ETPS User's Manual

2010 February 12

Frank Pfenning
Sunil Issar
Dan Nesmith
Peter B. Andrews
Hongwei Xi
Matthew Bishop
Chad E. Brown

Version for
Basic Logic
Mathematical Logic I and II

This manual is based upon work supported by NSF grants MCS81-02870, DCR-8402532, CCR-8702699, CCR-9002546, CCR-9201893, CCR-9502878, CCR-9624683, CCR-9732312, CCR-0097179, and a grant from the Center for Design of Educational Computing, Carnegie Mellon University. Any opinions, findings, and conclusions or recommendations are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1. ETPS User Interface

1.1. Introduction

You may find the Table of Contents at the end of this manual.

ETPS is a program which is designed to help you do your logic homework. It was developed from **TPS**, an ongoing research project in automated theorem proving. It is written in Common Lisp.

To do your homework on the computer, first enter **ETPS**. It is highly recommended that you run **ETPS** in an X-window (on a Unix or Linux system) or use the Java interface for **ETPS** (if these facilities are available on your system), so that formulas can be printed on the screen using special fonts which contain logical symbols, and so that the proof you are developing can be printed and updated in special proofwindows as you work.

You can start the Java interface for **ETPS** using the `JAVAWIN` command; see Section 2.3. If you are running **ETPS** in an X-window equipped with the special fonts used to print logical symbols, you need to tell **ETPS** that you want to use the special fonts for output. At the **ETPS** prompt, issue the command

```
setflag style
```

then, at the subsequent prompt

```
xterm
```

This will cause the special symbols to appear when any wffs are printed by **ETPS**. (If for some reason the special symbols won't print properly, just change the style back from `xterm` to `generic`.)

If you are running **ETPS** in an X-window or using the Java interface, you will probably also wish to use the `BEGIN-PRFW` command to start up windows containing the current subproof and the complete proof; see Section 2.4. You may need to iconify a window or move it up on your screen by the usual methods for manipulating windows so that you will have room to issue **ETPS** commands. You can eventually close the proofwindows with the `END-PRFW` comand. See Section 2.5 for commands which control what is regarded as the current subproof.

To help you learn how to use **ETPS**, transcripts of sample sessions are provided in Chapter 4. Just follow these examples exactly on your own computer, and you will soon have a general idea of what to do. You can also do some practice (unassigned) exercises (which you can find with the aid of the `PROBLEMS` command), and make frequent use of the `ADVICE` command, which will offer suggestions about what to do next.

You should note that the only means of identification available to **ETPS** is the *userid* of the account from which it is run. It will credit all work to the owner of that account *and to no other user*. Thus, in order to receive credit for an exercise, you *must* run **ETPS** from *your own* account. Run it from a private directory so that the files which **ETPS** creates containing your proofs will not be accessible to others.

Start work on an exercise with the command `EXERCISE` *exercise-name*; see Section 2.2. Next construct a complete proof using the inference rules described in Chapter 3. However, some of the more powerful inference rules may not be allowed for certain exercises. To find out which rules are prohibited, you should invoke the `HELP` command on the exercise. If you cannot figure out what the next step in a proof should be, you may get hints by using the `ADVICE` command; but beware: these hints are not always helpful and can be misleading.

A partially completed proof will be called a *proof outline* or simply an *outline*. When you start proving a theorem with the `EXERCISE` or `PROVE` command, **ETPS** will create an outline for you which contains a single line: the theorem you would like to prove. It is not yet justified since you are only planning to prove it. In place of the justification there will be the word `PLAN1`. This last line of the outline is therefore called a *planned* line. Lines which are justified can be introduced by justifying a planned line, introducing a hypothesis, or deriving consequences of already justified lines. Proofs may be built down from the top, adding consequences of established lines, or up from the bottom, justifying planned lines by asserting new planned lines. It is a good idea to work up from the bottom as much as possible, so that **ETPS** will already know about most of the wffs you need, and you will not have to type them in.

ETPS was originally developed for use with the textbook **An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof** by Peter B. Andrews. A second edition was published by Kluwer Academic Publishers in 2002, and is now available from Springer, which which has taken over Kluwer. Exercises from this textbook are available in **ETPS**. However, the inference commands available in your version of **ETPS** may depend on the particular logical system chosen by your teacher. Inference commands are specified as inference rules and are listed in Chapter 3. Ideally, after repeatedly applying rules to planned and deduced lines, they will ‘‘meet’’ and every planned line will be justified. The examples in Chapter 4 should make all this clearer.

When you have finished the proof, issue the `DONE` command. This will record the completion of the exercise *if* you used the `EXERCISE` command. The `PROVE` command lets you prove arbitrary wffs, *but it will not give you credit for any exercises to which these wffs correspond.* **ETPS** sends only a partial record of your session to a file on the teacher’s area. You must submit a printed copy of the finished proof, as well. To make this copy, first print the proof into a file, using the `TEXPROOF` command. When you have exited from **ETPS**, run this file through `TEX`. Then print the press file generated by `TEX` and hand this output to your teacher.

A word of advice to the user: **ETPS** is intended to aid you in learning logic, but if you use it thoughtlessly, you might be able to do exercises without learning much. **ETPS** does not allow you to use the rules of logic in an incorrect manner, but it’s important that you learn for yourself how to apply the rules correctly. By allowing only correct applications of the rules, **ETPS** encourages the user to spend more time learning the techniques of proving theorems in logic. It is strongly recommended that when **ETPS** does not allow you to do something, you think about why what you tried was incorrect.

1.2. Saving and Restoring Your Work

ETPS saves your work automatically as you try to prove a theorem. This facility is very similar to Emacs’ auto-save feature. **ETPS** commands `SAVE-WORK`, `STOP-SAVE`, `RESUME-WORK` and `RESUME-SAVE` allow you to explicitly use this feature. **ETPS** also provides the commands `SAVEPROOF` and `RESTOREPROOF` for saving your proofs. Unlike the automatic saving facility, which saves everything typed by the user (and later re-executes every command in the file), this feature only saves a proof (and later restores this proof, thus avoiding the re-execution of everything that was done to achieve this state). It is, however, up to you to decide when to save a proof. Although the auto saving feature starts to save your work whenever you start an exercise, you need to explicitly use the command `SAVEPROOF` when you wish to save a proof. Typically, you will want to save the proof whenever you need to interrupt a session in which you are constructing a proof and wish to continue this proof later. **ETPS** commands associated with saving work are described in Section 2.6.

As soon as you start an exercise (but not a proof of your own theorem), **ETPS** will save your commands with the appropriate arguments in a file. The name of this file will be *exercise*`.work` where *exercise* is the first exercise attempted in your current session with **ETPS**. When **ETPS** is saving work, it echoes every character you type in a stream obtained by opening the save-work file. The echo stream is closed only when the user issues the command `STOP-SAVE` or the command `EXIT` (to exit **ETPS** in the usual way). The save-work file is not available until **ETPS** closes this stream.

One more caution: When starting the same exercise in two different sessions, the *same filename* will be used. The work for the new attempt will overwrite the old file *exercise*`.work`. To save the old attempt, rename it before restarting **ETPS**.

After your work has been restored by `RESTORE-WORK`, **ETPS** will continue to save subsequent work into the same file by appending it to the end. If you would like to prevent that, give the `STOP-SAVE` command right after `RESTORE-WORK`; better yet, use the `EXECUTE-FILE` command.

When commands are read from a save-work file, most errors such as illegal arguments or logical mistakes are faithfully re-executed, since some of them have side-effects. Only Lisp errors will lead to an abort of the `RESTORE-WORK` and the state of the proof will be the same as after the last correct command read from the file.

You may edit the save-work file in Emacs to delete wrong commands or correct illegal arguments, but you'll be skating on thin ice. It's easy to make a mistake in editing the save-work file and you may not be able to recover the proof you wanted to restore! The inquisitive user may note that lines beginning with a semi-colon are ignored when the proof is being restored.

There are several options you have when using this auto-save feature. You may switch off saving with the `STOP-SAVE` command. Also, you can explicitly save into a different file with the `SAVE-WORK` command. To check whether your work is being saved, use the system command `PSTATUS`.

You also have the option of keeping a complete record of the session, including the system responses, in a file. If you want to prepare such a copy, issue the **ETPS** command `SCRIPT`. Note, however, that the file obtained this way cannot be used to restore your work as described earlier in this section.

1.3. Exiting and Reentering ETPS

If you wish to end a session with **ETPS** or temporarily interrupt your work, use the `EXIT` command. This allows **ETPS** to halt gracefully and to close an open save-work file.

If you are running **ETPS** under Unix and wish to interrupt a session temporarily, you can also use `Ctrl-Z`. This will return you immediately to the monitor; if you are currently saving work, the save-work file **will not be closed**. Thus any work will be lost unless you return to **ETPS**. Once out of **ETPS**, you can run other programs, such as `TEX`. To reeneter **ETPS**, use the Unix command `fg`.

1.4. Top-level Interaction

In **ETPS** every command line is identified with a statement number, shown in angle brackets. After a while, command line numbers are reused, i.e., commands issued long ago are forgotten.

The following is a list of useful control characters and their meaning in **ETPS**. Some of them may not work under certain operating systems or display environments. Additional control characters are discussed in Section I.1.

`<Rubout>`	Delete the last character and back over it.
`Ctrl-C`	Quit current command and return to top-level. (In some implementations of **ETPS** you must use `Ctrl-G` instead.) Using this command may cause problems for any save-work file that is being created, so it may be better to use ABORT.
`Ctrl-U`	Delete current input line.
`Ctrl-W`	Delete last input word.

The next three characters are special on the top-level only and are currently not to be used when typing in arguments to commands.

`@` Complete the name of the current command. If COMPLETION-OPTIONS is NIL, this works only if there is a unique completion; if it is T, you will be offered a list of completions to choose from.

`<Escape>` Exactly equivalent to `@`. This character can confuse some terminals; we recommend using `@` instead.

`?` When typed at the top-level, **ETPS** will list all the available commands. Note that `?` must be followed by a `<Return>`.

`<Linefeed>`
This starts a new line on the terminal without terminating the input. This is useful for long command arguments.

1.5. Using Commands and Defining Wffs

The commands available in **ETPS** are classified into system commands and inference commands. The system commands, which are discussed in Chapter 2, deal with communication facilities, starting and finishing a session with **ETPS**, the Java interface, proofwindows, the current subproof, saving work, printing, rearranging the proof, getting assistance, displaying types in higher-order logic, setting flags, and locating and constructing wffs. The inference commands, which are discussed in Chapter 3, correspond to inference rules. They transform one proof outline into another by applying the rules.

Common to all commands is the way they are invoked. Simply type the name of the command (you may use `<Esc>` to complete its name) and then type `<Return>`. The command may be printed in either upper or lower case. If the command takes arguments, **ETPS** will prompt you in succession for each argument. The prompt takes the general form

argument name (*argument type*) : *help* [*default*]>

```
For example:

<1>PROVE
WFF (GWFF0): Prove wff [No Default]>
PREFIX (SYMBOL): Name of the proof [No Default]>
NUM (LINE): Line number for theorem [100]>
```

You may also specify command arguments on the command line itself. **ETPS** will then prompt you only for the arguments you haven’t specified. This is a useful option for commands like `PL 2 50`, which directly prints every line in the range from `2` to `50`.

After **ETPS** issues a prompt, you have the following options for your reply besides typing in the argument.

`<Return>` This selects the default, if there is one.

`!` This selects the default, not only for this argument, but all remaining arguments. This can also be used on the command line itself.

`?` This gives help about the type of the argument **ETPS** is waiting for.

`??` This gives help about the command for which you are supplying arguments. In particular, when applying an inference rule, this will give you the statement of the rule.

`ABORT` Aborts current command.

`PUSH` Temporarily suspend whatever you’re doing and start a new top-level. The command `POP` will return you to the point at which you typed `PUSH`.

`Ctrl-G<Return> or Ctrl-C`
To abort. This is a system-dependent feature, and one or the other, or both, may not work on your

system. Using this command may cause problems for any save-work file that is being created, so it may be better to use ABORT.

If a mistake in an argument can be detected right away, **ETPS** will complain and let you try again. Sometimes **ETPS** will note that something is wrong after all the arguments are typed in. You will then get an error message and be thrown back to the top-level of **ETPS**.

The *argument name* is usually only important when typing in arguments to inference commands. If you are in doubt what wff you are supposed to type in now, look at the description of the inference rule. The name of the argument will be the same as the name of a wff in the rule description.

The *argument type* tells you what kind of object **ETPS** expects. The most important argument types are listed below. (You may omit most of the rest of this section when first reading this manual. However, be sure to read the description of GWFF.)

`ANYTHING` Any legal LISP object.

`SYMBOL` Any legal LISP symbol.

`BOOLEAN` A Boolean value (NIL for false, T for true).

`YESNO` `y` or `yes` to mean `YES`, `n` or `no` to mean `NO`.

`INTEGER+` A nonnegative integer.

`POSINTEGER`
A positive integer .

`STRING` A string delimited by double-quotes. For example `"This is a remark."`. Strings may contain `<Return>`, but no double-quotes.

`FILESPEC` A file specification of the form `"`*<dir>name*`.`*ext*`"`, `"`*name*`.`*ext*`"`, or simply *name*. For example:`"<FP01>EXAMPLE1.MSS"`. The defaults for *dir* and *ext* are usually correct and it is enough to specify *name*.

`LINE` The number of a line.

`LINE-RANGE`
A range of lines from M through N, written M--N, where M and N are positive integers and M <= N. As shortcuts, one may write M, which represents the range M--M; M--, which stands for the range from line M through the last line of the current proof; and --N, which represents the range from the first line of the proof through line N. Hence -- represents the range consisting of every line in the proof.

`EXISTING-LINE`
The number of a line in the current outline.

`PLINE` The number of a planned line.

`GWFF` A well-formed formula (wff). **ETPS** prompts you for a `GWFF` if it needs a wff, term or variable, and it will usually tell you which one of these it expects in the brief help message in the prompt.

A `GWFF` can be one of the following:

1. A string representing a wff in double quotes. Strings may contain `<Return>`'s for formatting purposes. Case does matter for variables and constants like `"x"`, `"y"`, `"P"`. For example `"x"` is not the same as `"X"`. Case is ignored, however, for special keywords, like quantifiers, connectives, etc. All logical symbols must be separated by spaces. In addition, the bracketing conventions used in the logic textbook are used in **ETPS**, and the symbol `"~"` can be used as an abbreviation for `"NOT"`; thus `"forall x.P x y implies ~ [Q x or Q y]"` represents the same gwff as `"FORALL x[P x y IMPLIES NOT .Q x OR Q y]"`. In general you may type wffs just as they appear in the logic text. See Chapter 4 for some examples of typed wffs and variables, and Chapter 5 (especially Section 5.1.3) for examples of wffs of higher-order logic (type theory). For more examples, execute the command `PROBLEMS` in ETPS with style GENERIC and answer "yes" when you are asked if you wish to see

definitions. Superscripts can be used, but unlike the textbook, they are not used to indicate the arity of functions. Instead, they are used to distinguish variables. Superscripts are indicated by using a "^". Valid superscripts must follow these rules.

- Only strings of the form [0-9]+ can be superscripts.
- The user will explicitly indicate a superscript by the use of the "^". E.g., "x^0", "foo^1234567". A "^" which is not followed by a legal superscript is treated as any (non-logical-constant) character would be. Thus "x^" is legal input, as is "^" or "^^", or "x^y".
- A superscript can only be used at the end of a variable, not in the middle. Hence "x^1y" will be parsed as "x^1(II) y(I)" (x^1 applied to y) not as "x^1y(I)" (a single variable).
- Generic style will show the superscripts with the ^, i.e., if you enter "x^1(I)", then it will print as "x^1(I)" when the style is generic and PRINTTYPES is T.
- Entering "x1" results in "x1", not "x^1", i.e., superscripts will not be created from the user's input unless explicitly indicated.

2. A number of a line in the proof, for example `100`. **ETPS** will replace it with the wff asserted in that line.
3. A label referring to a wff or the name of an exercise or lemma. A label can be assigned by the `CW` command (see page 15).
4. `(ed gwff)` which allows you to locate a sub-expression of another `GWFF`. For example `(ed 100)` can be used to extract a subformula from the assertion of line `100`. See Section 2.12 for an explanation of how to use this option.
5. A backquoted form, calling a wffop. For example, `‘(S x y A)` is the wff resulting from the substitution of `x` for `y` in `A`. See Appendix III for the most commonly used wffops.

`GWFF0` A gwff of type o. Two special constants of type o are provided: `TRUTH` and `FALSEHOOD`. These gwffs act just as you might expect, e.g., TRUTH ≡ `p∨~p` and FALSEHOOD ≡ `p∧~p`. After running a proof through `SCRIBE`, TRUTH will print as **T** and FALSEHOOD will print as ⊥.

`GVAR` A gwff which must be a logical variable.

`TYPESYM` The string representation of a type symbol. See Section 5.1.1.

`BOOK-THEOREM`
A theorem in the book. See the `PROBLEMS` command (page 9).

`EXERCISE` An exercise which may be assigned.

`PRACTICE` An unscored practice exercise.

`TEST-PROBLEM`
A potential test problem.

Several argument types are lists or pairs of other types. They are specified in parentheses, for example `(1 2 99)`. The empty list is specified by `()`. Pairs are entered as two element lists where the two elements are separated by a period. For example, you might enter the pair `("x" . "y")`. Do not put commas into your input!

We list the most common of the composite argument types below:

`OCCLIST` A list of occurrences (counted left-to-right) of a subwff in a wff. This list may be entered as a list of positive numbers or the symbol `ALL`. ALL refers to all occurrences of the subwff.

`LINELIST` A list of numbers of lines.

`LINE-RANGE-LIST`
A list of line ranges.

`EXISTING-LINELIST`
A list of numbers of lines in the current outline.

`GWFFLIST` A list of `GWFF`s.

`GVAR-LIST`
A list of `GVAR`s.

1.6. Flags and Amenities

Many aspects of **ETPS** are controlled by flags. See section 2.11 for some information about flags.

ETPS incorporates several features of the Unix C-shell (csh) top-level. These features include various control characters, command sequences, a history mechanism, and aliases. See Appendix I for details.

You may wish to set certain flags and define certain aliases each time you run **ETPS**. A good way to do this without having to repeat the commands is to start a work file (using `SAVE-WORK`), then set the flags and define your aliases, then use `STOP-SAVE` to stop saving into the file. When you subsequently use **ETPS**, you can use `EXECUTE-FILE` to automatically execute all the commands in the work file to set the flags and define the aliases.

1.7. Bugs and Error Messages

Typing or logical errors are usually noticed by **ETPS**, which issues an appropriate diagnostic message and typically throws you back to top-level.

Most bugs in **ETPS** itself will be caught by an error handler which appends an appropriate message to a file in the teacher's area. This of course only applies to real bugs in the **ETPS** software or Common Lisp, not typing errors which are caught by the command interpreter. You may try again after you get a bug error message, and often you will discover that you just made a mistake which was not caught by the usual error handling routines. If you still get an error send mail to the teacher or send a message with the `REMARK` command. If you think that you have discovered a bug in **ETPS**, don't delete the `.WORK` file for that session but rename that file (say, to `Y`*exercise-number*`.work`) so that your work is not overwritten, then allow read access for it and send mail to the teacher with a pointer to that file.

2. System Commands

2.1. Communication

`HELP` *subject*
: Will list help available on the subject. The help messages for inference rules can be very long; you may wish to set the flag SHORT-HELP to T, to prevent this. (The default value of this flag is NIL.)

`?` Will list all available commands.

`LIST-RULES`
: Will list all the available inference rules.

`PROBLEMS` Lists all exercises available in **ETPS**, and shows which are practice exercises for which `ADVICE` is available. Also lists theorems which can be asserted with the `ASSERT` command.

`NEWS` Will list all bugs fixed, changes, improvements, et cetera. The most recent **ETPS** news items are announced whenever you start **ETPS**.

`REMARK` *string*
: Will mail the *string* to the teacher, or will include it as a comment in a recordfile created in the teacher's directory.

2.2. Starting and Finishing

`EXERCISE` *label*
: Set up the proof outline to attempt the proof of exercise *label* from the logic text.

`PROVE` *gwff0 label line*
: Similar to `EXERCISE`, but lets you prove your own wff. *label* is the name of the proof, *line* is number of the last line of the proof, i.e., the number of the line which will assert the theorem.

`PROOFLIST`
: Gives a list of all the completed or partially completed proofs in memory, any of which may be `RECONSIDER`ed.

`RECONSIDER` *label*
: Allows you to return to a previous proof. *label* is the name that you gave the proof when you originally started it using `EXERCISE` or `PROVE`; all of the proofs that you have worked on in the current session with ETPS may be `RECONSIDER`ed.

`CLEANUP` Deletes, in the current completed proof, unnecessary lines and redundant lines introduced by the `SAME` rule. This command is applicable only if the proof is complete. `CLEANUP` asks for confirmation before actually deleting lines.

`DONE` Signals that you think that you have completed the current proof. **ETPS** will not believe you if you are not really done. The `DONE` command appends a message to the recordfile in the teacher's directory. If you fail to use it, you may not get credit for your work.

`SUMMARY` Tells the user what exercises have been completed.

`EXIT` Leave **ETPS**. See Section 1.3 for some information on reentering **ETPS**. This command will automatically close open work files.

`HISTORY` *n reverse*
: Show history list. Shows the N most recent events; N defaults to the value of HISTORY-SIZE, showing entire history list. REVERSE defaults to NO; if YES, most recent commands will be shown first.

`ALIAS` *name def*
: Define an alias DEF for the symbol NAME. Works just like the alias command in the Unix csh. If the value of NAME is *ALL*, all aliases will be printed; if the value of DEF is the empty string, then the current alias definition of NAME will be printed. See UNALIAS.

`UNALIAS` *name*
: Remove an alias for the symbol NAME. Like the Unix csh unalias, except that NAME must exactly match the existing alias; no filename completion is done.

See Section I.4 for more discussion of aliases.

2.3. Starting the Java Interface

There is a Java interface for **ETPS** running under Allegro Lisp (version 5.0 or greater). Special symbol fonts, proofwindows (see Section 2.4) and editor windows (see Section 2.12) are available when using the Java interface.

`JAVAWIN` Start the Java interface. This should open a Java window with menus, a display area for **ETPS** output, and possibly a prompt at the bottom of the window. All **ETPS** output after the Java window opens will be printed into the Java window. Also, all user input must be entered via the Java window, either using the menus or using the prompt at the bottom of the window. To enter input into the prompt, the user may need to click on the prompt area to bring it into focus.

2.4. Proofwindows

When **ETPS** is running under X-windows or through the Java interface (see section 2.3), it is possible to start up separate windows displaying the current subproof (which is described in Section 2.5 and can be printed on the screen with the `^P` command), the current subproof plus line numbers (which can be printed with the `^PN` command) and the complete proof (which can be printed with the `PALL` command). These windows will be automatically updated as commands are issued to modify the proof interactively. (By scrolling up in these windows, you can see the previous displays.) The windows may be moved around and resized by the usual methods for manipulating windows. `PSTATUS` will update the proofwindows. Printing in the proofwindows can be modified by changing the flags PROOFW-ACTIVE, PROOFW-ALL, PROOFW-ACTIVE+NOS, and PRINTLINEFLAG. For more information about the proofwindows, type `HELP BEGIN-PRFW`.

`BEGIN-PRFW`
: Begin proofwindow top-level; open Current Subproof, Current Subproof & Line Numbers, and Complete Proof windows.

`END-PRFW`
: End proofwindow top-level; close all the proofwindows.

If you forget to use the `END-PRFW` command before issuing the `EXIT` command to leave ETPS, the proofwindows may not disappear. To get rid of such a window, put the cursor into it and hit `^C` (control-C).

2.5. The Current Subproof

ETPS maintains a list which contains the *status* information for the current proof outline. The status information consists of the *planned lines* (lines not yet justified) and the lines (called *sponsoring lines*) which **ETPS** thinks you may wish to use in the proof of the associated planned line. The planned line which is the focus of current attention and its sponsoring lines constitute the Current Subproof. This is displayed in the windows mentioned in Section 2.4 and can be printed on the screen by using the `^P` command. The following commands allow you to examine and modify the current subproof and the status information.

`PSTATUS` This will print the status information in the form

$(p_1\ l_{11}\ \ldots\ l_{1n})\ \ldots\ (p_m\ l_{m1}\ \ldots\ l_{mk})$

where p_1 ... p_m are the planned lines and the rest of each list are the sponsors for the planned line. The first list corresponds to the ‘‘current’’ plan. In addition, it'll issue a message if you are currently saving work.

`SUBPROOF` *pline*
: Tells **ETPS** that you now wish to focus on the planned line *pline*. This changes the current subproof; it mainly affects the displays in the proofwindows, the results of the `^P` and `^PN` commands, and the defaults offered for outline manipulations commands.

`SPONSOR` *pline existing-linelist*
: Tells **ETPS** to add the lines in the list *existing-linelist* of existing proof lines to the list of sponsors for the planned line *pline*.

`UNSPONSOR` *pline existing-linelist*
: Tells **ETPS** to remove the lines in the list *existing-linelist* of existing proof lines from the list of sponsors for the planned line *pline*.

2.6. Saving Work

`SAVEPROOF` *filename*
: Saves the current natural deduction proof to the specified file in a form in which it can be restored. Use RESTOREPROOF to restore the proof. Overwrites the file if it already exists.

`RESTOREPROOF` *filename*
: Reads a natural deduction proof from a file created by SAVEPROOF and makes it the current proof. A security feature prevents the restoration of saved proofs which have been altered in any way.

`SAVE-WORK` *filename*
: Starts to save subsequent commands in the save-work file *filename*. Notice that this is not necessary, unless you want to specify your own filename before starting an exercise or if you did a `STOP-SAVE` some time before. A typical use would be to switch save-work files when you are done with one exercise and are starting the next one without leaving **ETPS**. The extension of *filename* defaults to `.WORK`.

`STOP-SAVE`
: Stops saving into the current save-work file. All commands that have been given but not yet saved will be written out to the file.

`RESTORE-WORK` *filename show-lines exec-print outfile*
: Executes commands from *filename* and continues to save in that file. When the end of the file is reached, you will be back at **ETPS**' command level. *show-lines* controls whether proof lines will be shown when restoring the proof. This is very time consuming, therefore the default is `NO`. *exec-print* controls whether printing commands will be executed when restoring the proof. These are commands like `PLINE`, `PRINTPROOF`, `HELP`, or `PROBLEMS`. The default is `NO`. *outfile* is the name of a file where the commands and the output is echoed to, while they are re-executed. The default is `TTY:`, the terminal, so you can see how far **ETPS** has progressed. To speed up the process you may select `NUL:`. `RESTORE-WORK` will not re-execute any of the following: `EXIT`, `RESUME-SAVE`, `RESTORE-WORK`, `EXECUTE-FILE`, `SAVE-WORK`, `STOP-SAVE`. They usually don't make sense when reading commands from a save-work file. If you aborted a command with a Ctrl-C, the Ctrl-C will be in the file and will abort the execution of the commands.

`RESUME-SAVE`
: Use this command to resume saving commands into the most recent save-work file. Unlike `RESTORE-WORK`, this command doesn't execute commands from the file, but simply appends subsequent commands to the file. You may not use this command if you are already saving work. Also, you may run into trouble if you forgot to save some commands.

`EXECUTE-FILE` *filename show-lines exec-print outfile*
: Works like `RESTORE-WORK`, but does not continue saving into a file after executing the commands in the file.

`SCRIPT` *scriptfile if-exists-append*
: Saves a transcript of session to a file.

`UNSCRIPT` Closes the most recent file opened with the SCRIPT command.

2.7. Printing

`DEPTH` *n* — *n* is a number. This command causes all subformulas at depth greater than *n* to be printed as `&`. For example the wff `"FORALL x FORALL y FORALL z.P x y z"` will be printed as below after the command `DEPTH 4`: `FORALL x FORALL y FORALL z.&`. This command may save time in printing huge formulas, particularly in higher-order logic.

`PW` *gwff* — Print *gwff*.

`PWSCOPE` *gwff*
: Print *gwff* with all brackets restored. This is sometimes useful if you are not sure which connective has precedence over another.

`PLINE` *line* — Print a specified line.

`PL` *lower upper*
: Print all lines in the range from *lower* to *upper*.

`PL*` *print-ranges*
: Print all proof lines in given ranges.

`PPLAN` *pline*
: Prints the planned line *pline* and all of its sponsors. A similar effect can be achieved with the `^P`, provided *pline* is the current planned line. `SUBPROOF` will change the current planned line. See Section 2.5 for more information on `SUBPROOF`.

`^P` — Same as `PPLAN` for the current planned line. Note that `^P` is not a control-character, but two characters `^ P` followed by a `<Return>`.

`^PN` — As for `^P`, but also prints the line numbers (only) of all the other lines of the proof. `^PN` is not a control character, but three characters `^`, `P` and `N`.

`PALL` — Print all the lines in the current outline.

`PRINTPROOF` *filespec*
: This will print the current proof into a file.

`SCRIBEPROOF` *filespec*
: This will also print the current proof into a file, but uses special symbols. In order to print this file, you must first run it through `SCRIBE`. *filespec* has the same format as in `PRINTPROOF`. The extension defaults to `MSS`.

`TEXPROOF` *filespec*
: Print the current proof into a tex file. After leaving tps, run this .tex file through `TeX` and print the resulting file.

2.8. Rearranging the Proof

This section describes commands which rearrange the proof outline, which is described in Section 3.1. The first two commands are frequently useful for starting over a part of the proof after you realize you have tried a wrong approach.

`DELETE` *existing-linelist*
: Delete the lines in *existing-linelist* from the proof. If you delete a hypothesis line, all lines which use this hypothesis will also be deleted. If a line justifying another line is deleted, the justification of that line is changed to `PLAN`*n*. Lines are shown as they are deleted.

`DELETE*` *ranges*
: Delete ranges of lines from the proof outline.

`PLAN` *existing-line*
: Change a justified line back to a planned line.

The next few commands allow you to change the numbers of lines in the proof, or even change the order of lines, as long as the conclusion of a rule of inference comes after the justifying lines. All references to line numbers are changed automatically whenever the numbers are changed.

`MOVE` *from to*
: Moves a line in the proof. **ETPS** checks to make sure the move is legal, i.e., the lines justifying a given line appear before it in the proof.

`MOVE*` *range-to-move new-start*
: Move all proof lines in given range to begin at new start number, but preserving the relative distances between the lines.

`RENUMBERALL` *increment*
: Renumbers all the lines in the proof with an increment of *increment*.

As a proof is constructed, new lines must be inserted into the outline and given new line numbers between occupied line numbers. A space allotted for this task is called a gap. Gaps are indicated in the outline by ellipses (...) and may be adjusted by the command `MODIFY-GAPS`.

`INTRODUCE-GAP` *existing-line increment*
: Introduce a new gap (or increase an existing gap) above *existing-line* by increasing the line numbers by *increment* of all lines beginning with line *existing-line*.

`MODIFY-GAPS` *lower upper*
: Removes unnecessary gaps in line numbers from the proof structure. Also, gaps with length less than *lower* have their length increased to *lower*, while gaps with length greater than *upper* have their length decreased to *upper*. *lower* must be less than or equal to *upper*.

`SQUEEZE`
: Removes unnecessary gaps in line numbers from the proof structure. Leaves necessary gaps (those just above planned lines) alone.

There is no UNDO command in ETPS. Usually one can undo the results of commands fairly easily by such measures as deleting lines from the proof. However, if this seems complicated, the following procedure can often be used to restore the proof to one of its previous states. **ETPS** is probably creating a save-work file. Execute the `STOP-SAVE` command, make a backup copy of the save-work file for safety, edit the save-work file by deleting the commands you wish you had not executed, then start a new **ETPS** and use `RESTORE-WORK` with the edited save-work file.

2.9. Proof Assistance

`ADVICE`
: Initially gives hints based on the current structure of the proof. The next time it is executed, it suggests the inference command based on the previous hint. It repeats this flip-flopping between hints and suggestions until it has no more suggestions. Advice may not be available for some exercises. **ETPS** will tell you if advice cannot be given and ask for confirmation if the advice would deduct points from your score.

`CHECK-STRUCTURE`
: Finds those lines which are not integrated into the proof and suggests their deletion. These lines are deduced lines which have not been used to justify another line and are no longer supports for any planned line. In addition, `CHECK-STRUCTURE` looks for extraneous hypotheses in each of the lines of the proof.

2.10. Higher-Order Logic

`PWTYPES` *gwff*
: Print *wff* with type symbols.

`SHOWTYPES`
From now on show the types of all wffs.

`SHOWNOTYPES`
From now on suppress types of all wffs.

2.11. Flags and Review

Many aspects of **ETPS** are controlled by flags. Most of the time you can ignore these, but if you wish to change some aspect of **ETPS** (such as the widths of the lines in a proof), you may be able to do so by changing the value of a flag (such as RIGHTMARGIN). HELP *flag* will provide information about a particular flag. Use the REVIEW top-level to find what flags are available. Enter `?` for a list of all the commands in this top-level; the following is just a selection of those available.

`REVIEW` Enter the review top-level.

`SETFLAG` Change the value of a flag.

`SUBJECTS` Each flag is associated with one or more subjects; this command lists all the known subjects. Some of these subjects may be irrelevant to **ETPS**, but used in a larger system of which **ETPS** is a component; you can ignore them.

`LIST` *subjects*
List all the flags associated with these subjects.

`LEAVE` Return to the main top-level.

2.12. ETPS Editor

The **ETPS** editor can be used to construct new formulas and extract subformulas from formulas already known to **ETPS**. You can enter the editor with the top-level command `ED`. Use `(ed` *gwff*`)` when asked for a `GWFF`. This will prompt you with `<Ed`*n*`>`. The wff you are editing will be referred to as `EDWFF`. Using the editor commands, move to a subformula and/or modify the `EDWFF` until you have the `GWFF` you desire. Then exit the editor by using the command `OK`; the current value of `EDWFF` will be returned.

For example, suppose that **ETPS** has asked you for a `GWFF` and the `GWFF` you would like to supply is `B`, a subwff of the assertion in line 1, which is `A AND B IMPLIES C`. Enter `(ed 1)` to enter the editor with that formula. Then use the following sequence of commands:

```
<Ed1>L        This moves to the left of the implication sign.
A AND B
<Ed2>R        This moves to the right of the conjunction.
B
<Ed3>OK       Since we have what we want, we exit the editor.
```

This is of course a trivial example, but if `B` had been a very complicated formula, using the editor would have been both faster and less susceptible to error than typing it in would have been.

You can also use multiple commands on a single editor input line, which will save more time. We could have done the above example as follows:

```
<Ed1>L R OK
```

When **ETPS** is running under X-windows or through the Java interface (see Section 2.3), the command `ED` will also start up two windows which display the top formula and the current formula in the editor. These windows are automatically updated as commands are issued to modify the formula, and they will disappear when you use `OK` to leave the editor. (By scrolling up in these windows, you can see the previous displays.) The windows may be moved around and resized by the usual methods for manipulating windows.

To prevent the windows from appearing, modify the flags EDWIN-TOP and EDWIN-CURRENT.

The following sections give brief descriptions of the most commonly used editor commands. Other editor commands are listed in Appendix II.

2.12.1. Top-Levels

`<Edn>OK` Exit the editor with the current wff.

2.12.2. Printing

`<Edn>P` Print the current expression in short format, i.e., some subformulas will be replaced by `&`.

`<Edn>PP` Pretty-print a wff.

`<Edn>PS` Print a wff showing all brackets and dots.

`<Edn>PT` Print a wff showing types.

2.12.3. Labels

`<Edn>CW` *label*
Assigns a label to the edwff, but does not change the edwff. You can use the label to refer to this wff later.

2.12.4. Moving Commands

`<Edn>0` Move up one-level, i.e., undo the last `L`, `R`, `D`, or `A` command. Note that 0 stands for the numeral zero.

`<Edn>A` For an expression like `P x y`, delete the rightmost element; in this example the result will be to make `Px` the current expression. For a quantified expression, move to the quantified variable.

`<Edn>D` For an expression like `P x y`, move to the rightmost element; in this example `y`. For a quantified expression, move to the scope of the quantifier.

`<Edn>FB` Find the first binder (left to right).

`<Edn>FI` Find an infix operator.

`<Edn>L` For an infix-operator, move to the left argument.

`<Edn>R` For an infix-operator, move to the right argument.

`<Edn>UNDO`
Moves up (like 0), but throws away any editing since your last downward moving command (typically A,D,L,or R).

`<Edn>XTR` Makes the current edwff the top wff.

`<Edn>^` Move upwards through enclosing wffs all the way to the top.

2.12.5. Substitution

`<Edn>AB` *newvar*
Alphabetic change of variable at top-level.

`<Edn>IB` *term*
Instantiate a top-level universal or existential binder with a term.

`<Edn>REW-EQUIV` *gwff*
: Replaces each equivalence in the gwff with a conjunction of implications.

`<Edn>RP` *rep-sym rep-by*
: Replace one occurrence of a symbol rep-sym (such as AND) by a predefined equivalent wff involving the symbol rep-by (such as [λ p λ q.~.p ⊃ ~q]). In this example, rep-sym is AND and rep-by is IMPLIES. To see if a symbol can be replaced using this command, enter HELP symbol; any such replacements will be listed under the heading ‘Replaceable Symbols’.

`<Edn>RPALL` *rep-sym rep-by*
: Replace all occurrences of a symbol by a predefined equivalent wff.

`<Edn>SUB` *gwff*
: Replaces the current wff by the wff supplied.

`<Edn>SUBST` *term var*
: Substitute a term for the free occurrences of variable in a gwff. Bound variables may be renamed, using the function in the global variable REN-VAR-FN.

`<Edn>SUBSTYP` *typevar typesym*
: Substitute typevar with typesym.

2.12.6. Negation movers

`<Edn>NEG` Negates current wff, erasing double negations.

`<Edn>NNF` Return the negation normal form of the given wff.

`<Edn>PULL-NEG`
: Pulls negations out one level.

`<Edn>PUSH-NEG`
: Pushes negation through the outermost operator or quantifier.

3. Inference Rules

3.1. How to Read Inference Rules

The user works within a *proof-outline*, which is a sequence of lines. Each line is either a hypothesis, a consequence of lines with lower numbers or an unjustified planned line. In general, a line of the proof has the following form:

(*n*) H_1,...,H_m ! *assertion* *justification*: *wffs* l_1 ... l_k

n is a number which serves as a label for the line. Each of the H_i's is the number of a line asserting a hypothesis. ! stands for the turnstyle |– , and l_1 ... l_k are the numbers of the lines which are used to justify this line.

Every description of a logical inference rule states that certain lines of a proof may be inferred from certain other lines, provided that certain restrictions are satisfied, and notes the change which the rule effects on the proof *status* (see Section 2.5). Inference commands apply inference rules, and they may be used in various ways to complete the proof. They may generate either new planned or sponsoring lines, or close up a proof by justifying planned lines with sponsoring lines. However, there is usually a most natural way to use a rule. This is indicated in the statement of the rule by labelling those lines which are usually sponsors, before or after the rule is applied, with a `D` (for deduced) followed by a number. Similarly, those lines the rule expects to be planned lines are labelled with a `P` followed by a number.

The *transformation* statement in an inference rule description indicates the change in the proof status effected by the most natural use of the rule. The lists before the arrow `==>` are matched against the initial status; those after the arrow describe what the new status should be. The first element of a list is always a planned line and the remaining elements are its sponsors. Each element of a list is either a label for a line in the rule, `pp` or `ss`. The symbol `pp` refers to all matched planned lines and the symbol `ss` to all (other) sponsoring lines for each of the matched planned lines.

Certain lines in a rule description are expected to exist in the proof before the rule is applied; these are indicated by an asterisk. If a line does not already exist when you apply an inference command, and its corresponding line in the rule asserts a wff which cannot be formed from the wffs in rule lines corresponding to existing proof lines, then the wff asserted by that new line will have to be entered. Thus, in order to avoid typing in long wffs, you should try to organize your work (often by working up from the bottom) so that such lines will be introduced before they are needed.

ETPS automatically applies the metatheorem that if $H_1 \vdash$ A and $H_1 \subseteq H_2$ then $H_2 \vdash$ A, so that normally you do not have to worry about expanding sets of hypotheses before applying inference rules.

Some rules use *wffops*, operations on well-formed formulae, in their descriptions. For example, a rule might form the assertion of one line by substituting a term, `t`, for all free occurrences of a variable, `x`, in a wff, `A`, asserted in another line. The new assertion would then be `‘(S t x A)`, where `S` is the wffop performing the substitution. (The backquote tells **ETPS** that the application of a wffop is being asserted, and not a wff.) However, this substitution will only be allowed when `t` is free for `x` in `A`. Thus, the form `(FREE-FOR t x A)` would appear in the restrictions for the rule. `FREE-FOR` is the wffop which checks that `t` is free for `x` in `A`. A catalogue of the wffops used in inference rules is provided in Appendix III. It is included in this manual only to help you understand the descriptions of the inference rules.

Now we shall give an example which demonstrates how to read a command. Before proceeding the reader should look at the description of `DEDUCT` in Section 3.5 below.

Suppose the proof originally contains the line

```
(P3)  H ⊢   A ⊃ B                                                    PLAN4
```

and we apply the command:

```
<n>DEDUCT    P3     H1     D2
```

Here

- P3 stands for the planned line that you are trying to prove.
- H1 stands for the number of the line which asserts the new hypothesis (the wff A in this case).
- D2 stands for the number of the new planned line (whose assertion is B in this case).

After the rule is applied, the proof will contain the lines:

```
 (H1)  H,A      ⊢   A                                                  Hyp
 (D2)  H,A      ⊢   B                                                PLAN5
 (P3)  H        ⊢   A ⊃ B                                       Deduct: D2
```

3.2. Quick Reference to Rules

Here is a list of the most generally useful rules of inference available in **ETPS** for quick reference. Some additional rules of inference may be found by typing "?" or "LIST-RULES" in **ETPS**. See the page indicated for a precise description of each rule listed below. You can also type "HELP *rule*" in **ETPS**. The rules of inference in **ETPS** are applicable to both first-order logic and higher-order logic. The user who is interested only in first-order logic should ignore the rules for higher-order logic.

Special Rules

RULEP	19	Justify a line by Rule P.
ASSERT	20	Assert a theorem known to **ETPS**.
ADD-HYPS	20	Weaken a line to include extra hypotheses.
DELETE-HYPS	20	Delete some hypotheses from the given line.

Miscellaneous Rules

HYP	20	Introduce a new hypothesis line into the proof outline.
LEMMA	20	Introduce a Lemma.
SAME	20	Use the fact that two lines are identical to justify a planned line.

Propositional Rules

CASES	20	Rule of Cases.
DEDUCT	21	The deduction rule.
DISJ-IMP	21	Rule to replace a disjunction by an implication.
ECONJ	21	Rule to eliminate a conjunction.
EQUIV-IMPLICS	21	Rule to convert an equivalence into twin implications.
ICONJ	21	Rule to introduce a conjunction.
IMP-DISJ	21	Rule to replace an implication by a disjunction.
IMPLICS-EQUIV	21	Rule to convert twin implications into an equivalence.
INDIRECT	21	Rule of Indirect Proof.
INDIRECT1	22	Rule of Indirect Proof using one contradictory line.
INDIRECT2	22	Rule of Indirect Proof using two contradictory lines.
MP	22	Modus Ponens.

Negation Rules

ABSURD	22	From falsehood, deduce anything.

`ENEG`	22	Eliminate a negation.
`INEG`	22	Introduce a negation.
`PULLNEG`	22	Pull out negation.
`PUSHNEG`	23	Push in negation.

Quantifier Rules

`AB*`	23	Rule to alphabetically change embedded bound variables.
`ABE`	23	Rule to change a top-level occurrence of an existentially quantified variable.
`ABU`	23	Rule to change a top-level occurrence of a universally quantified variable.
`EGEN`	23	Rule of Existential Generalization.
`RULEC`	23	Rule C.
`UGEN`	23	Rule of Universal Generalization.
`UI`	24	Rule of Universal Instantiation.

Substitution Rules

`SUBSTITUTE`	24	Rule to substitute a term for a variable.

Equality Rules

`EXT=`	24	Rule of Extensionality.
`EXT=0`	24	Rule to derive an equality at type o from an equivalence.
`LET`	24	Rule to produce a new variable which will represent an entire formula during part of a proof.
`SUBST=`	24	Substitution of Equality. Performs whichever of the `SUBST=L` and `SUBST=R` rules is appropriate.
`SUBST=L`	25	Substitution of Equality. Replaces some occurrences of the left hand side by the right hand side.
`SUBST=R`	25	Substitution of Equality. Replaces some occurrences of the right hand side by the left hand side.
`SUBST-EQUIV`	25	Substitution of Equivalence.
`SYM=`	25	Symmetry of Equality.

Definition Rules

`EDEF`	25	Rule to eliminate first definition, left to right.
`EQUIV-WFFS`	25	Rule to assert equivalence of lines up to definition.
`IDEF`	25	Rule to introduce a definition.

Lambda Conversion Rules

`LAMBDA*`	26	Convert between two equivalent lambda-expressions.
`LCONTR*`	26	Rule to put an inferred line into Lambda-normal form.
`LEXPD*`	26	Rule to put a planned line into Lambda-normal form.

3.3. Special Rules

`RULEP` Justify a line by Rule P.

Infer B_o from A^1_o and ... and A^n_o, provided that $[[A^1_o \land \ldots \land A^n_o] \supset B_o]$ is a substitution instance of a tautology. As a special case, infer B_o if it is a substitution instance of a tautology. The first argument must be the line to be justified; the second argument must be a list of lines (possibly empty) from which this line follows by Rule P. The flag RULEP-MAINFN controls which of two functions will be used by `RULEP`. When RULEP-MAINFN is set to `RULEP-SIMPLE`, `RULEP` will merely ensure that the planned line follows by Rule P from the specified support lines. When RULEP-MAINFN is set to `RULEP-DELUXE` (which is the default), `RULEP` will find a minimal subset of the support lines which suffices to prove the planned line by Rule P (if any). Note that `RULEP-DELUXE` will be somewhat slower than `RULEP-SIMPLE`. In order to check the setting of RULEP-MAINFN, merely enter

"RULEP-MAINFN" at the top-level. You will be prompted for a new value and the current value will be displayed. Hit <return> to accept the current value, or enter the new value.

`ASSERT` Assert a theorem known to **ETPS**.

Use a theorem known to **ETPS** (see Appendix IV) as a lemma in the current proof. Such a theorem can only be used if allowed by the teacher for that exercise. The first argument is the name of the theorem; the second argument is the line number. If the line already exists, ETPS will check whether it is a legal instance of the theorem schema, otherwise it will prompt for the metavariables in the theorem schema (usually x or P, Q, ...).

`ADD-HYPS` Weaken a line to include extra hypotheses.

Adding the hypotheses to the line may cause some lines to become planned lines. If possible, the user is given the option of adding hypotheses to lines after the given line so that no lines will become planned.

`DELETE-HYPS` Delete some hypotheses from the given line.

This may leave the given line as a planned line. The user is given the option of also deleting some hypotheses from lines after the given line. If possible, the user is given the option of deleting some hypotheses from lines before the given line so that the given line does not become a planned line.

3.4. Miscellaneous Rules

`HYP` Introduce a new hypothesis line into the proof outline.

```
 (H1)  H1      ⊢ A_o                                              Hyp
*(P2)  H       ⊢ B_o
Transformation: (P2 ss) ==> (P2 H1 ss)
```

`LEMMA` Introduce a Lemma.

```
 (P1)  H       ⊢ A_o
*(P2)  H       ⊢ B_o
Transformation: (P2 ss) ==> (P2 P1 ss) (P1 ss)
```

`SAME` Use the fact that two lines are identical to justify a planned line.

```
*(D1)  H       ⊢ A_o
*(P2)  H       ⊢ A_o                                     Same as: D1
Transformation: (P2 D1 ss) ==>
```

3.5. Propositional Rules

`CASES` Rule of Cases.

```
*(D1)  H       ⊢ A_o ∨ B_o
 (H2)  H,H2    ⊢ A_o                                      Case 1: D1
 (P3)  H,H2    ⊢ C_o
 (H4)  H,H4    ⊢ B_o                                      Case 2: D1
 (P5)  H,H4    ⊢ C_o
*(P6)  H       ⊢ C_o                                 Cases: D1 P3 P5
Transformation: (P6 D1 ss) ==> (P3 H2 ss) (P5 H4 ss)
```

DEDUCT The deduction rule.

(H1) H,H1 $\vdash A_o$ Hyp
(D2) H,H1 $\vdash B_o$
*(P3) H $\vdash A_o \supset B_o$ Deduct: D2
Transformation: (P3 ss) ==> (D2 H1 ss)

DISJ-IMP Rule to replace a disjunction by an implication.

*(D1) H $\vdash \sim A_o \vee B_o$
(D2) H $\vdash A_o \supset B_o$ Disj-Imp: D1
Transformation: (pp D1 ss) ==> (pp D2 ss)

ECONJ Rule to eliminate a conjunction by inferring its two conjuncts.

*(D1) H $\vdash A_o \wedge B_o$
(D2) H $\vdash A_o$ Conj: D1
(D3) H $\vdash B_o$ Conj: D1
Transformation: (pp D1 ss) ==> (pp D2 D3 ss)

EQUIV-IMPLICS
Rule to convert an equivalence into twin implications.

*(D1) H $\vdash P_o \equiv R_o$
(D2) H $\vdash [P_o \supset R_o] \wedge .R \supset P$ EquivImp: D1
Transformation: (pp D1 ss) ==> (pp D2 ss)

ICONJ Rule to introduce a conjunction by inferring it from two conjuncts.

(P1) H $\vdash A_o$
(P2) H $\vdash B_o$
*(P3) H $\vdash A_o \wedge B_o$ Conj: P1 P2
Transformation: (P3 ss) ==> (P1 ss) (P2 ss)

IMP-DISJ Rule to replace an implication by a disjunction.

*(D1) H $\vdash A_o \supset B_o$
(D2) H $\vdash \sim A_o \vee B_o$ Imp-Disj: D1
Transformation: (pp D1 ss) ==> (pp D2 ss)

IMPLICS-EQUIV
Rule to convert twin implications into an equivalence.

(P1) H $\vdash [P_o \supset R_o] \wedge .R \supset P$
*(P2) H $\vdash P_o \equiv R_o$ ImpEquiv: P1
Transformation: (P2 ss) ==> (P1 ss)

INDIRECT Rule of Indirect Proof.

(H1) H,H1 $\vdash \sim A_o$ Assume negation
(P2) H,H1 $\vdash \bot$
*(P3) H $\vdash A_o$ Indirect: P2
Transformation: (P3 ss) ==> (P2 H1 ss)

INDIRECT1
Rule of Indirect Proof using one contradictory line.

```
 (H1)  H,H1   ⊢ ~ A_o                                  Assume negation
 (P2)  H,H1   ⊢ B_o ∧ ~ B_o
*(P3)  H      ⊢ A_o                                       Indirect: P2
Transformation: (P3 ss) ==> (P2 H1 ss)
```

INDIRECT2
Rule of Indirect Proof using two contradictory lines.

```
 (H1)  H,H1   ⊢ ~ A_o                                  Assume negation
 (P2)  H,H1   ⊢ B_o
 (P3)  H,H1   ⊢ ~ B_o
*(P4)  H      ⊢ A_o                                 Indirect: P2 P3
Transformation: (P4 ss) ==> (P2 H1 ss) (P3 H1 ss)
```

MP Modus Ponens.

```
 (P1)  H      ⊢ A_o
*(D2)  H      ⊢ A_o ⊃ B_o
 (D3)  H      ⊢ B_o                                         MP: P1 D2
Transformation: (pp D2 ss) ==> (P1 ss) (pp D3 ss P1)
```

3.6. Negation Rules

ABSURD From falsehood, deduce anything.

```
 (P1)  H      ⊢ ⊥
*(P2)  H      ⊢ A_o                                         Absurd: P1
Transformation: (P2 ss) ==> (P1 ss)
```

ENEG Eliminate a negation.

```
*(D1)  H      ⊢ ~A_o
 (P2)  H      ⊢ A_o
*(P3)  H      ⊢ ⊥                                       NegElim: D1 P2
Transformation: (P3 D1 ss) ==> (P2 ss)
```

INEG Introduce a negation.

```
 (H1)  H,H1   ⊢ A_o                                                Hyp
 (P2)  H,H1   ⊢ ⊥
*(P3)  H      ⊢ ~A_o                                      NegIntro: P2
Transformation: (P3 ss) ==> (P2 H1 ss)
```

PULLNEG Pull out negation.

```
 (P1)  H      ⊢ ‘(PUSH-NEGATION  [~ A_o])
*(P2)  H      ⊢ ~ A_o                                          Neg: P1
Restrictions:  (NON-ATOMIC A_o)
Transformation: (P2 ss) ==> (P1 ss)
```

`PUSHNEG` Push in negation.

```
*(D1)  H      ⊦ ~ A_o
 (D2)  H      ⊦ ‘(PUSH-NEGATION  [~ A_o])                           Neg: D1
Restrictions:  (NON-ATOMIC A_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

3.7. Quantifier Rules

`AB*` Rule to alphabetically change embedded bound variables.

```
*(D1)  H      ⊦ A_o
 (D2)  H      ⊦ B_o                                                  AB: D1
Restrictions:  (WFFEQ-AB A_o B_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`ABE` Rule to change a top-level occurrence of an existentially quantified variable.

```
*(D1)  H      ⊦ ∃x_α A_o
 (D2)  H      ⊦ ∃y_α ‘(S  y  x_α  A_o)                             AB: y D1
Restrictions:  (FREE-FOR y_α x_α A_o) (NOT-FREE-IN y_α A_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`ABU` Rule to change a top-level occurrence of a universally quantified variable.

```
 (P1)  H      ⊦ ∀y_α ‘(S  y  x_α  A_o)
*(P2)  H      ⊦ ∀x_α A_o                                           AB: x P1
Restrictions:  (FREE-FOR y_α x_α A_o) (NOT-FREE-IN y_α A_o)
Transformation: (P2 ss) ==> (P1 ss)
```

`EGEN` Rule of Existential Generalization.

```
 (P1)  H      ⊦ ‘(LCONTR  [[λx_α A_o] t_α])
*(P2)  H      ⊦ ∃x_α A_o                                       EGen: t_α P1
Transformation: (P2 ss) ==> (P1 ss)
```

`RULEC` Rule C.

```
*(D1)  H      ⊦ ∃x_α B_o
 (H2)  H,H2   ⊦ ‘(LCONTR  [[λx_α B_o] y_α])                       Choose: y
 (D3)  H,H2   ⊦ A_o
*(P4)  H      ⊦ A_o                                            RuleC: D1 D3
Restrictions:  (IS-VARIABLE y_α) (NOT-FREE-IN-HYPS y_α)
(NOT-FREE-IN y_α [∃x_α B_o]) (NOT-FREE-IN y_α A_o)
Transformation: (P4 D1 ss) ==> (D3 H2 ss)
```

`UGEN` Rule of Universal Generalization.

```
 (P1)  H      ⊦ A_o
*(P2)  H      ⊦ ∀x_α A_o                                         UGen: x P1
Restrictions:  (NOT-FREE-IN-HYPS x_α)
Transformation: (P2 ss) ==> (P1 ss)
```

UI Rule of Universal Instantiation.

*(D1) H $\vdash \forall x_\alpha\ A_o$
(D2) H $\vdash$ ‘(LCONTR [[$\lambda x_\alpha\ A_o$] t_α]) UI: t D1
Transformation: (pp D1 ss) ==> (pp D2 D1 ss)

3.8. Substitution Rules

SUBSTITUTE
Rule to substitute a term for a variable.

*(D1) H $\vdash A_o$
(D2) H $\vdash$ ‘(S t_α x_α A_o) Subst: t x D1
Restrictions: (NOT-FREE-IN-HYPS x_α) (FREE-FOR t_α x_α A_o)
Transformation: (pp D1 ss) ==> (pp D2 ss D1)

3.9. Equality Rules

EXT= Rule of Extensionality.

(P1) H $\vdash \forall x_\beta . f_{\alpha\beta}\ x = g_{\alpha\beta}\ x$
*(P2) H $\vdash f_{\alpha\beta} = g_{\alpha\beta}$ Ext=: P1
Transformation: (P2 ss) ==> (P1 ss)

EXT=0 Rule to derive an equality at type o from an equivalence.

(P1) H $\vdash P_o \equiv R_o$
*(P2) H $\vdash P_o = R_o$ Ext=: P1
Transformation: (P2 ss) ==> (P1 ss)

LET Rule to produce a new variable which will represent an entire formula during part of a proof.

(D1) H $\vdash A_\alpha = A$ Refl=
(D2) H $\vdash \exists x_\alpha . x = A_\alpha$ EGen: x D1
(H3) H,H3 $\vdash x_\alpha = A_\alpha$ Choose: x
(P4) H,H3 $\vdash C_o$
*(P5) H $\vdash C_o$ RuleC: D2 P4
Restrictions: (NOT-FREE-IN-HYPS x_α) (NOT-FREE-IN x_α C_o)
Transformation: (P5 ss) ==> (P4 ss D1 D2 H3)

SUBST= Substitution of Equality. Performs whichever of the SUBST=L and SUBST=R rules is appropriate.

(P1) H $\vdash P_o$
*(D2) H $\vdash s_\alpha = t_\alpha$
(D3) H $\vdash R_o$ Sub=: P1 D2
Restrictions: (SAME-MODULO-EQUALITY P_o R_o s_α t_α)
Transformation: (pp D2 ss) ==> (P1 ss) (pp D3 ss P1 D2)

`SUBST=L` Substitution of Equality. Replaces some occurrences of the left hand side by the right hand side.

```
 (P1)  H      ├ P_o
*(D2)  H      ├ s_α = t_α
 (D3)  H      ├ R_o                                Subst=: P1 D2
Restrictions:  (R-PRIME-RESTR s_α P_o t_α R_o)
Transformation: (pp D2 ss) ==> (P1 ss) (pp D3 ss P1 D2)
```

`SUBST=R` Substitution of Equality. Replaces some occurrences of the right hand side by the left hand side.

```
 (P1)  H      ├ P_o
*(D2)  H      ├ t_α = s_α
 (D3)  H      ├ R_o                                Subst=: P1 D2
Restrictions:  (R-PRIME-RESTR s_α P_o t_α R_o)
Transformation: (pp D2 ss) ==> (P1 ss) (pp D3 ss P1 D2)
```

`SUBST-EQUIV`
Substitution of Equivalence. Useable when R and P are the same modulo the equivalence s EQUIV t.

```
 (P1)  H      ├ P_o
*(D2)  H      ├ s_o ≡ t_o
 (D3)  H      ├ R_o                             Sub-equiv: P1 D2
Restrictions:  (SAME-MODULO-EQUALITY P_o R_o s_o t_o)
Transformation: (pp D2 ss) ==> (P1 ss) (pp D3 ss P1 D2)
```

`SYM=` Rule of Symmetry of Equality.

```
 (P1)  H      ├ A_α = B_α
*(P2)  H      ├ B_α = A_α                              Sym=: P1
Transformation: (P2 ss) ==> (P1 ss)
```

3.10. Definition Rules

`EDEF` Rule to eliminate first definition, left to right.

```
*(D1)  H      ├ A_o
 (D2)  H      ├ '(INST-DEF  A_o)                       Defn: D1
Restrictions:  (CONTAINS-DEFN A_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`EQUIV-WFFS`
Rule to assert equivalence of lines up to definition.

```
*(D1)  H      ├ P_o
 (D2)  H      ├ R_o                              EquivWffs: D1
Restrictions:  (WFFEQ-DEF P_o R_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`IDEF` Rule to introduce a definition.

```
 (P1)  H      ├ '(INST-DEF  A_o)
*(P2)  H      ├ A_o                                    Defn: P1
Restrictions:  (CONTAINS-DEFN A_o)
Transformation: (P2 ss) ==> (P1 ss)
```

3.11. Lambda Conversion Rules

`LAMBDA*` Rule to infer a line from one which is equal up to lambda conversion using both beta and eta rules and alphabetic change of bound variables.

```
*(D1)  H      ⊢ A_o
 (D2)  H      ⊢ B_o                                   Lambda=: D1
Restrictions:  (WFFEQ-AB-LAMBDA A_o B_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`BETA*` Rule to infer a line from one which is equal up to lambda conversion using beta rule (but NOT eta rule) and alphabetic change of bound variables.

```
*(D1)  H      ⊢ A_o

 (D2)  H      ⊢ B_o
                                                    Beta Rule: D1
Restrictions:  (WFFEQ-AB-BETA A_o B_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`ETA*` Rule to infer a line from one which is equal up to lambda conversion using eta rule (but NOT beta rule) and alphabetic change of bound variables.

```
*(D1)  H      ⊢ A_o

 (D2)  H      ⊢ B_o
                                                    Beta Rule: D1
Restrictions:  (WFFEQ-AB-ETA A_o B_o)
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`LCONTR*` Rule to put an inferred line into Lambda-normal form using both beta and eta conversion.

```
*(D1)  H      ⊢ A_o
 (D2)  H      ⊢ ‘(LNORM  A_o)                          Lambda: D1
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`LCONTR*-BETA`
Rule to put an inferred line into beta-normal form.

```
*(D1)  H      ⊢ A_o
 (D2)  H      ⊢ ‘(LNORM-BETA  A_o)                     Lambda: D1
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`LCONTR*-ETA`
Rule to put an inferred line into eta-normal form.

```
*(D1)  H      ⊢ A_o
 (D2)  H      ⊢ ‘(LNORM-ETA  A_o)                      Lambda: D1
Transformation: (pp D1 ss) ==> (pp D2 ss)
```

`LEXPD*` Rule to put a planned line into Lambda-normal form using both beta and eta conversion.

```
 (P1)  H      ⊢ ‘(LNORM  A_o)
*(P2)  H      ⊢ A_o                                    Lambda: P1
Transformation: (P2 ss) ==> (P1 ss)
```

LEXPD*-BETA

Rule to put a planned line into beta-normal form.

```
 (P1)  H      ⊦ ‘(LNORM-BETA  A_o)
*(P2)  H      ⊦ A_o                                     Lambda: P1
Transformation: (P2 ss) ==> (P1 ss)
```

LEXPD*-ETA

Rule to put a planned line into eta-normal form.

```
 (P1)  H      ⊦ ‘(LNORM-ETA  A_o)
*(P2)  H      ⊦ A_o                                     Lambda: P1
Transformation: (P2 ss) ==> (P1 ss)
```

4. Sample Proofs

The following are transcripts of proofs of sample theorems obtained by using `script` before starting **ETPS**. Remarks are added in italics. It may be a good idea to look ahead a little bit, i.e., look at the final proof first to see what we are trying to obtain. You can execute these proof steps on your own computer and use the `PALL` command frequently to get a good picture of how the proof grows, or (if you are running **ETPS** under X-windows or using the Java interface) use the `BEGIN-PRFW` command to open proofwindows, and watch the proof grow in them. As mentioned in Section 1.1, you will probably find it best to set the style flag to xterm and have logical formulas displayed with logical symbols if you can. However, in this chapter we display formulas in generic style.

4.1. Example 1

```
>etps

etps for issar. Version from Saturday, September 23, 1989 at  5:59:15..
(c) Copyrighted 1988 by Carnegie Mellon University. All rights reserved.
****************************************************************************
WARNING -- Be sure that you when you begin ETPS, your current directory is
           one for which you have write access, e.g., your home directory.
****************************************************************************
****************************************************************************
WARNING -- You cannot use the Unix ~ convention in specifying file names.
           Use the full pathname instead, e.g., instead of entering
           "~/foo.work", enter "/afs/andrew/usr11/dn0z/foo.work".
****************************************************************************
****************************************************************************
ANNOUNCING -- ETPS can now be run on the sun3_35 workstation type, as well
              as on the Microvax.
              The more memory on the machine, the faster ETPS will run.  To
              check the amount of memory available on a Sun-3, type
              "/etc/dmesg | grep avail" in your typescript.
****************************************************************************

[Loading changes ...
                 ...done]
Loading /afs/andrew.cmu.edu/math/etps/etps.ini
Finished loading /afs/andrew.cmu.edu/math/etps/etps.ini
```

If you are using ETPS in an environment where proofwindows are available, issue the `BEGIN-PRFW` *command now to open proofwindows.*

```
<1>exercise x2108
(100)     ! FORALL x EXISTS y.P x IMPLIES P y                         PLAN1
```

Since this theorem is universally quantified, we will first use universal generalization. Note that to accept the defaults that **ETPS** *offers, we just hit a <Return>.*

```
<2>ugen
P2 (LINE): Universally Quantified Line [100]>
P1 (LINE): Line with Scope of Universal Quantifier [99]>
(99)      ! EXISTS y.P x IMPLIES P y                                  PLAN2
```

As we will see later, the justification of line `100` *has been changed from* `PLAN1` *to* `UGen`*, which stands for universal generalization. Now the formula we are trying to prove is existentially quantified, so we use the appropriate rule.*

```
<3>egen
P2 (LINE): Existentially Quantified Line [99]>
```

```
P1 (LINE): Line to be Existentially Generalized [98]>
t (GWFF): Term to be Generalized Upon [No Default]>(ED 99)
```

Let's use the editor to pick out the term we want from line 99.

```
<Ed1>p
EXISTS y.P x IMPLIES P y       Here's the current formula.
<Ed2>d
P x IMPLIES P y                We move inside the quantifier.
<Ed3>l d                       We use two commands to get to x.
x
<Ed4>ok                        Return x as the GWFF we were asked for.
(98)      ! P x IMPLIES P x                                    PLAN3
```

All that remains is an easy application of RULEP.

```
<4>rulep
P1 (PLINE): Plan Line [98]>
L (EXISTING-LINELIST): List of Lines [()]>

<5>squeeze
```

SQUEEZE *removes any unnecessary gaps. Now we take a look at the completed proof.*

```
<6>pall

(1)       ! P x IMPLIES P x                                    RuleP
(2)       ! EXISTS y.P x IMPLIES P y                        EGen: x 1
(3)       ! FORALL x EXISTS y.P x IMPLIES P y               UGen: x 2

<7>done
Score file updated.
```

The DONE *command is crucial! Not only does it verify that the proof is complete, it also ensures that you get credit for doing it. Now let's make a nice copy of the proof.*

```
<8>texproof
FILENAME (FILESPEC): Filename [x2108.tex]>
Written file x2108.tex

<9>^Z
```

Let's interrupt **ETPS** *and print the current proof.*

```
>tex x2108
This is TeX, Version 3.14159 (C version 6.1)
(x2108.tex
Hyphenation patterns for english, german, loaded.
(/afs/cs/project/tps/tps/doc/lib/tps.tex) [1] )
Output written on x2108.dvi (1 page, 628 bytes).
Transcript written on x2108.log.

>dvips x2108 -o x2108.ps
This is dvipsk 5.58f Copyright 1986, 1994 Radical Eye Software
' TeX output 1998.09.04:1219' -> x2108.ps
<tex.pro>. [1]

>lpr -Pprinter x2108.ps
```

We'll now resume our **ETPS** *session. The unix command* fg *continues the last job that was interrupted.*

```
>fg
```

Let's now prove this same theorem in a different way, and save it in a new file.

```
<1>stop-save
File x2108.work written.

<2>save-work
SAVEFILE (FILESPEC): SAVE-WORK file [work.work]>"x2108b.work"

<3>exercise x2108
(100)     ! FORALL x EXISTS y.P x IMPLIES P y                          PLAN1

<4>indirect
P3 (LINE): Line to be Proven by Contradiction [100]>
P2 (LINE): Line with Contradiction [99]>
H1 (LINE): Line with Assumed Negation [1]>
(1)   1   ! ~ FORALL x EXISTS y.P x IMPLIES P y              Assume negation
(99)  1   ! FALSEHOOD                                                  PLAN2
```

We can always use indirect proof.
As you can see, line `1` *is negated, so let's push in that negation.*

```
<5>pushneg
D1 (LINE): Line with Negation [1]>
D2 (LINE): Line after Pushing in Negation one Step [2]>
(2)   1   ! EXISTS x.~ EXISTS y.P x IMPLIES P y                       Neg: 1
```

We use the `^P` *command to show the lines which are now relevant.*

```
<6>^p
(2)   1   ! EXISTS x.~ EXISTS y.P x IMPLIES P y                       Neg: 1
               ...
(99)  1   ! FALSEHOOD                                                  PLAN2
```

`RULEC` *is often required when trying to prove a statement from an existentially quantified line. It is probably the most complicated rule you will use, so you might wish to study the description of* `RULEC` *in the previous chapter first, as well the description in the textbook.*

```
<7>rulec
P4 (LINE): Conclusion without Additional Hypothesis [99]>
D1 (LINE): Existentially Quantified Line [2]>
D3 (LINE): Conclusion with Additional Hypothesis [98]>
H2 (LINE): Hypothesis with Chosen Variable [3]>
y (GWFF): Chosen Variable Name [No Default]>"x"
(3)   1,3 ! ~ EXISTS y.P x IMPLIES P y                             Choose: x
(98)  1,3 ! FALSEHOOD                                                  PLAN5
```

The last command created a negated statement, so we can use `PUSHNEG` *again.*

```
<8>pushneg 3 4
(4)   3,1 ! FORALL y.~.P x IMPLIES P y                                Neg: 3

<9>ui
D1 (LINE): Universally Quantified Line [4]>
D2 (LINE): Instantiated Line [5]>
t (GWFF): Substitution Term [No Default]>"x"
(5)   1,3 ! ~.P x IMPLIES P x                                        UI: x 4
```

```
<10>^p
(4)   3,1 ! FORALL y.~.P x IMPLIES P y                               Neg: 3
(5)   1,3 ! ~.P x IMPLIES P x                                       UI: x 4
              ...
(98)  1,3 ! FALSEHOOD                                                 PLAN5
```

Note that line 5 *is a contradiction, so we can use it to justify line* 98 *by* RULEP. *Line* 4 *isn’t necessary for this step.*

```
<1>rulep 98
L (EXISTING-LINELIST): List of Lines [(5 4)]>(5)

<2>squeeze

<3>cleanup
No lines can be deleted.
```

CLEANUP *will remove unnecessary lines and hypotheses from your finished proof.*

```
<4>pall

(1)   1   ! ~ FORALL x EXISTS y.P x IMPLIES P y            Assume negation
(2)   1   ! EXISTS x.~ EXISTS y.P x IMPLIES P y                     Neg: 1
(3)   3   ! ~ EXISTS y.P x IMPLIES P y                           Choose: x
(4)   3   ! FORALL y.~.P x IMPLIES P y                              Neg: 3
(5)   3   ! ~.P x IMPLIES P x                                      UI: x 4
(6)   3   ! FALSEHOOD                                             RuleP: 5
(7)   1   ! FALSEHOOD                                           RuleC: 2 6
(8)       ! FORALL x EXISTS y.P x IMPLIES P y                  Indirect: 7
```

Have we finished?

```
<5>done
Score file updated.
```

Yes. Let’s make a nice copy of this proof. Note that we have to specify a new file name to keep **ETPS** *from overwriting the first file we made.*

```
<6>texproof
FILENAME (FILESPEC): Filename [x2108.tex]>"x2108b"
Written file x2108b.tex
```

If you have open proofwindows, close them now with the command END-PRFW.

```
<7>exit
File x2108b.work written.
```

4.2. Example 2

```
>etps

etps for issar. Version from Saturday, September 23, 1989 at  5:59:15..
(c) Copyrighted 1988 by Carnegie Mellon University. All rights reserved.
****************************************************************************
WARNING -- Be sure that you when you begin ETPS, your current directory is
           one for which you have write access, e.g., your home directory.
****************************************************************************
****************************************************************************
```

```
WARNING -- You cannot use the Unix ~ convention in specifying file names.
           Use the full pathname instead, e.g., instead of entering
           "~/foo.work", enter "/afs/andrew/usr11/dn0z/foo.work".
****************************************************************************
****************************************************************************
ANNOUNCING -- ETPS can now be run on the sun3_35 workstation type, as well
              as on the Microvax.
              The more memory on the machine, the faster ETPS will run.  To
              check the amount of memory available on a Sun-3, type
              "/etc/dmesg | grep avail" in your typescript.
****************************************************************************

[Loading changes ...
                 ...done]
Loading /afs/andrew.cmu.edu/math/etps/etps.ini
Finished loading /afs/andrew.cmu.edu/math/etps/etps.ini

<1>prove
WFF (GWFF0): Prove Wff [No Default]>"exists x forall y P x y implies
forall y exists x P x y"
PREFIX (SYMBOL): Name of the Proof [No Default]>example1
NUM (LINE): Line Number for Theorem [100]>
(100)     !  EXISTS x FORALL y P x y IMPLIES FORALL y EXISTS x P x y   PLAN1
```

If we were trying to prove one of the exercises in the text, we would have used `EXERCISE` *instead of* `prove`.

Note that `EXISTS` *(for example) was typed in lower case, but is always printed in upper case.*

```
<2>deduct
P3 (LINE): Line with Implication [100]>
D2 (LINE): Line with Conclusion [99]>
H1 (LINE): Line with Hypothesis [1]>
(1)   1   !  EXISTS x FORALL y P x y                                      Hyp
(99)  1   !  FORALL y EXISTS x P x y                                    PLAN2
```

`DEDUCT` *is often the right way to start the proof of an implication. Note that the defaults were just what we wanted anyway, so we selected them by simply typing* `<Return>`.

```
<3>ugen
P2 (LINE): Universally Quantified Line [99]>
P1 (LINE): Line with Scope of Universal Quantifier [98]>
(98)  1   !  EXISTS x P x y                                             PLAN3

<4>rulec
P4 (LINE): Conclusion without Additional Hypothesis [98]>
D1 (LINE): Existentially Quantified Line [1]>
D3 (LINE): Conclusion with Additional Hypothesis [97]>
H2 (LINE): Hypothesis with Chosen Variable [2]>
y (GWFF): Chosen Variable Name [No Default]>"x"
(2)   1,2 !  FORALL y P x y                                         Choose: x
(97)  1,2 !  EXISTS x P x y                                             PLAN5
```

We now do a `^P` *(note that this is not a control-character) to see what still has to be proven.*

```
<5>^P
(2)   1,2 !  FORALL y P x y                                         Choose: x
               ...
(97)  1,2 !  EXISTS x P x y                                             PLAN5
```

```
<6>ui 2
D2 (LINE): Instantiated Line [3]>
t (GWFF): Substitution Term [No Default]>"y"
(3)   2,1 !  P x y                                                UI: y 2
```

We are closing in. You can now see that line 97 *follows immediately from line* 3 *by existential generalization. Therefore we use the* EGEN *command, and display the proof with the* PALL *command.*

```
<7>egen
P2 (LINE): Existentially Quantified Line [97]>
P1 (LINE): Line to be Existentially Generalized [96]>3
t (GWFF): Term to be Generalized Upon [No Default]>"x"

<8>pall

(1)   1   !  EXISTS x FORALL y P x y                                     Hyp
(2)   1,2 !  FORALL y P x y                                        Choose: x
(3)   2,1 !  P x y                                                   UI: y 2
(97)  1,2 !  EXISTS x P x y                                        EGen: x 3
(98)  1   !  EXISTS x P x y                                       RuleC: 1 97
(99)  1   !  FORALL y EXISTS x P x y                               UGen: y 98
(100)     !  EXISTS x FORALL y P x y IMPLIES FORALL y EXISTS x P x y
                                                                   Deduct: 99
```

This is what our completed proof looks like. Let’s make sure that we are done and print the proof into a file before exiting **ETPS***.*

```
<9>done
You completed the proof.  Since this is not an assigned exercise,
the score file will not be updated.

<10>texproof
FILENAME (FILESPEC): Filename [example1.tex]>
Written file example1.tex

<10>exit
```

4.3. Example 3

```
>etps

etps for issar. Version from Saturday, September 23, 1989 at  5:59:15..
(c) Copyrighted 1988 by Carnegie Mellon University. All rights reserved.
****************************************************************************
WARNING -- Be sure that you when you begin ETPS, your current directory is
           one for which you have write access, e.g., your home directory.
****************************************************************************
****************************************************************************
WARNING -- You cannot use the Unix ~ convention in specifying file names.
           Use the full pathname instead, e.g., instead of entering
           "~/foo.work", enter "/afs/andrew/usr11/dn0z/foo.work".
****************************************************************************
****************************************************************************
ANNOUNCING -- ETPS can now be run on the sun3_35 workstation type, as well
              as on the Microvax.
              The more memory on the machine, the faster ETPS will run.  To
              check the amount of memory available on a Sun-3, type
              "/etc/dmesg | grep avail" in your typescript.
```

```
*****************************************************************************

[Loading changes ...
                 ...done]
Loading /afs/andrew.cmu.edu/math/etps/etps.ini
Finished loading /afs/andrew.cmu.edu/math/etps/etps.ini

<1>prove
WFF (GWFF0): Prove Wff [No Default]>"forall x [forall y P x y implies Q x x]
implies. forall z [P a z and P b z] implies . Q a a and Q b b"
PREFIX (SYMBOL): Name of the Proof [No Default]>example2
NUM (LINE): Line Number for Theorem [100]>
(100)     !          FORALL x [FORALL y P x y IMPLIES Q x x]
                 IMPLIES.FORALL z [P a z AND P b z] IMPLIES Q a a AND Q b b
                                                                      PLAN1
```

This example does not involve an existential quantifier, but has a more complicated structure. Since our theorem is an implication, we use the deduction theorem again right away.

```
<2>deduct
P3 (LINE): Line with Implication [100]>!
(1)   1   !  FORALL x.FORALL y P x y IMPLIES Q x x                     Hyp
(99)  1   !  FORALL z [P a z AND P b z] IMPLIES Q a a AND Q b b      PLAN2
```

Note that we used `!` *to specify that we want to choose the defaults for the remaining arguments.*

It is clear that we need to instantiate `x` *with* `a` *and* `b`*. We do this in the next two steps.*

```
<3>ui
D1 (LINE): Universally Quantified Line [1]>!
Some defaults could not be determined.
t (GWFF): Substitution Term [No Default]>"a"
(2)   1   !  FORALL y P a y IMPLIES Q a a                          UI: a 1
```

We again used `!`*, but* **ETPS** *couldn't determine all the defaults, so it prompted us again for the arguments for which it couldn't figure the defaults.*

```
<4>ui
D1 (LINE): Universally Quantified Line [2]>1
D2 (LINE): Instantiated Line [3]>
t (GWFF): Substitution Term [No Default]>"b"
(3)   1   !  FORALL y P b y IMPLIES Q b b                          UI: b 1
```

The planned line `99` *is again an implication, which suggests using* `DEDUCT` *again.*

```
<5>deduct
P3 (LINE): Line with Implication [99]>
D2 (LINE): Line with Conclusion [98]>
H1 (LINE): Line with Hypothesis [4]>
(4)   1,4 !  FORALL z.P a z AND P b z                                  Hyp
(98)  1,4 !  Q a a AND Q b b                                         PLAN5
```

We now use universal instantiation again, this time to distribute the universal quantifier over a conjunction.

```
<6>ui
D1 (LINE): Universally Quantified Line [4]>
D2 (LINE): Instantiated Line [5]>
```

```
t (GWFF): Substitution Term [No Default]>"y"
(5)   4,1 !  P a y AND P b y                                        UI: y 4
```

Just to make sure `ECONJ` *is the inference rule we want, let’s call the* `HELP` *command.*

```
<7>help econj
*(D1) H   !A AND B
 (D2) H   !A                               Conj: D1
 (D3) H   !B                               Conj: D1
Transformation: (pp D1 ss) ==> (pp D2 D3 ss)
```

Yes, that’s what we need.

```
<8>econj
D1 (LINE): Line with Conjunction [5]>
D3 (LINE): Line with Right Conjunct [7]>
D2 (LINE): Line with Left Conjunct [6]>
(6)   1,4 !  P a y                                                  Conj: 5
(7)   1,4 !  P b y                                                  Conj: 5
```

Let’s look at the current planned line and its support lines.

```
<9>pplan
PLINE (PLINE): Print planned line [98]>
(1)   1   !  FORALL x.FORALL y P x y IMPLIES Q x x                      Hyp
(2)   1   !  FORALL y P a y IMPLIES Q a a                           UI: a 1
(3)   1   !  FORALL y P b y IMPLIES Q b b                           UI: b 1
(4)   1,4 !  FORALL z.P a z AND P b z                                   Hyp
(6)   1,4 !  P a y                                                  Conj: 5
(7)   1,4 !  P b y                                                  Conj: 5
               ...
(98)  1,4 !  Q a a AND Q b b                                          PLAN5

<10>iconj
P3 (LINE): Line with Conjunction [98]>
P2 (LINE): Line with Right Conjunct [97]>
P1 (LINE): Line with Left Conjunct [52]>
(52)  1,4 !  Q a a                                                    PLAN9
(97)  1,4 !  Q b b                                                    PLAN8

<1>mp
D2 (LINE): Line with Implication [6]>2
D3 (LINE): Line with Succedent of Implication [30]>52
P1 (LINE): Line with Antecedent of Implication [29]>
(29)  1,4 !  FORALL y P a y                                          PLAN11

<2>pplan
PLINE (PLINE): Print planned line [29]>
(1)   1   !  FORALL x.FORALL y P x y IMPLIES Q x x                      Hyp
(3)   1   !  FORALL y P b y IMPLIES Q b b                           UI: b 1
(4)   1,4 !  FORALL z.P a z AND P b z                                   Hyp
(6)   1,4 !  P a y                                                  Conj: 5
(7)   1,4 !  P b y                                                  Conj: 5
               ...
(29)  1,4 !  FORALL y P a y                                          PLAN11

<3>ugen
P2 (LINE): Universally Quantified Line [29]>
P1 (LINE): Line with Scope of Universal Quantifier [28]>6

<4>pplan
PLINE (PLINE): Print planned line [97]>
```

```
(1)   1   !  FORALL x.FORALL y P x y IMPLIES Q x x                   Hyp
(3)   1   !  FORALL y P b y IMPLIES Q b b                         UI: b 1
(4)   1,4 !  FORALL z.P a z AND P b z                                Hyp
(6)   1,4 !  P a y                                               Conj: 5
(7)   1,4 !  P b y                                               Conj: 5
(29)  1,4 !  FORALL y P a y                                     UGen: y 6
(52)  1,4 !  Q a a                                              MP: 29 2
               ...
(97)  1,4 !  Q b b                                                 PLAN8

<5>mp
D2 (LINE): Line with Implication [52]>3
D3 (LINE): Line with Succedent of Implication [75]>97
P1 (LINE): Line with Antecedent of Implication [74]>
(74)  1,4 !  FORALL y P b y                                       PLAN14

<6>ugen
P2 (LINE): Universally Quantified Line [74]>
P1 (LINE): Line with Scope of Universal Quantifier [73]>7
```

The proof is complete. Let's look at the entire proof.

```
<7>pall
(1)   1   !  FORALL x.FORALL y P x y IMPLIES Q x x                   Hyp
(2)   1   !  FORALL y P a y IMPLIES Q a a                         UI: a 1
(3)   1   !  FORALL y P b y IMPLIES Q b b                         UI: b 1
(4)   1,4 !  FORALL z.P a z AND P b z                                Hyp
(5)   4,1 !  P a y AND P b y                                      UI: y 4
(6)   1,4 !  P a y                                               Conj: 5
(7)   1,4 !  P b y                                               Conj: 5
(29)  1,4 !  FORALL y P a y                                     UGen: y 6
(52)  1,4 !  Q a a                                              MP: 29 2
(74)  1,4 !  FORALL y P b y                                     UGen: y 7
(97)  1,4 !  Q b b                                              MP: 74 3
(98)  1,4 !  Q a a AND Q b b                                  Conj: 52 97
(99)  1   !  FORALL z [P a z AND P b z] IMPLIES Q a a AND Q b b   Deduct: 98
(100)     !          FORALL x [FORALL y P x y IMPLIES Q x x]
              IMPLIES.FORALL z [P a z AND P b z] IMPLIES Q a a AND Q b b
                                                               Deduct: 99
```

We'll next use `SQUEEZE` *to remove gaps from the proof structure.*
We could also have used `MODIFY-GAPS` *1 1).*

```
<9>squeeze
```

Let's see how the proof looks now.

```
<10>pall

(1)   1   !  FORALL x.FORALL y P x y IMPLIES Q x x                   Hyp
(2)   1   !  FORALL y P a y IMPLIES Q a a                         UI: a 1
(3)   1   !  FORALL y P b y IMPLIES Q b b                         UI: b 1
(4)   1,4 !  FORALL z.P a z AND P b z                                Hyp
(5)   4,1 !  P a y AND P b y                                      UI: y 4
(6)   1,4 !  P a y                                               Conj: 5
(7)   1,4 !  P b y                                               Conj: 5
(8)   1,4 !  FORALL y P a y                                     UGen: y 6
(9)   1,4 !  Q a a                                               MP: 8 2
(10)  1,4 !  FORALL y P b y                                     UGen: y 7
(11)  1,4 !  Q b b                                              MP: 10 3
(12)  1,4 !  Q a a AND Q b b                                   Conj: 9 11
(13)  1   !  FORALL z [P a z AND P b z] IMPLIES Q a a AND Q b b   Deduct: 12
(14)      !          FORALL x [FORALL y P x y IMPLIES Q x x]
```

```
              IMPLIES.FORALL z [P a z AND P b z] IMPLIES Q a a AND Q b b
                                                             Deduct: 13

<1>done
You completed the proof.  Since this is not an assigned exercise,
the score file will not be updated.
```

The proof is complete. Let’s print it into a file, so we can print it later

```
<2>printproof
FILENAME (FILESPEC): Filename [example2.prt]>
Written file example2.prt

<3>exit
```

5. Type Theory in ETPS

5.1. Using ETPS for Type Theory

ETPS can be used for higher-order logic as well as for first-order logic. Wffs of type theory are written essentially as they are expressed in the logic book. There are a few additional inference rules and the parsing and printing of wffs is slightly different, while everything else described in the previous chapters is still valid.

5.1.1. Types in Higher-Order Logic

There is a very direct translation from the way types are represented in the logic book and the way types are represented in **ETPS**. Since Greek subscripts are not available on most terminals, the Greek letters are transliterated to uppercase Roman letters. The most commonly used types are

`I` for ι	`O` for o	`S` for σ
`A` for α	`B` for β	`C` for γ

The same conventions for parentheses are used as in the logic book, i.e., association to the left is assumed. Note, however, that the outermost pair of parentheses must be preserved in order to distinguish types from identifiers.

Types are entered as strings, such as "`(O(OA))`"; typically they are substrings of a string representing a wff and serve to give type information about the symbols in that wff, e.g., "`p(O(OA))`". If entered separately, the opening and closing double-quotes must still be provided. Indeed, all of the string input rules apply; for example, carriage returns may be embedded. For more examples of entering typed wffs, see Section 5.1.3.

ETPS has a powerful type-inference mechanism which makes explicit typing mostly unnecessary within wffs. Often the type of one variable is enough to uniquely determine the type of every identifier in a wff. Within a wff, all occurrences of a variable are assumed to have the same type, unless the contrary is specifically indicated. If the type of a variable remains undetermined after all other type information has been used, ι is assumed. Take care to specify types if you use ‘‘type variables’’ like α. Also note that type-inference is local, i.e., the type of an identifier is determined anew for each wff parsed by **ETPS**.

5.1.2. Abbreviations and Special Symbols

ETPS allows polymorphic abbreviations. These are abbreviations with variable type, which may have multiple occurrences with different types in the same wff. Since their special symbols cannot be typed on most keyboards, there is a ‘‘long’’ form of each of them, which has to be used in **ETPS**. The following is a temporary list of special symbols, the binary operators ordered according to their binding priority and abbreviations marked with *(abb)*.

Improper symbols

λ	`LAMBDA`	The λ-binder
$\forall$	`FORALL`	
$\exists$	`EXISTS`	
$\exists_1$	`EXISTS1`	$\Sigma^1_{o(o\alpha)}.\lambda x_\alpha\ A_o$
	`EXISTSN`	$\exists z_\sigma.\mathrm{NAT}\ z \wedge A_o$
	`FORALLN`	$\forall z_\sigma.\mathrm{NAT}\ z \supset A_o$
μ	`MU-BIND`	$mu_{\sigma(o\sigma)}.\lambda z_\sigma\ A_o$
	`THAT`	$\iota.\lambda z_\chi\ A_o$

Unary operators with equal binding priority (except `NOT` which has the least binding priority):

~	`NOT`

% *(abb)*	%	$\lambda f_{\alpha\beta}\ \lambda x_{o\beta}\ \lambda z_{\alpha}\ \exists t_{\beta}.x\ t \land z = f\ t.$
$\wp_{o(o\alpha)(o\alpha)}$ *(abb)*	POWERSET	$\lambda p_{o\alpha}\lambda q_{o\alpha}.q \subseteq p$
$\bigcap$ *(abb)*	SETINTERSECT	$\lambda s_{o(o\alpha)}\lambda x_{\alpha}\forall\ p_{o\alpha}.s\ p \supset p\ x$ Intersection of a collection of sets
$\bigcup$ *(abb)*	SETUNION	$\lambda s_{o(o\alpha)}\lambda x_{\alpha}\exists p_{o\alpha}.s\ p \land p\ x$ Union of a collection of sets
$\bot$	FALSEHOOD	
T	TRUTH	

Binary operators, strongest binding first:

$\cap$ *(abb)*	INTERSECT	$\lambda p_{o\alpha}\lambda q_{o\alpha}\lambda x_{\alpha}.p\ x \land q\ x$ Intersection of two sets
$\cup$ *(abb)*	UNION	$\lambda p_{o\alpha}\lambda q_{o\alpha}\lambda x_{\alpha}.p\ x \lor q\ x$ Union of two sets
$\subseteq$ *(abb)*	SUBSET	$\lambda p_{o\alpha}\lambda q_{o\alpha}\forall\ x_{\alpha}.p\ x \supset q\ x$
$=_{o\alpha\alpha}$	=	Equality at type α
=S *(abb)*	SETEQUIV	$\lambda p_{o\alpha}\lambda q_{o\alpha}.p \subseteq q \land .\ q \subseteq p$ Equality between sets.
=S *(abb)*	EQUIVS $\lambda P_{o\alpha}\ \lambda R_{o\alpha}\ \forall x_{\alpha}.P\ x \equiv R\ x$	Elementwise equality between sets. This is equivalent to equality between sets, if one assumes extensionality.
<= *(abb)*	<=	$\lambda x_{\sigma}\ \lambda y_{\sigma}\ \forall p_{o\sigma}.p\ x \land \forall z_{\sigma}\ [p\ z \supset p.\mathrm{SUCC}_{\sigma\sigma}\ z] \supset p\ y$ Less than or equal to, for natural numbers.
$\land$	AND	
$\lor$	OR	
$\supset$	IMPLIES	
$\equiv$ *(abb)*	EQUIV	Equality at type o

Other abbreviations:

Conditional	COND $\lambda x_{\chi}\ \lambda y_{\chi}\ \lambda p_{o}\ \mathrm{THAT}\ q_{\chi}.p \land x = q \lor \sim p \land y = q$	
Equipollence	EQP $\lambda p_{o\beta}\lambda q_{o\alpha}\exists s_{\alpha\beta}.\forall x_{\beta}[p\ x \supset q.s\ x] \land \forall y_{\alpha}.q\ y \supset \exists_1 x_{\beta}.p\ x \land y = s\ x$	
Zero	ZERO	$[\lambda p_{o\iota}.\sim\exists x_{\iota} p\ x]$

Successor	SUCC	$\lambda n_{o(o\iota)} \lambda p_{o\iota} \exists x_{\iota} . p\ x \land n[\lambda t_{\iota} . t = x \land p\ t]$
One	ONE	$SUCC_{\sigma\sigma} 0_{\sigma}$
Finite	FINITE	$\lambda p_{o\iota}\ \exists n_{o(o\iota)} . NAT\ n \land n\ p$
μ	MU	$\lambda p_{o\sigma}\ THAT\ x_{\sigma} . NAT\ x \land p\ x \land FORALLN\ y_{\sigma} . p\ y \supset x <= y$
Natural No.	NAT	$\lambda n_{o(o\iota)}\ \forall p_{o\sigma} . p\ ZERO_{\sigma} \land \forall x_{\sigma}\ [p\ x \supset p . SUCC_{\sigma\sigma}\ x] \supset p\ n$
NC	NC	$\lambda u_{o(o\beta)}\ \exists p_{o\beta} . u = \mathcal{E}_{o(o\beta)(o\beta)}\ p$
Recursion	RECURSION	$\lambda h_{\sigma\sigma\sigma}\ \lambda g_{\sigma}\ \lambda n_{o(o\iota)}\ THAT\ m_{\sigma}\ \forall w_{o\sigma\sigma} . w\ ZERO_{\sigma}\ g \land \forall x_{\sigma}\ \forall y_{\sigma}\ [w\ x\ y \supset w\ [SUCC_{\sigma\sigma}\ x] . h\ x\ y] \supset w\ n\ m$
Σ^1	SIGMA1	$[\lambda p_{o\alpha} \exists y_{\alpha} . p_{o\alpha} = .= y]$
U	UNITSET	$\lambda x_{\alpha}\ \lambda y_{\alpha} . x = y$

5.1.3. Some Examples of Higher-Order Wffs

Here are some examples of higher-order wffs. The first line shows how the formula is printed in the logic book, the second line shows how it could be entered into **ETPS** as a string, and the third line shows how **ETPS** would print it with type symbols (for example with the PWTYPES command). Look at these carefully and make sure you understand how to type in wffs of higher-order logic.

$\exists f_{\iota(o\iota)} \forall S_{o\iota} . \exists x_{\iota} [S\ x] \supset S . f\ S$

```
"EXISTS f FORALL S. EXISTS x S x IMPLIES S . f S"
EXISTS f(I(OI)) FORALL S(OI).EXISTS x(I)[S x] IMPLIES S.f S
```

$\exists f_{o\iota(o(o\iota))} \forall S_{o(o\iota)} . \exists x_{o\iota} [S\ x] \supset S . f\ S$

```
"EXISTS f(OI(O(OI))) FORALL S. EXISTS x S x IMPLIES S.f S"
EXISTS f(OI(O(OI))) FORALL S(O(OI)).EXISTS x(OI)[S x] IMPLIES S.f S
```

$\exists f_{\alpha(o\alpha)} \forall S_{o\alpha} . \exists x_{\alpha} [S\ x] \supset S . f\ S$

```
"EXISTS f FORALL S. EXISTS x(A) S x IMPLIES S.f S"
EXISTS f(A(OA)) FORALL S(OA).EXISTS x(A)[S x] IMPLIES S.f S
```

$p_{o} \equiv q_{o} \equiv [\lambda r_{o} \lambda s_{o} . [r \supset s] \land . s \supset r]\ p\ q$

```
"[p EQUIV q] EQUIV . [LAMBDA r LAMBDA s . [r IMPLIES s] AND .s IMPLIES r] p q"
      p(O) EQUIV q(O)
 EQUIV[LAMBDA r(O) LAMBDA s(O).[r IMPLIES s] AND.s IMPLIES r] p q
```

$\forall A_{o} \exists f_{o\iota} \forall P_{o(o\iota)} . P[\lambda p_{\iota}\ A] \equiv P\ f$

```
"FORALL A(O) EXISTS f FORALL P . P [LAMBDA p A] EQUIV P f"
FORALL A(O) EXISTS f(OI) FORALL P(O(OI)).P[LAMBDA p(I) A] EQUIV P f
```

$\forall A_{o} \exists f_{oo} \forall P_{o(oo)} . P[\lambda p_{o}\ A] \equiv P\ f$

```
"FORALL A(O) EXISTS f(OO) FORALL P. P[LAMBDA p A] EQUIV P f"
FORALL A(O) EXISTS f(OO) FORALL P(O(OO)).P[LAMBDA p(O) A] EQUIV P f
```

5.2. Example of a Proof of a Higher-Order Theorem

The following is an annotated transcript of part of a proof in type theory. Basic familiarity with **ETPS** is assumed.

```
>etps

etps for issar. Version from Saturday, September 23, 1989 at  5:59:15..
(c) Copyrighted 1988 by Carnegie Mellon University. All rights reserved.
****************************************************************************
WARNING -- Be sure that you when you begin ETPS, your current directory is
           one for which you have write access, e.g., your home directory.
****************************************************************************
****************************************************************************
WARNING -- You cannot use the Unix ~ convention in specifying file names.
           Use the full pathname instead, e.g., instead of entering
           "~/foo.work", enter "/afs/andrew/usr11/dn0z/foo.work".
****************************************************************************
****************************************************************************
ANNOUNCING -- ETPS can now be run on the sun3_35 workstation type, as well
              as on the Microvax.
              The more memory on the machine, the faster ETPS will run.  To
              check the amount of memory available on a Sun-3, type
              "/etc/dmesg | grep avail" in your typescript.
****************************************************************************

[Loading changes ...
                 ...done]
Loading /afs/andrew.cmu.edu/math/etps/etps.ini
Finished loading /afs/andrew.cmu.edu/math/etps/etps.ini

<1>Exercise X5209
(100)        !    POWERSET [D(OA) INTERSECT E(OA)]
                 = POWERSET D INTERSECT POWERSET E                     PLAN1

<2>ext=
P2 (LINE): Line with Equality [100]>
P1 (LINE): Universally Quantified Equality [99]>
x (GWFF): Universally Quantified Variable [No Default]>"S(OA)"
(99)         !  FORALL S(OA).  POWERSET [D(OA) INTERSECT E(OA)] S
                              = [POWERSET D INTERSECT POWERSET E] S    PLAN2

<3>ugen !
(98)         !    POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                 = [POWERSET D INTERSECT POWERSET E] S                 PLAN3

<4>ext=0
P2 (LINE): Line with Equality [98]>
P1 (LINE): Line with Equivalence [97]>
(97)         !        POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                 EQUIV [POWERSET D INTERSECT POWERSET E] S             PLAN4

<5>implics-equiv
P2 (LINE): Line with Equivalence [97]>
P1 (LINE): Line with Implications in Both Directions [96]>
(96)         !      [        POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                        IMPLIES [POWERSET D INTERSECT POWERSET E] S]
                 AND.        [POWERSET D INTERSECT POWERSET E] S
                        IMPLIES POWERSET [D INTERSECT E] S             PLAN5
```

```
<6>iconj !
(48)        !         POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                 IMPLIES [POWERSET D INTERSECT POWERSET E] S           PLAN7
(95)        !         [POWERSET D(OA) INTERSECT POWERSET E(OA)] S(OA)
                 IMPLIES POWERSET [D INTERSECT E] S                    PLAN6
```

In this example we will prove only line 95*. It may be a a good exercise to try to prove line* 48*.*

```
<7>subproof
PLINE (PLINE): Line to prove [48]>95

<8>deduct !
(49)  49    !  [POWERSET D(OA) INTERSECT POWERSET E(OA)] S(OA)          Hyp
(94)  49    !  POWERSET [D(OA) INTERSECT E(OA)] S(OA)                 PLAN8
```

Now we eliminate the POWERSET*.* S *is in the powerset of* D ∩ E *iff* S *is a subset of* D ∩ E*.*

```
<9>idef !
(93)  49    !  S(OA) SUBSET D(OA) INTERSECT E(OA)                     PLAN9
```

Now we eliminate the INTERSECT *from the justified line* 49*. We therefore have to use the command symmetric to* IDEF*, which is* EDEF*.*

```
<10>edef !
(50)  49    !  POWERSET D(OA) S(OA) AND POWERSET E(OA) S           Defn: 49

<1>^P
(50)  49    !  POWERSET D(OA) S(OA) AND POWERSET E(OA) S           Defn: 49
              ...
(93)  49    !  S(OA) SUBSET D(OA) INTERSECT E(OA)                     PLAN9

<2>econj !
(51)  49    !  POWERSET D(OA) S(OA)                                Conj: 50
(52)  49    !  POWERSET E(OA) S(OA)                                Conj: 50
```

From here on we go through a sequence of routine elimination of definitions.

```
<3>edef !
(53)  49    !  S(OA) SUBSET D(OA)                                  Defn: 51

<4>edef
D1 (LINE): Line with Definition [53]>52
D2 (LINE): Line with Instantiated Definition [54]>
(54)  49    !  S(OA) SUBSET E(OA)                                  Defn: 52

<5>^P
(53)  49    !  S(OA) SUBSET D(OA)                                  Defn: 51
(54)  49    !  S(OA) SUBSET E(OA)                                  Defn: 52
              ...
(93)  49    !  S(OA) SUBSET D(OA) INTERSECT E(OA)                     PLAN9
```

We are on the right track! From S ⊆ D *and* S ⊆ E *we should be able to infer that* S ⊆ D ∩ E

```
<6>edef 53
D2 (LINE): Line with Instantiated Definition [55]>
(55)  49    !  FORALL x(A).S(OA) x IMPLIES D(OA) x                 Defn: 53

<7>edef 54
D2 (LINE): Line with Instantiated Definition [56]>
```

```
(56)  49     !  FORALL x(A).S(OA) x IMPLIES E(OA) x                Defn: 54

<8>idef !
(92)  49     !  FORALL x(A).S(OA) x IMPLIES [D(OA) INTERSECT E(OA)] x PLAN16

<9>^P
(55)  49     !  FORALL x(A).S(OA) x IMPLIES D(OA) x                Defn: 53
(56)  49     !  FORALL x(A).S(OA) x IMPLIES E(OA) x                Defn: 54
               ...
(92)  49     !  FORALL x(A).S(OA) x IMPLIES [D(OA) INTERSECT E(OA)] x PLAN16
```

We have to get rid of the universal quantifier, but we have to be careful to give our variable the right type, namely α.

```
<10>ui 55
D2 (LINE): Instantiated Line [57]>
t (GWFF): Substitution Term [No Default]>"x(A)"
(57)  49     !  S(OA) x(A) IMPLIES D(OA) x                         UI: x 55

<1>ui 56
D2 (LINE): Instantiated Line [58]>
```

Let’s use the editor to extract the variable.

```
t (GWFF): Substitution Term [No Default]>(ed 56)

<Ed1>a
x(A)
<Ed2>ok
(58)  49     !  S(OA) x(A) IMPLIES E(OA) x                         UI: x 56

<2>ugen !
(91)  49     !  S(OA) x(A) IMPLIES [D(OA) INTERSECT E(OA)] x         PLAN19

<3>deduct !
(59)  49,59  !  S(OA) x(A)                                               Hyp
(90)  49,59  !  [D(OA) INTERSECT E(OA)] x(A)                          PLAN20

<4>idef !
(89)  49,59  !  D(OA) x(A) AND E(OA) x                                PLAN21

<5>^P
(55)  49     !  FORALL x(A).S(OA) x IMPLIES D(OA) x                Defn: 53
(56)  49     !  FORALL x(A).S(OA) x IMPLIES E(OA) x                Defn: 54
(57)  49     !  S(OA) x(A) IMPLIES D(OA) x                         UI: x 55
(58)  49     !  S(OA) x(A) IMPLIES E(OA) x                         UI: x 56
(59)  49,59  !  S(OA) x(A)                                               Hyp
               ...
(89)  49,59  !  D(OA) x(A) AND E(OA) x                                PLAN21
```

We don’t need the universally quantified sponsoring lines any more in order to prove line 89*, so let’s use* UNSPONSOR.

```
<6>unsponsor
PLINE (PLINE):Planned line [89]>
LINELIST (EXISTING-LINELIST): Sponsoring lines [(59 58 56 57 55)]>(55 56)

<7>rulep !

<8>pall


               ...
(48)         !          POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                 IMPLIES [POWERSET D INTERSECT POWERSET E] S          PLAN7
(49)  49     !  [POWERSET D(OA) INTERSECT POWERSET E(OA)] S(OA)          Hyp
```

```
(50)  49     !  POWERSET D(OA) S(OA) AND POWERSET E(OA) S           Defn: 49
(51)  49     !  POWERSET D(OA) S(OA)                                Conj: 50
(52)  49     !  POWERSET E(OA) S(OA)                                Conj: 50
(53)  49     !  S(OA) SUBSET D(OA)                                  Defn: 51
(54)  49     !  S(OA) SUBSET E(OA)                                  Defn: 52
(55)  49     !  FORALL x(A).S(OA) x IMPLIES D(OA) x                 Defn: 53
(56)  49     !  FORALL x(A).S(OA) x IMPLIES E(OA) x                 Defn: 54
(57)  49     !  S(OA) x(A) IMPLIES D(OA) x                          UI: x 55
(58)  49     !  S(OA) x(A) IMPLIES E(OA) x                          UI: x 56
(59)  49,59  !  S(OA) x(A)                                               Hyp
(89)  49,59  !  D(OA) x(A) AND E(OA) x                      Rulep: 59 58 57
(90)  49,59  !  [D(OA) INTERSECT E(OA)] x(A)                        Defn: 89
(91)  49     !  S(OA) x(A) IMPLIES [D(OA) INTERSECT E(OA)] x      Deduct: 90
(92)  49     !  FORALL x(A).S(OA) x IMPLIES [D(OA) INTERSECT E(OA)] x
                                                                  Ugen: x 91
(93)  49     !  S(OA) SUBSET D(OA) INTERSECT E(OA)                  Defn: 92
(94)  49     !  POWERSET [D(OA) INTERSECT E(OA)] S(OA)              Defn: 93
(95)         !          [POWERSET D(OA) INTERSECT POWERSET E(OA)] S(OA)
                 IMPLIES POWERSET [D INTERSECT E] S               Deduct: 94
(96)         !     [        POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                      IMPLIES [POWERSET D INTERSECT POWERSET E] S]
                 AND.        [POWERSET D INTERSECT POWERSET E] S
                      IMPLIES POWERSET [D INTERSECT E] S        Conj: 48 95
(97)         !        POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                 EQUIV [POWERSET D INTERSECT POWERSET E] S      ImpEquiv: 96
(98)         !    POWERSET [D(OA) INTERSECT E(OA)] S(OA)
                 = [POWERSET D INTERSECT POWERSET E] S              Ext=: 97
(99)         !  FORALL S(OA).  POWERSET [D(OA) INTERSECT E(OA)] S
                              = [POWERSET D INTERSECT POWERSET E] S
                                                                  Ugen: S 98
(100)        !    POWERSET [D(OA) INTERSECT E(OA)]
                 = POWERSET D INTERSECT POWERSET E                  Ext=: 99

<9>exit
```

Appendix I
Amenities

.

ETPS incorporates several features of the Unix C-shell (csh) top-level. These features include various control characters, command sequences, a history mechanism, and aliases.

I.1. Control characters for Unix

If you are running **ETPS** under Unix (or Linux), you may be able to use the following control characters in addition to those discussed in Section 1.4.

`Ctrl-S`	Freeze output.
`Ctrl-Q`	Proceed with output.
`Ctrl-Z`	Suspend the current program (**ETPS**), and return to the monitor.
`Ctrl-R`	Redisplay the current input line.

I.2. Command Sequences

You may enter a series of commands on the same command line by using the ampersand (&) as a separator. This is analogous to the C-shell's use of the semicolon (`;`). That is, entering

<0> command$_1$ & command$_2$ & ... & command$_n$

will cause **ETPS** to sequentially execute *command$_1$* through *command$_n$* as though you had typed them in one at a time.

For example, after you have finished a proof, you may want to enter the sequence:

```
<0> cleanup & squeeze & done & texproof !
```

I.3. History Substitutions

It is often convenient to be able to refer to commands and arguments that you have already typed. As in the C-shell, the exclamation point (`!`) is used to indicate a history substitution, with two exceptions. An exclamation point that is followed by whitespace will not be interpreted as a history reference, nor will an exclamation point that is immediately preceded by a a backslash (\). Any input line that contains history substitutions will, before execution, be echoed on the screen as it would appear without the history references.

In **ETPS**, each command line is given a unique number; this number is part of the top-level prompts. A certain number of previous commands are saved by **ETPS**; the number saved is determined by the flag HISTORY-SIZE. The previous command is always saved. In addition, each line is parsed into a series of *tokens*. Unlike the C-shell, these tokens are not distinguished simply by surrounding whitespace, but rather by their Lisp syntax. All that the user needs to know is that, in general, each argument entered on a command line will be considered a separate token. On each input line, the tokens are numbered from left to right, beginning at 0. For example, the input line

```
<n> rulep 27 (1 2 7 14)
```

would be parsed into three tokens: `rulep`, `27` and `(1 2 7 14)`, which would be numbered 0, 1 and 2, respectively.

The `HISTORY` command is used to examine the list of input lines that have been saved by **ETPS**. It takes two arguments, the first being the number of lines to show (defaulting to the entire list), and the second being whether to show them in reverse numerical order (defaulting to no). The number of each input line is also given. The lines are saved in the history list as they appear after all history substitutions are made.

Previous lines can be referred to using the following input line specifiers:

`!n`	the command line whose number is `n`.
`!-n`	the input line that was entered `n` lines above the current one.
`!!`	the previous line.
`!`*str*	the most recent command that begins with the string *str*.
`!?`*str*`?`	the most recent command that has a token containing the substring *str*.

Here are some examples. Suppose we had the following output from the `HISTORY` command

```
<10> history 5 !
    6 exercise x2106
    7 pstatus
    8 ^p
    9 pall
   10 history 5 !
```

Then, as input line 11, we could refer to line 7 by `!ps` or `!-4`, or even by `!?tat?`.

Used alone, the above references merely insert every token of the line referred to into the the current input line. In order to select particular tokens from an input line, a colon (`:`) follows the input line specifier, along with a selector for the intended tokens. Here is the syntax for the token selectors, where *x* and *y* indicate arbitrary token selectors.

`0`	first (command) word
n	*n*’th argument
`^`	first argument, i.e. `1`
`$`	last argument
`%`	word matched by (immediately preceding) `?`*str*`?` search
x`-`*y*	range of words from the *x*’th through the *y*’th
`-`*y*	abbreviates `0-`*y*
`*`	abbreviates `^-$`, or nothing if only 1 word in input line referred to
x`*`	abbreviates *x*`-$`
x`-`	like *x*`*` but omitting word `$`

The `:` separating the event specification from the token selector can be omitted if the token selector begins with a `^`, `$`, `*`, `-` or `%`.

Going back to our example, we can then create the input line

```
<11> help x2106
```

by entering `help !5:*`, or `help !ex:$`, or `help !?2?%`.

Here is a longer example of the use of history substitutions. We will omit the output of the commands themselves, showing only the results of history substitutions in italics.

```
<38> prove "[[A and B] and C] implies [B or C]" foo 100
<39> deduct !!:$ 99 50
deduct 100 99 50
<40> econj !39:$ !
econj 50 !
<41> !e:0 !
econj !
<42> texproof "!?implies?:2_proof1.mss"
texproof "foo_proof1.mss"
```

One cautionary note: It is unwise to use absolute references to input line numbers (e.g., `!25`) in your work files, because when the file is executed again, it is unlikely that a particular line numbered `n` will be the same as line `n` was when the work file was created.

You may wish to know what a command history substitution will look like without executing it. In order to do that, merely choose a word that is not a command (such as "foobar"), and prefix your history substitution by that word. **ETPS** will first echo the substituted line, then just complain that "foobar" is an unknown command.

I.4. Aliases

ETPS maintains a list of aliases which can be created, printed and removed by the `ALIAS` and `UNALIAS` commands. Each input line is separated into distinct commands and the first word of each command is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting tokens replace the command and argument list. If no history references appear, then the argument list is not changed.

As an example, if the alias for `ded` is `"deduct"`, the command `ded 100 !` would be interpreted as `deduct 100 !`. If the alias for `pr` was `"prove \!\!:1 foo 100"`, then `pr x2106` would become `prove x2106 foo 100`. Note that any occurrences of `!` in the alias definition that are meant to be expanded when the alias is invoked must be escaped with a backslash (\) to keep them from being interpreted as history substitutions when the alias is defined.

If an alias is found, the token transformation of the input text is performed and the aliasing process begins again on the new input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing.

The command `ALIAS` can be used to create or display an alias, or to display all existing aliases. The command `UNALIAS` can be used to delete an existing alias. The following example will illustrate:

We define an alias.

```
<135>alias d "deduct \!\!:1-$ \!"
```

We show its definition.

```
<136>alias !
d         deduct !!:1-$ !
<137>exercise x2106
<138>d
```

This expands to "deduct !!:1-$!" but here $ is 0.

```
TPS error while reading.
!!:1-$: Bad ! arg selector.  Last of range is less than first.
```

We’ll remove this definition, and try again.

```
<139>unalias d
```

*The * selector is what we want.*

```
<140>alias d "deduct \!\!:* \!"
<141>d
```

This expands to "deduct !", which is what we intend.

Suppose now that we have finished the proof.

```
<155>alias finish "cleanup & squeeze & done & pall & texproof \!\!:*"
T
<156>finish "myproof.mss"
```

Appendix II
More Editor Commands

The most commonly used editor commands were listed in Section 2.12. Here we list the rest of the commands available in the editor. Note that some of the commands available within the editor are needed for type theory, and may be ignored by the user who is interested only in first-order logic.

II.1. Labels

`<Edn>DELWEAK` *label*
: Replace all occurrences of the label in the current edwff by the wff it represents.

`<Edn>DW` Replace a top-level occurrence of label by the wff it represents.

`<Edn>DW*` Replace all labels in a wff by the wffs represented by them.

`<Edn>NAME` *label*
: Assign a label to the edwff, and replace the edwff with this label.

`<Edn>RW` *label*
: Makes current edwff the new value of label (which must already exist).

II.2. Basic Abbreviations

`<Edn>EXPAND=`
: Instantiates all equalities.

`<Edn>INST` *gabbr*
: Instantiate all occurrences of an abbreviation. The occurrences will be lambda-contracted, but not lambda-normalized.

`<Edn>INST1`
: Instantiate the first abbreviation, left-to-right.

`<Edn>INSTALL` *exceptions*
: Instantiate all definitions, except the ones specified in the second argument.

II.3. Lambda-Calculus

`<Edn>ABNORM`
: Convert the gwff to alphabetic normal form.

`<Edn>ETAB` *gwff*
: Eta-expands until original wff is part of a wff of base type.

`<Edn>ETAC`
: Reduces [lambda x.fx] to f at the top.

`<Edn>ETAN`
: Reduces [lambda x.fx] to f from inside out.

`<Edn>ETAX`
: Performs one step of eta expansion of the gwff.

`<Edn>LEXP` *var term occurs*
: Converts the wff into the application of a function to the term. The function is formed by replacing given valid occurrences of a term with the variable and binding the result.

`<Edn>LNORM`
: Put a wff into lambda-normal form; equivalent to `LNORM-BETA` followed by `LNORM-ETA`.

`<Edn>LNORM-BETA`

Put a wff into beta-normal form.

`<Edn>LNORM-ETA`
Put a wff into eta-normal form (exactly equivalent to `ETAN`).

`<Edn>RED` Lambda-contract a top-level reduct. Bound variables may be renamed.

II.4. Quantifier Commands

`<Edn>DB` Delete the leftmost binder in a wff.

`<Edn>EP` Delete all essentially existential quantifiers in a wff.

`<Edn>OP` Delete the leftmost binder in a wff. Delete all essentially existential quantifiers in a wff.

II.5. Embedding Commands

There are a range of embedding commands, which take the current edwff and embed it below a quantifier or connective. These commands all begin `MBED`, and then have a suffix denoting the quantifier or connective, and a further suffix (if appropriate) denoting whether the current wff is to become the left or right side of the new formula.

They are:

`<Edn>MBED-AL`
Embed the current wff below AND, on the left side.

`<Edn>MBED-AR`
Embed the current wff below AND, on the right side.

`<Edn>MBED-E`
Embed the current wff below an EXISTS quantifier.

`<Edn>MBED-E1`
Embed the current wff below an EXISTS1 quantifier.

`<Edn>MBED-F`
Embed the current wff below a FORALL quantifier.

`<Edn>MBED-IL`
Embed the current wff below IMPLIES, on the left side.

`<Edn>MBED-IR`
Embed the current wff below IMPLIES, on the right side.

`<Edn>MBED-L`
Embed the current wff below a LAMBDA binder.

`<Edn>MBED-OL`
Embed the current wff below OR, on the left side.

`<Edn>MBED-OR`
Embed the current wff below OR, on the right side.

`<Edn>MBED-QL`
Embed the current wff below EQUIV, on the left side.

`<Edn>MBED-QR`
Embed the current wff below EQUIV, on the right side.

`<Edn>MBED=L`
Embed the current wff below =, on the left side.

`<Edn>MBED=R`
Embed the current wff below =, on the right side.

II.6. Changing and Recursive Changing Commands

Many of these commands operate only on the current wff, but have a recursive form that will perform the same operation on the current wff and all of its subwffs.

`<Edn>ASRB`
: Absorbs unnecessary `AND` and `OR` connectives; see the help message for examples. This command also has a recursive form, `ASRB*`.

`<Edn>ASSL`
: Applies the left associative law, changing `(A * (B * C))` to `( (A * B) * C)`. This command also has a recursive form, `ASSL*`.

`<Edn>ASSR`
: Applies the right associative law, changing `( (A * B) * C)` to `(A * (B * C))` This command also has a recursive form, `ASSR*`.

`<Edn>CMRG`
: Deletes the constants `TRUTH` and `FALSEHOOD` from a wff; see the help message for examples. This command also has a recursive form, `CMRG*`.

`<Edn>CMUT`
: Applies the commutative laws to a formula; see the help message for examples. This command also has a recursive form, `CMUT*`.

`<Edn>CNTOP` *connective-or-quantifier*
: Changes the outermost connective or quantifier to that specified.

`<Edn>DIST-CTR`
: Applies the laws of distributivity to a wff in the contracting direction; see the help message for examples. This command also has a recursive form, `DIST-CTR*`.

`<Edn>DIST-EXP`
: Applies the laws of distributivity to a wff in the expanding direction; see the help message for examples. This command also has a recursive form, `DIST-EXP*`.

`<Edn>DNEG`
: Removes a double negation from a wff. This command also has a recursive form, `DNEG*`.

`<Edn>MRG` Merges redundant connectives in a wff; see help message for examples. This command also has a recursive form, `MRG*`.

`<Edn>PMUT`
: Permutes the two components of an infix operator. This command also has a recursive form, `PMUT*`.

`<Edn>SUBEQ`
: Reduces an equivalence to a conjunction of implications. This command also has a recursive form, `SUBEQ*`.

`<Edn>SUBIM`
: Reduces an implication to a disjunction. This command also has a recursive form, `SUBIM*`.

II.7. Miscellaneous

`<Edn>CNF` Find the conjunctive normal form of a wff.

`<Edn>HEAD`
: Finds the head of a gwff.

`<Edn>HVARS`
: Returns all the head variables of a gwff.

`<Edn>MIN-SCOPE`
: Minimises the scope of quantifiers in a gwff.

II.8. Wellformedness

`<Edn>DUPW` *connective*
Duplicates a wff across a connective.

`<Edn>EDILL`
Finds a minimal ill-formed subformula.

`<Edn>ILL` Returns a list of messages, each describing the error in a minimal ill-formed subformula.

`<Edn>TP` Returns the type of a gwff.

`<Edn>WFFP`
Tests for a gwff (general well-formed formula).

II.9. Saving and Recording Wffs

Saving wffs into a file is governed by the two flags PRINTEDTFILE and PRINTEDTFLAG, which determine the name of the file being written and whether or not wffs are currently being written to it, respectively.

`<Edn>O` Toggles recording on and off (i.e. inverts the current value of `PRINTEDTFLAG`).

`<Edn>REM` *string*
Writes a comment into the current output file.

`<Edn>SAVE` *label*
Saves a gwff by appending it to the file `savedwffs`.

Appendix III
Wff Operations

This is a list of those wffops mentioned in Chapter 3; you can get a complete list of wffops by typing `ENVIRONMENT` and then `WFFOP`.

III.1. Equality between Wffs

`(WFFEQ-AB` *wff1 wff2*`)`
: Tests for equality modulo alphabetic change of bound variables.

`(WFFEQ-AB-BETA` *wff1 wff2*`)`
: Tests for equality modulo alphabetic change of bound variables and beta-conversion.

`(WFFEQ-AB-ETA` *wff1 wff2*`)`
: Tests for equality modulo alphabetic change of bound variables and eta-conversion.

`(WFFEQ-AB-LAMBDA` *wff1 wff2*`)`
: Tests for equality modulo alphabetic change of bound variables and both beta- and eta-conversion.

`(WFFEQ-DEF` *wff1 wff2*`)`
: Tests for equality modulo definitions, lambda conversion and alphabetic change of bound variables.

III.2. Predicates on Wffs

`(FREE-FOR` *term var inwff*`)`
: Tests whether a term is free for a variable in a wff.

`(IS-VARIABLE` *gwff*`)`
: Tests whether a wff is a logical variable.

`(NON-ATOMIC` *gwff*`)`
: Tests whether a wff is not atomic, that is, negated, quantified or the result of joining two wffs with a binary connective.

`(NOT-FREE-IN` *gvar inwff*`)`
: Tests whether a variable is not free in a wff.

`(NOT-FREE-IN-HYPS` *gvar*`)`
: Tests whether a variable is not free in the set of hypotheses of a rule.

`(R-PRIME-RESTR` *term1 wff1 term2 wff2*`)`
: Verifies that *wff2* follows from *wff1* by Rule R’ using equality *term1=term2*.

`(SAME-MODULO-EQUALITY` *wff1 wff2 term1 term2*`)`
: Verifies that *wff2* follows from *wff1* by Rule R’ (possibly iterated) using the equality *term1=term2*.

III.3. Substitution

`(S` *term var inwff*`)`
: Substitute a term for the free occurrences of variable in a gwff.

III.4. Basic Abbreviations

`(CONTAINS-DEFN` *wff*`)`
: Tests whether the argument contains a definition.

`(INST-DEF` *inwff*`)`
: Instantiate the first abbreviation, left-to-right.

III.5. Lambda-Calculus

`(LCONTR` *reduct*`)`
Lambda-contract a top-level reduct.

`(LNORM` *wff*`)`
Put a wff into lambda-normal form (equivalent to `LNORM-BETA` followed by `LNORM-ETA`).

`(LNORM-BETA` *wff*`)`
Put a wff into beta-normal form.

`(LNORM-ETA` *wff*`)`
Put a wff into eta-normal form.

III.6. Negation movers

`(PUSH-NEGATION` *gwff*`)`
Pushes negation through the outermost operator or quantifier.

Appendix IV Theorems and Exercises

IV.1. Book Theorems

Substitution instances of the theorems below can be inserted into a proof by using the ASSERT command.

DESCR (Axiom of description at all types.)
$\iota\ [= Y_{o\alpha}] = Y$

EXT (Axiom of extensionality at all types.)
$\forall x_{\beta}\ [f_{\alpha\beta}\ x = g_{\alpha\beta}\ x] \supset f = g$

REFL= (Reflexivity of Equality.)
$A_{\alpha} = A$

SYM= (Symmetry of Equality.)
$A_{\alpha} = B_{\alpha} \supset B = A$

T5302 (Symmetry of Equality.)
$x_{\alpha} = y_{\alpha} =. y = x$

T5310 (Theorem about descriptions.)
$\forall z_{\alpha}\ [p_{o\alpha}\ z \equiv y_{\alpha} = z] \supset \iota\ p = y$

T5310A (Theorem about descriptions.)
$\forall z_{\alpha}\ [p_{o\alpha}\ z \equiv z = y_{\alpha}] \supset \iota\ p = y$

IV.2. First-Order Logic

X2106	$\forall x\ [R\ x \supset P\ x] \land \forall x\ [\sim Q\ x \supset R\ x] \supset \forall x.P\ x \lor Q\ x$
X2107	$R\ a\ b \land \forall x\ \forall y\ [R\ x\ y \supset R\ y\ x \land Q\ x\ y] \land \forall u\ \forall v\ [Q\ u\ v \supset Q\ u\ u] \supset Q\ a\ a \land Q\ b\ b$
X2108	$\forall x\ \exists y.P\ x \supset P\ y$
X2109	$\exists x\ [p \land Q\ x] \equiv p \land \exists x\ Q\ x$
X2110	$\exists x\ R\ x \land \forall y\ [R\ y \supset \exists z\ Q\ y\ z] \land \forall x\ \forall y\ [Q\ x\ y \supset Q\ x\ x] \supset \exists x\ \exists y.Q\ x\ y \land R\ y$
X2111	$\forall x\ [\exists y\ P\ x\ y \supset \forall y\ Q\ x\ y] \land \forall z\ \exists y\ P\ z\ y \supset \forall y\ \forall x\ Q\ x\ y$
X2112	$\exists v\ \forall x\ P\ x\ v \land \forall x\ [S\ x \supset \exists y\ Q\ y\ x] \land \forall x\ \forall y\ [P\ x\ y \supset \sim Q\ x\ y] \supset \exists u.\sim S\ u$
X2113	$\forall y\ \exists w\ R\ y\ w \land \exists z\ \forall x\ [P\ x \supset \sim R\ z\ x] \supset \exists x.\sim P\ x$
X2114	$\forall x\ R\ x\ b \land \forall y\ [\exists z\ R\ y\ z \supset R\ a\ y] \supset \exists u\ \forall v\ R\ u\ v$
X2115	$\forall x\ [\exists y\ P\ x\ y \supset \forall z\ P\ z\ z] \land \forall u\ \exists v\ [P\ u\ v \lor M\ u \land Q.f\ u\ v] \land \forall w\ [Q\ w \supset \sim M.g\ w] \supset \forall u\ \exists v.P\ [g\ u]\ v \land P\ u\ u$
X2116	$\forall x\ \exists y\ [P\ x \supset R\ x\ [g.h\ y] \land P\ y] \land \forall w\ [P\ w \supset P\ [g\ w] \land P.h\ w] \supset \forall x.P\ x \supset \exists y.R\ x\ y \land P\ y$
X2117	$\forall u\ \forall v\ [R\ u\ u \equiv R\ u\ v] \land \forall w\ \forall z\ [R\ w\ w \equiv R\ z\ w] \supset.\exists x\ R\ x\ x \supset \forall y\ R\ y\ y$
X2118	$\forall x\ [p \land Q\ x \lor \sim p \land R\ x] \supset \forall x\ Q\ x \lor \forall x\ R\ x$
X2119	$\exists y\ \forall x.P\ y \supset P\ x$
X2120	$\forall u\ \forall v\ \forall w\ [P\ u\ v \lor P\ v\ w] \supset \exists x\ \forall y\ P\ x\ y$
X2121	$\exists v\ \forall y\ \exists z.P\ a\ y\ [h\ y] \lor P\ v\ y\ [f\ y] \supset P\ v\ y\ z$
X2122	$\exists x\ R\ x\ x \supset \forall y\ R\ y\ y \supset \exists u\ \forall v.R\ u\ u \supset R\ v\ v$
X2123	$\exists y\ [P\ y \supset Q\ x] \supset \exists y.P\ y \supset Q\ y$
X2124	$\exists x\ [P\ x \supset Q\ x] \equiv \forall x\ P\ x \supset \exists x\ Q\ x$
X2125	$\exists x\ \forall y\ [P\ x \equiv P\ y] \equiv.\exists x\ P\ x \equiv \forall y\ P\ y$

X2126 $\forall x\ [P\ x \equiv \exists y\ P\ y] \equiv . \forall x\ P\ x \equiv \exists y\ P\ y$

X2127 $\exists x\ \forall y\ [P\ y \equiv P\ x] \supset \forall x\ P\ x \vee \forall x . \sim P\ x$

X2128 $\forall x\ [P\ x \equiv \forall y\ P\ y] \equiv . \exists x\ P\ x \equiv \forall y\ P\ y$

X2129 $\exists x\ \forall y\ [P\ x \equiv P\ y] \equiv [\exists x\ Q\ x \equiv \forall y\ P\ y] \equiv . \exists x\ \forall y\ [Q\ x \equiv Q\ y] \equiv . \exists x\ P\ x \equiv \forall y\ Q\ y$

X2130 $\forall x\ P\ x \supset \sim \exists y\ Q\ y \vee \exists z . P\ z \supset Q\ z$

X2131 $\forall x\ P\ x \supset \exists y . \forall x\ \forall z\ Q\ x\ y\ z \supset \sim \forall z . P\ z \wedge \sim Q\ y\ y\ z$

X2132 $\forall w\ [\sim R\ w\ w] \supset \exists x\ \exists y . \sim R\ x\ y \wedge . Q\ y\ x \supset \forall z\ Q\ z\ z$

X2133 $\forall x\ [\exists y\ Q\ x\ y \supset P\ x] \wedge \forall v\ \exists u\ Q\ u\ v \wedge \forall w\ \forall z\ [Q\ w\ z \supset Q\ z\ w \vee Q\ z\ z] \supset \forall z\ P\ z$

X2134 $\forall z\ \exists x\ [\forall y\ P\ x\ y \vee Q\ x\ z] \supset \forall y\ \exists x . P\ x\ y \vee Q\ x\ y$

X2135 $\exists x\ \forall y . P\ x \wedge Q\ y \supset Q\ x \vee P\ y$

X2136 $\exists x\ \exists y\ \forall u . P\ x\ y\ z \supset P\ u\ x\ x$

X2137 $\exists x\ \forall y . P\ x \supset Q\ x \vee P\ y$

X2138 $\forall x\ \exists y\ F\ x\ y \wedge \exists x\ \forall e\ \exists n\ \forall w\ [S\ n\ w \supset D\ w\ x\ e] \wedge \forall e\ \exists d\ \forall a\ \forall b\ [D\ a\ b\ d \supset \forall y\ \forall z . F\ a\ y \wedge F\ b\ z \supset D\ y\ z\ e] \supset \exists y\ \forall e\ \exists m\ \forall w . S\ m\ w \supset \forall z . F\ w\ z \supset D\ z\ y\ e$

IV.3. Higher-Order Logic

X5200 $x_{o\alpha} \cup y_{o\alpha} = \bigcup . \lambda v_{o\alpha} . v = x \vee v = y$

X5201 $x_{o\alpha} \cap y_{o\alpha} = \bigcap . \lambda v_{o\alpha} . v = x \vee v = y$

X5202 $\%\ f_{\alpha\beta}\ [x_{o\beta} \cup y_{o\beta}] = \%\ f\ x \cup \%\ f\ y$

X5203 $\%\ f_{\alpha\beta}\ [x_{o\beta} \cap y_{o\beta}] \subseteq \%\ f\ x \cap \%\ f\ y$

X5204 $\%\ f_{\alpha\beta}\ [\bigcup w_{o(o\beta)}] = \bigcup . \%\ [\%\ f]\ w$

X5205 $\%\ f_{\alpha\beta}\ [\bigcap w_{o(o\beta)}] \subseteq \bigcap . \%\ [\%\ f]\ w$

X5206 $\%\ f_{\alpha\beta}\ [x_{o\beta} \cup y_{o\beta}] = \%\ f\ x \cup \%\ f\ y$

X5207 $\%\ f_{\alpha\beta}\ [x_{o\beta} \cap y_{o\beta}] \subseteq \%\ f\ x \cap \%\ f\ y$

X5208 $\exists S_{o\iota}\ \forall x_{\iota}\ [[S\ x \vee P_{o\iota}\ x] \wedge . \sim S\ x \vee Q_{o\iota}\ x] \equiv \forall y_{\iota} . P\ y \vee Q\ y$

X5209 $\wp_{o(o\alpha)(o\alpha)}\ [D_{o\alpha} \cap E_{o\alpha}] = \wp\ D \cap \wp\ E$

X5210 $[= x_{\alpha}] = \lambda z_{\alpha}\ \exists y_{\alpha} . y = x \wedge z = y$

X5211 $y_{o\alpha} = \bigcup . \lambda z_{o\alpha}\ \exists x_{\alpha} . y\ x \wedge z = [= x]$

X5212 $\lambda z_{\alpha}\ \exists x_{\beta}\ [g_{o\beta}\ x \wedge z = f_{\alpha\beta}\ x] = \%\ f\ g$

X5304 $\sim \exists g_{o\alpha\alpha}\ \forall f_{o\alpha}\ \exists j_{\alpha} . g\ j = f$

X5305 $\forall s_{o\alpha} . \sim \exists g_{o\alpha\alpha}\ \forall f_{o\alpha} . f \subseteq s \supset \exists j_{\alpha} . s\ j \wedge g\ j = f$

X5308 $\exists j_{\beta(o\beta)}\ \forall p_{o\beta}\ [\exists x_{\beta}\ p\ x \supset p . j\ p] \supset . \forall x_{\alpha}\ \exists y_{\beta}\ r_{o\beta\alpha}\ x\ y \equiv \exists f_{\beta\alpha}\ \forall x\ r\ x . f\ x$

X5309 $\sim\exists h_{\iota(o\iota)}\ \forall p_{o\iota}\ \forall q_{o\iota} . h\ p = h\ q \supset p = q$

X5310 $\forall r_{o\beta(o\beta)}\ [\forall x_{o\beta}\ \exists y_{\beta}\ r\ x\ y \supset \exists f_{\beta(o\beta)}\ \forall x\ r\ x . f\ x] \supset \exists j_{\beta(o\beta)}\ \forall p_{o\beta} . \exists z_{\beta}\ p\ z \supset p . j\ p$

X5500 $\forall P_{o\beta}\ [\exists x_{\beta}\ P\ x \supset P . J_{\beta(o\beta)}\ P] \supset \forall f_{\alpha\beta}\ \forall g_{\alpha\beta} . f\ [J . \lambda x . \sim f\ x = g\ x] = g\ [J . \lambda x . \sim f\ x = g\ x] \supset f = g$

X6004 $E_{o(o\alpha)(o\beta)}\ [= x_{\beta}] . = y_{\alpha}$

X6101 $\overline{1} = \Sigma^{1}{}_{o(o\iota)}$

X6104 $\exists i_{o(\alpha\alpha)(\alpha\alpha)} . \forall g_{\alpha\alpha}\ [i\ g\ [\lambda x_{\alpha}\ x] \wedge i\ g . \lambda x\ g . g\ x] \wedge \forall f_{\alpha\alpha}\ \forall y_{\alpha} . i\ [\lambda x\ y]\ f \supset f\ y = y$

X6105 (This is a lemma for X6106. You may need to ASSERT DESCR or T5310 or T5310A)
$\forall n_{o(o\iota)} . \mathrm{NAT}\ n \supset \forall q_{o\iota} . n\ q \supset \exists j_{\iota(o\iota)}\ \forall r_{o\iota} . r \subseteq q \wedge \exists x_{\iota}\ r\ x \supset r . j\ r$

X6106 $\mathtt{FINITE}\ [\lambda x_{\iota}\ \mathbf{T}] \supset \exists j_{\iota(o\iota)}\ \forall r_{o\iota} . \exists x\ r\ x \supset r . j\ r$

X6201 $\exists r_{o\alpha\alpha}\ \forall x_{\alpha}\ \forall y_{\alpha}\ \forall z_{\alpha}\ [\exists w_{\alpha}\ r\ x\ w \land \sim r\ x\ x \land . r\ x\ y \supset . r\ y\ z \supset r\ x\ z] \supset \exists R_{o(o\alpha)(o\alpha)}\ \forall X_{o\alpha}\ \forall Y_{o\alpha}\ \forall Z_{o\alpha} . \exists W_{o\alpha}\ R\ X\ W \land \sim R\ X\ X \land . R\ X\ Y \supset . R\ Y\ Z \supset R\ X\ Z$

X8030A $[g_{oo}\ \mathbf{T} \land g\ \bot] = \forall x_{o}\ g\ x$

Index

Table of Contents