

# Math 187 Logic Lab Manual and Exercises

Dr. Holmes

January 31, 2010

## 1 Setup

Open a terminal window. In the terminal window, type

```
cp /home/public/holmes/math187/marcel.sml .
```

The period is really there, it is part of the command. This command copies the marcel program source into your local directory.

Now, in the terminal window type

```
mosml
```

This command opens the Moscow ML command interpreter. This is an implementation of the Standard ML programming language. By the way, if you are interested in getting Moscow ML and/or the Marcel theorem prover on your own computer, you can download both – instructions are found on the Marcel page, a link to which is on my home page. I have had undergraduate and graduate students do research projects with this and a previous theorem prover.

The next commands are not to the operating system but to the ML interpreter.

```
compile "marcel.sml";  
load "marcel";  
open marcel;  
  
OneConclusion();
```

The first command compiles Marcel, the logic software we are using (which I wrote as part of my research). The second and third commands open Marcel so we can use it. The last command is actually a Marcel command, which makes the way in which the logic is displayed a bit more friendly to beginners. On second and subsequent times that you use the prover, you do not need to type the `compile` line, though it is not harmful to do so.

You are now sitting at an ML interpreter window, with the commands of the Marcel interpreter loaded as functions. You are ready to do logic.

## 2 Brief Reference

### 2.1 Format of ML command lines

Commands to the Marcel prover are actually calls to ML functions. An ML command line begins with a function name and is followed by one or more arguments, and must end with a semicolon. A command with no arguments always has a dummy argument `()` (a pair of parentheses).

A very useful command is `quit()`; which exits the ML interpreter.

### 2.2 Common Prover Commands

The basic commands of the prover which you will use or see used are

```
Start <string>; (* start proving the expression denoted by the string;  
the string needs to be in quotes *)
```

Start can be abbreviated `s`.

```
r(); (* apply a proof strategy to the conclusion of the argument
```

```

        we are looking at *)

l(); (* apply a proof strategy to the first premise of the argument
      we are looking at *)

gl <number>; (* move the premise with the indicated number to the
             beginning of the list of premises *)

Done(); (* recognize that the first premise and the conclusion are the
        same, which means the argument is valid! *)

startlogging <filename>; (* the filename is a string and so must be in quotes:
                        opens a file named {\tt <filename>.mlg} and starts logging all proven

stoplogging(); (* closes the log file *)

showall(); (* shows a human-readable though very boring proof of a theorem
           just proved; it will show partial proofs if you run it before
           your proof is done. You need to hit enter repeatedly to see
           the whole thing *)

LogTheProof(); (* sends the entire output of showall(); [without pausing]
              to the log file *)

```

### 2.3 Marcel's logical notation

The notation of Marcel for logical operations is dictated by the limitations of ASCII, and the fact that the Marcel parser is simple. Propositional letters are of the form P1, P2, P3...: P followed by a number.

$\sim$  P1 is NOT P1

P1 & P2 is P1 AND P2

P1 v P2 is P1 OR P2

P1 -> P2 is IF P1 THEN P2

P1 == P2 is P1 IFF P2

there are a couple of extras

P1 <- P2 is P1 IF P2

P1 /= P2 is P1 XOR P2 (exclusive or)

There is built-in order of operations. NOT binds most tightly, followed by AND, followed by OR, followed by IF...THEN (and IF), followed by IFF (and XOR). This is the same order of operations which is standard for these operators.

## 2.4 Arguments

The Marcel display shows you an **argument**, a list of premises (assumptions) from which you are trying to show that a conclusion follows logically. An argument is written linearly in the form A1, A2, A3... |- C. It is displayed vertically by Marcel:

1. A1

2. A2

3. A3

...

|-

1. C

An argument is said to be *valid* if any assignment of meanings to the variables in the argument which makes all the premises (assumptions) true

also makes the conclusion true. In the context we are working in at first, the only variables are propositional variables (letters to be assigned the values “true” or “false”).

Marcel works by simplifying arguments. The ways in which it simplifies arguments correspond to natural ways to reason about sentences of these logical forms (which is the point of this exercise). The strategies being implemented in symbols and computer code here are identical to or very similar to the usual strategies for structuring arguments in mathematics.

For example, an argument  $A1, A2, A3\dots \vdash C \rightarrow D$  simplifies to the argument  $C, A1, A2, A3\dots \vdash D$ : this expresses our strategy for proving implications exactly (add the hypothesis of the implication as a new assumption and prove the conclusion).

$D \ \& \ E, A2, A3\dots \vdash C$  simplifies to  $D, E, A2, A3\dots \vdash C$ . This is really trivial (we have done it in actual arguments and not even called attention to it particularly). A conjunction of assumptions acts just like a list of assumptions.

$A1, A2, A3\dots \vdash C \ \& \ D$  simplifies to *two* simpler arguments, as there are two things to prove:  $A1, A2, A3\dots \vdash C$  and  $A1, A2, A3\dots \vdash D$ . This is also very simple and natural: to prove a conjunction (from assumptions) prove each of the parts of the conjunction separately.

The rules for OR are a bit trickier.

$D \vee E, A2, A3\dots \vdash C$  is an argument in which one of the hypotheses is a disjunction:  $D$  is true *or*  $E$  is true. This is the time to use the strategy of *proof by cases*: this argument breaks down into the two simpler arguments  $D, A2, A3\dots \vdash C$  and  $E, A2, A3\dots \vdash C$ .

$A1, A2, A3\dots \vdash C \vee D$  simplifies to  $\sim D, A1, A2, A3, \dots \vdash C$ . To show that  $C$  is true OR  $D$  is true is the same as to show that if  $D$  is false,  $C$  must be true ( $C \vee D$  is equivalent to  $\sim D \rightarrow C$  – you can show this easily with a truth table). It is equally true that it could be simplified to  $\sim D, A1, A2, A3, \dots \vdash C$ , but that is not what Marcel does (and it is easy to convert to that form as well).

The rules for NOT require us to introduce a variation on our notation.

$A1, A2, A3\dots \vdash \sim C$  simplifies to  $A1, A2, A3, C \vdash \perp$ , which means that from the assumptions  $A1, A2, A3, \dots, C$  we can deduce a contradiction. Arguments to contradiction can be simplified in basically the same ways as arguments with explicit conclusions.

$\sim D, A1, A2, A3\dots \vdash C$  transforms to  $\sim C, A1, A2, A3\dots \vdash D$  (because  $\sim D \rightarrow C$  is logically equivalent to  $\sim C \rightarrow D$ , a special case of ar-

gument by contrapositive.)  $\sim D, A1, A2, A3 \dots \vdash \sim C$  simplifies to  $C, A1, A2, A3 \dots \vdash D$ , by an additional application of double negation. or simply by the rule of contrapositive:  $\sim D \rightarrow \sim C$  is equivalent to  $C \rightarrow D$ .

The rules for IFF follow easily for those from implication and conjunction, and correspond to the ways we would prove an IFF statement or use one:

$A1, A2, A3 \dots \vdash C == D$  simplifies to  $C, A1, A2, A3 \vdash D$  and  $D, A1, A2, A3 \vdash C$ : this looks exactly like the strategy for proving IFF statements which already appears in the book.

$D == E, A2, A3 \dots \vdash C$  simplifies to  $D \rightarrow E, E \rightarrow D, A1, A2, A3, \dots \vdash C$ : this relies on the equivalence of  $D == E$  with  $(D \rightarrow E) \ \& \ (E \rightarrow D)$ .

An important consideration is how to finish an argument.

$A1, A2, A3 \dots \vdash A1$  is valid! If the conclusion is one of the premises we are done. Further, we can freely rearrange the order of the assumptions (this is important when we want to apply a rule to an assumption other than the first one).

We have one remaining item of business. This is the treatment of implications as assumptions. We simplify  $D \rightarrow E, A2, A3 \dots \vdash C$  to  $A2, A3, \dots, \sim C \vdash D$  and  $E, A2, A3, \dots \vdash C$ . This requires explanation even for those of you with some experience with logic. The rule which would usually be given here is **modus ponens**: from  $A$  and  $A \rightarrow B$  deduce  $B$ . So, the way to use a hypothesis  $A \rightarrow B$  is to deduce  $D$  (which we do, with a refinement, in the first argument), and then (since we then know that  $E$  follows by modus ponens), show that  $C$  follows from  $E$  and the other assumptions. The refinement is that we only need to prove  $D$  if the conclusion  $C$  is supposed false: if  $C$  is true we have nothing to prove: that is the reason that  $\sim C$  is added as an assumption in the first argument.

To prove  $A$  is to show that  $\vdash A$  is a valid argument. The command `Start <theorem to be proved>` sets up an argument  $\vdash$  <theorem to be proved> which we then attempt to prove.

### 3 Examples

- Start "P1→P1";

Line number 1:

```
|-  
1: P1 -> P1  
  
> val it = () : unit
```

We prove a very simple statement. The `> val it = () : unit` line is chatter from the ML interpreter which you should ignore. Notice that Marcel set up an argument with no assumptions and the desired theorem as a conclusion.

```
- r();  
  
Line number 2:  
  
1: P1  
  
|-  
  
1: P1  
  
> val it = () : unit
```

The `r();` command applies the appropriate logical strategy to the conclusion of an argument. In this case the conclusion is an implication (an if-then statement) and we apply the strategy we have already learned for proving an implication. The new argument is obviously a valid one.

```
- Done();  
  
Q. E. D.  
> val it = () : unit
```

The `Done()` ; command tells the computer to check for the condition that the first assumption is the same as the conclusion, under which the argument is of course valid. We could easily have the program do this automatically (and there are various other things we could have it do automatically) but our design philosophy in writing Marcel is that we are writing a proof checker: the user does the proving.

The message `Q. E. D.` tells us that we are done. This was requested by students using the prover. Originally Marcel just shut up when it was done proving a theorem.

The theorem we prove next is also obvious. It lets us see another couple of features.

```
- s "(P1&P2)->P2";
```

```
Line number 1:
```

```
|-
```

```
1: P1 & P2 -> P2
```

```
> val it = () : unit
```

```
- r();
```

```
Line number 2:
```

```
1: P1 & P2
```

```
|-
```

```
1: P2
```

```
> val it = () : unit
```

We apply the familiar strategy for proving an implication (an if-then statement).

```
- l();
```

```
Line number 3:
```

```
1: P1
```

```
2: P2
```

```
|-
```

```
1: P2
```

```
> val it = () : unit
```

The `l();` command applies the appropriate logical strategy to the first assumption. In this case the logical strategy (which we *have* used in chalk proofs on the board) is so obvious as to be invisible: to assume `A & B` (a single assumption) is the same thing as to assume `A` and assume `B` (two assumptions).

The argument above is obviously valid. The conclusion is one of the assumptions!

```
- gl 2;
```

```
Line number 3:
```

```
1: P2
```

```
2: P1
```

```
|-
```

```
1: P2
```

```
> val it = () : unit
```

Since Marcel acts (for simplicity) on the first assumption, we need to be able to reorder assumptions when we want to act on a different one. The command `gl n` moves the  $n$ th assumption to the first position (moving assumptions above the  $n$ th to the bottom).

Now we can finish.

```
- Done();
```

Q. E. D.

```
> val it = () : unit
```

As above, we are done.

Now we give the big example we did in class, though we are really only going to run through half of the proof (the same general approach will work for the second half and indeed for any proof in propositional logic).

```
- s "(P1&P2)->P3 == P1->(P2->P3)";
```

Line number 1:

```
|-
```

```
1: P1 & P2 -> P3 == P1  
   -> P2 -> P3
```

```
> val it = () : unit
```

Notice that Marcel drops parentheses that it does not need in order to understand the expression. Everything groups to the right ( $A \rightarrow B \rightarrow C$  means  $A \rightarrow (B \rightarrow C)$ ), and the order of operations described above is used.

```
- r();
```

Line number 2:

```
1: P1 & P2 -> P3
```

```
|-
```

```
1: P1 -> P2 -> P3
```

```
> val it = () : unit
```

The usual strategy for implication.

```
- r();
```

Line number 4:

```
1: P1
```

```
2: P1 & P2 -> P3
```

```
|-
```

```
1: P2 -> P3
```

```
> val it = () : unit
```

The usual strategy for implication, again.

```
- r();
```

Line number 5:

```
1: P2
```

```
2: P1
3: P1 & P2 -> P3
|-
1: P3
> val it = () : unit
```

Notice that the only interesting assumption or conclusion is the third assumption...

```
- gl 3;
Line number 5:
1: P1 & P2 -> P3
2: P2
3: P1
|-
1: P3
> val it = () : unit
```

...so we bring it to the front with the `gl 3` command.

```
- l();
Line number 6:
```

1: P2

2: P1

3:  $\sim P3$

|-

1: P1 & P2

Remember that our rule for a left assumption  $A \rightarrow B$  generates two arguments, one that proves  $A$  (the immediately preceding line 6) [assuming that the original conclusion is false] and one that adds  $B$  as an assumption and proves the original conclusion. This appears here as line 7 far below: Marcel remembers line 7 and serves it up to us when we have finished showing that line 6 is valid.

We now have a conjunction in the conclusion. To prove a conjunction, we have to prove both of the statements linked by AND. So there will be two separate arguments to show valid (line 8 and line 9 below).

```
> val it = () : unit
```

```
- r();
```

Line number 8:

1: P2

2: P1

3:  $\sim P3$

|-

1: P1

```
> val it = () : unit
```

Here we need to bring the second assumption to the front and observe that we are finished. We do this on the next line. Notice we can type two Marcel commands on the same line.

```
- gl 2; Done();
```

```
Line number 8:
```

```
1: P1
```

```
2: ~P3
```

```
3: P2
```

```
|-
```

```
1: P1
```

```
> val it = () : unit
```

The *Done()* command finishes showing that line 8 is valid, but we are not done with the proof: we still have line 9 to do (and indeed we have a couple more hanging agenda items which will appear). So we do not get a QED, we get the next agenda item:

```
-
```

```
Line number 9:
```

```
1: P2
2: P1
3: ~P3
|-
1: P2
> val it = () : unit
- Done();
```

This shows the other half of the conjunction. It is immediate. We then finish the proof of line 5 (introducing the other goal generated by the implication assumption there).

Line number 7:

```
1: P3
2: P2
3: P1
|-
1: P3
> val it = () : unit
```

This is the other goal generated by the implication assumption above. It is obvious and once we are done with it we are served with the second part

of the main proof.

```
- Done();
```

Line number 3:

```
1: P1 -> P2 -> P3
```

```
|-
```

```
1: P1 & P2 -> P3
```

```
> val it = () : unit
```

We do not present the other half of this proof here.

## 4 Printed Proofs and Command Logs

If you type `Showall()`; when you are done with your proof, and keep hitting return, you will see a proof which a human being might be able to read, if they were very patient. Take a look at these proofs.

You will need to record your work (so you have something to turn in as evidence that you have completed your lab). The command

```
startlogging "filename";
```

will open a log file named `filename.mlg` (you don't really want to call it "filename", do you? You can use any name).

The prover will then start recording all the commands you type to it in the log file.

When you have completed your proof, type

```
LogTheProof();
```

This will insert the “human-readable” proof that `showall()`; shows you into the log file.

Then type `stoplogging()`; . This will close the log file. I don’t recommend trying to edit or view the log file before that point. The log file `filename.mlg` is a text file, containing the commands you issued to the prover and the “human readable” proof as a comment at the end. If you pasted your log file into the ML window, it would run your proof again: it is designed to be executable by the interpreter. It is useful in a serious prover project for saving your work when you are part of the way through a proof as well.

## 5 Exercises

Prove the following tautologies. Follow the outline above to make sure you save a log of your commands to a file and embed the human-readable proof in it. E-mail the files to me when you are done. Use a different file name for each proof so you don’t overwrite your work!

1.  $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$
2.  $(P \rightarrow Q) \wedge (Q \rightarrow R) \rightarrow (P \rightarrow R)$  [watching carefully what happens in this proof might help you understand what is going on with implications as assumptions].
3.  $(P \vee Q) \rightarrow R \leftrightarrow (P \rightarrow R) \wedge (Q \rightarrow R)$

Be aware that it is a theorem that if you keep applying prover commands (except for cycling on negations in an obviously circular way) you *will* arrive at a proof: unless the statement you are trying to prove is not a tautology, in which case you will arrive at a counterexample! I’ll give an example to support that last remark.

The complete manual and command reference for Marcel is found at <http://math.boisestate.edu/holmes/marcelstuff/marcelmanual.pdf>; I cannot imagine why you would need it, but there it is.