Hitch Hiker's Guide to

Lygon 0.7

Michael Winikoff [1]

Technical Report 96/36

Department of Computer Science
The University of Melbourne
Parkville, Victoria 3052
Australia

[1] Department of Computer Science, the University of Melbourne, Parkville, Melbourne, 3052. `winikoff@cs.mu.oz.au`, `http://www.cs.mu.oz.au/~winikoff`

Contents

1 The Lygon Language **1**

1.1 Introduction 1

1.2 Syntax 2

1.3 Semantics 3

1.3.1 Resolution 5

1.3.2 Selecting the Formula to be Reduced 7

1.4 Current and Future Work 9

1.5 Literature 9

2 The Lygon System **11**

2.1 Installation 11

2.1.1 Porting 12

2.1.2 Local Usage 13

2.2 Usage 13

2.3 Accessing TCL / Tk from Lygon 18

2.4 Running Lygon without the GUI 19

2.5 Bugs 20

3 Example Lygon Programs **23**

3.1 Graph Problems 24

3.2 Representing states and updates 27

3.3 Representing actions and planning 28

3.4 Concurrency 30

3.5 Counting clauses 31

A Lygon Screendump **37**

Chapter 1

The Lygon Language

1.1 Introduction

Lygon is a logic programming language that is based on linear logic. Lygon can be viewed as Prolog extended with features derived from linear logic. These features include a clean declarative notion of state and the ability to express problems involving concurrency. In addition, the availability of *use once* predicates allows simple solutions to problems such as graph manipulation [HPW96b]. Just as Prolog programmers do not require a knowledge of proof theory, a knowledge of proof theory (used in the derivation of Lygon [PH94]) is not needed to program in Lygon.

Linear Logic can be described as a logic of *resources*. Whereas facts in classical logic are reusable, facts in linear logic must be used (by default) exactly once. This leads to a more distinguishing logic which is constructive yet retains the symmetries of classical logic. Both classical and intuitionistic logic can be embedded in linear logic.

Linear logic has applications in a large range of areas [Ale94, Ale93] including natural language processing, concurrency, logic programming and resource management (for example garbage collection).

A full introduction to linear logic is not necessary and is beyond the scope of this report Brief introductions to linear logic can be found in most papers on Lygon (see section 1.5). Slightly longer introductions can be found in [Laf88, Ale94, Ale93, Sce90]. The definitive references on linear logic are of course those by its designer [Gir87, Gir95].

In this report we describe the language and the implementation (Version 0.7). Current information about Lygon and its implementation can be found on the World Wide Web [Win96].

History

Lygon grew out of some work on the proof-theoretic foundations of logic programming languages. The fundamental idea behind Lygon is the completeness properties of certain kinds of proofs in the linear sequent calculus, and the initial work in this area was done in the second half of 1990. Over the period of the next two years, the operational model of the language was defined, revised, extended, and, in part, applied to other logics as well, and the language received its name over dinner one night late in 1992.

The first Lygon implementation appeared in the following year, although there were still some technical problems with the operational semantics, which were ironed out in early 1994, and a prototype implementation was completed later that year.

The Name

Lygon street is close to Melbourne University and is known for its restaurants and cafes.

Acknowledgements

Lygon is the work of a number of people. People currently working on Lygon include James Harland and David Pym who did the original design of the language. The debugger was designed and implemented by Yi Xiao Xu. Chapter 3 of this report is an extended version of [HPW96b].

1.2 Syntax

Lygon's syntax is similar to Prolog's with the main difference that the bodies of clauses are a (subset) of linear logic formulae rather than a sequence of atoms.

Program clauses are reusable (ie. nonlinear) by default. However it is possible to specify program clauses that must be used exactly once in each query. This is done by prefixing the clause with the word `linear`.

Lygon syntax is described by the following BNF. The notation [*x*] indicates that *x* is optional. Nonterminals are written in *italics* and terminals are in `teletype`.

G ::= G `*` G | G `@` G | G `&` G | G `#` G | `!` G | V `exists` G | `once` G
| `?` `neg` D | `one` | `bot` | `top` | A | `neg` A

D ::= [`linear`] A_1`#`$\ldots$`#`A_n `<-` G`.` | [`linear`] A`.`

A ::= A prolog atom

V ::= A lower-case symbol (eg. “x”)

Lexical syntax and comments follow the underlying Prolog system. For example:

- `p <- q(X).` is a valid program clause.
- `p * (q(1) # (a @ b))` is a valid query.

Note that almost anything is valid as an atom. The program clause `p :- q(X).` is valid and is parsed as a definition of the atom `:-` (ie. `':-'(p,q(X))`).

The default bracketing can be summarised by the following set of Prolog operator declarations. As always, when in doubt use excessive bracketing.

```
:- op(1200,xfy,exists).
:- op(1200,fx,linear).
:- op(1100,xfx,(<-)).
:- op(900, fy, [once,(!),(?)]).
:- op(500, xfy, [(&),(@)]).
:- op(450, xfy, (#)).
:- op(400, xfy, (*)).
:- op(300, fx, neg).
```

For example `(once a # b * c # d @ c & d # e)` is parsed as `(once ((a # (b * c) # d) @ (c & (d # e))))`.

1.3 Semantics

The semantics of Lygon can best be explained by the definition of an abstract interpreter. Although there is a sound mathematical basis for linear logic its declarative semantics is complicated and is not particularly useful in explaining Lygon.

An abstract Lygon interpreter operates in a simple cycle:

1. Select a formula to be reduced
2. Reduce the selected formula

This repeats until there are no goals left.

The selection step does not occur in prolog and is worth explaining. In Lygon a goal may consist of a number of formulae which evolve (conceptually) in parallel. At each step, we reduce one of these formulae. In general which we choose to reduce first *can* make a difference to whether a proof is found.

The Lygon interpreter thus may need to select a formula, and - if the proof attempt fails - backtrack and select another formula. The interpreter attempts to reduce this nondeterminism where possible using known properties of linear logic. This is outlined in [WH95b] and is summarised in section 1.3.2.

After selecting a formulae the interpreter reduces the selected formula. The reduction used is determined by the top level connective of the selected formula. In the rules below we denote the linear context (ie. the rest of the goal and any linear parts of the program) by *C*. When we say *"look for a proof of ..."* we mean that the goals indicated *replace* the goal being reduced.

- A`*`B: Split C into $C1$ and $C2$ and look for proofs of $(C1, A)$ and $(C2, B)$.
- A#B: Look for a proof of (C, A, B).
- A&B: Look for proofs of (C, A) and (C, B).
- A@B: Look for either a proof of (C, A) or a proof of (C, B).
- V `exists` G: Replaces occurences of V in G with a fresh variable yielding G' and then look for a proof of (C, G').
- `top`: The goal succeeds.
- `bot`: Look for a proof of (C).
- `one`: The goal succeeds *if* the context C is empty.
- A: *resolve* (see section 1.3.1).
- `neg` A: This is a linear fact. Linear facts are used by the axiom rule (see section 1.3.1).
- `? neg` A: Add A to the program and look for a proof of (C).
- `!`A: If C is empty then look for a proof of (A) otherwise fail.
- `once`A: This is an impure pruning operator. It is equivalent to A except that only the first possible proof will be found. Note that (a) `once` can only be used in an (effectively) single conclusion setting - that is C can only consist of goals of the form `neg`A. (b) `once` should be used with care. It can be

used to define most of Prolog's impure features (for example var, nonvar, negation as failure *etc.*).

Example

The goal `((one @ bot) # (one & top))` has a number of proofs. One proof involves the following steps:

1. There is only one formula so we select it. We then apply the # rule yielding a goal consisting of the two formulae `one @ bot` and `one & top`.
2. There are now two formulae in the goal. Let us select the first. We apply the @ rule yielding the goal `(bot, (one & top))`.
3. Assume we now decide to select the second formula. We apply the & rule yielding two goals: `(bot, one)` and `(bot, top)`.
4. We now have two goals which need to be proven independently. Let us consider the second goal first since it is the simpler. We can choose its second formula and apply the `top` rule. It succeeds and we are left with the first goal.
5. The remaining goal is now `(bot, one)`. We attempt to select the second formula but find that the context is nonempty and we can't apply the `one` rule.
6. We backtrack and select the first formula. the `bot` rule yields the goal `(one)` which is provable using the `one` rule.

1.3.1 Resolution

There are a number of ways of proving an atom:

1. It can be a builtin predicate. Builtins are logically equivalent to `one`.
2. It can match the axiom rule. The axiom rule states that a context of the form *(A,* `neg` *B)* where *A* and *B* unify is provable. Note that the context must not have any formulae other than *A* and `neg` *B*.
3. It can match a program clause. In this case, we nondeterministically select a (unifiable) program clause, create fresh variables for it, unify its head with the atom and replace the atom with its body. This behaviour is identical to Prolog's for clauses of the form `A <- G`. If the clause has a compound head (for example `(A # B) <- G`) then *both* `A` *and* `B` must be present as goals and both are deleted before `G` is added to the goal.

Example

Consider the program:

```
toggle <- (off * neg on) @ (on * neg off).
linear on.
```

The goal `toggle # off` is provable as follows:

1. Firstly, we add the linear parts of the program to the goal. The linear program fact `linear on` is equivalent to adding `neg on #` to the goal.

2. Our goal then is `(neg on # toggle # off)`. Using two applications of the # rule we obtain a goal consisting of three formulae: `(neg on, toggle, off)`.

3. There is never any point in selecting formulae of the form `neg` A. Hence we consider the other two formulae. Selecting `off` won't work - it is not a builtin, there is no program clause and we can't use the axiom rule.

4. Hence we select `toggle`. Using the program clause we obtain the goal `(neg on, off, ((off * neg on) @ (on * neg off)))`.

5. We select the compound formula and use the @ rule to reduce the goal to `(neg on, off, (off * neg on))`.

6. We select the compound formula and use the `*` rule. In applying this rule we have a choice as to how we split the context C. The Lygon implementation does this splitting lazily and deterministically. In this case it determines that no matter which way we split the linear context a proof is not possible.

7. We backtrack and try the other alternative of the @ rule. The goal is now `(neg on, off, (on * neg off))`.

8. We apply the `*` rule. This gives us the two goals `(neg on, on)` and `(off, neg off)` both of which are provable using the axiom rule.

Example

Consider the program:

```
a # b <- d @ c.
c # d <- top.
```

The goal `a # b # d # e` behaves as follows:

1. We apply the # rule repeatedly to obtain the goal `(a, b, d, e)`.

2. We now have to select atoms and apply resolution (since none of atoms are builtins and there are no negated atoms which the axiom rule can make use of). The second rule in the program can not be applied since `c` is not present. However, we can apply the first program rule. This replaces `a` and `b` with `d @ c` yielding the goal
`(d @ c, d, e)`

3. We select the compound goal and use the @ rule to obtain `(d, d, e)`

4. At this point we are stuck - neither program rule can be applied. We backtrack and select the other possible application of the @ rule to obtain `(c, d, e)`

5. We can now apply the second program rule which replaces `c` and `d` with `top` yielding `(top, e)`

6. We select `top` and the goal is provable using the `top` rule.

1.3.2 Selecting the Formula to be Reduced

When selecting the formula to reduce in a goal the Lygon interpreter uses a number of heuristics to reduce nondeterminism.

Firstly though, a few definitions. The connectives `# bot & top` and `?` are *asynchronous*. The connectives `* one @` and `!` are *synchronous*. A formula is classified as synchronous or asynchronous depending on its top level connective. This is discussed in the papers by Galmiche and Perrier [GP94] and by Andreoli [And92]. The following two facts can be used to reduce the nondeterminism associated with selecting formulae in a goal:

1. If there are formulae in the goal whose top level connectives are asynchronous then we can select any one of these formulae and commit to this selection. If a proof exists then we can find it by selecting an asynchronous formula first.

2. If we have just processed a synchronous connective and a / the sub-formula is itself synchronous then we can select the sub-formula for reduction. This is known as *focusing*.

Example

In the goal `(a * b, c # d, p(X))` we can select the second formula and commit to it since its top level connective - # - is asynchronous.

In the goal `(a * b, p(X) * (q(X) * r(X)))` if we select the second formula we may need to backtrack and try the first, however after processing the outermost * we can safely select the subformula `q(X) * r(X)` using focusing.

Built In Predicates

The Lygon system provides a number of primitive predicates for operations such as arithmetic, meta-programming and I/O:

- **print/1**: prints its argument. Example: `print('hello, world!')`.

- **nl/0**: prints a newline.

- **input/1**: reads a Prolog term and binds it to its argument. Note that when using the GUI input comes from the TCL/Tk process and not from the command line.

- **system/1**: passes its argument to the system to be executed as a command. Example: `system('dir ..')`.

- **is/2**: evaluates its second argument as a mathematical expression and binds the result to the first argument. Inherited from prolog. Example: `X is 1+Z`. Note that there is a potential parsing problem in that * is both a Lygon connective and an arithmetic operator. For example, the goal `X is 1 + 2 * print(X)` will not work. Two alternative ways of writing this goal are `(X is 1+2) * print(X)` or `is(X,1+2) * print(X)`.

- **lt/2**: succeeds if both its arguments are numbers and the first is less than the second. Example: `lt(1,3)` succeeds whilst `lt(3,1)` fails.

- **builtin/1**: succeeds if its argument matches a builtin predicate. Example: `builtin(builtin(_))` and `builtin(print(asd))` both succeed.

- **readgoal/1** and **readprog/1**: read terms and convert them to Lygon goals and programs respectively. The difference between these and **input/1** is that atoms are tagged. Example: if the user types `p * q # r` the call to **readgoal/1** returns `atom(p)*atom(q)#atom(r)`.

- **prolog/1**: passes the argument to the underlying prolog system for execution. This can be used to extend the set of builtins. Example: `prolog(see(file))`.

- **tcl/1**: passes its argument to the TCL / Tk interface for evaluation. This can be used to access graphics from within Lygon or to extend the user interface (see section 2.3).

- **call/1**: `call(X)` is equivalent to `X`. It is needed since internally Lygon represents `p(X)` as `atom(p(X))`.

1.4 Current and Future Work

There is a range of interesting research topics relating to Lygon specifically and to linear logic programming more generally. The following are simply the ones which we are currently pursuing.

When programming in Lygon certain idioms arise naturally. For example a certain configuration of connectives expresses state handling. It is possible to add syntactic sugar which encodes (and makes more readable and convenient) certain programming idioms.

The current Lygon debugger is based on a fairly standard four port tracing debugger. Unfortunately Lygon typically has even more backtracking than Prolog. It is thus highly desirable to look into the design and construction of a declarative debugger. This is an interesting research question in that a goal can not be considered to be true or false in isolation - in general the *context* in which a goal is called matters.

Work is continuing on investigating applications of Lygon. Currently under investigation are applications with an AI flavour such as non-monotonic reasoning and belief revision.

Finally, it appears feasible to base Lygon on a logic programming language other than Prolog. Specifically we are looking at integrating Lygon with Mercury [SHC$^+$96, SHCO95, Hen96]. This would probably involve adding Mercury's type, mode and determinism systems to Lygon.

1.5 Literature

The development of Lygon is described in a series of papers which trace the various stages of its development:

- **Design:** The initial design is reported and developed on in [HP90, HP91, HP92] and is published in its final form in [PH94]. The question of design has recently been reopened in [WH96a].

- **Implementation:** The first stab at implementing Lygon is reported on in [WH95a]. The final version of the rules for deterministic splitting of resources are published as [WH95b, WH94]. The Lygon system was demonstrated at AMAST'96 [HPW96a].

- **Application:** Initial ideas for applications predate the implementation [HP94]. Further papers discussing applications of Lygon include [HPW95, WH96b, HPW96b].

Lygon papers can be found on the World Wide Web at http://www.cs.mu.oz.au/~winikoff/papers/papers/ABSTRACTS.html and at http://www.cs.rmit.edu.au/~jah/linear-pubs.html

Other Linear Logic Programming Languages

Since its development in 1986 linear logic has excited considerable interest in the computer science research community. Not surprisingly there are a number of logic programming language which make use of linear logic in some form. A short description of these languages is contained in a survey of linear logic programming [Mil95b].

- ACL [KY93]
- Forum [Mil94, Mil95a, HP96]
- $\mathcal{LC}$ [Vol94]
- LinLog [And92]
- LO [AP91a, AP91b, AP90]
- Lolli [HM94, Hod94]

Chapter 2

The Lygon System

2.1 Installation

In order to run Lygon you need to have installed:

- A recent-ish version of TCL / Tk[1].
- BinProlog version 5.00 or later[2] (But see section 2.1.1)

The Lygon distribution can be found at the Lygon homepage [Win96] and comprises the following files:

1. `test.lyg:` When Lygon starts up it loads this file as the default program. If absent a warning is given. (Optional)
2. `lygon.xbm:` This is the icon displayed at the top left. (Optional)
3. `lygon.pl:` The prolog code for the interpreter.
4. `lygon:` The TCL / Tk code for the interface.

To install Lygon you may need to make some changes to the configuration of the TCL / Tk script:

1. Change the line `#!/usr/local/bin/wish` to point to where wish is on your system.
2. Change the following lines to specify the fonts for different GUI elements. (A list of fonts available can be obtained by running `xlsfonts`)

[1] Available from `http://www.sunlabs.com:80/research/tcl/`
[2] Available from `http://clement.info.umoncton.ca/~tarau/`

```
set fontfile "-*-courier-medium-*-normal--*-140-*-*-*-*-*-*"
set fontquery "-*-courier-medium-r-normal--*-140-*-*-*-*-*-*"
set fonttext "-*-courier-medium-r-normal--*-120-*-*-*-*-*-*"
set fontlabel "-*-Helvetica-medium-r-normal--*-180-*-*-p-*-*-*"
```

3. Change the following line to specify which editor to use
   `set editor  "exec /usr/bin/X11/xterm -rv -T Lygon -e vi %s"`

4. Follow the instructions in the following lines:

```
# Make sure the next line points to the lygon icon ...
label .restart -bitmap @lygon.xbm -command {p "alive"}
# ... or replace it with this line:
#label .restart -text "Lygon 0.7\nPing" -font $fontlabel \
#           -command {p "alive"}
```

5. Change the following line so that (a) it points to where `lygon.pl` is and (b) it points to where the BinProlog executable `bp` resides. (Note that step (b) is not required if `bp` is in your path).
   `return [start_slave "bp -h30000 -s10000 -t10000 -c2000 ./lygon.pl"]`

6. You may find it useful to add the location of `lygon` to your path. and it may be necessary to do `chmod u+x lygon`.

2.1.1 Porting

Lygon is written in fairly standard Prolog and should be easy to port to other Prolog systems. For example, to port Lygon to NU-prolog the following changes were required:

1. In `lygon` change the following line to `return [start_slave "np"]`
   `return [start_slave "bp -h30000 -s10000 -t10000 -c2000 ./lygon.pl"]`

2. In `lygon` add `-state disabled` to the end of the line
   `button .stop -text "Interrupt!" -command {interrupt_prolog }`
   (NU-Prolog doesn't catch interrupts in the appropriate way by default).

3. In `lygon` towards the end of the file before `p "gui"` add the lines

```
 puts $f "\[lygon\]."
 flush $f
```

These load Lygon (which starts itself running).

4. The main change to `lygon.pl` concerns the way in which quoting is handled. Find every occurence of `\n`, `\[` and `\]` and add an extra backslash (eg. change `\n` to `\\n`).

5. Add as the last line of `lygon.pl` the line `:- seen, run.` This closes the file being consulted and starts the Lygon interpreter.

6. In `lygon.pl` replace `halt` with `exit(0)`.

2.1.2 Local Usage

Melbourne University people with an account on mundook (or murlibobo) can run Lygon without having to do any installation. Simply execute the file `/home/pgrad/winikoff/lygon/lygon`

2.2 Usage

To start Lygon simply run the TCL/Tk script `lygon`. It takes a single optional argument which is the name of a file containing a Lygon program. The TCL/Tk script will start up BinProlog and load the Lygon interpreter.

The Lygon GUI is divided into four panels. From top to bottom these are:

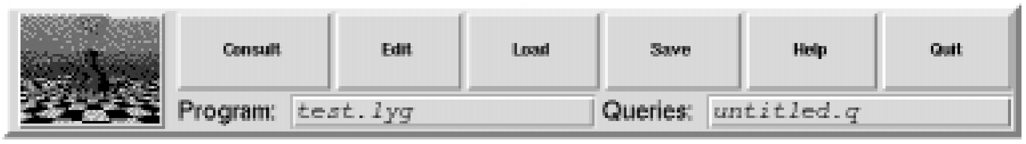
1. Main control

2. Queries

3. Debugging

4. Results

Main Control



The top row of buttons acts as follows:

- **Consult:** Loads the program into the Lygon interpreter.

- **Edit:** Fires up an editor on the program file. When the command returns it does a consult. If you run your editor in the background you will need to manually consult the file.

- **Load:** Loads all of the GUI's text fields from a file (default is `untitled.q`). This is used to store typical program queries.

- **Save queries:** Saves text fields to a file

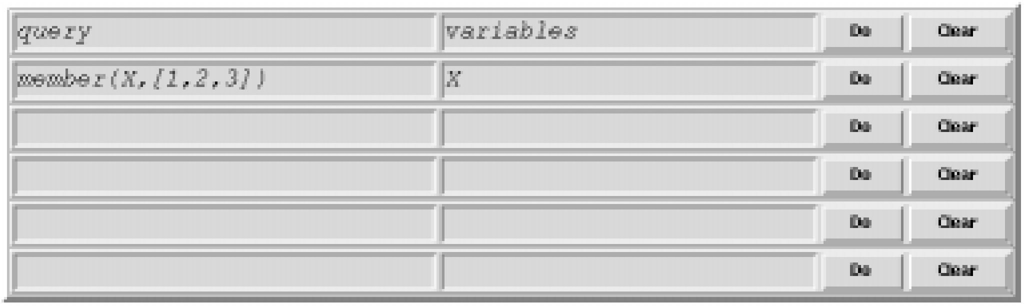
- **Help:** Displays a (very!) brief help message.

- **Quit:** Quits.

The text entry fields immediately below the top row of buttons allows the user to specify the program file and query file. The query file is (despite the name) used to store all text fields of the GUI (ie. including **show** and **watch**).

Hitting the icon in the top right hand corner sends an "are you there?" message to the underlying Prolog system. If present, Prolog will respond with a "yes, I'm alive" message.

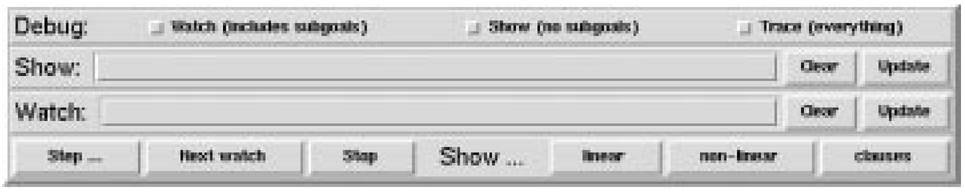
Queries



query	variables	Do	Clear
member(X,[1,2,3])	X	Do	Clear
		Do	Clear
		Do	Clear
		Do	Clear
		Do	Clear

The leftmost column of text entry fields contains goals, the second leftmost column contains a (space separated) list of variables whose values we would like to be reported. The **do** button runs the query and the **clear** button clears the query and variables fields. Note that a terminating full stop is not needed.

Debugging



The debugging panel has four rows of controls. The top row controls the form of the debugging trace. The bottom row of buttons is used during a trace to get information and control the trace. The middle two are used to specify which formulae to look out for.

To understand how to use the debugger we need to understand the distinction between **watch** and **show**[3].

The debugger will print out any formula that matches a formula in the show or watch lists. In the case of a watched formula however it will also print out *all* of that formula's subgoals.

For example given the query a and the program

```
a <- b.
b <- c.
```

If we **show** b then the debugging output is

```
>>[2]Call   b
>>[2]Fail   b
```

However, if we **watch** b then the debugging output is

```
**[2]Call   b
[3]Call   b <- c
[4]Call   c
[4]Fail   c
[3]Fail   b <- c
**[2]Fail   b
```

The first two toggles (from the left) turn watching and showing on and off. Tracing (which is controlled by the next toggle) prints *everything* and overrides watch and show. (although formulae that match the watch or show lists will be preceded by an indicator).

The **show** (respectively **watch**) text field should contain a (space separated) list of formulae. For example to show all calls to eq and once it would be

[3]If you can suggest better terminology please let me know - winikoff@cs.mu.oz.au

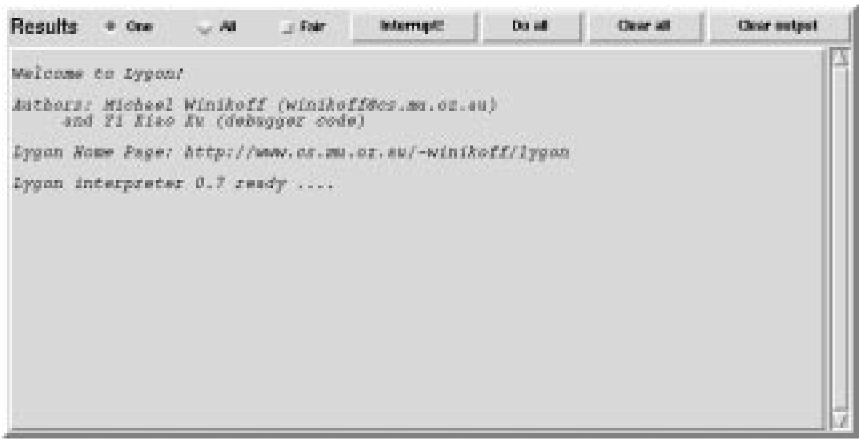
```
eq(_,_) once(_)
```

The **clear** button does the obvious thing. The **update** button is used when changing the lists in the middle of a trace. It tells the Lygon interpreter about any changes you have made to the list of goals to be shown or watched. Note that evaluating a new goal (using **do**) or flipping the toggles which control whether watching or showing happens will also update the Lygon interpreter.

The bottom row of buttons in the debug panel are used during a trace to control the output:

- **Step:** Takes a single step forward.
- **Next watch:** This button is to be used when tracing is disabled, watching is enabled and the system is single stepping through the subgoals of a watched goal. This button stops single tracing. Single tracing will be re-enabled when a watched goal is encountered.
- **Stop:** Stops all tracing.
- Show **linear:** Prints out the linear facts currently present.
- Show **non-linear:** Prints out the non-linear facts currently present.
- Show **clauses:** Prints out a list of clauses that were used.

Results



```
Welcome to Lygon!

Authors: Michael Winikoff (winikoff@cs.mu.oz.au)
     and Yi Xiao Xu (debugger code)

Lygon Home Page: http://www.cs.mu.oz.au/~winikoff/lygon

Lygon interpreter 0.7 ready ....
```

Do all runs all of the queries in sequence and **clear all** clears all of the queries. **Clear output** clears the output window. The button labelled **Interrupt!** sends a control-C to the Lygon system. This (hopefully!) will interrupt a running query and return to the top level. If it works the system will redisplay a welcome message.

The toggle labelled **Fair** has to do with the way Lygon selects formulae in a goal. Recall that when a goal consists of multiple formulae we need to select a formula then apply the appropriate rule to it. Normally Lygon will select the leftmost formula (if none of the optimisations outlined in section 1.3.2 apply). Turning **fair**ness on *randomises* the selection. For example, consider the goal `a # b` and the program:

```
a <- print(a) * a.
b <- nl * top.
```

Without fairness, the formula `a` is chosen repeatedly and an infinite loop results. However with fairness turned on there is an equal chance of `b` being chosen and the computation terminated by using the `top` rule. Note that different choices may be made if the goal is called again:

```
Lygon a # b.
aaa
Lygon a # b.
aaaaaaaa
Lygon a # b.
aaa
```

The **One** and **All** buttons select whether a single solution or all solutions to a query are found and printed. They behave differently for queries with no variables. There are four types of queries:

1. A query with no variables and with **One** selected: this is only run for side effects and no results will be reported. No message is given if the query fails.

2. A query with variables and with **One** selected: the query is run and the first solution found is reported. Note that unbound variables are not reported.

3. A query with no variables and with **All** selected: The query is run and the system reports whether it succeeded or failed.

4. A query with variables and with **All** selected: All solutions are found and displayed.

For example with the program

```
member(X,[X|_]).
member(X,[_|Y]) <- member(X,Y).
```

And the goals:

```
member(X,[1,2,3])
tcl(['puts "asd"'])
```

where the first query has the variable X requested, the system gives the following response for **one**:

```
X = 1
```

Note that only a single solution is reported and that nothing is reported for the second goal (which writes to the terminal from which Lygon was started). If **all** is selected the output is:

```
X = 1

X = 2

X = 3

yes
```

Note that all answers are returned and that the second query resulted in the `yes`.

Note: These are implemented by sending Lygon a modified query. This of course can be seen if any debugging is done. For this reason it is probably simpler to use the first query type when debugging.

2.3 Accessing TCL/Tk from Lygon

The Lygon predicate **tcl/1** (which takes a list and sends its concatenation to TCL / Tk for execution as a command) can be used to access graphics functions from Lygon. For example consider the following Lygon program:

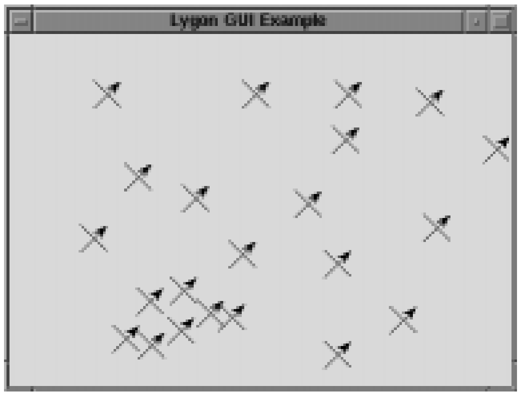
```
go <- tcl(['toplevel .lygtop']) *
      tcl(['canvas .lygtop.c']) *
      tcl(['pack .lygtop.c']) *
      tcl(['wm title .lygtop "Lygon GUI Example"']) *
      tcl(['bind .lygtop.c <Button-1> {p "line(%x,%y)"}']).

end <- tcl(['destroy .lygtop']).

line(X,Y) <- is(X1,X + 20) * is(Y1,Y + 20) * line(X,Y,X1,Y1) *
```

```
        arrow(X,Y1,X1,Y).

line(X,Y,X1,Y1) <-
    tcl(['.lygtop.c create line ', X,' ',Y,' ',X1,' ',Y1]).

arrow(X,Y,X1,Y1) <-
    tcl(['.lygtop.c create line ', X,' ',Y,' ',X1,' ',Y1,
         ' -arrow last']).
```

The predicate `go` creates a window and binds a left mouse button click to sending the goal `line(X,Y)` to Lygon where `X` and `Y` are the coordinates of the click.

The next few clauses call back TCL/Tk and place a little graphic figure into the window.

In order to run this program load it into Lygon then run `go`. A window will appear in which you can click. When done run `end` and the window will go away again.



2.4 Running Lygon without the GUI

Although the GUI is easier to use it only runs on Unix/X platforms. However, it is possible to run Lygon without the GUI. To do this run BinProlog and then load the file `lygon.pl`. Note that Lygon does *not* print out query results - these must be requested explicitly. The commands available when running Lygon without a GUI are:

- **bye:** Quits Lygon. (note that control-D and control-C don't work)
- **compile(X):** Compiles the file `X`. (eg. `compile('test.lyg').`)

- **traceOn & traceOff:** Turn tracing on and off.
- **watchOff:** Turns watching off.
- **setshow(X) & setwatch(X):** Specify which formulae you want to watch and show. (eg. `setshow([eq(_,_), once(_)])`)
- **show(X) & watch(X):** *Add* a single formula to the list of watched and shown formulae. (eg. `watch(eq(_,_))`)
- **rmshow(X), rmallshows, rmwatch(X) & rmallwatches:** Used to remove formulae from the watch and show lists. (eg. `rmshow(once(_))`)
- **listshows & listwatches:** Display the show and watch lists respectively.
- **cleanup:** Deletes the show and watch lists and turns all tracing off.

These commands are accepted at all times. In addition the following commands are relevant while debugging:

- **lr, nlr & cls:** Display respectively the linear context, the non-linear context and the list of clauses that have been used.
- **addwf & rmwf:** Add/remove the current formula to/from the watch list.
- **towatchf:** The same as hitting the "Next Watch" button.
- **step:** This is used to step forward. Unlike older versions of the debugger you can't just hit return.

2.5 Bugs

- An extra click on **step** is needed at the end of a tracing run
- When run from the command line the system comes up with messages of the form:

  ```
  atom/1 ??? warning: *** bad constructor starts 2nd arg of member_scan/3 ***
  ```

  This doesn't occur under prologs other than BinProlog.
- Due to the way in which the Lygon system gives a `yes` or `no` response to queries the use of linear *clauses* can cause problems. For example given the program `linear a <- b` the query `a` is translated[4] to the Lygon query

[4] With **all** selected

```
once( (a * !(print(yes) * nl))
    @ (print(no) * nl))
```

This is then translated by Lygon to

```
(neg a * b) # once( (a * !(print(yes) * nl))
                  @ (print(no) * nl))
```

This query will print `yes` and then end up failing. The long term solution is to add the reporting of query results to Lygon rather than modifying the query. The short term solution is to avoid linear clauses and use only linear facts. Note that a linear clause can be simulated using linear facts. For example the program given above could be simulated by the program

```
linear x.
a <- x * b.
```

Chapter 3

Example Lygon Programs

In this chapter, we discuss various programming techniques which distinguish Lygon from Prolog (see also [HP94, WH96b, WH95b, HPW96b, Win96]) and give examples of Lygon programs. This chapter is based on [HPW96b].

More Lygon programs - including programs to do depth and breadth first traversal of DAGs, model exceptions, do topological sorting and meta-interpret Lygon programs can be found at http://www.cs.mu.oz.au/~winikoff/lygon/progs

Since Lygon is an extension of Prolog any (pure) Prolog program will run under Lygon. In general Lygon's `*` connective substitutes for Prolog's "`,`". For example here is a well known Prolog program transcribed into Lygon:

```
append([],Y,Y).
append([X|Xs],Y,[X|Z]) <- append(Xs,Y,Z).

nrev([],[]).
nrev([X|Xs],R) <- nrev(Xs,R1) * append(R1,[X],R).
```

As noted in section 1.3 the proof of the goal A `*` B partitions the context between the proof of A and the proof of B. Two special cases that are common in Lygon programs are when A is a builtin and when A is an atom which has corresponding linear facts. In the first case the entire context gets passed to B since builtins are logically equivalent to `one` and can't use any of the context. In the second case a single linear fact matching A is used by the axiom rule in the proof of A and the remaining context is used in the proof of B.

For example consider the goal `(neg p(1) , neg q , p(X) * print(X) * top)`. The only formula which can be usefully selected is the third. This gives us the two goals `(neg p(1) , p(X))` and `(neg q , print(X) * top)`. The first is provable using the axiom rule which unifies `X` with `1`. In proving the second goal we again use the `*` rule yielding the two new goals `(print(1))` and `(neg q , top)`. The first of these succeeds and as a side effect prints `1`. The second succeeds using the `top` rule.

Another connective in Lygon that is often used in a particular way is #. Recall that the proof of the goal A # B simply looks for a proof of A, B. If A and B are both atoms which have matching program clauses then the two formulae evolve in parallel. A commonly occuring special case is `neg` A # B which adds the linear fact A to the context and then continues with B. This is equivalent to $A \multimap B$.

3.1 Graph Problems

Graphs are an important data structure in computer science. Indeed, there are many applications of graph problems, such as laying cable networks, evaluating dependencies, designing circuits and optimization problems. The ability of Lygon to naturally state and satisfy constraints, such as that every edge in a graph can be used at most once, means that the solution to these problems in Lygon is generally simpler than in a language such as Prolog. The solutions presented are concise and lucid.

One of the simplest problems involving graphs is finding paths. The standard Prolog program for path finding is the following one, which simply and naturally expresses that the predicate `path` is the transitive closure of the predicate `edge`, in a graph.

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Whilst this is a simple and elegant program, there are some problems with it. For example, the order of the predicates in the recursive rule is important, as due to Prolog's computation rule, if the predicates are in the reverse order, then goals such as `path(a,Y)` will loop forever. This problem can be avoided by using a memoing system such as XSB [War94], or a bottom-up system such as Aditi [VRD+94]. However, it is common to re-write the program above so that the path found is returned as part of the answer. In such cases, systems such as XSB and Aditi will only work for graphs which are acyclic. For example, consider the program below.

```
path(X,Y,[X,Y]) :- edge(X,Y).
path(X,Y,[X|Path]) :- edge(X,Z), path(Z,Y,Path).
```

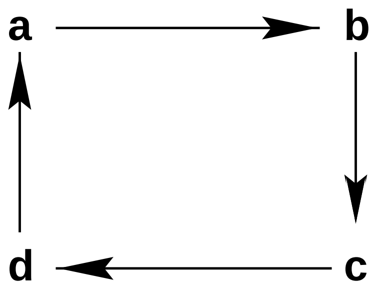
If there are cycles in the graph, then Prolog, XSB and Aditi will all generate an infinite number of paths, many of which will traverse the cycle in the graph more than once.

The main problem is that edges in the graph can be used an arbitrary number of times, and hence we cannot mark an edge as used, which is what is done in many imperative solutions to graph problems. However, in a linear logic programming language such as Lygon, we can easily constrain each edge to be used at most once on any path, and hence eliminate the problem with cycles causing an infinite number of paths to be found.

The code is simple; the main change to the above is to load a 'linear" copy of the `edge` predicate, and use the code as above, but translated into Lygon. Most of this is mere transliteration, and is given below.

```
graph <- neg edge(a,b) # neg edge(b,c)
       # neg edge(c,d) # neg edge(d,a).

trip(X,Y,[X,Y]) <- edge(X,Y).
trip(X,Y,[X|P]) <- edge(X,Z) * trip(Z,Y,P).

path(X,Y,P) <- top * trip(X,Y,P).
```

The `graph` clause above represents the following graph:



The extra predicate `trip` is introduced so that not every path need use every edge in the graph. As written above, `trip` will only find paths which use every edge in the graph (and so `trip` can used directly to find Eulerian circuits, *i.e.*, circuits which use every edge in the graph exactly once). However, the `path` predicate can ignore certain edges, provided that it does not visit any edge more than once, and so the `path` predicate may be considered the affine form of the predicate `trip`.

The goal `graph` is used to load the linear copy of the graph, and as this is a non-linear rule, we can load as many copies of the graph as we like; the important feature is that within each graph no edge can be used twice. We can then find all paths, cyclic or otherwise, starting at node `a` in the graph with the goal

```
graph # path(a,_,P).
```

This goal yields the solutions below.

```
P = [a,b,c,d,a]   P = [a,b,c,d]
P = [a,b,c]       P = [a,b]
```

We can also find all cycles in the graph with a query such as `graph # path(X,X,P).` which yields the solutions:

```
X = c, P = [c,d,a,b,c]
X = d, P = [d,a,b,c,d]
X = b, P = [b,c,d,a,b]
X = a, P = [a,b,c,d,a]
```

This example suggests that Lygon is an appropriate vehicle for finding "interesting" cycles, such as Hamiltonian cycles, *i.e.*, those visiting every node in the graph exactly once. We can write such a program in a "generate and test" manner by using the `path` predicate above, and writing a test to see if the cycle is Hamiltonian. The key point to note is that we can delete any edge from a Hamiltonian cycle and we are left with an acyclic path which includes every node in the graph exactly once. Assuming that the cycle is represented as a list, then the test routine will only need to check that the "tail" of the list of nodes in the cycle (*i.e.*, the returned list minus the node at the head of the list) is a permutation of the list of nodes in the graph. Hodas and Miller [HM94] have shown that such permutation problems can be solved simply in linear logic programming languages by "asserting" each element of each list into an appropriately named predicate, such as `list1` and `list2`, and testing that `list1` and `list2` have exactly the same solutions.

The full Lygon program for finding Hamiltonian cycles is given below.

```
go(P) <- graph # (top * (nodes # hamilton(P))).

graph <- neg edge(a,b) # neg edge(b,c) # neg edge(c,d) # neg edge(d,a).
nodes <- neg node(a) # neg node(b) # neg node(c) # neg node(d).

trip(X,Y,[X,Y]) <- edge(X,Y).
trip(X,Y,[X|P]) <- edge(X,Z) * trip(Z,Y,P).

all_nodes([]).
all_nodes([Node|Rest]) <- node(Node) * all_nodes(Rest).

hamilton(Path) <- trip(X,X,Path) * eq(Path,[_|P]) * all_nodes(P).

eq(X,X).
```

The rôle of the `top` in `go` is to make the `edge` predicate affine (*i.e.*, not every edge need be used). Given the query `go(P)`, the program gives the solutions:

```
P = [c,d,a,b,c]  P = [d,a,b,c,d]
P = [b,c,d,a,b]  P = [a,b,c,d,a]
```

A problem related to the Hamiltonian path is that of the travelling salesman. In the travelling salesman problem we are given a graph as before. However each edge now has an associated *cost*. The solution to the travelling salesman problem is the (or a) Hamiltonian cycle with the minimal total edge cost. Given a facility for finding aggregates, such as `findall` or `bagof` in Prolog, which will enable all solutions to a given goal to be found, we can use the given program for finding Hamiltonian cycles as the basis for a solution to the travelling salesman problem. This would be done by simply finding a Hamiltonian cycle and computing its cost. This computation would be placed within a `findall`, which would have the effect of finding all the Hamiltonian cycles in the graph, as well as the associated cost of each. We would then simply select the minimum cost and return the associated cycle. Note that as this is an NP-complete problem, there is no better algorithm known than one which exhaustively searches through all possibilities.

In order to directly implement the solution described above, aggregate operators in Lygon are needed. As these are not yet present we do not give the code for this problem here.

3.2 Representing states and updates

When attempting to find a proof of $(G_1$ `*` $G_2, C)$, we use the technique of passing the unused resources from one conjunct to the other. This can be used as a kind of state-mechanism, in that the first conjunct can pass on information to the second. In particular, we can use this feature to simulate a memory. For example, consider a memory of just two cells, represented by two instances of the predicate `m`, the first argument being the address and the second the contents of the cell. The state in which these two cells contain the values t_1 and t_2 would then be represented by the goal containing the two linear facts `m(1,`t_1`)` and `m(2,`t_2`)`. A (non-destructive) read for cell 2, say, would be given by the goal formula `m(2,X) * (neg m(2,X) #` G`)` where G is to be executed after the read. The states in this computation are (i) that `m(2,X)` is unified with `m(2,`t_2`)`, (ii) that the latter atom is deleted from the program, and then (iii) added again via the # connective, before G is executed. Similarly, writing the value t' into the memory can be done using the goal `m(2,X) * (neg m(2,`t'`) #` G`)`, where it is possible that t' can contain `X`, so that either the new value can be dependent on the old, or t'

can be totally independent of the old value. In this way we can use the "delete after use" property of the linear system to model a certain form of destructive assignment.

Using a continuation passing style to encode sequentiality, with a predicate `call` to invoke continuations, we can create an abstract data type for memory cells using the operations `newcell`, `lookup` and `update`.

```
newcell(Id,Value,Cont) <- neg m(Id,Value) # call(Cont).
lookup(Id,Value,Cont) <- m(Id,Value) * (neg m(Id,Value) # call(Cont)).
update(Id,NewValue,Cont) <- m(Id,_) * (neg m(Id,NewValue) # call(Cont)).
```

For example, consider summing a list using a variable which is updated. (The `top` in the second clause is needed to consume the cell once it is no longer needed.)

```
sum(List,Result) <- newcell(sum,0, sumlist(List,Result)).
sumlist([],Result) <- lookup(sum,Result,top).
sumlist([N|Ns],Result)
 <- lookup(sum,S,(is(S1,S+N) *
    update(sum,S1, sumlist(Ns,Result)))).
```

We can then run the program using a goal such as `sum([1,5,3,6,7],X)` which yields the solution `X = 22`.

3.3 Representing actions and planning

The notion of state present in Lygon can also be applied in planning type problems where there is a notion of a state and operators which change the state.

The Yale shooting problem [HM87] is a prototypical example of a problem involving actions. The main technical challenge in the Yale shooting problem is to model the appropriate changes of state, subject to certain constraints. In particular:

1. Loading a gun changes its state from unloaded to loaded;
2. Shooting a gun changes its state from loaded to unloaded;
3. Shooting a loaded gun at a turkey changes its state from alive to dead.

To model this in Lygon, we have predicates `alive, dead, loaded,` and `unloaded`, representing the given states, and predicates `load` and `shoot`, which, when executed, change the appropriate states. The initial state is to assert `alive` and `unloaded`, as initially the turkey is alive and the gun unloaded. The actions of loading and shooting are governed by the following rules:

```
load <- unloaded * neg loaded.
shoot <- alive * loaded * (neg dead # neg unloaded).
```

Hence given the initial resources `alive` and `unloaded`, the goal `shoot # load` will cause the state to change first to `alive` and `loaded`, as `shoot` cannot proceed unless `loaded` is true, and then `shoot` changes the state to `dead` and `unloaded`, as required.

A (slightly) less artificial planning problem is the blocks world. The blocks world consists of a number of blocks sitting either on a table or on another block and a robotic arm capable of picking up and moving a single block at a time. We seek to model the state of the world and of operations on it.

The predicates used to model the world in the Lygon program below are the following:

- `empty:` the robotic arm is empty;
- `hold(A):` the robotic arm is holding block A;
- `clear(A):` block A does not support another block;
- `ontable(A):` block A is supported by the table;
- `on(A,B):` block A is supported by block B.

There are a number of operations that change the state of the world. We can `take` a block. This transfers a block that does not support another block and that is supported by the table into the robotic arm. It requires that the arm is empty.

```
take(X) <- (empty * clear(X) * ontable(X)) * neg hold(X).
```

We can `remove` a block from the block beneath it, which must be done before picking up the bottom block.

```
remove(X,Y) <-
 (empty * clear(X) * on(X,Y)) * (neg hold(X) # neg clear(Y)).
```

We can also `put` a block down on the table or `stack` it on another block.

```
put(X) <- hold(X) * (neg empty # neg clear(X) # neg ontable(X)).

stack(X,Y) <- (hold(X) * clear(Y)) *
              (neg empty # neg clear(X) # neg on(X,Y)).
```

Finally, we can describe the initial state of the blocks and change the state of the world:

```
initial <- neg ontable(a) # neg ontable(b) # neg on(c,a)
         # neg clear(b) # neg clear(c) # neg empty.
```

```
Lygon (initial # take(c) # put(c) # take(a) # stack(a,b) # showall(R)).
[empty,on(a,b),clear(a),clear(c), ontable(c), ontable(b)]
Succeeded.
```

The order of the instructions `take`, `put` *etc.* is not significant: there are actions, specified by the rules, such as `put(c)`, which cannot take place from the initial state, and others, such as `take(b)` which can. It is the problem of the implementation to find an appropriate order in which to execute the instructions, so giving the final state.

3.4 Concurrency

Our next example is the classical dining philosophers (or logic programmers) problem and illustrates the use of Lygon to model concurrent behaviour.[1] This solution is adapted from [CG89].

For N logic programmers there are $N-1$ "room tickets". Before entering the room each logic programmer must take a roomticket from a shelf beside the door. This prevents all of the programmers from being in the room at the same time.

The program uses a number of linear predicates: `rm` represents a roomticket, `log(X)` represents the Xth programmer and `ch(X)` the Xth chopstick.

It is recommended that this program be run with **fair**ness turned on.

```
go <-  log(a) # neg ch(a)
     # neg rm # log(b) # neg ch(b)
     # neg rm # log(c) # neg ch(c)
     # neg rm # log(d) # neg ch(d)
     # neg rm # log(e) # neg ch(e).

log(N) <-
   hack(N) * rm *
   succmod(N,N1) * ch(N) * ch(N1) *
   eat(N) * (neg ch(N) # neg ch(N1) # neg rm # log(N)).
```

Procedurally, this code is read as: get a room ticket; get the chopsticks in sequence; eat; return the chopsticks and room ticket; go back to hacking.

[1]This problem is particularly apt — Lygon's name is of gastronomic origin.

```
succmod(a,b).    succmod(d,e).
succmod(b,c).    succmod(e,a).
succmod(c,d).

eat(N) <- print('log(') * print(N) * print(') eating') * nl.
hack(N) <- print('log(') * print(N) * print(') hacking') * nl.
```

3.5 Counting clauses

Another problem, in which the properties of linear logic make a significant simplification and which has been discussed as a motivation for the use of embedded implications in the presence of Negation-as-Failure is the following: given a number of clauses $r(1), \ldots\ r(n)$, how can we determine whether n is odd or even ? The program below has been used for this purpose.

```
even :- not odd.
odd :- select(X), (mark(X) => even).
select(X) :- r(X), not mark(X).
```

Note the dependence on the co-existence of Negation-as-Failure and embedded implications. In the linear case, there is no need to do the explicit marking, as this will be taken care of by the Lygon system. This can be thought of as a simple aggregate problem; a good solution to this would indicate potential for more involved problems (and possibly some meta-programming possibilities). Clearly the marking step can be subsumed by the linear properties of Lygon, resulting in a conceptually simpler program, which is given below.

```
check(Y) <- r(X) * (toggle # check(Y)).
check(X) <- count(X).

toggle <- count(even) * neg count(odd)
toggle <- count(odd) * neg count(even).
```

The goal

```
neg count(even) # neg r(1) # neg r(2) # check(X).
```

returns the answer `X = even`.

Bibliography

[Ale93] Vladimir Alexiev. Applications of linear logic to computation: An overview. Technical Report TR93-18, University of Alberta, December 1993.

[Ale94] Vladimir Alexiev. Applications of linear logic to computation: An overview. *Bulletin of the IGPL*, 2(1):77–107, March 1994.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[AP90] Jean-Marc Andreoli and Remo Pareschi. LO and behold! concurrent structured processes. *SIGPLAN Notices*, 25(10):44–56, 1990.

[AP91a] Jean-Marc Andreoli and Remo Pareschi. Communication as fair distribution of knowledge. In Andreas Parpcke, editor, *OOPSLA*, pages 212–229, 1991.

[AP91b] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir95] Jean-Yves Girard. Linear logic : its syntax and semantics. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, chapter 0. Cambridge University Press, 1995.

[GP94] Didier Galmiche and Guy Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, December 1994.

[Hen96] Fergus Henderson. Home page of the Mercury project. http://www.cs.mu.oz.au/mercury, 1996.

[HM87] S. Hanks and D. MacDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic logic. *Journal of Information and Computation*, 10(2):327–365, 1994.

[Hod94] Joshua Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation*. PhD thesis, University of Pennsylvania, 1994.

[HP90] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming. LFCS Report Series ECS-LFCS-90-124, University of Edinburgh, 1990.

[HP91] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming (extended abstract). In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 304–318. MIT Press, 1991.

[HP92] James Harland and David Pym. A synopsis on the identification of linear logic programming languages. LFCS Report Series ECS-LFCS-92-248, University of Edinburgh, 1992.

[HP94] James Harland and David Pym. A note on the implementation and applications of linear logic programming languages. In Gopal Gupta, editor, *Seventeeth Annual Computer Science Conference*, pages 647–658, 1994.

[HP96] Joshua S. Hodas and Jeffrey Polakow. Forum as a logic programming language: Preliminary results and observations. In *Linear Logic 1996*, 1996.

[HPW95] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: an overview. In John Lloyd, editor, *International Logic Programming Symposium*, page 636, Portland, Oregon, December 1995. MIT Press.

[HPW96a] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: A system demonstration. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, page 599. Springer, July 1996.

[HPW96b] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, pages 391–405. Springer, July 1996.

[KY93] Naoki Kobayashi and Akinori Yonezawa. ACL – a concurrent linear logic programming paradigm. In *International Logic Programming Symposium*, pages 279–294, 1993.

[Laf88] Yves Lafont. Introduction to linear logic. Lecture Notes for the Summer School in Constructive Logics and Category Theory, August 1988.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In *Logic in Computer Science*, pages 272–281, 1994.

[Mil95a] Dale Miller. Forum: A multiple-conclusion specification logic (draft). Submitted to *Theory of Computer Science*. Also at `ftp://ftp.cis.upenn.edu/pub/papers/miller/tcs95draft.dvi.Z`, 1995.

[Mil95b] Dale Miller. A survey of linear logic programming. *Computational Logic: The Newsletter of the European Network in Computational Logic*, 2(2):63–67, December 1995.

[PH94] David Pym and James Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[Sce90] Andre Scedrov. A brief guide to linear logic. *Bulletin of the European Association for Theoretical Computer Science*, 41:154–165, June 1990.

[SHC$^+$96] Zoltan Somogyi, Fergus Henderson, Thomas Conway, Andrew Bromage, Tyson Dowd, David Jeffery, Peter Ross, Peter Schachte, and Simon Taylor. Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation technology for (Constraint) Logic Programming Languages*, 1996.

[SHCO95] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard O'Keefe. Logic programming for the real world. In Donald A. Smith, editor, *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*, pages 83–94, 1995.

[Vol94] Paolo Volpe. Concurrent logic programming as uniform linear proofs. In Giorgio Levi and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming*, pages 133–149. Springer-Verlag, September 1994.

[VRD$^+$94] J. Vaghani, K. Ramamohanarao, D.Kemp, Z. Somogyi, P. Stuckey, T. Leask, and J. Harland. The aditi deductive database system. *VLDB Journal*, 3(2):245–288, April 1994.

[War94] David S. Warren. Programming the ptq grammar in xsb. In Raghu Ramakrishna, editor, *Applications of Logic Databases*. Kluwer Academic, 1994.

[WH94] Michael Winikoff and James Harland. Deterministic resource management for the linear logic programming language Lygon. Technical Report 94/23, Melbourne University, 1994.

[WH95a] Michael Winikoff and James Harland. Implementation and development issues for the linear logic programming language Lygon. In *Australasian Computer Science Conference*, pages 563–572, February 1995.

[WH95b] Michael Winikoff and James Harland. Implementing the linear logic programming language Lygon. In John Lloyd, editor, *International Logic Programming Symposium*, pages 66–80, Portland, Oregon, December 1995. MIT Press.

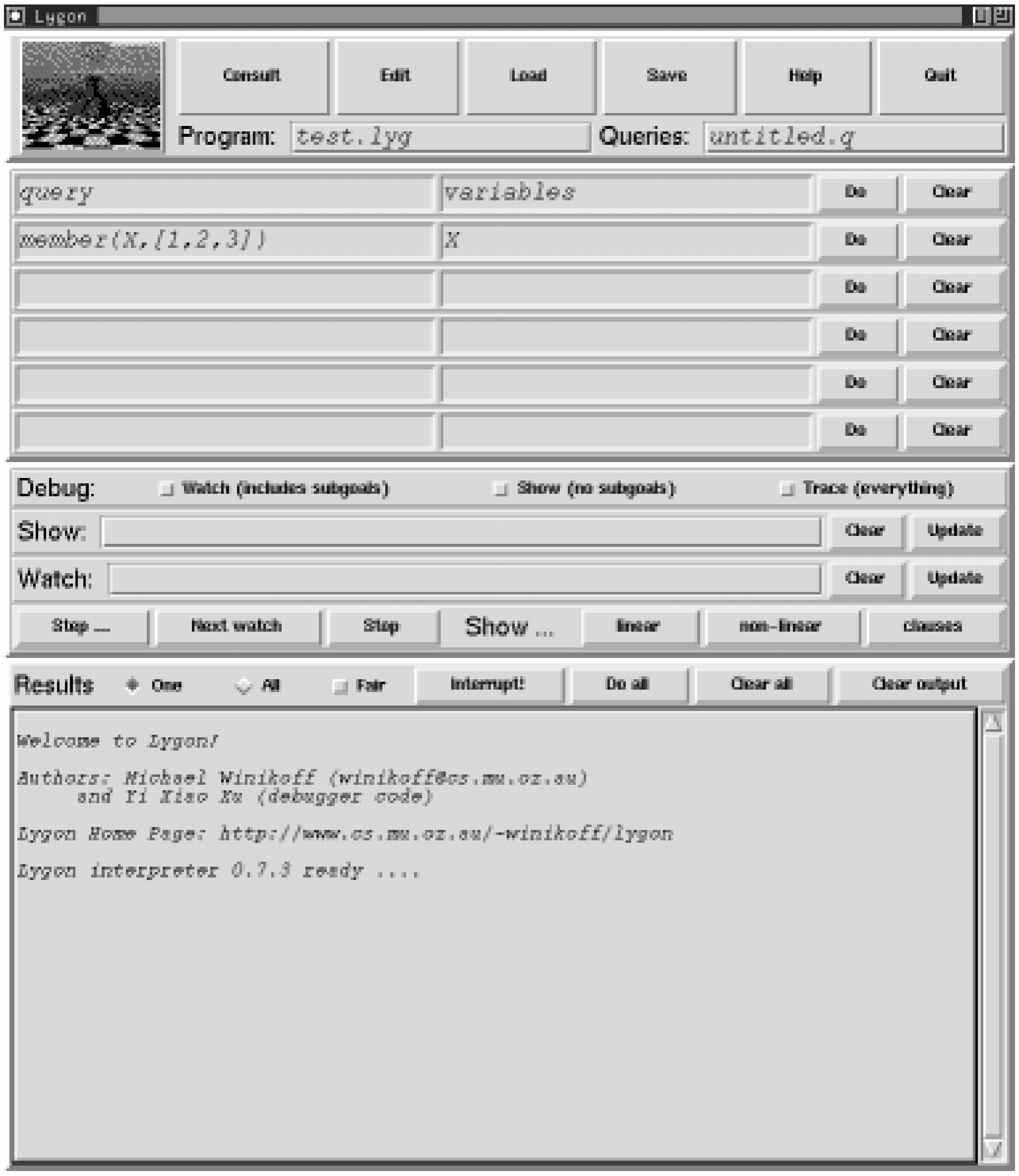
[WH96a] Michael Winikoff and James Harland. Deriving logic programming languages. Technical Report 95/26, Melbourne University, 1996.

[WH96b] Michael Winikoff and James Harland. Some applications of the linear logic programing language Lygon. In Kotagiri Ramamohanarao, editor, *Australasian Computer Science Conference*, pages 262–271, February 1996.

[Win96] Michael Winikoff. Lygon home page. `http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html`, 1996.

Appendix A

Lygon Screendump



Program: test.lyg
Queries: untitled.q

query	variables
member(X,[1,2,3])	X

```
Welcome to Lygon!

Authors: Michael Winikoff (winikoff@cs.mu.oz.au)
     and Yi Xiao Xu (debugger code)

Lygon Home Page: http://www.cs.mu.oz.au/-winikoff/lygon

Lygon interpreter 0.7.3 ready ....
```