L & O
on IC PROLOG][
User's Guide DRAFT

Z.Bobolakis
email:zb@doc.ic.ac.uk

1 Introduction

In this chapter, Logic & Objects (*L&O* for short) on top of IC PROLOG][is presented. It is not intended as a complete description of *L&O* but as a user's guide to the system. *L&O* is defined in detail in the book *Logic & Objects* by F.G.McCabe, Prentice Hall, 1992. What is presented in the following section is *only* an outline of some important concepts.

L&O is an extension to Prolog which supports an object-oriented notation. An object consists of a *label* which identifies it (the label may be parametrized) and program which defines its behaviour. The program describing its behaviour is a Prolog program with some additional features. The most important features will be described here.

2 *L&O* programs

2.1 Object definitions

An *L&O* object is defined as follows:

```
Label: { Program }.
```

`Label` is a Prolog atom or compound term which identifies the object. `Program` is a set of clauses which can be one of the following:

Relation definitions Relations can be defined for the object locally using ordinary Prolog clauses. Standard Prolog syntax applies here, but the body of a clause can contain specific kinds of calls which are defined in *L&O*. The following clauses are an example of relation definitions in an object[1]:

---

[1] The calls in the body will be explained later.

```
age(Ag).
likes(X) :- X:age(A),A<Ag.
likes(X) :- X:likes(self).
```

Equations Equations are used for defining functions.

An example of equation is:

```
route_length(T)=0 :-atom(T).
```

or

```
no_of_legs=2.
```

The function (on the LHS of the head) evaluates to the value (on the RHS of the head) if the body of the clause succeeds. If there is no body, it is enough that the call unify with the LHS of the head. A function can be defined using more than one clauses but the equations should not overlap in defining the function[2]. Guaranteeing this, is not a mere syntactic checking, so the responsibility for it lies with the programmer.

Variable initializations for example:

```
foible:=34.
```

Such variables in *L&O* are mutable, i.e. their value can change during the execution of the program. The effect of the above clause is to assign the variable to the value at the time the object is created, i.e. when the program is loaded or consulted.

A call in the body of a clause can be:

Prolog call It can be a call to an ordinary Prolog relation, built-in or user-defined, or to a relation defined within the same object. In both cases the call is executed as an ordinary Prolog call[3].

A labelled call for example

---

[2] Otherwise they do not define a function in the strict sense.

[3] Inheritance can also be used for such a call, see later for inheritance.

```
animal:mode(X)
```

The general form of a labelled call is

```
Label:Call
```

A labelled call is a call to another object. `Label` (`animal` in the example above) is an object and `Call` (`mode(X)` in the example) is a call to a predicate defined in that object.

A special *L&O* call for example

```
self:mode(X)
```

or

```
super:mode(X)
```

Special calls will be presented later in this section.

2.2 Inheritance

Apart from object definitions, an *L&O* program can also contain inheritance definitions. Different kinds of inheritance can be defined:

Ordinary inheritance is defined as follows:

```
L <= M.
```

where `L` and `M` are two objects, for example

```
horse <= animal.
```

The effect of this rule is that any call of the form `horse:Call` can be transformed to the call `animal:Call`. Therefore, the definition of the relation specified in `Call` in `horse` is augmented with the corresponding definition in `animal`.

Overriding inheritance can be defined as follows:

```
L << M.
```

for instance

```
horse << animal.
```

In this case, a call `horse:Call` is transformed to `animal:Call` but only if the predicate to which `Call` is a call is not defined in the object `horse`.

Differential inheritance relationships are defined as follows:

```
L <= M-Preds.
```

or

```
L << M-Preds.
```

for example

```
bird <= animal-mode(walk).
```

In this case, `bird` inherits `animal` except for calls of the form `bird:mode(walk)`.

2.3 Expressions

Prolog provides the ability to use expressions in programs but in a clumsy and not natural way. A variable should be used and the value of the variable should be assigned to the expression's result after an explicit evaluation using the `is` primitive. For example, instead of

```
..., foo(X+1), ...
```

the programmer should write

```
..., X1 is X+1, foo(X1), ...
```

This is only possible for primitive functions (such as `+/2`). For a user-defined function, a relation should be defined and called explicitly, for example, instead of defining the function `length/1` which returns the length of a list, one would write:

```
length([], 0).
length([_|X], N) :- ...
```

and call it as follows:

```
..., length(List, L), foo(L), ...
```

instead of

```
..., foo(length(List)), ...
```

L&O provides this ability to use expressions which can be evaluated. In particular, built-in functions are provided as well as the ability for the user to define functions.

In practice though (for various reasons), the kinds of expressions that are evaluated by the *L&O* system are more restricted.

In particular, an expression is evaluated if:

- Its functor is a built-in function. The built-in functions are listed later in this chapter. Therefore, a call of the form

  ```
  p(2+3, length([a,b,c]))
  ```

  is the same (unlike in Prolog) as

  ```
      p(5, 3)
  ```

- It is explicitly labelled, i.e. it uses an equation defined in a different object. For example

  ```
  horse:no_of_legs
  ```

  is evaluated using the definition for `no_of_legs` in the object `horse`.

- It is a mutable variable, for example:

`p(c)`

is interpreted as `p(V)` where `V` is the current value of a mutable variable `c`, if such a variable exists in the object (i.e. if there exists an initialization clause for the variable).

L&O provides a 'quote' operator `` ` `` which prevents the expression on which it is applied from being evaluated. So if an equation for `f(X)` in an object defines `f(X) = 2*X`, then the expression `f(4)` evaluates to `8` while `` `f(4) `` is not evaluated, it stands for `f(4)` as a Prolog term.

Moreover, an 'unquote' operator is provided, namely `#`, which enforces evaluation of the expression it is applied to, allowing so nested expressions.

Finally, the quote-hash operator `` `# `` causes the function symbol of an expression to be quoted while each of the arguments is unquoted. So the expression `` `#(X+Y < Z-W) `` is equivalent to `` `(#(X+Y) < #(Z-W)) ``, where only the expressions `X+Y` and `Z-W` are evaluated.

2.4 Special *L&O* calls

2.4.1 `self` and `super`

The `self` keyword A call in the body of a clause in an object can have the form

```
        self:Call
```

In order to see how the call is evaluated, consider an example:

```
person:{
        child :- self:age(X),
                 X < 18.
}.

tom:{
        age(14).
}.

tom <= person.
```

Consider now the call

```
tom:child
```

This call cannot succeed using any clause in `tom`. Because `tom` inherits `person`, the call is transformed to `person:child`. This call is now tried in the object `person` using the clause for `child`. In the body of the clause, `self:age(X)` has to be solved. This call is tranformed to `tom:age(X)` because `self` refers to the original object that the call originated before the inheritance. The last call succeeds binding `X` to `14` and the original call succeeds as expected.

More generally, `self` is a keyword which evaluates to the object to which the call originated before inheritance transformations and can be used not only as a label but in any place in a call.

When a call is transformed to another one through inheritance, `self` does not change, it evaluates to the same value as before the transformation. On the other hand, when a call like `tom:child` is made, `self` during the evaluation of this call becomes `tom`, no matter what it was before.

The `super` label `super` can be use as a label to transfer the evaluation of a call from the current object to one which it inherits. `super` cannot be used but as a label.

2.4.2 Broadcasting

A call can be broadcasted to more than one objects and the results can be combined in different ways to evaluate the result to the broadcasted call.

filter broadcasting A call of the form

```
&^(L,C,M).
```

returns a binding for `M` which is a sublist of `L`, namely those elements `O` of the list for which `O:C` is true. A functional notation can also be used, i.e.

```
&^(L,C)
```

evaluates to the filtered list.

and-casting A call of the form

```
L&:C.
```

succeeds iff for all `O` on the list `L`, `O:C` is true.

or-casting A call of the form

```
L\:C.
```

succeeds iff for some `O` on the list `L`, `O:C` is true.

2.4.3 Variable assignment

A mutable variable (defined by an initialization in the definition of an object) can be assigned a new value using a call of the form:

```
Variable := Expression
```

where `Variable` is the variable and `Expression` evaluates to its new value. `Expression` can contain the variable itself (as in an imperative language) allowing thus assignments like

```
c := c+1
```

2.5 An example *L&O* program

Consider the following program which describes an animal-world.

```
person(Ag) : {
    age(Ag).
    likes(X) :- X:age(A),A<Ag.
    likes(X) :- X:likes(self).
    no_of_legs=2.

    foible := 34
            }.
person(Ag)<<animal.
```

In this object, `person(Ag)` is the label. The label is parametrized (`Ag` is interpreted here as the age of the person). Notice that the label arguments are global variables with respect to the program defining the object. So the clauses `age(Ag).` and likes(X) :- X:age(A),A¡Ag. can access this variable.

`age/1` and `likes/1` are relations for the object, while `no_of_legs=2.` defines a function (which happens to take no arguments, i.e.it is a constant). On the other hand `foible := 34` initializes a mutable variable.

A person (of any age) is also defined to inherit the object `animal` (defined later) using overriding inheritance. Thus, although an animal (as we will see) has 4 legs, a person has only 2 because `no_of_legs` is defined in `person(Ag)` and therefore the definition is not inherited from `animal`.

```
tom <= person(32).
frank <= person(37).
```

`tom` and `frank` are defined to inherit one instance of `person` each. Their age is automatically defined through this inheritance.

```
animal:{
   mode(walk) :- self:no_of_legs>=2.
   mode(run) :- self:no_of_legs=2.
   mode(gallop) :- self:no_of_legs=4.
   mode(slither) :- self:no_of_legs=0.
   no_of_legs=4.
   }.
```

For the object `animal`, the relation `mode/1` is defined. Its definition uses the function `no_of_legs/0`. Notice that the calls to the function use the keyword `self` and threfore are executed using the definition of the particular animal we are refering to, not `animal` in general. So `mode(X)` for a horse will use the definition of `no_of_legs` in `horse` while if the expression to be evaluated is `animal:no_of_legs` then `no_of_legs=4` is used (as in `animal`.

```
mammal : {
   mode(walk) :- self:no_of_legs>=2
   }.

mammal <= animal.
```

```
reptile <= animal.

horse <= mammal.
horse : {
   no_of_legs=4
   }.

beauty <= horse.
beauty : {
   mode(fly).
   }.

bird : {
   mode(fly).
   no_of_legs=2
   }.
bird <= animal.

penguin <= bird-[mode(fly)].
penguin : {
   mode(swim).
   }.
```

Here, we define `penguin` to inherit `bird`. The inheritance is differential, so although for `bird` it is defined that `mode(fly)`, `penguin` does not inherit it. `mode(swim)` is defined for `penguin`.

3 The *L&O* system

3.1 Starting *L&O* in IC-PROLOG][

Once ICP is has started, *L&O* can be loaded using the command

```
[lo].
or
        ensure_loaded(lo).
```

L&O files are now recognized and consulted/compiled.

3.2 *L&O* file handling

L&O programs should be included in files which have the extension `.lo`. At this stage, *L&O* and ordinary Prolog code should not be mixed in the same file. The IC PROLOG][compiler can invoke a user-defined preprocessor to read the source file and produce Prolog code which the compiler itself takes as input. The *L&O* preprocessor is hooked to the compiler in this way. The file extension is an indication to the compiler that the file to be compiled is a special one and the `.lo` extension is reserved for *L&O* for this purpose[4].

L&O files are compiled using the same Prolog commands as with ordinary Prolog files. For example, the file called `animals.lo` (which contains the example program presented in the previous section) can be consulted using:

```
| ?- consult(animals).
```

Prolog will add the missing ending to the filename. Notice here that if a `animals.pl` file exists, it will be consulted instead of `animals.lo` because `.pl` files have priority. In order to avoid conflict, full filename should be used in such cases, i.e.

```
| ?- consult('animals.lo').
```

All conventions that hold for standard Prolog files also hold here, e.g. filenames contaning `.` or `/` should be quoted, absolute filenames (i.e. the ones that include the path) should contain the `.lo` extension etc.

A file can also be consulted using square brackets, i.e. as follows:

```
| ?- [animals].
```

Once a file has been consulted and subsequently modified, it should be reconsulted using `reconsult/1` as in

```
| ?- reconsult(animals).
```

or using

```
| ?- [-animals].
```

---

[4] In this way, IC PROLOG][provides the facility to write any preprocessor and redirect its output to the standard Prolog compiler.

(square brackets with a `-` preceding the filename).

Alternatively, `compile/1` and `load/1` should be used if optimized compilation is preferred. These primitives behave as in standard Prolog, but the comments made above for `consult/1` and `reconsult/1` also hold here.

The differences between ordinary and optimized compilation hold here exactly as with ordinary Prolog files.

3.3 *L&O* query evaluation

L&O queries have the form

```
Label:Call
```

where `Label` is an object label and `Call` is a call to a predicate defined in `Label`.

L&O queries behave as usual Prolog queries (returning bindings, multiple solutions etc).

In the simplest case, `Label` is ground and `Call` is a non-variable. It can also be the case that `Label` is not ground. In this case, *L&O* replies with bindings on variables occuring in `Label`, so that `Label` is an object for which `Call` holds.

It is not allowed though that `Call` be a variable, it should be a predicate call.

For example,

```
| ?- horse:mode(X).

X = walk ;

X = walk ;

X = gallop ;

no
```

and

```
| ?- X:mode(gallop).
```

```
X = animal ;

X = beauty ;

X = horse ;

X = mammal ;

no
```

are valid *L&O* queries and give the corresponding solutions while

```
| ?- horse:X.
```

is not valid.

L&O queries can also be ones that evaluate expressions[5]. In particular, they can have the form

```
        Expression =? Value.
```

The above query, evaluates `Expression` and unifies it with `Value`. The query can also be given in a different form:

```
        Expression =? .
```

It is important that in the last case `=?` and `.` are separated by a space character. Otherwise a syntax error occurs.

The first form (using `=?` as infix) is necessary when the query is given in a context where `Value` is a variable shared with other subqueries.

For example, in the animals' program we can ask:

```
| ?- beauty:no_of_legs =? .
4

yes
```

because `beauty` is a `horse` which has 4 legs and

---

[5] How expressions are evaluated is discussed later.

```
| ?- penguin:no_of_legs =? .
2

yes
```

because `penguin` is a `bird` (except for its `mode`) and therefore has 2 legs. because

3.4 Dynamic code in *L&O* programs

As in ordinary Prolog programs, programs can be modified dynamically in *L&O*. There are though some restrictions to what can be modified dynamically. This will be explained in this section.

3.4.1 Adding new programs

Objects, programs or rules can be asserted using the `assert_object` primitive. The primitive can be used as following:

- An object can be asserted by a call `assert_object(L:B)` where `L` is the label of the object and `B` is a program included in {}, that is, using the same syntax as with static objects. The only difference is that the last clause in the program `B` should not be followed by `'. '` as in static programs. The object asserted should not already exist.
- One or more clauses can be asserted to an already existing *dynamic* object, using the call `assert_object(L:B)`. Again, `L` is the label and `B` is either a single clause or a set of clauses Included in {}.
- A new inheritance rule can be asserted using the call `assert_object(L<=R)` or `assert_object(L<<R)`. The objects need not be dynamic.

The restrictions in the kind of programs asserted are:

- No expressions are allowed in the objects.
- No equations are allowed.
- `self` can only be used as a label in explicitly labelled calls in the body of a clause.

3.4.2 Retracting programs

Clauses or rules (not complete objects) can be retracted using the `retract_object` primitive. The primitive can be used as following:

- A clause can be retracted from an already existing *dynamic* object, using the call `retract_object(L:B)`. Again, `L` is the label of the object and `B` is the clause to be retracted.
- An inheritance rule can be retracted using the call `retract_object(L<=R)` or `retract_object(L<<R)`. but the rule retracted should have been asserted dynamically.

3.5 Debugging *L&O* programs

L&O in ICP, provides a debugging facility for tracing queries.

3.5.1 How the *L&O* tracer works

The Prolog tracer might be used for *L&O* programs since the *L&O* preprocessor produces Prolog code. The problem is that the output from the preprocessor is not in a form easy to understand from the *L&O* programmer's point of view.

The *L&O* tracer is not implemented as an ordinary metainterpreter on programs. Instead, at the preprocessing/compilation phase, information related to tracing is introduced to the program in the form of hooks.

Different kinds of hooks exist corresponding to different kinds of events that might happen during the execution of an *L&O* query. In particular, the hooks introduced in the Prolog code that is output by *L&O*, uses the following hooks:

- `#r#` The initial entry to a relation in a class body.
- `#e#` The entry to a clause for a call.
- `#s#` The successful exit from a clause.
- `#f#` The initial entry to a function in a class body.
- `#q#` The entry to an equation or function.

- `#v#` The completion of an equation.
- `#i#` The use of an inheritance rule.
- `#d#` The initialization of a dynamic variable.
- `#a#` The reassignment of such a variable.
- `#u#` A user specified print statement. This hook is compiled in
- `#l#` The entry to a labelled variable call.
- `#k#` Evaluation of a labelled expression. when the programmer has inserted a `trace` (*message*) subgoal into the body of a clause or equation. The *message* is printed out in the trace of the execution whenever this hook is activated.

In this way, the programmer can activate some of the hooks only. Moreover, a different interface for the tracer can be adapted to it (by redefining the hooks only), as the one provided in this version is relatively low level.

In order for a hook to be invoked during the execution of a query, it has to:

- be compiled in the code
- be activated at the time of execution.

This means that the user can compile as many hooks as needed in the program and then temporarily activate/deactivate any of them.

3.5.2 Using the *L&O* tracer

Setting the tracing mode Which of the hooks are compiled by the preprocessor depends on a set of flags being set. The flags are the following:

- `entry`
- `user`
- `exit`
- `fail`

- `redo`
- `clauses`
- `equations`
- `inheritance`
- `dynamic_code`
- `percentfs` (restricted use - only for low-level programming)

Notice that the flags do not correspond one-to-one to the hooks. For example, in order to trace a clause, the `clause` flag should be `on` and some of `entry`, `exit`, `fail`, `redo` depending on which points the user needs to trace it (normally all of them).

The flags can be set on/off using the following primitives:

- `lo_trace` sets all tracing flags (except `percentfs`) on.
- `lo_notrace` sets all tracing flags (except `percentfs`) off.
- `tracing(X,Y)` sets all the flags on `X` and `Y` and the rest of them off. `X` should be a sublist of `[entry,exit,fail,redo]` and `Y` a sublist of `[clauses,equations,inheritance,dynamic_code,percentfs, user]`.
- `tracing(X)` has the same effect as `tracing([entry,exit,fail,redo],X)`.

Once the flags have been set, the program should be compiled/consulted. If at some stage the user wants to deactivate/reactivate some of the tracing hooks, the corresponding flags should be set on/off. This can be done using the `tracing/1` or `tracing/2` primitives.

Notice here that only hooks that have been compiled can be activated or deactivated. Flags can be set on or off but the corresponding hooks are only modified at compilation.

For example, if before compiling a file the query `tracing([clauses])` has been given, only clauses (at all points: entry, redo, exit, fail) will be traced.

If after compilation the user gives `tracing([clauses, equations])`, it will have no immediate effect, as `clauses` is already set on and `equations` related hooks have not been compiled.

Though, the flag `equations` will be set on and if the program is compiled again, it will cause the equations related hooks to be activated (unless `equations` is set off before this compilation).

If on the other hand the user gives `tracing(off)` (which is the same as `tracing([],[])`) or `tracing([])`, then `clauses` will be set off and, although the hook is compiled, the clauses will not be traced.

A subsequent call `tracing([clauses])` will set `clauses` on again and, because the related hook is compiled, clauses *will* be traced.

Tracing a query When the tracing mode has been set, a query can be traced. The query is given as usually and the system traces any activated hooks in the code and reports them. The reporting information includes:

- the object currently active (and `self` if it is different from the object)
- the type of the traced point (`enter`ing a relation/clause/equation etc, `redo`ing, `exit`ing successfully, `fail`ing, `report`ing and inheritance rule)
- the traced point (`clause`, `relation` etc.)

At every step, the system waits until a command is given. A command can be one of the following:

- '`c`' or *return* to continue with the next step.
- '`s`' to skip the tracing of the current goal (allowed only when entering a call/clause/etc).
- '`f`' to fail the current goal.
- '`q`' to abort the top level query.
- '`h`' for help (display these options)

If an illegal command is given then the tracer insists until a legal one is given.

3.5.3 An example using the tracer

We now present and explain an *L&O* session using the tracer.

```
laotzu(22)% icp

IC-Prolog ][ version 0.93  -   05 Nov 1992
(c) Imperial College 1992, Imagine Project

{consulting system.pl ...}

| ?- ensure_loaded(lo).
{consulting lo.pl ...}
{consulting looperators.pl ...}
loading loenv.icp ... ok
loading lolow_level.icp ... ok
loading lodynamic.icp ... ok
loading lotranslator.icp ... ok
loading lotracing.icp ... ok
loading loexpressions.icp ... ok
loading lotop.icp ... ok

yes
| ?- [animals].
{consulting animals.lo ...}

yes
| ?- lo_trace.

yes
| ?- beauty:mode(X).

X = fly ;

X = walk ;

X = walk ;
```

```
X = gallop ;

no
```

Here, although the tracing mode has been set, the source file has been consulted before, so the compiled code contains no calls to the hooks. So we need to reconsult (the `lo_trace` call should have preceded the `[animals]` call).

```
| ?- [-animals].
{consulting animals.lo ...}

yes
| ?- beauty:mode(X).
enter(relation) beauty:(mode _45) ?
enter(clause) beauty:(mode fly) ?
exit(clause) beauty:(mode fly) ?

X = fly ;
redo(clause) beauty:(mode fly) ?
report(inherit) beauty:(beauty << horse) ?
report(inherit) horse/beauty:(horse << mammal) ?
enter(relation) mammal/beauty:(mode _45) ?
enter(clause) mammal/beauty:(mode walk) ?
report(inherit) beauty:(beauty << horse) ?
enter(function) horse/beauty:no_of_legs ?
enter(equation) horse/beauty:no_of_legs ?
exit(equation) horse/beauty:(no_of_legs = 4) ?
exit(clause) mammal/beauty:(mode walk) ?

X = walk ;
redo(clause) mammal/beauty:(mode walk) ?
report(inherit) mammal/beauty:(mammal << animal) ?
enter(relation) animal/beauty:(mode _45) ?
enter(clause) animal/beauty:(mode walk) ?
report(inherit) beauty:(beauty << horse) ?
enter(function) horse/beauty:no_of_legs ?
enter(equation) horse/beauty:no_of_legs ?
exit(equation) horse/beauty:(no_of_legs = 4) ?
exit(clause) animal/beauty:(mode walk) ?

X = walk ;
redo(clause) animal/beauty:(mode walk) ?
enter(clause) animal/beauty:(mode run) ?
report(inherit) beauty:(beauty << horse) ?
enter(function) horse/beauty:no_of_legs ?
redo(clause) animal/beauty:(mode run) ?
enter(clause) animal/beauty:(mode gallop) ?
report(inherit) beauty:(beauty << horse) ?
enter(function) horse/beauty:no_of_legs ?
enter(equation) horse/beauty:no_of_legs ?
exit(equation) horse/beauty:(no_of_legs = 4) ?
exit(clause) animal/beauty:(mode gallop) ?

X = gallop ;
redo(clause) animal/beauty:(mode gallop) ?
enter(clause) animal/beauty:(mode slither) ?
report(inherit) beauty:(beauty << horse) ?
enter(function) horse/beauty:no_of_legs ?
redo(clause) animal/beauty:(mode slither) ?
fail(relation) animal/beauty:(mode _45) ?
fail(relation) mammal/beauty:(mode _45) ?
fail(relation) beauty:(mode _45) ?

no
```

Now the hook calls have been compiled in the code, so the tracer works. The example above is the full tracing of the query `beauty:mode(X)`. Since everything is traced (`lo_trace` call sets all flags on), all ports are reported (`enter`, `report`, `redo`, `exit`, `fail`) in all the cases (clauses, equations, inheritance etc). If we now change the tracing mode to trace only clauses, we have:

```
| ?- tracing([clauses]).

yes
```

```
| ?- beauty:mode(X).
enter(relation) beauty:(mode _45) ?
enter(clause) beauty:(mode fly) ?
exit(clause) beauty:(mode fly) ?

X = fly ;
redo(clause) beauty:(mode fly) ?
enter(relation) mammal/beauty:(mode _45) ?
enter(clause) mammal/beauty:(mode walk) ?
exit(clause) mammal/beauty:(mode walk) ?

X = walk
```

which causes clauses only to be traced (not equations, inheritance etc). This is obvious if we compare the traces of the query for the first two solutions. Although the calls to the hooks are still in the compiled code, they are deactivated (except for the ones related to clauses).

```
| ?- [-animals].
{consulting animals.lo ...}

yes
| ?- tracing([equations]).

yes
```

After the above queries, the only hook calls compiled are the ones for clauses (they were set on at the consultation time) but the only flags set on are the ones for clauses. As a result, we should expect that nothing will be traced.

```
| ?- beauty:mode(X).

X = fly ;

X = walk
| ?- [-animals].
{consulting animals.lo ...}
```

```
yes
| ?- beauty:mode(X).

X = fly ;
enter(function) horse/beauty:no_of_legs ?
enter(equation) horse/beauty:no_of_legs ?
exit(equation) horse/beauty:(no_of_legs = 4) ?

X = walk
```

After reconsulting, equations-related hook calls are compiled (and the corresponding flags are set on) so the above query shows (as expected) that only equations are traced.

```
| ?- tracing(all).

yes
| ?- penguin:no_of_legs=? .
enter(function) bird/penguin:no_of_legs ?
enter(equation) bird/penguin:no_of_legs ?
exit(equation) bird/penguin:(no_of_legs = 2) ?
2

yes
```

Now, although although all flags are set on, only equations-related hooks calls are compiled so only equations are traced.

Finally, we will see the effect of possible responses to the tracer (in the example above <*return*> was always the response which caused tracing to proceed step-by-step.

```
| ?- [-animals].
{consulting animals.lo ...}

yes
| ?- beauty:mode(X).
enter(relation) beauty:(mode _45) ?
enter(clause) beauty:(mode fly) ?
```

```
exit(clause) beauty:(mode fly) ?

X = fly ;
redo(clause) beauty:(mode fly) ? s

Options :
  <CR> - continue
   c   - continue
   s   - skip goal (only at entry)
   f   - fail goal
   q   - abort query
   h   - display this list

redo(clause) beauty:(mode fly) ?
report(inherit) beauty:(beauty << horse) ?
report(inherit) horse/beauty:(horse << mammal) ?
enter(relation) mammal/beauty:(mode _45) ? s
exit(clause) mammal/beauty:(mode walk) ?

X = walk ;
report(inherit) mammal/beauty:(mammal << animal) ? f
fail(relation) mammal/beauty:(mode _45) ?
fail(relation) beauty:(mode _45) ?

no
```

Notice that when `s` (for *skip*) was given at an inappropriate call, the system insisted with the same call (and displayed the options). Skipping at a later call (an `enter` call) caused the evaluation of the query to be skipped until `exit`ing this call. One solution was given but after more solutions were required, the user gave `f` which caused the evaluation to fail.

Finally, aborting (`q`) is also an option:

```
| ?- beauty:mode(X).
enter(relation) beauty:(mode _45) ?
enter(clause) beauty:(mode fly) ?
exit(clause) beauty:(mode fly) ?
```

```
X = fly ;
redo(clause) beauty:(mode fly) ?
report(inherit) beauty:(beauty << horse) ?
report(inherit) horse/beauty:(horse << mammal) ?
enter(relation) mammal/beauty:(mode _45) ?
enter(clause) mammal/beauty:(mode walk) ?
report(inherit) beauty:(beauty << horse) ? q

--- ABORTED ---


Unbound variable : X
```

which causes the whole evaluation to stop.

3.6 *L&O* built-in primitives

3.6.1 Keywords

`self` and `super` are *L&O* keywords and therefore they should not be used for other purposes.

3.6.2 Functions

The following functions (with trhe corresponding arity) are built-in in *L&O*:

Mathematical functions

- `+/2` for addition (integer or floating point)
- `-/2` for subtraction (integer or floating point)
- `-/2` for multiplication (integer or floating point)
- `/ /2` for division (integer or floating point)
- `mod/2` for the remainder of integer division
- `\ /1` for bitwise negation
- `/\ /2` for bitwise and
- `\/ /2` for bitwise or
- `++ /2` for integer addition

- `-- /2` for integer subtraction
- `** /2` for integer multiplication
- `// /2` for integer division
- `sqrt/1` for square root of a number
- `abs/1` for absolute value of a number
- `int/1` for integer part of a number
- `ln/1` for logarithm (base e)
- `pwr/2` or `^/2` for power (`pwr(N,M)` is the `M`th power of `N`)
- `sin/1`
- `cos/1`
- `tan/1`
- `atan/1` for arctan
- `exp/1` for exponential function

Other functions Several primitive functions have the form `F/A` and return what `F/A+1` would return in Prolog in the last argument. The following built-in functions are provided:

- `concat/2` returns the concatenation of two strings
- `name/1` and `pname/1` (see ICP built-in primitives)
- `length/1` returns the length of a list
- `nint/1` returns the nearest integer of a number
- `-/1` returns the opposite of a number
- `&^/1` (see definition of filter casting)
- `max/2` for maximum
- `min/2` for minimum

3.6.3 Reserved operators

L&O reserves some operators for special purposes, either for the language itself or for the internal representation. The operators with special interpretation are defined as follows:

- `op(1198,xf,'=?')`, `op(1198,xfx,'=?')`, `` op(698,fx,`) ``, `op(698,fx,#)` and `` op(698,fx,`#) `` are used for expression evaluation.

- `op(699,xfx,&^)`, `op(700,xfx,'&:')` and `op(700,xfx,\:)` are used for broadcasting.

- `op(700,xfx,':=')` is the assignment operator (assigns a mutable *L&O* variable to a value).

- `op(1198,xfx,'<=')` and `op(1198,xfx,'<<')` are used for inheritance declarations.

- `op(1200,yfy,'.  ')` and `op(1200,xfy,'..')` are used as clause separators in a class body.

- `op(699,fx,':')` is also reserved for internal representation.

A Implementation of the tracer hooks

Here we present in brief the way the tracer hooks are implemented. For the method by which the tracer is built, see the discussion in the corresponding section above.

Depending on the environment used, the tracer may be implemented in a different way, by changing only the definitions of the hooks.

For each hook call that is introduced in the Prolog code produced by *L&O*, it is initially checked (using the `is_traced/1` calls to flags) whether the related flags are set on (i.e. whether the hook is active or not) and, if they are, a dialogue is invoked with parameters depending on the type of the hook. The dialogue is invoked using the `trace_dialogue/5` call. The arguments passed to the `trace_dialogue/5` predicate are the following (as they are ordered):

- The type of hook, in fact the information displayed to the user, e.g. `enter(relation)`.
- Information about the position in the source code in the form `'@x'(Id,P,IX,C)` where `Id` is the name of the source file, `P` is the position of the current object in the file (as the position of the first character of the label), `IX` is the order of the clause within the object and `C` is the order of the Prolog clause for the Prolog predicate output from the preprocessor. This is not always possible since some hooks do not correspond to a particular clause in the source code (e.g.when entering a relation call) and in this case `true` is the value passed in this argument. All this information can be used for a source-level debugging interface (which is not provided now).
- The `self` of the current object.
- The label of the current object.
- The call traced (also displayed to the user).

A.1 Code for the tracer hooks

- `#r#` The initial entry to a relation in a class body.

```
'#r#'(A,L,S):-
   is_traced(clauses),is_traced(entry),!,
   (trace_dialog(enter(relation),true,S,L,A);
   trace_dialog(fail(relation),true,S,L,A),fail).
'#r#'(_,_,_).
```

- `#e#` The entry to a clause for a call.

```
'#e#'(A,L,S,C):-
   is_traced(clauses),is_traced(entry),!,
   (trace_dialog(enter(clause),C,S,L,A);
   trace_dialog(redo(clause),C,S,L,A),fail).
'#e#'(_,_,_,_).
```

- `#s#` The successful exit from a clause.

```
'#s#'(A,L,S,C):-
   is_traced(clauses),is_traced(exit),!,
   trace_dialog(exit(clause),C,S,L,A).
'#s#'(A,L,S,C):-
   skip_trace(S,L,A),!,
   '#s#'(A,L,S,C).
'#s#'(_,_,_,_).
```

- `#f#` The initial entry to a function in a class body.

```
'#f#'(E,L,S):-
   is_traced(equations),is_traced(entry),!,
   trace_dialog(enter(function),true,S,L,E).
'#f#'(_,_,_).
```

- `#q#` The entry to an equation or function.

```
'#q#'(E,L,S,C):-
   is_traced(equations),is_traced(entry),!,
   trace_dialog(enter(equation),C,S,L,E).
'#q#'(_,_,_,_).
```

- `#v#` The completion of an equation.

  ```
  '#v#'(EX,R,L,S,C):-
     is_traced(equations),is_traced(exit),!,
     trace_dialog(exit(equation),C,S,L,EX=R).
  '#v#'(EX,R,L,S,C):-
     skip_trace(S,L,EX),!,
     '#v#'(EX,R,L,S,C).
  '#v#'(_,_,_,_,_).
  ```

- `#i#` The use of an inheritance rule.

  ```
  '#i#'(L,M,_,C,S):-
     is_traced(inheritance),is_traced(entry),!,
     trace_dialog(report(inherit),C,S,L,(L<<M)).
  '#i#'(_,_,_,_,_).
  ```

- `#d#` The initialization of a dynamic variable.

  ```
  '#d#'(V,R,L,S,C):-
     is_traced(dynamic_code),is_traced(entry),!,
     trace_dialog(report(declare),C,L,S,V:=R).
  '#d#'(_,_,_,_,_).
  ```

- `#a#` The reassignment of such a variable.

  ```
  '#a#'(L,V,R,C):-
     is_traced(dynamic_code),is_traced(entry),!,
     trace_dialog(report(assign),C,L,L,V:=R).
  '#a#'(_,_,_,_).
  ```

- `#l#` The entry to a labelled variable call.

  ```
  '#l#'(A,L,S):-
     is_traced(clauses),is_traced(entry),!,
     (trace_dialog(entry(relation),true,S,L,A);
     trace_dialog(fail(relation),true,S,L,A),fail).
  '#l#'(_,_,_).
  ```

- `#k#` Evaluation of a labelled expression.

  ```
  '#k#'(E,L,S):-
     is_traced(equations),is_traced(entry),!,
     trace_dialog(entry(function),true,S,L,E).
  '#k#'(_,_,_).
  ```

- `#u#` A user specified print statement.

  ```
  '#u#'(M,L,C,S):-
     is_traced(user),is_traced(entry),!,
     trace_dialog(report(user),C,S,L,M).
  '#u#'(_,_,_,_).
  ```

A.2 The tracing dialogue

`trace_dialogue` is implemented as explained here. The possible responses of the user are:

- `Help` in a non-standard response (none of the below).
- `Quit` in which case the query is aborted (using `abort/0`).
- `Continue` in which case the call succeeds.
- `Fail` in which case the call is enforced to fail (using `fail/0`).
- `Skip` which is more complicated. It is important here that any implementation of the reaction to a `Skip` response be compatible with the implementation of the hooks. `Skip` is only valid when `enter`ing a call and `trace_dialogue` should check that (reject an invalid `Skip` message and, either insist or react in a default manner, e.g.`Continue`). If the `Skip` response is valid, it should deactivate the hooks, save the tracing mode (i.e.which flags are set on/off) and succeed. Saving the tracing mode is done using the calls

  ```
  is_tracing(T),
  set_prop('#tracing#','#mode#',trace(Sf,LbL,Call,T)),
  ```

which checks which flags are set on and uses the IC PROLOG][properties database to save the state. (`Sf` is the self of the current object, `LbL` is its label, `Call` is the current call and `T` is the tracing mode. Of course, the tracing mode should be recovered later on, when the call skipped has succeeded. The implementation of the hooks should ensure that this is done (that is what we meant by 'compatibility' a few lines above. Only the `#s#` and `#v#` hooks need to recover the tracing mode. In our implementation, this is done through the `skip_trace(S,L,A)` call which is defined as:

```
skip_trace(S,L,A):-
  get_prop('#tracing#','#mode#',trace(S,L,A,T)),!,
  tracing(T),
  del_prop('#tracing#','#mode#').
```

This call, checks whether the call that it currently exits was skipped, finds the previous tracing mode, resets it and removes any saved tracing mode related info. Notice here that these two actions (tracing mode saving and skip-checking while exiting) should be implemented in any tracer interface, although not necessarily in this way.