# I(2) Programming Languages
# Logic Programming Laboratories

Bernard Sufrin

Version of February 7, 2002

**Abstract**

This document describes the *complete* set of three Labs for the Programming Languages course. The first two labs are identical to those in the earlier revision of this document.
Changes:

1. The syntax of *LogPro* version 1.44 has been extended to accomodate "fat brackets".
2. *LogPro* 1.44 has different primary and secondary prompts (`--` and `>>` respectively)

## 1 First Lab

This Lab is designed to get you acquainted with the *LogPro* system and with Prolog.

1. Read Appendix A to find out how to run the *LogPro* system.
2. Define `append` in the usual way, and

   ```
   isrot(XS, YS) :- append(LS, RS, XS), append(RS, LS, YS).
   ```

   Now find the answers to the query `isrot(1:2:3:nil, YS)`?

   What happens if you make the query `isrot(XS, 1:2:3:nil)`? Can you explain why?[1]
3. In the lectures we partly defined the relation *is* to implement basic arithmetic notation.

   ```
   #infix1 is
   #infix4 +
   #infix5 *
   V is V   :- num(V).
   V is E+F :- Ev is E, Fv is F, sum(Ev, Fv, V).
   V is E*F :- Ev is E, Fv is F, prod(Ev, Fv, V).
   ```

   complete the definition of this relation so that it includes division, subtraction, and remainder.
4. Investigate, and explain,[2] what happens when you try to use `is` "backwards", to solve a query such as

[1] You may not be able to answer this part of the question fully until we have explained the Prolog execution model more fully in lectures, but if you turn on tracing in the *LogPro* interpreter you may be able to describe what is happening in a little more detail, and that will be enough for the moment.

[2] See the previous footnote.

```
3 is V+1?
```

5. Change the definition of `is` so that it works for strings as well as numbers. Overload (+) to signify catenation as well as addition, and use `-` and `/` for suffix and prefix removal.

```
"foobar" is "foo" + "bar"
"foo"    is "foobar" - "bar"
"bar"    is "foobar" / "foo"
```

   Avoid repeated evaluation of the subexpressions of composite expressions if you can.

# 2 Second Lab

A binary tree takes one of the forms

| | |
|---|---|
| `leaf(`$E$`)` | where $E$ is a term |
| $T_1$ ** $T_2$ | where $T_1$, $T_2$ are binary trees |

1. ("Cat-elimination")
   (a) Define a predicate `flatten/2` such that $flatten(T, L)$ holds if $L$ is the list formed by the elements at the leaves of the tree $Y$. For example

```
flatten((leaf(1)**leaf(2))**(leaf(3)**leaf(4)), 1:2:3:4:nil)?
```

   Use the predicate `append/3` to form the list, and investigate (using the directive `#count on`) how many inferences are needed in the best and worst cases when the result list is of length 4, 8, 12, 16, and 20.

   (b) Define a predicate `flat/2` with the same specification as `flatten`, but use open-lists and open-list-append to form the answer list. Investigate the number of inferences needed in the best and worst cases as in the previous exercise.

   What do you conclude from parts (a) and (b) above?

2. Can either `flatten` or `flat` be used "backwards" – in other words to form a tree from a list?

3. Define a predicate `formtree/2` such that $formtree(L, T)$ holds if $T$ is a balanced binary tree formed from the elements of the list $L$, and such that $formtree(L, T)$ holds whenever $flat(T, L)$ holds.

4. Can `formtree/2` be used backwards – in other words to form a list from a tree?

# 3 Third Lab

In this Lab we investigate the use of Prolog to prototype the operational semantics of the Plain language developed in the lectures. With a modicum of advance preparation the two parts can be taken at one sitting.

We take for granted the following *LogPro* definitions of the equality and inequality relations

```
#infix3 "="
#infix3 "/="
X=X         :- .
X/=Y        :- not X=Y.
```

We will represent finite mappings by terms of the form $\{\}$ (for the empty mapping), and $U \mapsto V \ \& \ M$ for the mapping written $M \oplus \{U \mapsto V\}$ in the lecture notes.

## Part 1

The following clauses define the relation $lookup(X, MAP, R)$ which holds between a term $MAP$ (representing a finite mapping $map$), and terms $X$ and $R$ when $X \in \mathbf{dom}\, map \wedge R = map(X)$.[3]

```
#infix6 "|->"
#infix5 "&"
lookup(X, U|->V & MAP, V)      :- X=U.
lookup(X, U|->V & MAP, R)      :- X/=U, lookup(X, MAP, R).
```

### Task

Define a relation $update(MAP, X \mapsto Y, MAP')$ which holds between terms $(MAP, MAP')$ representing mappings $(map, map')$ when $map' = map \oplus \{X \mapsto Y\}$. In addition: if $X$ is already in the domain of $map$ then *the representation of $map'$ should be no larger than that of $map$* (in other words should contain no more terms). For example

```
update(                  {}, a|->2, a|->2 & {})
update(          a|->1 & {}, a|->2, a|->2 & {})
update(b |-> 4 & a|->1 & {}, a|->2, b |-> 4 & a|->2 & {})
```

The additional requirement is present to ensure that while the operational semantics of a program are simulated, the term representing a store contains no more than a single value for each variable in the store.[4]

## Part 2

The inference rules of an operational semantics can often be modelled by Prolog clauses in which the consequent of the rule appears as the head of the clause, and the antecedents as the right hand side. This works satisfactorily only if the side conditions of the rules are encoded as predicates and are used as guards (*i.e.* early) on the right hand side.[5]

Here, for example, is an encoding of a few of the Plain rules as a `LogPro` program.

[3] We have deliberately left out the clause defining *lookup* in an empty mapping so that we can explore the possibility of having "stuck" configurations.

[4] This is essential if humans are to be able to make sense of the output of a simulation.

[5] The notational conventions we use to present the rules in the lecture notes must also be respected by the Prolog program, and the best way of doing so is to treat them as additional side-conditions.

We start with syntactic declarations

```
#infix0 "-|>"
#infix3 ";"
#infix4 ":="
#infix7 "+"
#infix7 "-"
```

Next we define predicates which capture some of the side conditions.

```
id(I)               :- atom(I), I/=skip.
con(E)              :- num(E).
```

Our configurations are encoded as *LogPro* terms in one of the forms[6]

```
[| Program    |] Store
[| Expression |] Store
```

and our judgements as relations of the form

```
CONFIGURATION -|> CONFIGURATION'
```

A configuration is in normal form if its program component is `skip`.

```
normalForm([| skip |] S) :- .
```

We want to show the names of the rules which are used during the execution of a program, so we define the predicate `rule` and end each rule by calling it.

```
#notfix rule
rule S :- show(" (by", S, ")").
```

The first few rules are encoded as follows[7]

```
[|  I       |] S        -|>    [|  V     |] S   :- id(I),
                                                   lookup(I, S, V),
                                                   rule "var".

[|  I:=E    |] S        -|>    [|  skip  |] S'  :- id(I),
                                                   con(E),
                                                   update(S, I|->E, S'),
                                                   rule "assign.2".

[|  I:=E    |] S        -|>    [|  I:=E' |] S   :- id(I), not con(E),
                                                   [| E |] S -|> [| E' |] S,
                                                   rule "assign.1".

[|  P; Q    |] S        -|>    [|  P'; Q |] S'  :- P/=skip,
                                                   [| P |] S -|> [| P' |] S',
                                                   rule "seq.1".

[|  skip; Q |] S        -|>    [|  Q     |] S   :- rule "seq.2".
```

[6] **NB: the "fat bracket" notation is only available in `LogPro` versions 1.44 and later. If you don't presently have access to this, you may encode configurations as `Program | Store` after declaring `#infix0 |`**

[7] The rule for identifiers is not quite the same as that given in the levcture notes, because *lookup* fails when presented with variables which not already in the store. This is deliberate – we want to be able to produce a "stuck" configuration easily.

The `-|>` relation describes only a single step in the (top-level) execution of a program, although this step may involve the use of several structural inference rules in addition to any computation rule it uses.

For example:

```
[| x := 3 ; y := 4 |] {}   -|>  CONFIG' ?
 (by assign.2 )
 (by seq.1 )
CONFIG' = [| skip ; y := 4 |] x |-> 3 & {}
```

The complete program execution relation `-|>*` is encoded as follows:[8]

```
#infix0 "-|>*"
CONFIG  -|>* CONFIG    :- normalForm(CONFIG),
                          show(CONFIG, "  finished.").

CONFIG  -|>* CONFIG''  :- not(normalForm(CONFIG)),
                          show(CONFIG, "  -|>"),
                          CONFIG  -|> CONFIG',
                          CONFIG' -|>* CONFIG''.
```

Notice that we take pains to show the configuration at each stage, and to inform the user when a program has finished. For example

```
...
[| x := 3 ; y := x |] {}   -|>
 (by assign.2 )
 (by seq.1 )
[| skip ; y := x |] x |-> 3 & {}   -|>
 (by seq.2 )
[| y := x |] x |-> 3 & {}   -|>
 (by var )
 (by assign.1 )
[| y := 3 |] x |-> 3 & {}   -|>
 (by assign.2 )
[| skip |] x |-> 3 & y |-> 3 & {}   finished.
```

If the program is just stuck,[9] the "finished" epithet is never applied. For example:

```
...
[| x := 3 ; y := z |] {}   -|>
 (by assign.2 )
 (by seq.1 )
[| skip ; y := z |] x |-> 3 & {}   -|>
 (by seq.2 )
[| y := z |] x |-> 3 & {}   -|>
```

[8] (*For the theoretically inclined:*) Notice that `-|>*` computes a subrelation of the transitive closure of `-|>`. In fact it is only the "stuck" configurations that are not actually reached by execution of `-|>*`, even though the last `show` it executes shows a configuration which is not in the domain of `-|>`.

[9] For the moment the only way this is possible is if a variable is referenced for which there is not yet a value in the store.

**Tasks**

1. Encode the rules for addition so that they prescribe a left-to-right order of evaluation.
2. Define the following "driver" relation

   ```
   #notfix run
   run Program :- [| Program |] {} -|>* Config'
   ```

   and give evidence that the encoding of the rules you gave in part 1 is appropriate by running the following predicates.

   ```
   prob1  :- run x:=3 .
   prob2  :- run x:=3; y:=x .
   prob2a :- run x:=3; y:=z .
   prob3  :- run x:=3; y:=4; z:=x .
   prob4  :- run x:=3; y:=4; z:=x+y .
   ```

3. Change the rules for addition so that they no longer prescribe left-to-right evaluation, and try running `prob4` again. What do you notice?
4. Encode the rules for `after` and gather evidence of their appropriateness by running (with the order-prescriptive rules for addition back in place)

   ```
   prob5 :- run x:=3; y:=4; z:=(x after x:=x+1) .
   ```

   **Hint:** You will need to change a few of the rules so that they permit the evaluation of expressions with side-effects. Make a note of what happens before you change them.
5. Optional tasks (independent of each other)

   (a) (*for the moderately ambitious*) Encode the rules for conditionals, and gather evidence of their appropriateness. You will need to add relational operators to the language. If you are successful in this then encode the iteration rules.

   **Hint:** It is wisest to represent these constructs directly (*i.e.* in their abstract syntax forms, as exemplified below), rather than trying to find syntactic declarations for ***LogPro*** which make it possible to write them in the (admittedly more readable) concrete forms given in the notes.

   `if(`*condition*`,` *TrueProgram*`,` *FalseProgram*`)`
   `while(`*condition*`,` *LoopBody*`)`

   (b) (*for the somewhat persistent*) Try running the following predicate with non-order-prescriptive rules for addition in place

   ```
   prob6 :- run x:=3; y:=4; z:=(x after x:=x+1)+x.
   ```

   Can you explain what is happening?

   Now, if you have time, *and you are more than just somewhat persistent* try the following with the same rules in place

   ```
   prob7 :- run x:=3; y:=4; z:=(x after x:=y)+(y after y:=x).
   ```

   Can you explain what is happening?

# A The LogPro Manual (for version 1.44)

## A.1 Running LogPro

The *LogPro* interpreters are available in the Laboratory filestore as

```
/ecslab/sufrin/logpro/bin/logpro
/ecslab/sufrin/logpro/bin/x86-solaris/logpro
/ecslab/sufrin/logpro/bin/sparc-solaris/logpro
/ecslab/sufrin/logpro/bin/x86-linux/logpro
```

The first of these is a bytecode interpreter, and can be used anywhere there is an installation of ocaml 2.02. The others are binary executables which can be used on a machine with the appropriate architecture and operating system.[10]

Once the appropriate path is added to your PATH variable, the *LogPro* interpreter is invoked with the command

```
logpro [path path ... path]
```

After the commands in the specified files have been read,[11] the intepreter enters interactive mode, initially prompting for each command with `--`,[12] and responding to queries by displaying the substitutions which satisfy them[13] in sequence. After each substitution is displayed, the interpreter prompts with a "`?`": responding to this with "`.`" terminates the search for substitutions; responding with a newline causes the search to continue. At any stage during the evaluation of a query, it can be interrupted by typing the appropriate interrupt character for the operating system on which it is being run.

## A.2 Commands

A *LogPro* command is either a *clause* taking the form

$$formula \texttt{ :- } formula\texttt{, } formula\texttt{, } \cdots\ formula\texttt{.}$$

or a *query* taking one of the (equivalent) forms

$$\texttt{:- } formula\texttt{, } formula\texttt{, } \cdots\ formula\texttt{.}$$
$$formula\texttt{, } formula\texttt{, } \cdots\ formula\texttt{?}$$

or a *directive* taking one of the forms in Table 1.

If there are no formulæ to the right of the entailment symbol `:-` in a clause then that symbol may be omitted.

[10] The Linux version was compiled and runs under a completely standard RedHat 6.1 distribution. I don't know of any reason why it might not run under other Linux distributions.

[11] For each *path*, *LogPro* reads the file *path*.`lp` if it exists, and otherwise reads the file *path*.

[12] If the interpreter requires more input in response to its prompt – perhaps because a query or directive has not yet been terminated properly, then it will prompt again with its secondary prompt: `>>`

[13] Or "`Yes`" if there are no variables in the query

## A.3 Notation

Any text appearing between a `/*` and the subsequent `*/` is treated as a comment and ignored, as is any text appearing between a `--` and the subsequent newline.

A *word* is any consecutive sequence of letters, digits, underscores or primes.

A *symbol* is any consecutive sequence of one or more of the characters "`!^%$@&/\;=><`", "`:+-*/~|`"

The symbol `:-` is the *entailment* symbol. Any other symbol or word may be declared to be an infix, prefix, or notfix operator using one of the directives outlined in Table 1. Unless declared otherwise by such a directive, all symbols behave as left associative infix operators of priority 8.

A *term* has one of the forms described below, whilst a *formula* has any of these but the first three.

1. a (possibly negated) *integer*. For example:

```
34567
-32
```

2. a *double-quoted string*.

3. a *variable* – which is a word beginning with an uppercase letter. For example:

```
X
Rumpelstiltskin
State_Space
```

4. an *atom* – which is a word beginning with a lowercase letter. For example:

```
it_is_raining
david
pruneSquallor
```

5. a *prefix composite* – an atom, followed by a parenthesized, comma-separated, list of *term*s. For example:

```
grandparent(X, Y)
connected(From, To, Via)
simply(purple)
negative(2)
```

6. an *infix composite* – a pair of terms separated by an infix operator. For example, after the directive `#infix3 "divides"`

```
Var + Const
3 divides 62
```

7. a *leftfix composite* – a term followed by a bracketed term. This is regarded as syntactic sugar for a prefix composite with functor `[]`. For example:

`f[Arg]` is sugar for `[](f, Arg)`
`g[x][y]` is sugar for `[]([](g, x), y)`

8. a *rightfix composite* – a term bracketed by the "fat brackets" `[| and |]`. This is regarded as syntactic sugar for a prefix composite with functor `[||]`. For example:

`[|I|] S` is sugar for `[||](I, S)`
`[|I|][|J|]S` is sugar for `[||](I, [||](J, S))`

Leftfix composites have higher priority than rightfix composites. For example:

`[|I|]J[S]` is sugar for `[||](I, [](J, S))`

9. an *outfix composite* -- a curly-bracketed, comma-separated, possibly-empty, list of *term*s. This is regarded as syntactic sugar for a prefix composite with functor {}. For example:

   {} is sugar for `{}()`
   `{foo, bar}` is sugar for `{}(foo, bar)`

10. a *notfix operator* followed by a *term*. The only built-in notfix operator is not, but others may be declared using the #notfix directive (Table 1). *Notfix operator*s have less priority than any other forms of operator. For example

```
not male(X)
not append(X, Y, Z)
not not contradiction(X)
```

    With directives #notfix "let", and #infix3 "in"

```
let x=3 in x+2
```

    yields the formula whose structure is

```
let(in((=)(x, 3), (+)(x, 2)))
```

11. a *prefix operator* followed by a *term*. There are no built-in prefix operators, but they may be declared using the #prefix directive (Table 1). *Prefix operator*s have higher priority than infix operators. For example, after the directives #prefix "factor" and #infix3 "or"

```
factor Y or magic(Y, Z)
```

## A.4 Built-In Relations

The relations and symbols built-in to *LogPro* are described in Table 2.

## A.5 Changes

V1.44 (a) The syntax of terms has been extended to accomodate "fat brackets".
(b) Different primary and secondary prompts (-- and >> respectively)

| Command | | Effect |
|---|---|---|
| `#use` | *path* | Read the file specified by the given path (no more than once per session) |
| `#width` | *number* | Set the output width to *number* columns |
| `#infix0` | "*symbol*" | Declare the symbol to be a right-associative infix operator, priority 0 |
| `#infix1` | "*symbol*" | Declare the symbol to be a right-associative infix operator, priority 1 |
| ... | | |
| `#infix7` | "*symbol*" | Declare the symbol to be a right-associative infix operator, priority 7 |
| `#infix8` | "*symbol*" | Declare the symbol to be a *left-associative* infix operator, priority 8 |
| `#prefix` | "*symbol*" | Declare the symbol to be a prefix operator with maximal priority |
| `#notfix` | "*symbol*" | Declare the symbol to be a prefix operator with minimal priority |
| `#trace` | on | Trace the invocation of each predicate |
| `#trace` | all | As above and show the complete current substitution at each invocation |
| `#trace` | off | Stop tracing |
| `#check` | on *or* off | Enable or disable the occurs check in unification |
| `#count` | on *or* off | Enable or disable the printing of inference count with answers |

Table 1: Directives

| Relation | Arguments | Effect (Restriction) |
|---|---|---|
| `!` | | the cut symbol |
| `fail` | | always fails |
| `sum` | $(A, B, C)$ | $C$ is the sum of $A$ and $B$. (No more than one variable) |
| `prod` | $(A, B, C)$ | $C$ is the product of $A$ and $B$. (No more than one variable) |
| `succ` | $(A, B)$ | $B$ is the natural number successor to $A$. (No more than one variable) |
| `>=` | $(A, B)$ | $A \geq B$. (No variables). |
| `>` | $(A, B)$ | $A > B$. (No variables). |
| `<=` | $(A, B)$ | $A \leq B$. (No variables). |
| `<` | $(A, B)$ | $A < B$. (No variables). |
| `num` | $(A)$ | $A$ is an integer. |
| `str` | $(A)$ | $A$ is a string. |
| `atom` | $(A)$ | $A$ is an atom. |
| `var` | $(A)$ | $A$ is an uninstantiated variable. |
| `nonvar` | $(A)$ | $A$ is not an uninstantiated variable. |
| `toString` | $(A, B)$ | $B$ is the string representing the term $A$. |
| `toFormula` | $(A, B)$ | $B$ is the formula represented by the string $A$. |
| `functor` | $(A, OP, B)$ | $A$ is the formula with principal connective $OP$ and list of arguments $B$. |
| `call` | $(A, B, ...)$ | The formulae $A, B, ...$ are evaluated in sequence. |
| `len` | $(A, B)$ | $B$ is the length of the string $A$. |
| `cat` | $(A, B, C)$ | $C$ is the catenation of the strings $A$ and $B$ (No more than one variable). |
| `hd` | $(A, B)$ | $B$ is a string consisting of the first character of the string $A$ (ditto). |
| `tl` | $(A, B)$ | $B$ consists of all but the first character of the string $A$ (ditto). |
| `ascii` | $(A, B)$ | $B$ is the ascii code of the first character of the string $A$ (ditto). |
| `read` | $(A)$ | The string $A$ is the next line read from the terminal. |
| `show` | $(A, B, ...)$ | The terms $A, B, ...$ are printed on the terminal followed by a newline. |
| `write` | $(A, B, ...)$ | The terms $A, B, ...$ are printed on the terminal. |
| The following relations manipulate the built-in updateable mapping from terms to terms | | |
| `map_Lookup` | $(A, B)$ | $B$ is the current value of the mapping at key $A$. |
| `map_Add` | $(A, B)$ | The value of the mapping at key $A$ is overlaid by $B$. |
| `map_Enter` | $(A, B)$ | The value of the mapping at key $A$ becomes $B$; previous values removed. |
| `map_Remove` | $(A)$ | Remove the topmost value of the mapping at key $A$. |
| `map_All` | $(A, B)$ | $B$ is the list of values of the mapping at key $A$. |

Table 2: Built-in Relations

# Contents

**1 First Lab** **1**

**2 Second Lab** **2**

**3 Third Lab** **2**

**A The `LogPro` Manual (for version 1.44)** **7**

A.1 Running *`LogPro`* . . . . . 7

A.2 Commands . . . . . 7

A.3 Notation . . . . . 8

A.4 Built-In Relations . . . . . 9

A.5 Changes . . . . . 9

Typeset on August 14, 2019