, and the body (with some constants in place of the textual variables) is left to be solved as a conjunction of goals. The execution continues by selecting the first goal in the body (parent(C, P), now rewritten at runtime to parent(charles, P)), and the process continues. There are two matching clauses for parent(charles, P), and the two are tried in textual order: that is the reason why two different subtrees are rooted at this node. The execution proceeds until a node with no atoms to solve is obtained (this is possible because a resolution against a fact, which has no body, removes an atom from the node).

The final result of the query, X = george, is obtained in the leaf labeled (precisely) X = george. This binding for X can be seen as propagated upwards in the tree and communicated to the variable present in the toplevel query, but, in fact, the variable this binding is made to, is the same one which was present in the toplevel query: as atoms were reduced in the execution process, variables in the same position in atoms and clause heads were *unified*, i.e., equated.

 

|  |
| --- |
| **Figure 2.2:** Traversing an execution tree |
| \resizebox{0.7\textwidth}{!}{\includegraphics{Figs/prolog1.ps}} |

![% latex2html id marker 3179
$\mathbf\therefore$]()

Knowing the operational behavior of the language is necessary for larger programs (especially because it is instrumental for achieving better performance), but for the time being, it is not essential: understanding the *declarative* semantics (i.e., the grandfather of someone is the father of his/her mother or the father or his father) is far more important at this stage.