

TR-0886

A Beginner's Guide to EUODHILOS

by

T. Minami, H. Sawamura
& T. Ohtani

August, 1994

© Copyright 1994-8-3 ICOT, JAPAN ALL RIGHTS RESERVED

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191~5

Institute for New Generation Computer Technology

A Beginner's Guide to EUODHILOS

Toshiro Minami Hajime Sawamura Takeshi Ohtani

Institute for Social Information Science (*ISIS*),
FUJITSU LABORATORIES LTD.

140 Miyamoto, Numazu-shi, Shizuoka 410-03, Japan
Email: {minami, hajime, ohtani}@iias.flab.fujitsu.co.jp

Abstract

EUODHILOS is a general-purpose reasoning assistant system which allows users to interactively define a logic and to construct proofs on the defined logic. Users get supports for specifying syntactic structures of a logic so that they can use the expressions that are familiar and easy to recognize. The logical structure is specified as a derivation system that consists of combination of axioms, inference rules, and rewriting rules. These rules are represented in tree form so that they are shown in a natural style and thus easy to recognize. A proof is displayed in tree form as well. The users can manipulate proofs directly on one or more windows where they are displayed. The window gives supports to the users constructing proofs in quite flexible ways. The users are allowed to mix both forward and backward derivations following to the way they think. Such a proof-assisting window is called a sheet of thought in EUODHILOS.

This document is intended to be a guidebook to EUODHILOS system for beginners. Most essential and important features of the system are described.

EUODHILOS runs on PSI(Personal Sequential Inference machine) with SIMPOS operating system.

Key words: Reasoning assistant system, Generic system, Logic specification, Proof construction, DCG, Parser, Unparser, Interactive System

9.4.4	Connecting Proof Fragments	56
9.5	Other Facilities of Sheet of Thought	58
9.5.1	Displaying the Structure of an Expression	58
9.5.2	is_axiom/theorem	59
9.5.3	Saving as Theorems or Derived Rules	60
A	Appendix	64
A.1	First-Order Logic	64
A.2	Category Theory	65
A.3	Hoare Logic	66
A.4	Relevant Logic	67

Contents

1	Introduction	3
2	Getting Started	5
2.1	Starting EUODHILOS	5
2.2	Creating a new Logic	7
2.3	Closing EUODHILOS	7
2.4	Manipulating a Logic	7
2.5	Functions Menu of a Logic	8
2.6	Defining Syntax	9
2.7	Defining Inference Rules	11
2.7.1	Rule Name	12
2.7.2	Inputting the Inference Rule Body	13
2.8	Proving on a Sheet of Thought	16
3	Overview of EUODHILOS	24
4	Creating a Logic	25
5	Special Symbols	27
6	Syntax Description	30
6.1	Syntax Editor	30
6.2	Extended DCG part	32
6.3	Constructor declaration	33
6.4	Remarks on Syntax Description	34
7	Inference and Rewriting Rules	35
7.1	Opening and Closing an Inference Rule Menu	35
7.2	Defining and Modifying an Inference Rule	36
7.3	Side Conditions	37
7.4	Rewriting Rules	38
8	Axioms	38
9	Proof Construction on a Sheet of Thought	40
9.1	Assumption	43
9.2	Derivation	44
9.2.1	Forward Derivation	45
9.2.2	Backward Derivation	49
9.3	Metavariables	50
9.3.1	Instantiation	51
9.4	Editing Proof Fragments	53
9.4.1	Deleting Proof Fragment and Node Expression	53
9.4.2	Copying Proof Data	54
9.4.3	Moving Proof Data	55

1 Introduction

EUODHILOS¹ is a general-purpose (or logic-independent) reasoning assistant system which interactively helps us users with reasoning on various number of logics. We have two stages of reasoning on EUODHILOS. In the first stage we define the logic we are going to deal with in the system. Then we move to the reasoning (or proving) stage where we prove theorems, define derived rules, and so on.

A logic in EUODHILOS is specified by describing language and derivation systems. The language system gives the specification for the allowable expressions such as terms, formulas and everything that we will use in the logic. The syntaxes of the expressions are given by using DCG(Definite Clause Grammar[Pereira 80]) augmented with constructor declarations[Ohashi 90]. The derivation system gives us the means to describe how to create a new statement out of the old ones by specifying the creation mechanism as rules that are allowed in the world where we are reasoning on. Since a theory and a logic are identical in EUODHILOS, we define all the rules in the intended theory as a derivation system. It consists of three parts: axioms, inference rules and rewriting rules. Axioms are logical expressions that are considered to postulates(or assumptions supposed to be true in advance) for reasoning. Each axiom may have an optional name if the user thinks that it is more convenient to use the name rather than the expression itself. Inference and rewriting rules are represented in the Natural Deduction[Prawitz 65]-style. An inference rule consists of three parts; a premise or premises of the rule, which may have optional assumption(s), the conclusion of the rule, and optional side conditions. We can apply the inference rule whenever we have the same number of statements which matches the premise(s) part of the rule and the side conditions, if given, are satisfied. A rewriting rule consists of two parts; upper and lower expressions. A rewriting rule indicates that the upper and lower expressions are equivalent so that we may rewrite an expression having a subexpression matching to either one of the expressions specified in the rule by substituting the expression corresponding to the other one in the rule as the rule is applied. Every rule is supposed to have its name for reference.

Once a logic specification has been ended, we move to the stage of proving theorems and constructing derived rules of the logic on "sheets of thought". A sheet of thought helps us with building up proof fragments² and giving justifications to the theorems and lemmas. We can put assumptions, axioms, or theorems; derive new results by applying rules; and connect two proof trees on a sheet of thought. The construction of proofs or partial proofs goes in a tree form so that we can easily see the natural proof structure. Unlike ordinary proof editors, we are allowed both forward and backward derivations on a sheet of thought. The forward derivation is a derivation which derives a conclusion from premises by applying a rule. The backward derivation, on the other hand, is the derivation which derives one or more premises from a conclusion. Two proof fragments can be combined into a big one by connecting an assumption of one of the fragments and the root (or the conclusion) of another fragment if the two expressions are unifiable. In this way, we are able to make a big proof out of assumptions and axioms put on the sheets of thought.

¹The name EUODHILOS is an acronym coming after the sentence 'Every universe of discourse has its logical structure.' by S. K. Langer[Langer 25] and is supposed to be pronounced "you-oh-dee-loss".

²We also use the word "proof fragments" for partially constructed proof trees.

It is a tedious job to build up a working environment for new logics on which we want to do some experiments. We are able to build up such an environment fairly quickly on EUODHILOS. Another advantage of EUODHILOS lies on developing a new logic itself. Since the definitions of syntax, rules, axioms are allowed to be changed freely, EUODHILOS can be a good platform for developing logics from the beginning.

We have so far experimented with defining and making proofs for a number of logics ([Sawamura 91b], [Sawamura 93], [Ohtani 93]). Among them are classical first-order logic, higher-order logic, propositional modal logic, intensional logic, Martin-Löf's intuitionistic theory, Hoare logic, general logic, elementary category theory, relevant logic, and so on.

This is an introductory guide to EUODHILOS system for those who have not used it before. We learn how to initiate the system, create a logic, define a language and derivation systems for it, and build up theorems in the defined logic. What are explained in this article are not comprehensive. However the reader will be able to learn the most important and useful features of EUODHILOS from this article.

The following sections are organized as follows. In Section 2, we will take up quite a simple logic as an example and see how a logic is defined on EUODHILOS and how a simple theorem is constructed. This short experience will be good enough for the reader to capture the rough idea of what the reasoning style on EUODHILOS is like.

In the following sections, we will see some other useful features of EUODHILOS such as how to define and use special symbols that are familiar with on papers but not appears on standard keyboards, how to make up a complete syntax description, how to construct proofs in a more sophisticated way, and so forth. In Section 3, we will see how the process of reasoning goes on EUODHILOS. One of the fundamental thoughts in designing EUODHILOS is that human reasonings proceed only through trial and error. Thus EUODHILOS is designed as flexible as possible in the interaction with users as well as modifying the definition data of logics.

We have two ways of creating a logic, one from scratch and the other by copying some of the data from a logic already defined. The former has been shown in Section 2. Section 4 explains the latter. We need to specify which data of the old logic to be copied to the new one. We usually use special symbols, especially some of the logical connectives, in a logic. Section 5 shows how to create a font for special symbols and how to use them in a specific logic. In Section 6, we learn how to specify syntax of expressions used in the logic. This feature allows us to use expressions in the form we want to use. Sections 7 and 8 show us how to give the derivation system of the logic we are dealing with. Rules are given in tree form and axioms in a list of names and expressions. The proof editing facilities are explained in Section 9. We start the proof procedure by putting assumptions, axioms, and theorems on a sheet of thought. Then we make them grow by applying rules and connecting proof fragments. A sheet of thought proving environment gives us the means of constructing proof fragments in a flexible way so that eventually the intended results would be proved quite easily. We have an appendix that shows us some of the samples of logics defined on EUODHILOS. These will help us feel potential usefulness of EUODHILOS.

2 Getting Started

It would be a good experience for beginners to take a simple example and see how EUODHILOS works in order to get a rough idea. This section is written for this purpose. The logic treated in this section is a propositional logic which has an implication as the only logical connective, which has only two inference rules; implication introduction and implication elimination. We will write the syntax of the logic, define the two inference rules, and construct a theorem of the logic.

This section is organized as follows. Section 2.1 shows how to initiate the system. In Section 2.2, we see how to create a new logic and give it a name. We can terminate the system whenever we want to. Section 2.3 shows us how to do it. We must be very careful when we terminate the system, because some data we have defined may be disappeared as the system is terminating. In Section 2.4, we have a description how to activate and manipulate a logic. Section 2.5 describes about the functions menu of a logic. We can show or modify various kinds of data of the logic through this menu. In Section 2.6 and Section 2.7 the ways of defining the syntax and inference rules are shown respectively. Finally, in Section 2.8, we will learn how to construct and manipulate proofs on a sheet of thought.

2.1 Starting EUODHILOS

We suppose EUODHILOS is properly installed³ so that the "EUODHILOS" item appears in the system menu. We will not mention here how to install the system. For installation, see the reference manual([Minami 92]).

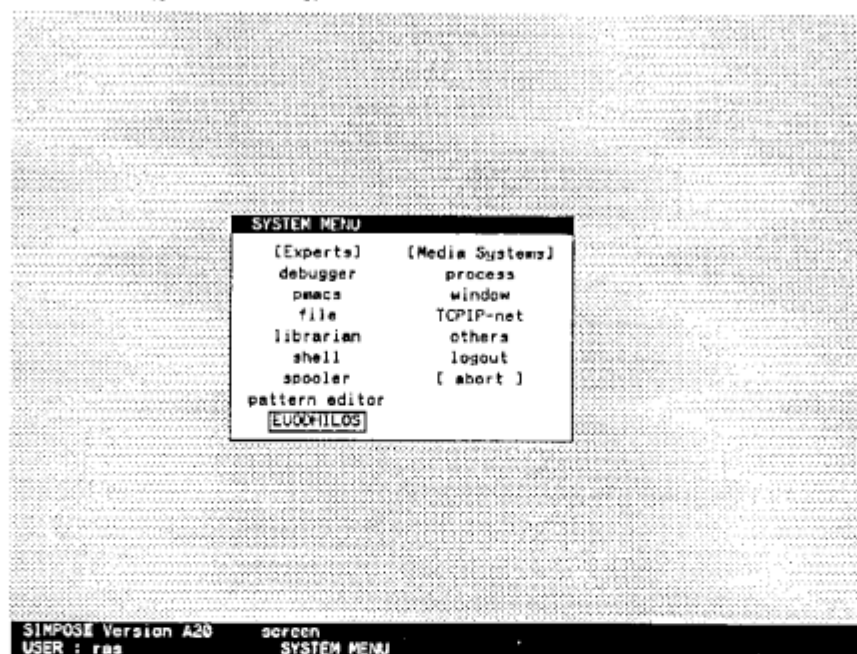


Figure 1: System Menu

³EUODHILOS runs on PSI with SIMPOS operating system.

We will get the system menu when we have logged in, and also whenever we give a double right clicks(i.e. the clicking of the right mouse button consecutively twice). The double right clicks is reserved for this purpose on PSI. The system menu looks like as shown in Figure 1.

EUODHILOS will be initialized when we select the "EUODHILOS" item⁴. We have to wait for a while if it is the first initialization of EUODHILOS after the operating system gets started, because the objects defined in EUODHILOS must be loaded in the first initialization.

Now we have the logic menu of EUODHILOS, which looks like as in Figure 2.

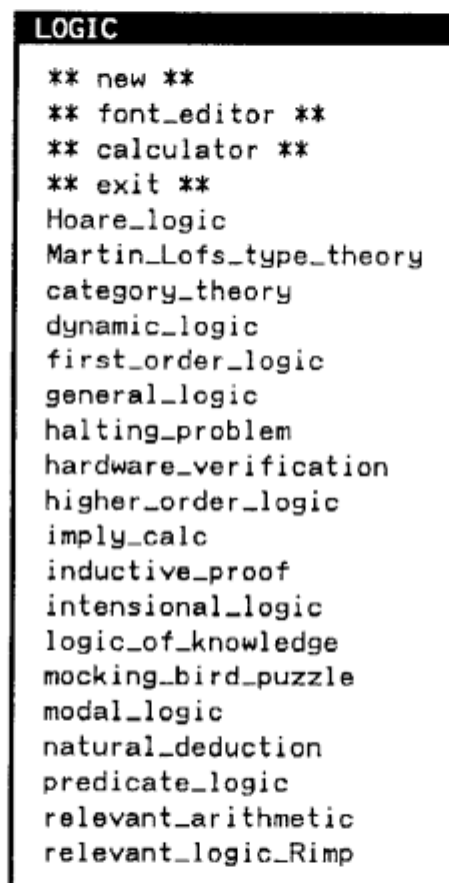


Figure 2: Logic Menu

The first four items represent the general functions whereas the rest are the list of the logics defined and stored in the system. The first item "**** new ****" of the menu is for creating a new logic. See the following Section 2.2 for details. The second item "**** font_editor ****" is for making new symbols. The standard font editor of SIMPOS will be invoked when we select this item. We can define the new symbols which we want to use in our logics. The third item "**** calculator ****" is for opening up a Boolean calculator, which is a tool for checking the validity of propositional formulas in the ordinary (i.e. Boolean) logic. The

⁴For selecting a menu item, move the mouse cursor on the item so that a rectangle which surrounds the item appears, then click any mouse button we want.

fourth one “** exit **” is for terminating EUODHILOS. All the EUODHILOS windows will be closed as we select this item. So, we have to be very careful when we select this item so that we will not lose data which should have been saved. The rest items are the names of the logics already defined. They are sorted in the lexicographical order. The logic menu is a scroll window so that we can see other logics which are not displayed in the initial menu by scrolling the window. For handling an old logic, select the name of the logic. Then we will get the manipulation menu, which is described in Section 2.4 at page 7.

2.2 Creating a new Logic

We will create a logic as a new one for our logic. Select the “** new **” item of the logic menu. We will get the prompting window for inputting the name of the logic. Let’s type “example” as the name of our logic.

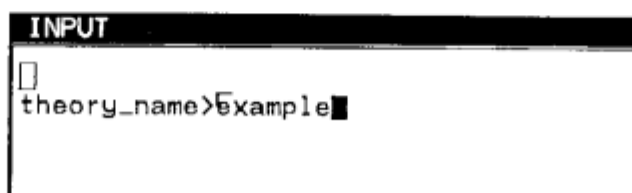


Figure 3: Inputting the theory/logic name

The new logic named “example” will be created, and be shown in the logic menu. Then the logic is activated and we get the functions menu of the logic. See Section 2.4 for the proceeding steps.

2.3 Closing EUODHILOS

Some system has a poor user interface so that we have a trouble finding a way to quit it. In EUODHILOS it is quite easy to find how. Clicking the “** exit **” item in the logic menu (Figure 2, page 6),⁵ is all we have to do to terminate EUODHILOS. Then all the windows used in the session are closed and the system terminates.

2.4 Manipulating a Logic

When we select a specific logic, we will get the menu for selecting what to do on this logic. We will call it the *manipulation menu* of the logic. The window in Figure 4 is the manipulation menu of the logic “example”.

The “information” item is for displaying information about the logic and also some remarks, if we have given, for the logic. See the reference manual ([Minami 92]) for detail. The “activate” item, which we are going to click, is for displaying the functions menu of the logic and getting into the work on the logic. The “rename” item is for changing the

⁵If the menu is under some other window(s), then give a left click anywhere on the menu, so that the target menu will make itself raise up.

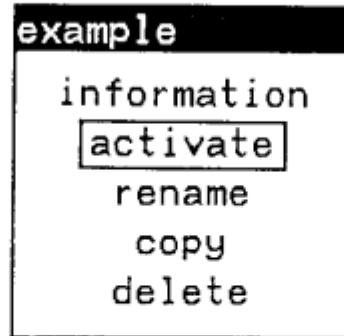


Figure 4: Manipulation Menu of a Logic

name of the logic. The “copy” item is for creating a new logic by copying some or all of the data of an old logic. The “delete” item is for deleting all the data of the logic.

2.5 Functions Menu of a Logic

Now suppose we select the “activate” item of the manipulation menu. We have the functions menu for the logic as shown in Figure 5.

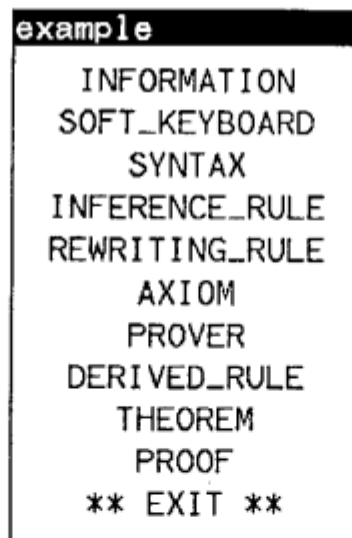


Figure 5: Functions Menu of a Logic

Here again the “INFORMATION” item appears in the menu. It has the same function as the “information” item in the manipulation menu of the logic. See Section 2.4 for brief description of this item. The “SOFT_KEYBOARD” item is for defining and displaying the key assignment of symbols used in the logic. We can use special symbols in the expressions of the logic as far as they are already defined and assigned onto some keys. Symbols are defined with

font editor and their assignments are done by this item. The detail is explained in Section 5. The "SYNTAX" item is for defining the syntactic structure for the expressions, e.g. formula, term, and so on which will be used in the logic. We are supposed to define the syntax first because we can define axioms and rules only after specifying what sort of expressions will be used in the logic. The "INFERENCE_RULE", "REWRITING_RULE", and "AXIOM" items are for defining inference rules, rewriting rules, and axioms respectively. The logical structure of the logic is given by these data. The "PROVER" item is for assigning theorem proving program for the logic. We are supposed to give the file name where a theorem prover program for the logic written in ESP is stored, the class name, and the method name. The "DERIVED_RULE" and "THEOREM" items are for displaying the list of derived rules and theorems, respectively. We will get these kinds of data on sheets of thought by derivations. They will be saved to be used in the later derivations. Theorems can be used exactly the same way as axioms whereas derived rules as primitive rules. These data do not change or augment the derivation power of the logic. However they will help us making large proofs shorter and easier to deal with. The item "** EXIT **" is for terminating this menu, as it says.

2.6 Defining Syntax

We are going to define the syntax for the logic "example". Figure 6 indicates the actual syntax definition window of the logic.

```

SYNTAX : example
save  make  test  structure  print  reshape  exit
formula --> formula, ">=>", formula ;
           "(", formula, ")"    ;
           meta_formula        ;
           "a"-"z";
meta_formula --> "A"-"Z".

operator ">=>":left.

```

Figure 6: Syntax Definition Window

The logic "example" has only one logical connective—implication. The syntax definition consists of two parts, DCG part and constructor declaration part. The former part describes how the expressions are constructed as strings. The latter part describes the constructors, i.e. operators and predicate/functional symbols, used in the first part. We need to designate

the constructor component in a compound expression, i.e. an expression composed of more than two subexpressions. For example, we read the expression "A \wedge B" as *conjunction*(\wedge) of "A" and "B". Thus, " \wedge " is the constructor of conjunctive expressions.

The expressions given to the system are stored in its *internal form* which is a tree consisting of the constructor as the root and other component(s) of the expression as its leaf(leaves). This data is used for creating internal data of the tree form by the parser as well as by the unparser as generating the external expressions from the internal forms. It will be used also in the structure displaying of the expressions. See Section 6 and Section 9.5.1 for structure displaying facility. The parser and unparser will be generated when we finish the syntax definition and select the "make" item.

The window in Figure 6 defines as follows. The syntax category (i.e. non-terminal) "formula" has four kinds of forms; implicational formula, parenthesized formula, meta formula, and either of the characters from "a" to "z". Meta formula is represented by either one of the characters from "A" to "Z". The prefix "meta_" of the syntactic category name indicate that it is for defining meta-variables. A meta-variable is a place holder where we can substitute any expressions belong to the syntactic category that the meta-variable has. The last line is the constructor description. Here we have defined " $=>$ " as an operator which is left associative.

What follows are descriptions of items in the functions menu for syntax definition windows. They form a line at the top row of the window.

- save:** Save the syntax data. It does not generate the parser nor the unparser.
- make:** Create the parser and the unparser from the syntax definition given on the window. A parser interprets the given string as a logical expression and create its internal representation. An unparser creates a string from the internal representation of a logical expression. A message is displayed if syntactically wrong definition is given and the creation of the parser and the unparser fails.
- test:** This item creates a testing window. We can give an expression and see whether this one belongs to one of the syntax categories. If it is acceptable all the matching categories are shown. The detail description is explained in Section 6.
- structure:** Display the structure of the expression. The constructor of an expression will be displayed higher then the other parts of the expression. This facility is helpful in checking if the tree structure we have defined is the same to that we want to define. See Figure 46(Page 32) for the actual image of structure displaying.
- print:** Print the syntax rules to the standard laser printer of PSI (if it is available). Please be patient. It will take a long time.
- reshape:** Change the size of the window.
- exit:** Terminate the syntax definition window.

2.7 Defining Inference Rules

The logic we are dealing with has no axioms nor rewriting rules. It has only two inference rules—implication introduction and implication elimination.

For opening the window for defining inference rules, we are supposed to select the “INFERENCE_RULE” item in the functions menu of the logic (See Figure 5 at page 8). Then we have a window which looks like as follows.

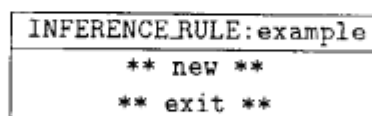


Figure 7: Inference Rule Menu

The “** new **” item is for making a new rule, and the “** exit **” for closing the menu. We click the “** new **” item to get a new inference rule window.

Now we have the window waiting for the definition as is shown in Figure 8.

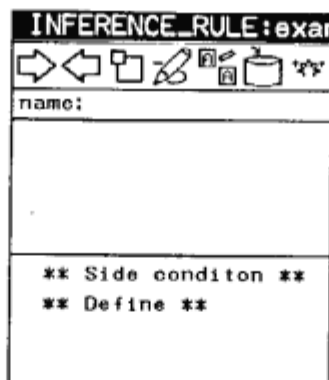








Figure 8: Inference Rule Window

From top to bottom the window region is divided into four areas. The top one is the icon displaying area where we see seven icons. Next comes the name area where the name of the rule is displayed. Then comes the rule body displaying and editing area. The bottom area is for giving and displaying the side conditions for the rule. The rule can be applied only if all the side conditions are satisfied.

Icons in the window show the following functions (from left to right).

- ➡ : Scroll the rule body region to the right.
- ⬅ : Scroll the rule body region to the left.
- 📏 : Resize the window.

-  /  : The pencil and the eraser icons indicate the writing and deleting mode respectively. If we give a middle click on this icon then it will change itself to the other mode. We are sure to change to the deleting mode before we erase a formula.
-  /  : The copy and move icons indicate the copying and moving mode respectively. If we give a click on this icon then it changes to the other mode.
-  : Save the current definition and make the rule be effective.
-  : Exit from the editor. The inference rule window will disappear.

The way how to input the name and body parts of the rule will be shown in the next subsection.

2.7.1 Rule Name

The rule name field is editable. Give a double left click(or two) while the mouse cursor is in the name field. Then we will get a window, shown in Figure 9, waiting for inputting the rule name.

INPUT
rule name=>I

Figure 9: Rule Name Prompting

Type the name of the rule we are going to define. In the figure above “=>I” is typed as the name. Type the return key at the end of the name. Then the new rule name we have just typed will be displayed in the rule name field of the inference rule window. It looks as in Figure 10.








INFERENCE_RULE: exat	
      	
name: =>I	
<div style="border: 1px solid black; height: 100px; width: 100%;"></div>	
<p>** Side conditon **</p> <p>** Define **</p>	

Figure 10: New Rule Name Added in the Inference Rule Menu

2.7.2 Inputting the Inference Rule Body

Here we are going to input the body part of the rule named " $\Rightarrow I$ " (implication introduction rule) and finish the definition of the inference rule. The procedure of inputting is similar to that of inputting the name of the rule.

The rule looks like as this.

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \Rightarrow B}$$

It has one premise formula "B" which has an assumption "A". The conclusion of the rule is " $A \Rightarrow B$ ". This rule says that if we have a proof of "B" from the assumption "A" then we can conclude that " $A \Rightarrow B$ " holds. We can input these three formulas in any order. In this example we will input the assumption "A" first, then "B", and " $A \Rightarrow B$ " finally.

Give a left click in the upper left part of the rule body region. A small box appears at the top-left corner of the rule-body region, which indicates where the assumption expression is supposed to be inserted. (See Figure 11.)

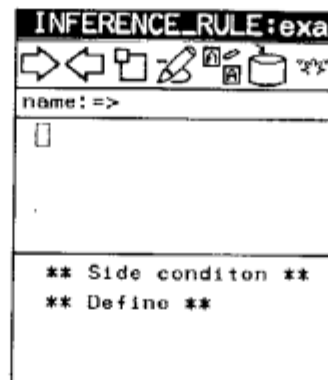


Figure 11: Defining an Assumption Part of the Inference Rule

Give a left double clicks in the region. Then we will get the window for inputting the expression for the box. Type the assumption "A" as in Figure 12. (And be sure to type the return key to terminate the input.)

INPUT
expression>A

Figure 12: Inputting the Expression "A"

The expression “A” is displayed as the assumption for the premise that has not been typed yet. Figure 13 shows the current situation.

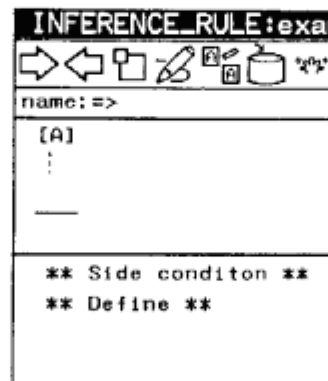


Figure 13: Inference Rule having an Assumption

Next we are going to type the premise part of the rule. Let's give a left click in the center-left part of the body region just under the assumption “A”. This time, again, the prompting window appears and waits for the premise expression “B” to be typed. When we have finished inputting the premise “B” the window looks as is shown in Figure 14.

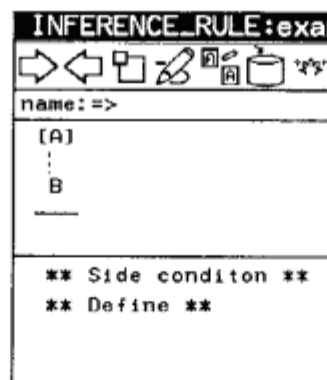



Figure 14: Inference Rule having an Assumption and a Premise

Finally we move to the conclusion formula of the rule. Give a left click in the lower left part where the conclusion of the rule would be inserted. Again the prompting window appears. Type the conclusion expression “ $A \Rightarrow B$ ” of the rule in the prompting window. The conclusion part is displayed on the rule definition window as the prompting window closes. We have completed the body part of the rule (Figure 15).

As we have finished the definition of the inference rule “ $\Rightarrow I$ ”, we have to save the definition so that the rule can be used in proofs. Give a middle click on the “” icon. We will get the message “saved” at the place where the mouse cursor is shown while the data are being saved. The name of the rule will be added to the menu of inference rules.

INFERENCE_RULE:exar	
<div> <div>→</div> <div>←</div> <div>□</div> <div>✎</div> <div>📄</div> <div>🗑️</div> <div>⚙️</div> </div>	
name: =>	
<div> <div>{A}</div> <div>⋮</div> <div>B</div> <div>—</div> <div>A=>B</div> </div>	
<div> <div>** Side conditon **</div> <div>** Define **</div> </div>	

Figure 15: Inference Rule(Implication Introduction)

Click next the “** new **” item of the menu again for the next rule “=>E”. The expression of the rule looks like this.

$$\frac{A \quad A=>B}{B}$$

Input the name of the rule and one of the premise “A” in the same way as we did in defining the implication introduction rule. Now we are going to input the second premise of the rule. Give a left click at the place just right of the first premise “A”. We will get the following rule window (Figure 16).

INFERENCE_RULE:exar	
<div> <div>→</div> <div>←</div> <div>□</div> <div>✎</div> <div>📄</div> <div>🗑️</div> <div>⚙️</div> </div>	
name: =>E	
<div> <div>A □</div> <div>—</div> </div>	
<div> <div>** Side conditon **</div> <div>** Define **</div> </div>	

Figure 16: The Second Premise of an Inference Rule

Again, give the left double clicks and input the second premise “A=>B”, then input the conclusion “B”. Be sure to save the definition data of the rule. Now we have finished of defining the implication elimination rule (Figure 17).

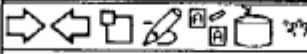
INFERENCE_RULE:exar	
	
name: =>E	
$\frac{A \quad A \Rightarrow B}{B}$	
** Side conditon ** ** Define **	

Figure 17: Inference Rule(Implication Elimination)

This concludes the definition of inference rules. We now have defined the logical structure of “example”. So it’s time to move to the proof-making phase on sheets of thought.

2.8 Proving on a Sheet of Thought

Sheet of thought helps us users with constructing proof trees on the logic that we are dealing with. We may say that it is a proof editing environment or a proof editor. A proving procedure begins by putting some assumptions⁶ and/or axioms (or theorems). These are the origins of more complicated proofs on the sheets of thought. Then we make them bigger by applying some rules and connecting some of the partially constructed proofs (i.e. proof fragment). In this way, we would have final results that eventually be stored as either theorems or derived rules. Once stored theorems can be used just like axioms and derived rules as rules.

Now let’s get back to the functions menu (Figure 5, page 8) of the logic “example”. Select the “PROOF” item of the menu. Then the proof manipulation menu looked like the following will appear.

PROOF_NAME:example
** new **
** exit **
** work **

Figure 18: Proof Manipulation Menu

The “** new **” item is for creating a new sheet of thought, the “** exit **” item for closing, and the “** work **” item for opening up the work area where proof fragments constructed in sheets of thought may be stored.

⁶We can think an assumptions as a goal, since we are allowed to apply a rule backward on it at any time, and thus it is treated a goal in this case.

A sample figure of sheet of thought is shown in Figure 19.

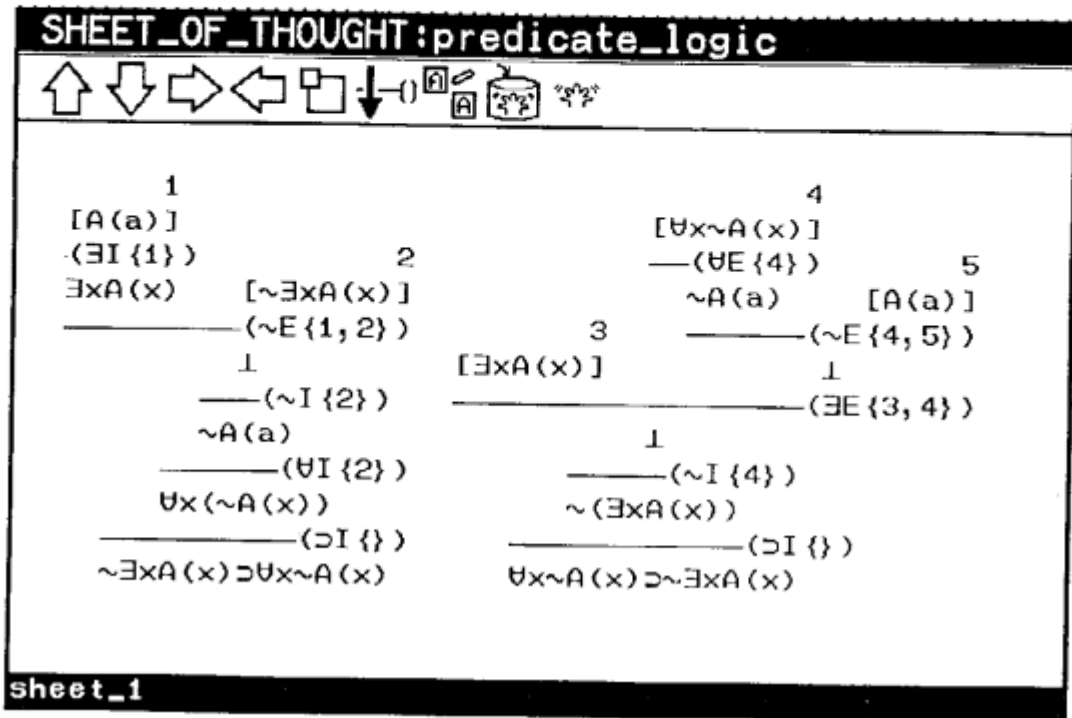










Figure 19: Sheet of Thought

The icons lined up on a sheet of thought have the following meanings.

- : Scroll the window to right for proof tree.
- : Scroll the window to left for proof tree.
- : Scroll the window to upward for proof tree.
- : Scroll the window to downward for proof tree.
- : Resize the window.
- / : Writing/Deleting mode indication. Give a middle click to change the mode. These icons appear when we have a marker for inserting or deleting a proof subtree or a formula on a sheet of thought.
- / : Forward/backward derivation mode. The icons indicate how the derivations go. In the forward derivation mode we will get the conclusion will be derived from the given premise(s). In the backward derivation mode the premise(s) will be got from the conclusion in the derivation.
- / : Copy/move a proof tree or an expression from right click position to the place where the left click was given. The original tree or expression will be erased as the copy/move operation is applied in the move mode.

 : Close the sheet after saving the proof data.

 : Close the sheet. We will have the confirmation menu for the proof data whether they should be saved or not(See Section 9).

Give the middle click on the icon in order to select the item. The “” and the “” icons as well as the “” and “” icons indicate modes. When we select the “” icon then the mode changes from writing to erasing, and the icon changes to the “” icon. The situation is exactly the same on copying and moving modes.

Let us get back and try to prove the theorem “ $(A \Rightarrow (A \Rightarrow B)) \Rightarrow (A \Rightarrow B)$ ”. First we put the two assumptions “ $A \Rightarrow (A \Rightarrow B)$ ” and “ A ” on the sheet.

Select “** new **” item of the proof manipulation menu and let a sheet of thought open. Give a left click on the sheet. Then a box marker appears in the lower left corner of the sheet. Next give a left double clicks. The selection menu appears (Figure 20).

COMMAND
input_assumption
input_axiom/theorem
cancel

Figure 20: Input Assumption Menu

Select the “input_assumption” item so that we have the prompting window (Figure 21).

INPUT
input> $A \Rightarrow (A \Rightarrow B)$

Figure 21: Input Assumption

The expression we gave is inserted to the place where the box marker was displayed.

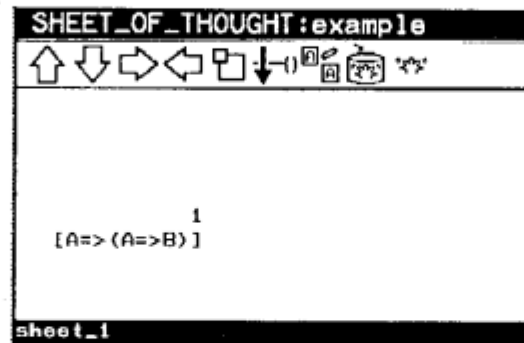


Figure 22: Assumption is Displayed

The assumption expression is surrounded by brackets “[” and “]” so that it is easy to recognize and the assumption identification number is assigned to differentiate one from the

others. Now let us give an another left click on the sheet at the right of the assumption mentioned above. When we get the box marker, we give left double clicks to the sheet and type the next assumption "A". We now have the sheet shown in Figure 23.

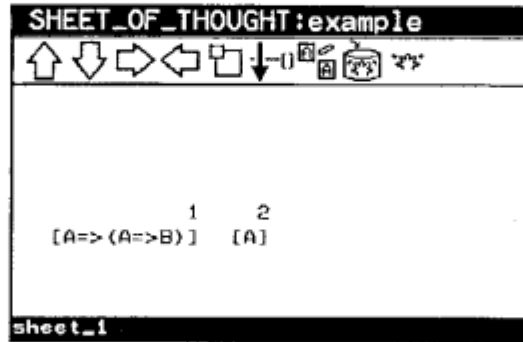


Figure 23: Two assumptions are displayed.

Next we apply the implication elimination rule. Give right clicks on both assumptions, then the assumptions are highlighted with being surrounded by black boxes (Figure 24).

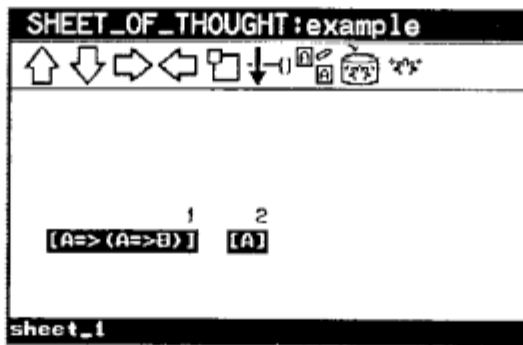


Figure 24: Two assumptions are highlighted.

Give left double clicks and get the application selection menu (Figure 25).

INFERENCE_RULE	REWRITING_RULE	DERIVED_RULE	COMMAND
$\Rightarrow E$ $\Rightarrow I$	--reverse_application--		cancel discharge search_rule(input) search_rule --call_prover--

Figure 25: Application selection menu

Select the “ $\Rightarrow E$ ” item. Then the selected rule will be applied to the highlighted formulas on the sheet and the conclusion will be derived. The result is displayed as a tree as is shown in Figure 26. The pair of numbers “{1,2}” at the right of the rule name indicates that the conclusion “ $A \Rightarrow B$ ” of this application depends on assumptions 1 and 2.

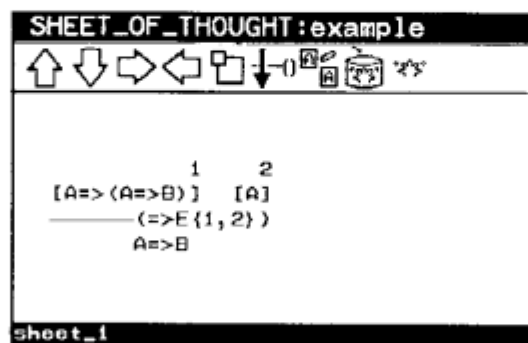


Figure 26: The rule “ $\Rightarrow E$ ” is applied.

Next we would like to derive “B” by applying the rule “ $\Rightarrow E$ ” again to “ $A \Rightarrow B$ ” and “A”. Since we want to use the assumption “A” as the same one as that having the assumption number 2, we need to copy the assumption “A” with the assumption ID 2 to the place next to the formula “ $A \Rightarrow B$ ”.

Give a right click on the assumption “A” and mark it with a black box. Give a left click next to the derived formula “ $A \Rightarrow B$ ” on the right and get a white box marker. The copy icon must be displayed. If not, give a middle click on the move icon and get it. Give left double clicks in the proof area. We will get the sheet in Figure 27.

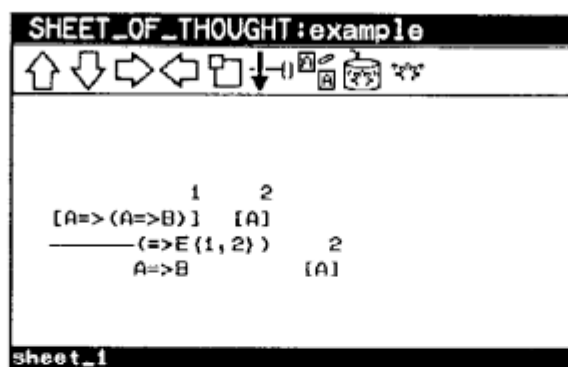


Figure 27: Assumption “A” is copied.

We are going to apply the inference rule “ $\Rightarrow E$ ” again. Select the two formulas “ $A \Rightarrow B$ ” and “[A]” by giving right clicks. Do not worry about that all the left proof tree is surrounded by a black box. It is equivalent to select the root formulas. Again, give left double clicks and select the rule “ $\Rightarrow E$ ”. Figure 28 shows where we are.

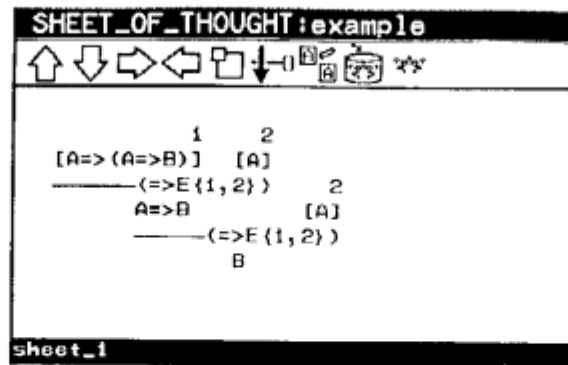


Figure 28: The rule “=>E” is applied.

Now we apply “=>I”, which has an discharging assumption. Select the formula “B” with a right click. Then give a middle click on the discharging assumption “A”. Select either one of two occurrences of “A”. The selected assumption is shown with underline.

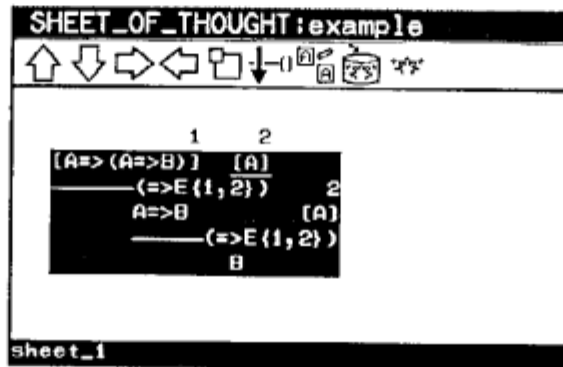


Figure 29: All the proof fragment is selected.

Here, we give double clicks on the sheet and select the inference rule “=>I” in menu.

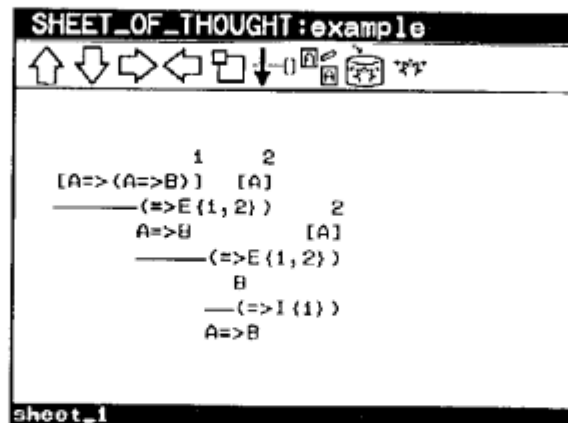


Figure 30: The rule “=>I” is applied.

We apply the implication introduction rule one more time, and discharge the assumption numbered 1. We now have completed proving the theorem " $A \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ ".

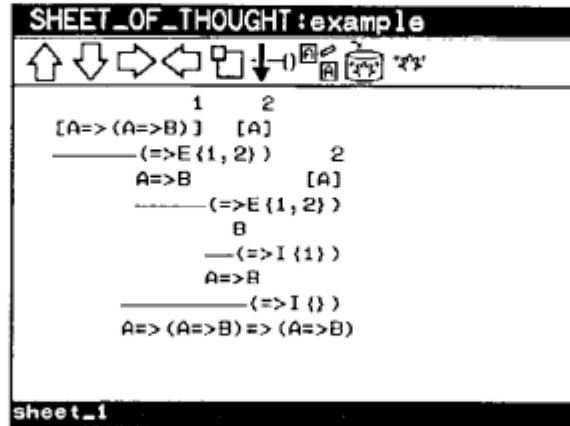


Figure 31: Proof completed.

The conclusion has no depending assumptions now. So it is a theorem of the logic. Let us save the result as a new theorem. Give a left click on the conclusion of the proof. Then a white box marker surrounding the proof tree appears. Give left double clicks. Then the following command menu will appear.

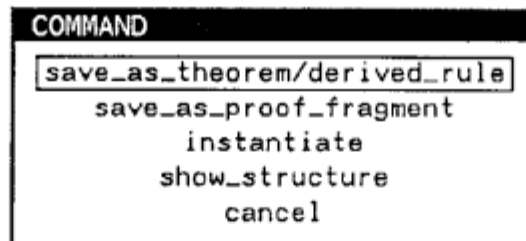


Figure 32: Select the saving item

Select the "save_as_theorem/derived_rule" item in the menu. This is the item for saving the selected proof fragment as a theorem or a derived rule. If the conclusion of the fragment has at least one assumption then it is saved as a derived rule. Otherwise, i.e. when it has no assumptions that depend on, it is saved as a theorem. In this case, the result is saved as a theorem. Here, the following input menu appears.

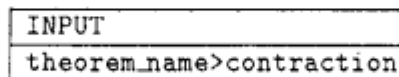


Figure 33: Input the theorem name.

As we type a name, say “contraction”, the theorem is saved which has the given name. If we just type the return key without specifying the name, the theorem will be saved as a theorem which has no name. In this case, we have to answer to the confirmation window as is shown in Figure 34.

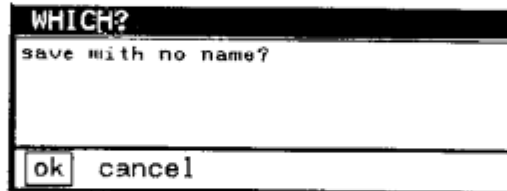


Figure 34: Confirmation of the theorem name for saving.

The proof fragment disappears when it is saved as a theorem. We see the body of the theorem, that is the conclusion of the proof fragment of the theorem, and its name also, if we have given.

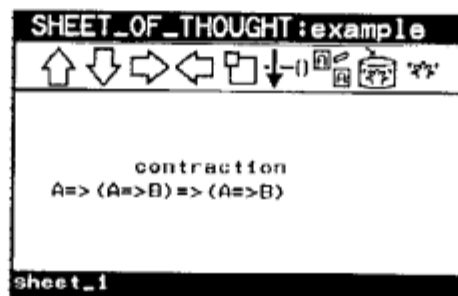



Figure 35: The theorem looks like this after it is saved.

We would like to close the sheet of thought. Give a middle click on “” icon. Then the window will be closed after saving the proof data into the working storage area. See Section 9 for detail about sheet of thought. Let us exit from the logic now. Select the “** exit **” item of the top menu of the logic “example”. If the window is under one or other windows then give a left click at a point on the top menu. Then it will come as the most front window in the screen, and we can select the item. We will have the confirmation window. Select “yes”. Then all the windows related to the logic disappears and only the logic menu appears in front of us.

It is time to exit from EUODHILOS. Select the “** exit **” item of the logic menu. Now all the windows will be disappeared, and the execution of the system will terminate.

3 Overview of EUODHILOS

In the previous section, we have seen briefly how to use EUODHILOS. In the following sections we will define a bigger logic and build up some simple proofs by using more functions of EUODHILOS.

The process of using EUODHILOS can be illustrated in the following figure.

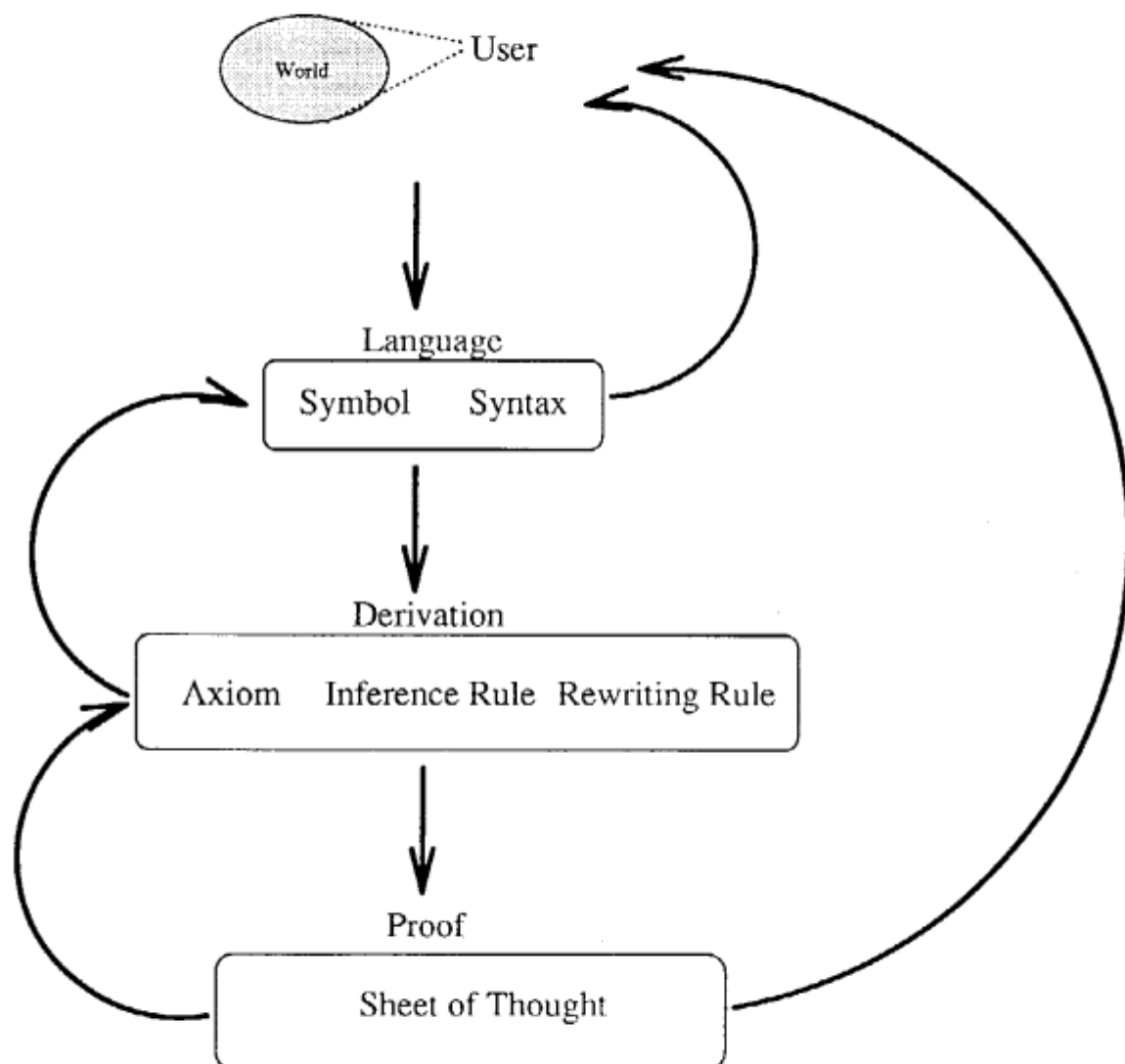


Figure 36: The process of using EUODHILOS

There are two phases in using EUODHILOS: defining a logic and constructing proofs. A logic in EUODHILOS consists of language and derivation systems. A language system describes what characters are used in the logic and how to combine them and construct expressions used as predicates, terms, and so forth. We can use all the standard characters assigned on the keyboard for this purpose, of course. However in most cases they are not

sufficient enough. We would like to use more characters for readability of these expressions. For example, for logical connectives it is better using “ \wedge ”, “ \vee ”, “ \supset ”, and “ \square ” rather than “ \wedge ”, “ \vee ”, “ \Rightarrow ”, and “ $[]$ ”. We can make such symbols using the standard font editor of SIMPOS and use them by assigning them on keys by the software keyboard facility given by EUODHILOS. Section 5 describes how to carry these out. We give a syntax description to the system and specify how to combine the characters and make reasonable expressions. Syntax description in EUODHILOS has two parts. One is the syntax definition based on the DCG (Definite Clause Grammar)[Pereira 80] notation with some augmentation such as “or” notation, character interval specification, and so on. The other part in syntax description which is unique in EUODHILOS is the constructor declaration. This part defines constructors that are suppose to be the main operations when expressions are combined. Take, for example, the expression “ $A \wedge B$ ”. This expression consists of three characters: “ A ”, “ \wedge ”, and “ B ”. As we see this expression we naturally interpret this as the “and”(or conjunctive)-expression, which has the operator “ \wedge ”, which takes the two sub-expressions “ A ” and “ B ” as arguments. We call the operator “ \wedge ” the constructor of the expression. For any combination which has more than one subexpressions defined in the syntax definition part, the system supposes it has a constructor declared in the constructor definition part. All the windows including the syntax definition window that are used for inputting character strings have the Pmacs⁷ interface. So, we can easily scroll the window, position the cursor, yank texts, and do other editings.

We move to the derivation system definition when we have finished the language system definition. Derivation system consists of three parts: inference rules, rewriting rules, and axioms. We just call “rules” when we refer both inference and rewriting rules. EUODHILOS has two types of rules based on our observation that we humans use both types of rules in our reasonings. We have an explanation in detail in Section 7.

We will move to the construction of proofs when we have defined the logic. A sheet of thought is a field in which we can reason by putting assumptions and axioms, applying new rules, composing a larger proof from smaller fragments, and so on. We can apply rules both forward and backward on sheets of thought. We can even mix both of them if it is convenient for us to do in order to get proofs. See Section 9 for detail.

It will probably happen that we find the logic we have defined lacks some rules, and it is not sufficient enough for our purpose. In this case we can go back to the logic definition phase and modify one or more rules, add rules, or even change the syntax description of the theory. When finished we can go forward in the suspended proof. This kind of flexibility is one of the most useful and characteristic features of EUODHILOS.

4 Creating a Logic

A logic (or a theory) is the most elementary concept in EUODHILOS. A logic is the framework of expressing the concepts and structure we are going to reason with. On EUODHILOS we will create the name and some small data for the new logic at first. Next we will give syntactic structure of the expressions used in the logic. Then we will give logical structure of the logic by giving a number of inference rules, rewriting rules, and axioms. A new logic

⁷Pmacs is the Emacs-like editor installed in SIMPOS.

will be defined in this way. It should occur that the logic we have needs some modification as we find the logic does not meet our purpose or it is even wrong.

In this section, we will see how to create a new logic. We will call it “mylogic”. We have two ways to create a new logic. The first way is to create a logic by selecting the “** new **” item of the logic menu as we saw in Section 4. The other way is to create a logic by copying some data of other logic. Suppose here that the logic menu of EUODHILOS is displayed. The logic “example” that we defined in the previous section is displayed in the menu as in Figure 37.

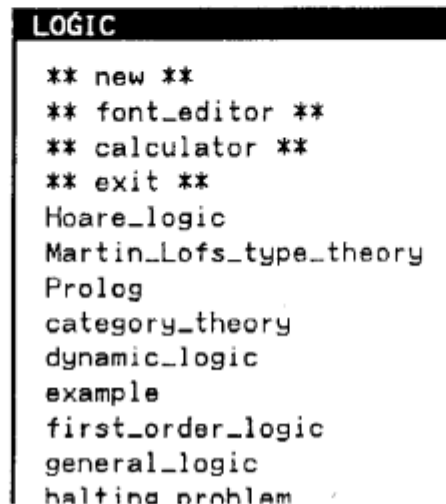


Figure 37: Logic Menu (part)

Let us make a new logic by copying the syntax data from the logic “example” and modifying them. Select the “example” item of the logic menu first. We will get the following menu (Figure 38).

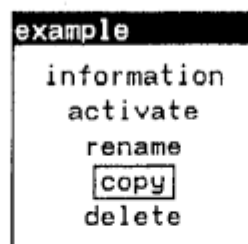


Figure 38: Copying a Logic

The “information” item is for showing several kinds of information on the logic such as remarks, the date when the data are updated, and so on. The “activate” item is for activating(opening) the logic. The “rename” and “delete” items are for renaming and deleting the logic respectively. The “copy” item is for copying the data of the logic and making a new logic.

Let us select the “copy” item. We will get the following input window (Figure 39).

INPUT
copy example to>mylogic

Figure 39: Specifying the target logic name for copying.

We type “mylogic” for the name of the logic we are going to create. Then we will have the copying items menu window shown in Figure 40.

ITEMS TO BE COPIED	
<input type="button" value="OK"/>	
cancel	

key_assignment	
syntax	
axiom	
inference_rule	
rewriting_rule	
derived_rule	
theorem	

Figure 40: Items menu for copying logic data

Here the items which are defined in the logic are highlighted except the “derived_rule” and “theorem” items. They are non-selected items as default. We need to choose them if we want to copy these data to the new logic.

Let us copy the syntax data only in this example. Select “inference_rule” item and change it from the highlighted item to the normal non-highlighted item so that it would not be copied to the new logic. Select the item “OK” and wait for a while during the data are copied. The name of the created logic is added to the logic menu. Select the name “mylogic” and select the “activate” item. Then we will get the top menu of the logic.

Here we are going to modify the syntax of the logic “mylogic” and to define the first-order classical logic. First, let us see how to define a language system in EUODHILOS.

5 Special Symbols

Since EUODHILOS intends to help the user with his or her reasoning, it is one of the important aspects to present easily recognizable expressions. In an ordinary logical system, many symbols, especially logical symbols which have no corresponding keys on the standard keyboard. Further, even if we have a key on the keyboard, it is not unusual that we want something different symbols since we get used to them. Considering this observation on

our preference on the way expressions are represented, to provide a facility to use arbitrary symbols in the expressions is one of the important features of EUODHILOS.

EUODHILOS provides two facilities in order to let us use non-standard symbols. First we can create new symbols by the font editor, which is provided as a built-in facility in SIMPOS. Next we assign these special symbols we have defined on the software keyboard. The assignment to software keyboard varies from logic to logic. Thus we may assign only those symbols that we want to use in the logic we are defining.

For creating a new symbol, we select the **** font_editor **** item in the logic menu(See Figure 37). Then the pattern editor⁸ will be invoked, and afterwards we are supposed to load the font file and create a new symbol as a collection of dots. Figure 41 is a sample of the image of the font editor. Since the font file of EUODHILOS already has a collection of symbols, calling the font editor is needed only when we want to use a very special symbol for the logic we are going to define.

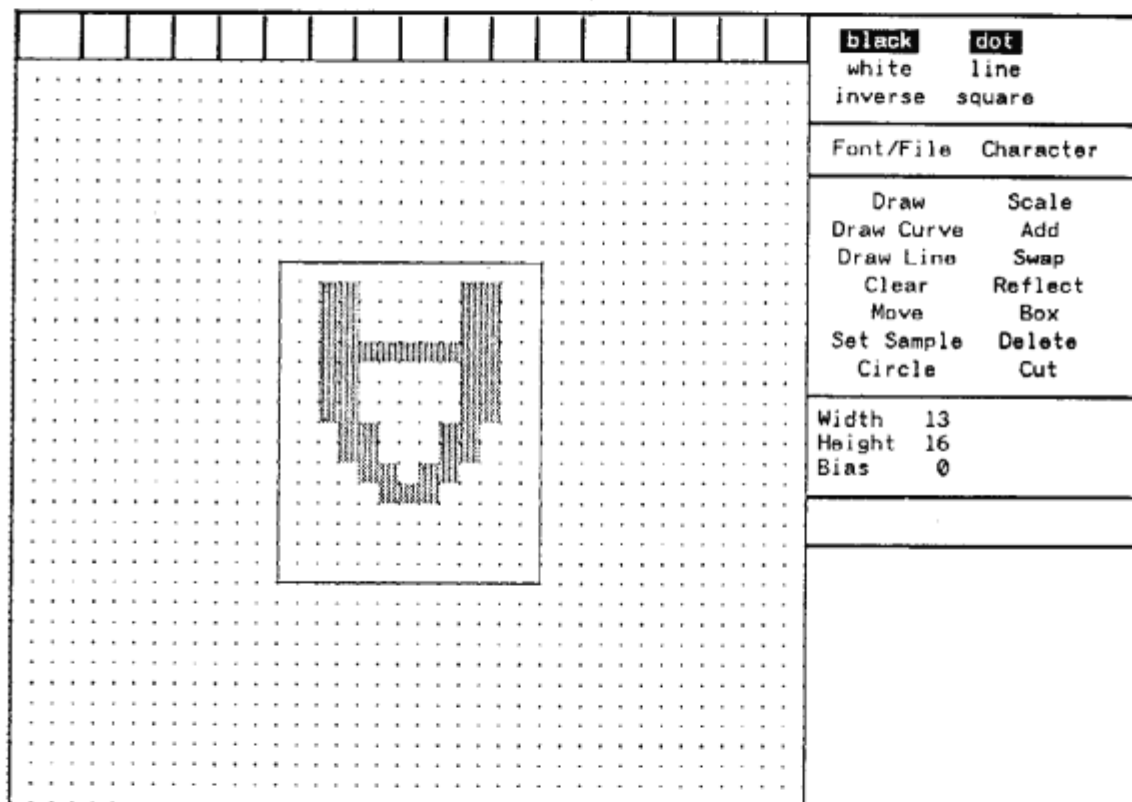


Figure 41: Font editor

We need the software keyboard when we want to use some special symbols defined with the font editor. Symbols can be assigned to the keys in the physical keyboard. Then we can input the symbols by typing the corresponding keys. Select the **"SOFT_KEYBOARD"** item of the top menu of the logic in order to invoke the software keyboard. Figure 42 shows how it looks like.

⁸Please refer to the PSI manual in order to know how to use the font editor in detail.

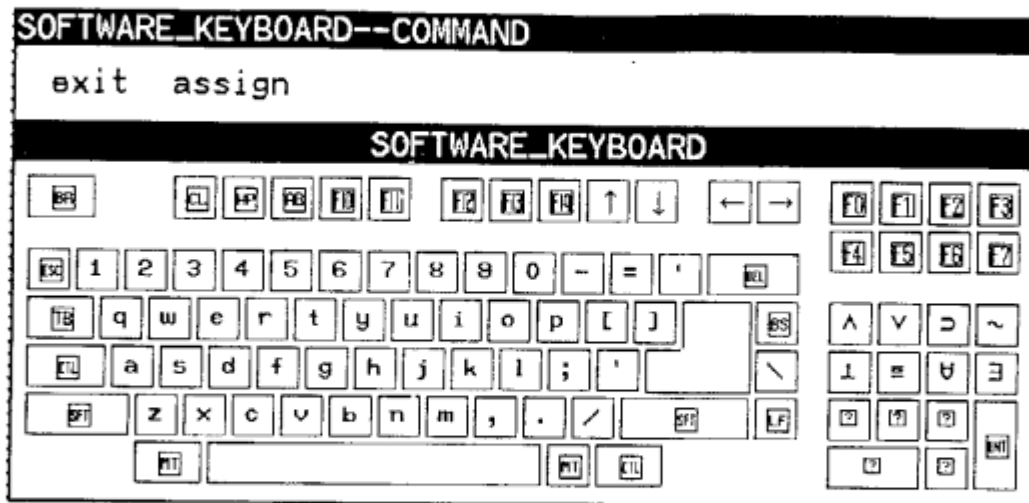


Figure 42: Software Keyboard

There are two regions in this window. In the upper region we see two command items "exit" and "assign". The first one terminates the software keyboard, and the second one change itself into the key-assignment mode (Figure 43). We are in the key-assignment displaying mode. We can see some special symbols in the right part of the keys, and we can see which key corresponds to what symbol.

We are going to assign some symbols to some keys in the actual keyboard. Select the "assign" item in the software keyboard. Then we will have another window of software keyboard for assignments as is shown in Figure 43.

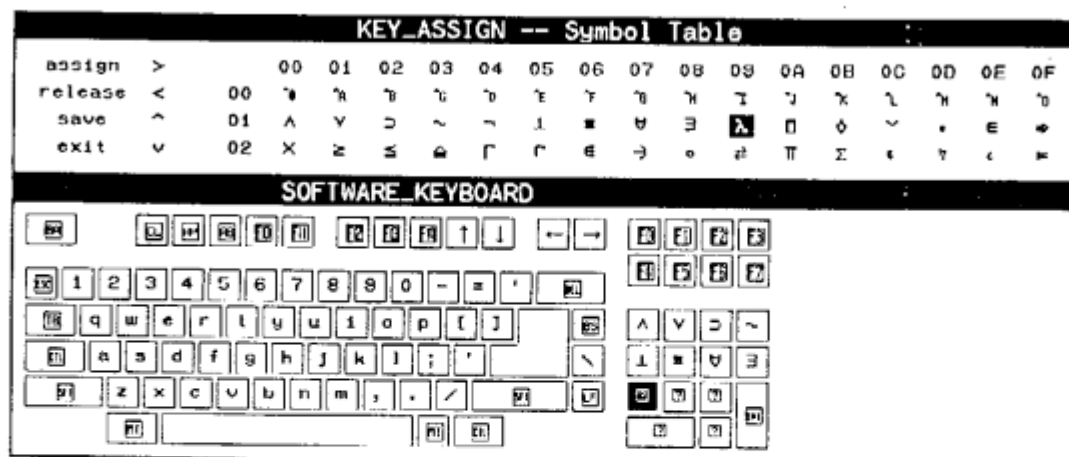


Figure 43: Software Keyboard (Assignment Mode)

The lower half displays how the keys are assigned whereas the upper left part displays the functions and upper right part shows the symbols defined in the standard font file. We can define or modify the symbols with font editor if we are not satisfied with the symbols

already defined, as has been explained before. However it is not recommendable to change the shape of characters into completely different ones, because the symbol table is shared with all the logics. So it is safe to define a new symbol in a new place rather than changing one of the old characters defined already.

We go back to the explanation how we use the software keyboard and see how symbols are assigned to keys. Give a left click on the “ λ ” symbol in the symbol table. It will change the key to be highlighted. Next we give another left click on the key where we want to assign. The key is also highlighted as is shown in Figure 43. Then we select the “**assign**” item in the menu area. Now the assignment completes and the symbol is displayed on the assigned key. In this way we have assigned the “ λ ” symbol on the numeral key “1” in the ten-key area. Similarly we repeat and assign “ \vee ”, “ \supset ”, “ \neg ”, “ \forall ”, and “ \exists ”. Finally select the “**save**” item so that the current assignment will be in effect in the following phases. Now all the assignments have completed. Select the “**exit**” item to get back to the display mode of the keyboard. We will see the new symbols on the assigned keys.

Here let us see how other functions in the assignment keyboard work. The item “**release**” is for releasing the assignment on a key. Give a click on a key which has an assigned symbol on it. And select the “**release**” item. Then the assignment will be released. The four direction items “ \rightarrow ”, “ \leftarrow ”, “ \sim ”, and “ \vee ” are for scrolling the symbol table.

6 Syntax Description

6.1 Syntax Editor

A syntax description window appears when we select the “**SYNTAX**” item in the functions menu of the logic (Figure 5 at page 8). Figure 44 shows how it looks like. In the upper part of the window we see function commands making a line. The lower part is the Pmacs window region for inputting and editing the syntax description of the logic. This is the syntax definition for “mylogic”.

SYNTAX : mylogic						
save	make	test	structure	print	reshape	exit
<pre> formula --> formula, ">", formula ; formula, "^", formula ; formula, "v", formula ; "~", formula ; "(", formula, ")" ; meta_formula ; "a"-"z"; meta_formula --> "A"-"Z". operator "~"; "^": left; "v": left; ">": left. </pre>						

Figure 44: Syntax for “mylogic”

We select the “make” item when we finish the syntax description. Then the description is saved and the parser and unparser are generated. The parser converts an expression given as a string of characters into the corresponding internal form. The unparser, on the other hand, converts an internal expression into an external string form which is used to display the logical expression to the user.

The item “test” is a feature for checking up if the syntax description is good enough. To describe the intended syntax looks fairly easy at first. But soon we will find it quite difficult to write an appropriate one. If we go further, in the proof phase for example, it will be more difficult to find what is or are wrong. The “test” function is quite useful for finding the mistakes in the syntax description. It is a testing facility in order to find syntax errors early and with ease. The “test” window looks as is shown in Figure 45.

```

Syntax_Test : mylogic
>
formula, meta_formula ... success.

Input an expression
>AAB

Input its syntax
>formula

formula ... success.

Input an expression
>

```

Figure 45: Test Window for Syntax Definition

The testing facility works as follows. First, input an expression which is to be tested. Then, input a name of syntactic category(non-terminal). If the given expression belongs to the given category the message “success” will be responded, otherwise we will have “failure”. We may type only the return key as the syntactic category. Then if the expression belongs to at least one of the categories defined in the syntax description, the system will show the list of the names of the categories that accept the given expression, and show also the resulting condition “success”.

The item “structure” displays the structure tree for acceptable expressions. The input window appears and we type an expression. If the expression succeeds in parsing then the tree structure of the expression is displayed. Figure 46 shows how the tree is displayed.

The string expression is displayed at the bottom line of the window. In this sample the expression is “ $\sim \exists x A(x) \supset \forall x \sim A(x)$ ”. The tree structure is displayed above this line. The top part “ \supset ” of the tree indicates it is the constructor of the whole expression. The operator is displayed exactly above the operator in the string expression so that we can see the correspondence between in the tree form and the string form easily. The left and right subtrees are the arguments of “ \supset ”. The root nodes of these subtrees are “ \sim ” and “ \forall ”, which indicate the constructors of the arguments.

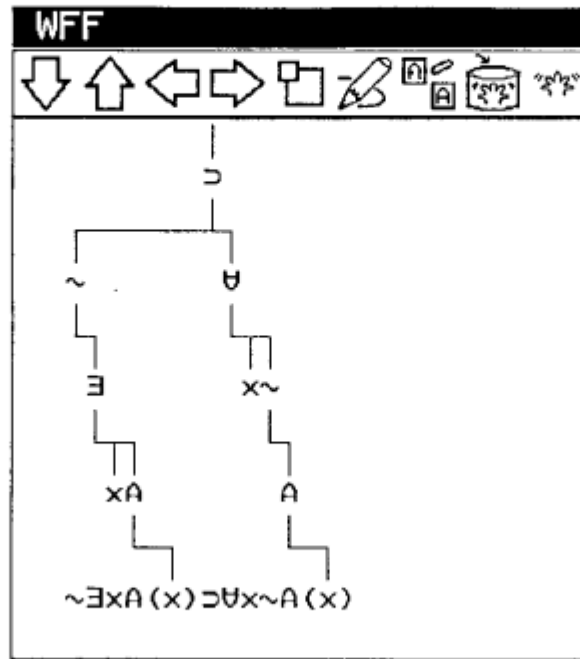


Figure 46: Structure Displaying for an Expression

6.2 Extended DCG part

The DCG part consists of sequences of clauses which define syntactic structure of logical expressions. The following is an example of DCG clauses used in EUODHILOS.

Example:

```
formula --> formula,"=",formula |
           constant;
constant --> "x";
```

However DCG clauses in EUODHILOS is different from those in ordinary DCG. The following is the list of features of EUODHILOS different from the ordinary DCG.

(a) Delimiter

Put semicolons(';') instead of period('.') at the end of the syntax definition clauses. Put a period at the end of the last clause.

Example:

DCG	EUODHILOS
formula --> formula,"=",formula.	formula --> formula,"=",formula;

(b) "Or" Notation

We can use the vertical bar('|') for giving alternative expressions in one clause.

Example:

DCG	EUODHILOS
<code>formula --> formula,"=",formula1.</code>	<code>formula --> formula,"=",formula </code>
<code>formula --> formula1.</code>	<code>constant</code>
<code>formula1 --> constant.</code>	<code>constant --> "x" "y" "z";</code>
<code>constant --> "x".</code>	
<code>constant --> "y".</code>	
<code>constant --> "z".</code>	

(c) Character Range Specification

We can specify consecutive characters for a non-terminal by putting minus symbol('-') in between the characters having the smallest and the largest codes.

Example:

DCG	EUODHILOS
<code>constant --> "x".</code>	<code>constant --> "x"-"z";</code>
<code>constant --> "y".</code>	
<code>constant --> "z".</code>	

(d) Calling an ESP Program

We can write any ESP-methods call as constituents in a syntax definition. The format is as follows.

```
call( <method call> , ... )
```

(e) ESP Program in the Syntax Definition

We can put some ESP programs in the syntax definition part. They will work as instance methods.

Example:

```
:check(A) :- integer(A)
```

Important Notice:

For each clause in the syntax definition, we have to be sure that:

Each clause including at least two non-terminals in its body has to have exactly one component which is declared as a constructor in the constructor declaration part.

6.3 Constructor declaration

The constructor declaration part consists of two subparts; the *operator declaration* and *predicate declaration*. The operator declaration subpart declares operators. Prefix, infix, and postfix binary operators, should be declared in this subpart.

Operators are lined after the key word "operator" in the order of precedence. The first one has the highest precedence, and the second one the next highest, and so on. We can parenthesize two or more operators if they have the same precedence. Put comma(',')

between the parenthesized consecutive operators. Put semicolon(';') at the end of the operators and right parentheses to delimit the operators list. Both constants (i.e. character strings in between two double quotation symbols) and non-terminals are allowed to be declared as operators. We may put ":left" or ":right" for specifying the associativity of the operator.

Predicates are lined after the word "predicate" delimited by comma(',') . Put period('.') at the end of the list. Here also both constants and non-terminals are allowed in the list.

Here is a sample of constructor declaration.

Example:

```
operator
  "Q";
  "M";
  (not, bind_op);
  "/" :left;
predicate
  pred_sym1, pred_sym2, function_sym.
```

6.4 Remarks on Syntax Description

1. The order defined in the DCG part and that in the constructor declaration must be matched. Mismatching of these two may lead failure and/or misinterpretation in EUODHILOS. So, it is highly recommendable to declare the operator precedence only in the constructor declaration part instead of putting the operator precedence in the DCG part.
2. We have to define one of the constituents of a clause as a constructor if the body part of the clause includes at least two non-terminals. In this case we might declare either a constant(string) or a non-terminal as the constructor. We can use an empty string("") as a constructor. This might be useful when we define a construction without explicit constructor as, for example, in dealing with combinatory logics.
3. Binding operators such as \forall and \exists in predicate logic, and λ in lambda calculus need special treatment. They bind a variable in the expression followed by the variable. That is, the variable occurrences in the expression are no more free occurrences of the variable. Binding operators must be declared so that they have the syntactic category "bind_op". The following is a typical example of defining binding operators.

Example:

```
formula --> bind_op, variable, formula;
:
bind_op --> "V"|"E";
```

```

      :
operator
      bind_op;
      :

```

4. Meta symbols are defined as the syntactic category name having the prefix "meta.". We can instantiate meta symbols by substituting an appropriate expressions which are supposed to include not only object symbols but also meta symbols.

7 Inference and Rewriting Rules

Inference and rewriting rules are key part in the logic. We apply them to axioms, theorems, assumptions, and proof fragments so that we get the new conclusion. Most of the proof fragments grow in this way. In EUODHILOS, these rules are represented in tree form and therefore easy to define as well as to recognize what they are saying. Inference rules may have an assumption for each premise. It also be able to have one or more side conditions in order to give restrictions to how the rule is applied.

This section explains how to open and close the rule menu, how to define the rule, and how to give one or more side conditions to inference rules. Since inference and rewriting rules have much similarities most part of this section can be applied to both cases.

7.1 Opening and Closing an Inference Rule Menu

When we select the "INFERENCE_RULE"("REWRITING_RULE") item in the function menu of the logic(See Figure 5 at page 8), the inference rule menu(rewriting rule menu) is shown. Inference rule menu looks as in Figure 47.

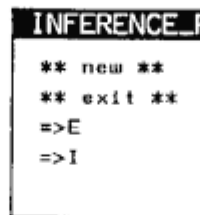


Figure 47: Inference Rule Menu

The items except the first two are the list of inference rules already defined. We are supposed to select the "** new **" item if we want to define a new inference rule. If we want to see or modify a rule already defined then we select the rule name item. Since we already saw how to define a new inference rule in Section 2, we suppose, in the following sections, that we have a rule and want to modify it.

The rule menu closes when we click the "** exit **" item.

7.2 Defining and Modifying an Inference Rule

By selecting the rule name item in the rule menu, we can get the functions menu of the rule which looks as follows.

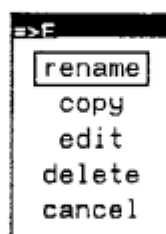


Figure 48: Functions Menu of a Rule

We can change the name of the rule by selecting the “rename” item. Then the input window will appear and we are supposed to type the new name.

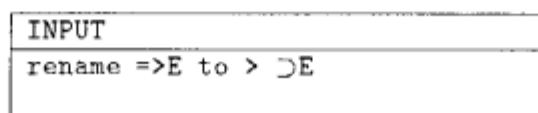


Figure 49: Inputting the new name

The old name disappears and new name of the rule appears in the rule menu.

The “copy” item is for copying the rule and make a new rule. The input window appears also in this case and we give the name of the new one. The “edit” item is for displaying and editing the rule and “delete” item is for deleting the rule definition.

When we select the “edit” item, then the rule window like Figure 50 appears.

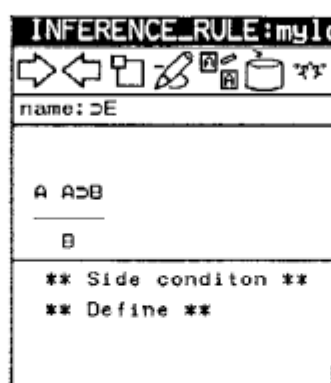


Figure 50: Inference Rule Window

We can change the name, body part (i.e. assumptions, premises, and conclusion) and the side condition(s) of the rule. Give a left click on the name field for changing the name, on

an assumption, a premise, the conclusion for changing the assumption, premise, conclusion, respectively. Then the input window appears on which the old expression is displayed. We are supposed to modify the expression and put the cursor at the end of the expression in the line and type carriage return to replace with the new one. The modified expression will be shown. See also Section 2.7 on what the icons mean and how we can input the name, premises, and conclusion of the rule.

7.3 Side Conditions

When we select the “** Define **” item with the middle click in the inference rule window, the side condition editor which is shown in Figure 51 will appear.

SIDE CONDITION :DE	
t	is free for *x* in *P(x)*
x	is bound in *P(x)*
x	is not free in *P(x)*
a	is an eigen variable
checker's	*Class_name*, *Method_name*, *File_name*
comment is	*Comment*

template_selection:	menu input all_occurrence
<input type="text"/>	
<input type="text"/>	
delete delete_all exit cancel	

Figure 51: Side Condition Definition Window

There are four regions in the window. From the top one down to the last, they are side condition selection menu region, side condition input region, side condition display region, and command menu. In the side condition selection menu region, there are four built-in elementary side condition items, side condition checker specification item, and comment item. The expressions between “*” indicate that it is representing a place holder of the condition. We have to give the contents of these place holders in order to define a side condition. Take the first condition for an example. The item is “*t* is free for *x* in *P(x)*”, which has three place holders. When we select this item, the prompting messages for these place holders appear in the side condition input region. Since the first one is “t”, we will get the following prompting message.

t?>

We type the corresponding expression for it. Let us give "T" for "t", "X" for "x", and "P" for "P(x)".


The expressions "T", "X", and "P" must appear in the expressions used in the definition of the inference rule. Then we have the side condition in the side condition display region as follows.

T is free for X in P

We can repeat this procedure of defining side conditions. If we want to give special side conditions, we can attach a condition checker through the checker specification item. We will not see the detail about this facility in this article. We can give a comment for giving a warning, remark, or some message like these.

The region also includes the items for template selection. Template is used for specifying the occurrences of an expression subsumed in the whole expression. We choose one among the tree items: "menu", "input", "all_occurrence". "menu" is the default value. These modes mean as follows. First the "menu" mode indicates that the menu is shown to the user and he/she selects which one to be chosen. Next the "input" mode indicates that the user gives the template expression that specifies which occurrences to be taken. Last the "all_occurrence" mode indicates that all the occurrences should be taken.

Using the items in the command menu we can delete all or some of the side conditions. By selecting the "delete" item we can delete a side condition. We need to select one of the conditions before using this function. If we want to delete all the side conditions all we have to do is to select the "delete_all" item. As is easily seen "exit" and "cancel" items are for closing the side condition window and cancelling all the changes, respectively.

When we finish the definition then select the "exit" item to get back to the inference rule window. Be sure to click on the  icon of the inference rule editor window and save the definition data so that the definition actually works as we apply the rule.

7.4 Rewriting Rules

Rewriting rules can be defined with almost the same way as of inference rules. The differences are the rewriting rule has no assumption, it has only one expression over the line, and it has no side conditions.

The most significant difference between the two kind of rules in application are the inference rule specifies the whole logical expression whereas rewriting rule specifies one or more subexpressions in a logical expressions. Since the rewriting rules have one upper expression and one lower expression, the rule may be applicable repeatedly several times. We can specify a number which gives the limit of times of application of rewriting rules. See the reference manual for detail.

8 Axioms

Axioms and assumptions are the starting formulas in a proof. Assumptions are statements that suppose temporarily and should be discharged. Axioms do not have such conditions. They stand as themselves. Axioms are declared as such by being put in the axiom definition window. This section explains how to declare axioms.

When we select the "AXIOM" item in the function menu of a logic we will get the axiom window as shown in Figure 52.

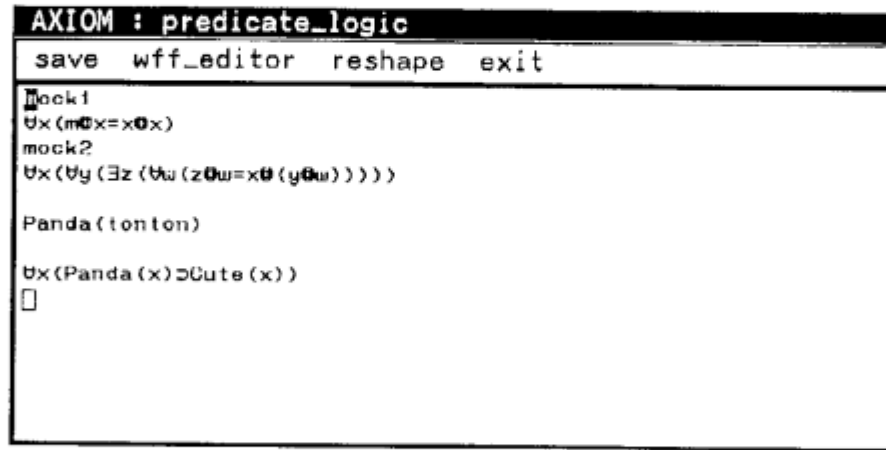


Figure 52: Axiom Window

In the upper part four command items make a line. They mean as follows.

save This item is used when we save the data so that the axioms in the window work in the sheets of thought.

wff_editor This item is for calling the tree structure editor for showing and editing the axioms.

reshape This item is for resizing the window.

exit This item is for closing the axiom window.

In the lower window we can modify the axioms as well as input them. Two lines are used for specifying one axiom. An odd-numbered line in the window is for naming the expression in the following line. For example the first line "mock1" in Figure 52 is the name of the axiom " $\forall x(m0x=x0x)$ " in the second line. Leave the name line empty if we do not give a name to the axiom. When we put an axiom in a sheet of thought, we are supposed to select an axiom in the axiom menu. In the axiom menu, each item is the name of the axiom if it is given, otherwise the expression itself is displayed.

9 Proof Construction on a Sheet of Thought

Providing a comfortable proof environment is one of the important features of EUODHILOS. Since EUODHILOS intends to provide the facilities to let the user deal with as a great variety of logics, we give up assuming we have one or more efficient theorem prover for them. So, it is the user to decide what to do in the search process for the theorems and/or the proofs for conjectures. A sheet of thought is the environment for constructing proofs in EUODHILOS.

Constructing a proof for a theorem on a sheet of thought starts by putting assumptions, axioms, and theorems. Proof fragments will grow as we apply rules, connect several fragments into one big fragment, and eventually we might get complete proofs for theorems. Sometimes we will get new results by combining some results we have already proved. Sometimes we have a conjecture which is waiting to be proved. Giving a proof of the formula is our goal in this case. If we have a goal to be proved, we will apply rules backward to the goal and will get one or more subgoals to be proved. Since we have a various styles of proof construction, the reasoning assistant system has to be flexible enough to deal with these all. Therefore the functions of a sheet of thought are designed so that they are flexible and the proof fragments on the sheets are easy to recognize. We can derive both forward and backward. We can make a proof fragment by forward reasoning and some others by backward and combine them to get a complete proof. The results we have can be saved as theorems and derived rules. A theorem can be used just like an axiom and a derived rule as an inference rule in the following process of constructing more sophisticated results.

In Section 2.8 we learned how to make a simple proof in a very basic way. In this section we learn some more sophisticated facilities which are quite useful in proof construction.

A sheet of thought can be opened from several different kinds of menus. The most typical one might be as follows. When we select the "PROOF" item in the functions menu of a logic (Figure 5, page 8), we get the proof group name menu.

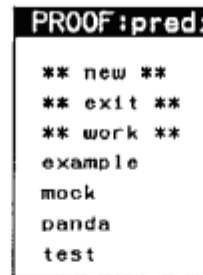


Figure 53: Proof Name Menu

We will get a new sheet of thought when we select the "** new **" item.

The "** work **" item and the rest are the list of *proof groups*. A proof group is a collection of proof fragments. This concept is useful if we develop several kinds of theorems in one logic. The "** work **" proof group is a special proof group in which proof fragments having no proof group name specified are saved. Let us select the work proof group. Then the proof fragments menu will appear (Figure 54).

The "** new **" item in the menu can be used also for opening a new sheet of thought. The "** exit **" is for closing the menu. The other items are the list of root formulas of



Figure 54: Working Proof Group

proof fragments that belong to the group.

As we select a proof group name item in the proof name menu, we have the command menu for the proof group.

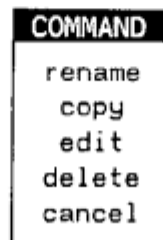


Figure 55: Command Menu for Proof Group

The functions displayed in this menu have the same meaning to those already described. If we select the “edit” item then the proof fragment menu just like that of work group appears.

We select a formula item in the proof fragment menu, so that we get the command menu for this proof fragment.

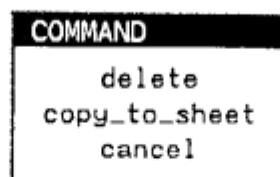


Figure 56: Command Menu for Proof Fragment

The “delete” item is for deleting the proof fragment and “cancel” for closing the menu itself. The “copy_to_sheet” item is for copying the data into a sheet of thought. We will have the selection menu for choosing the sheet where this fragment is copied into.

The “** new **” item opens a new sheet of thought and the proof fragment will be copied into this sheet. The other items are the names of the sheets which are already shown on the screen. If we select one of them, the fragment is copied into the sheet and the proof fragment appears as the rightmost fragment in the sheet.

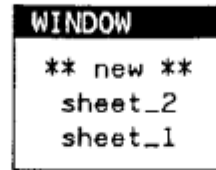



Figure 57: Menu for Selecting the Fragment where the Fragment is Copied into

In order to close a sheet of thought we have to decide which proof fragments should be saved and where to be saved. If we want to save all the fragments in the work group then we just click the “” icon. Then all the proof fragments are saved and the sheet terminates. If we want to save one fragment, we choose the fragment and give a left double clicks or just type the return key. Then the following command menu appears (Figure 58).

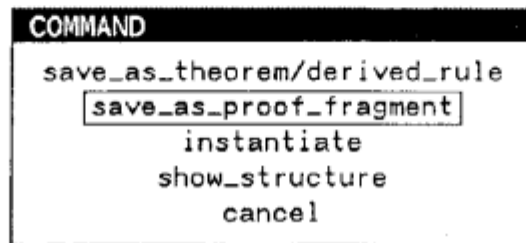



Figure 58: Command Menu for `save_as_proof_fragment`

Let us select the second item “`save_as_proof_fragment`”. Then the menu for specifying the proof group name appears (Figure 59).



Figure 59: Proof Name Menu

Now we select the “`** new **`” item for creating a new group and saving the proof fragment in the group. Then we select the “`** work **`” or the group name item according to which one we would like to save the proof fragment into.

If we give a click on the “” icon, then the confirmation window appears for selecting which fragments to save in the work group and which one to delete (Figure 60).

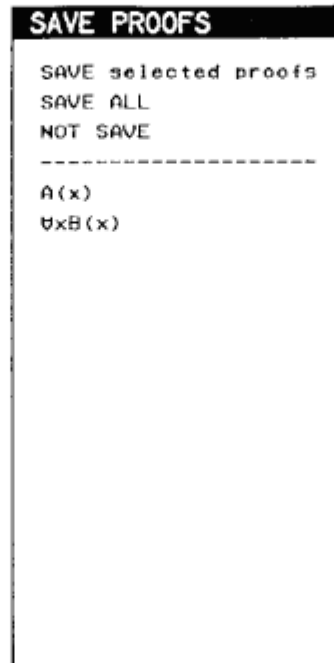


Figure 60: Menu for Saving Data as a Sheet of Thought Closes

The uppermost three items are functions we need to choose and the rest below the dotted line is the list of the results of the proof fragments in the sheet. If we want to save or not to save all the fragments then what we do is just select the “NOT SAVE” or “SAVE ALL”, respectively. Otherwise we select the results of the list those are supposed to be saved in the work group, and select the top function in the function items. These fragments will be saved in the work group.

9.1 Assumption

A proof on a sheet of thought begins with putting an assumption (or a goal for backward derivation), an axiom or a theorem. This section deals with assumptions. In order to put an assumption on a sheet of thought, we first give a left click in the sheet where the mouse cursor is not on a expression. Then we get a box marker; the nearest available place of the cursor. Next we give another click to get the input window for typing the assumption. When we type the carriage return, the expression appears where the box marker was shown. The assumption is parenthesized with “[” and “]”, which indicates this expression is an assumption (or a goal if it would be used in a backward derivation). Each assumption has a number called the assumption ID(identification) number. Two assumptions having the different ID numbers are treated differently even if they have the identical expressions. On the other hand, two distinctive assumptions have different ID numbers.

If we give a left click on a stand-alone assumption, assumption would be surrounded by a box marker. If we give another click or type the return key, a command menu appears (Figure 61).

COMMAND
edit
input_axiom/theorem
wff_editor
is_axiom/theorem
ID_number
instantiate
show_structure
cancel

Figure 61: Command Menu for Stand-Alone Assumptions

The items in the command menu for stand-alone assumption shown in Figure 61 represent the following functions.

edit: For modifying the assumption. As we select it, the assumption would appear in the ordinary input window and would be waiting for being modified.

input_axiom/theorem: As we choose one of the axioms or theorems the assumption would be replaced with it.

wff_editor: For modifying the assumption. As we select it the “wff_editor” is invoked. “wff_editor” is a structure editor we can manipulate formulas in tree form.

is_axiom/theorem: The system checks if the assumption is an axiom or a theorem. If it appears to be one of them, its status as an assumption changes to either an axiom or a theorem according to which one it belongs to.

ID_number: This item changes the ID number of the assumption. If the number given to this assumption has been assigned to other assumption and the expressions of these assumptions are not identical, the ID number can not be changed.

instantiate: Instantiating the metavariables included in the assumption. The metavariables will be displayed one by one in the input window and we are supposed to give the corresponding expressions to be substituted.

show_structure: Displaying the structure of the selected formula.

cancel: Terminating this menu.

9.2 Derivation

Two kinds of derivations are allowed on a sheet of thought: forward and backward derivations. Suppose we have an inference rule named “ $\wedge I$ ” which is defined as follows (Figure 62).

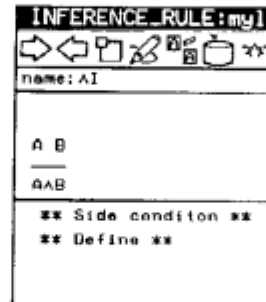


Figure 62: \wedge -Introduction Inference Rule

In the forward derivation, we have “A” and “B”, and apply the rule to get the result formula “ $A \wedge B$ ”. On the other hand, in the backward derivation we have “ $A \wedge B$ ” as the goal to be proved. We apply the rule to this goal formula and get the two subgoals “A” and “B”. In the following two subsections we will have precise explanations about how these derivations are achieved by way of sample constructions.

9.2.1 Forward Derivation

As the first example, We are going to see how the forward derivation actually goes on a sheet of thought in this section.

Suppose we have two formulas “A” and “B” on a sheet of thought, and we are going to apply the inference rule “ $\wedge I$ ” to these two and to get “ $A \wedge B$ ”. These formulas may be either of assumptions, axioms, theorems, or the conclusion(or root) formulas of proof trees(proof fragment) already constructed. If the sixth icon from the left at the icon area in a sheet of thought is down arrow (\downarrow), we are in the forward derivation mode. If it is the up arrow (\uparrow) then click the middle button of the mouse while the mouse cursor is on the up arrow icon and change the mode. Now we give right click to the formula “A” and also give another one to “B”. The following window(Figure 63) shows how it looks like.

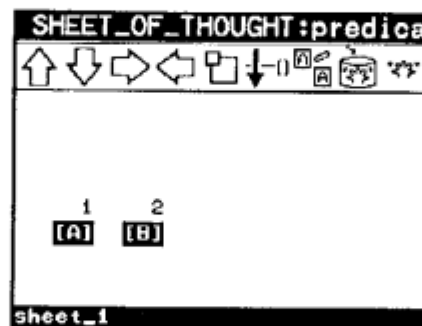


Figure 63: Two assumptions are selected for forward derivation

Next we make double left clicks or press the return key. We will get the rule menu list as shown in Figure 64.

INFERENCE_RULE	REWRITING_RULE	DERIVED_RULE	COMMAND
AE-left	Acom	AE-right	cancel
AI	=sum	\Leftarrow	discharge
BI-init	dot-assoc	\Leftarrow to \Leftarrow	search_rule(input)
BI-ored	pr1-E	=trans	search_rule
BI-sum	pr2-E	$\emptyset(X, Y) \vdash \text{dom}(\emptyset(X, Y)) = X$	--call_prover--
BI-term	--reverse_application--	Eq: largest_equalizer	
BF	Acom(R)	$\text{Eq}(F, G) \vdash \text{Eq}(F, G) : \text{mon}$	
$\Leftarrow E$	=sum(R)	$F \vdash F; \text{dom}(F) \dashv \vdash \text{cod}(F)$	
$\Leftarrow I$	dot-assoc(R)	$F \vdash F = F$	
squot-E(large)	pr1-E(R)	$F, Z : \text{init} \vdash \text{init}(Z, \text{dom}($	

Figure 64: Rule Selection Menu for Derivation

The rule selection menu actually consists of four smaller menus. From left to right they are inference rule menu, rewriting rule menu, derived rule menu, and command menu. We select the name in the inference rule menu for applying an inference rule, in the rewriting rule menu for rewriting rule, and in the derived rule menu for derived rule. We can apply the rule in a slightly different way with the items in the command menu. Each submenu allows scrolling to find out the intended rule in the long list of names.

The rewriting rule menu is separated in two. The upper part is the list of the rules for the application in the normal order, whereas the lower part is for the application in the reverse order. This makes sense because a rewriting rule is supposed to give a way of replacing an expression with an equivalent one.

The derived rule menu is for applying a derived rule. A proof fragment can be used for later proofs in two ways. If it does not depend on any assumptions the conclusion formula of the proof is a theorem. Otherwise the conclusion formula of the proof fragment depends on the assumptions that have not been discharged. In this case this proof fragment can be stored as a derived rule where assumptions are considered to be the premises and the conclusion is the conclusion of the application.

The rightmost command menu has various functions.

cancel: Canceling this application.

discharge: Discharging an assumption.

search_rule(input): Searching the applicable rules by giving the conclusion of the application. A prompting message for the resulting formula appears and the user input it. Then the system searches the rules that derive the result we have typed.

search_rule: Searching rules which has premises matching to some or all the results on the sheet which we have selected as premises just before we called this menu. Usually more than one rules can be applicable, so the system makes a list of applicable rules and we choose one of them.

The items area after “--call prover--” is for displaying the list of provers which have been implemented and have been attached to the logic. The detail will be explained in the next section.

Let us move back to the application of the rule. We have applied the “ $\wedge I$ ” rule in the inference rule menu and now we have the proof fragment which has the formula “ $A \wedge B$ ” as the conclusion. Figure 65 indicates where we are now.

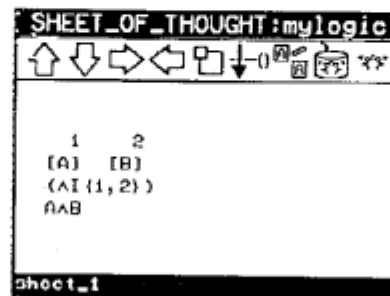


Figure 65: $A \wedge B$ is derived by forward application

We have seen how we can apply a rule and get a new conclusion in the normal way. The rule has no assumptions to be discharged as it is applied. Now we are going to see how to apply a rule that has one (or more) assumption(s). Corresponding to the assumption of the rule, the proof fragment would have the assumption to be discharged as the rule is applied to it. Here is how we specify the assumption to be discharged. The middle button click of the mouse corresponds to this specification. An underline appears at the specified assumption as we give the click.

Now let us have a look at the next figure.

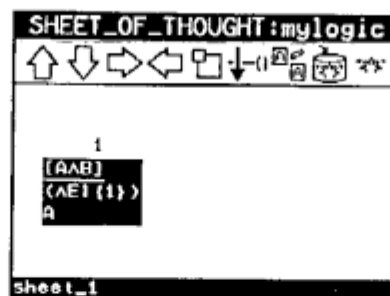


Figure 66: Specifying discharging assumption

In Figure 66 we have selected whole proof fragment as well as specifying an assumption that is going to be discharged in the next application of a rule. We can see the underline just below the assumption “[$A \wedge B$]”.

Just as we have done in order to get the conclusion formula “ $A \wedge B$ ” we make a double left clicks and choose “ $\supset I$ ” in the inference rule menu. The assumption is now discharged by this application. Let us have a look at Figure 67, where the conclusion formula “ $A \wedge B \supset A$ ” does not depend on any assumptions.

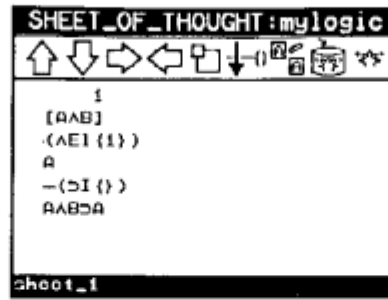


Figure 67: Assumption $A \wedge B$ is discharged.

We know this because we see something like “ $-(\text{OI}\{\})$ ” just above the conclusion. Here “ $\{\}$ ” says that the set of depending assumptions of the conclusion is empty.

We can postpone the specification of discharged assumption. Then the asterisk is attached at the name of the rule applied to the derivation to indicate that this derivation is not complete yet.

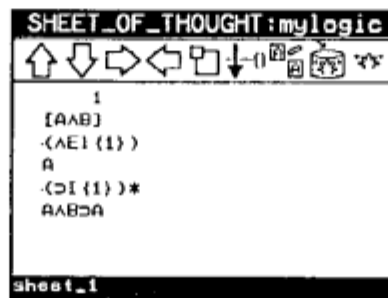


Figure 68: Assumption is not discharged as the rule applied.

We may discharge this expression whenever we like by specifying the appropriate assumption by middle click and execute discharging by giving a double left clicks.

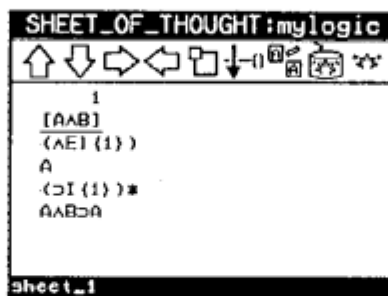


Figure 69: Discharging assumptions.

Now we get the theorem as is displayed in Figure 67.

9.2.2 Backward Derivation

We can change the direction of the derivation by clicking the forward/backward icon for derivation. To make a backward derivation, first select an assumption with a right click. Then the assumption is highlighted in a black box. This shows that the assumption is selected by a right click. The assumption to be selected may be the one standing alone, may be a leaf of a proof fragment.

Figure 70 shows when an assumption is selected.



Figure 70: Assumption is selected for backward derivation.

Next give a double left clicks on the sheet, and the rule selection menu appears. If we select the appropriate rule in the menu, the rule is applied and the premise('premises' if the rule has more than one premises) of the rule appears as the assumption of the proof fragment.



Figure 71: Rule $\wedge I$ has been applied backward.

We usually apply one rule in one application. However we have three ways to make the applications shorter.

The first way is to call a prover. The last items after "--call prover--" in the application rule menu are the list of provers which we have implemented and attach to the logic.

System calls the prover we have selected. If the prover program fails in searching a proof in the current situation, then it fails. If the prover has succeeded and returns the proof data then the data is displayed in an ordinary proof tree form. If it does not give the proof data and just returns in success, then the system treats the conclusion as a theorem without proof structure.

The second way to shorten the application is to use a derived rule. A derived rule is a sort of "macro-facility" in ordinary programming languages. A proof fragment can be seen as a rule the assumptions the conclusion (root formula) of the proof fragment depends on are premises while the conclusion formula of the fragment is the conclusion formula of the rule. If we store a fragment as a derived rule and give a name to it, then this pattern of proof can be treated as one application of a rule.

The third way is to use a theorem. A theorem can be interpreted as a derived rule which has no assumptions the conclusion depends on.

Actually, there is one more way to reduce the number of applications. It is the repetitive applications of a rewriting rule. When we apply a rewriting rule the system tries to apply the rule up to the limit number of times. The initial limit number is three, but we can change it through the customizing window.

9.3 Metavariables

A metavariable is a symbol which represents a class of expressions such as formulas, terms, numerical expressions, and so on. An expression that includes one or more metavariables is called a metaexpression. For example, a metaformula is a formula expression which includes one or more metavariables. We can substitute another expression for a metavariable in an expression. The expression we get in this way is called an instance of the original metaexpression. We can instantiate several variables simultaneously as well. A metavariable should be substituted with an expression having the same type as of the metavariable; e.g. a formula for a metavariable declared as a metaformula, a term for one declared as a metaterm. We will call an instantiation as a ground instantiation if the resulting formula has no metavariables in it. A metaexpression is considered to represent the collection of all the ground instances of the expression. A proof fragment which includes metaexpressions can be called a proof schema also. A proof schema represents the set of ground proof fragments which can be obtained by instantiating it.

Metavariables are conceptually equivalent to the normal (or object) variables in a logic. However, in EUODHILOS, we use metavariables as "place holders" which are used for representing the schematic expressions, whereas the ordinary variables are used as parts of logical expressions which we can not substitute expressions without explicit applications of rules to them, e.g. α -conversion rule in the λ -calculus. So, we can instantiate any metavariables at any time we want and the whole expression or even the whole proof fragment be instantiated with this substitution.

Metavariables are supposed to be declared as non-terminals whose names begin with the string "meta_". If we want to declare a symbol "A" as a metavariable for "formula" (i.e. metaformula), we can do it by adding the following clause in the syntax declaration.

```
meta_formula --> "A"
```

Note that the correspondence between the syntax for “formula” and “meta_formula” is left to the user. An easy way to do is put the following clause in the declaration.

```
formula --> meta_formula
```

On a sheet of thought assumptions and axioms can be schemata. Almost all the inference rules naturally should be the schematic relations between premises and conclusions.

9.3.1 Instantiation

What we have to do first for instantiating metavariables in a proof fragment is to select the object. The objects that we can select for instantiation are proof fragments, subtrees of proof fragments, or a node expression in a proof fragment. We give a left mouse click on the root formula for the first two cases and on the formula for the third case. Then a box surrounds the whole (or sub) proof fragment that the root formula represents. For the third case we need to give another left click on the same formula. Then the box changes itself to surround the specified formula. Figure 72 shows how the proof fragment having “ $P \wedge Q$ ” as its root is selected.

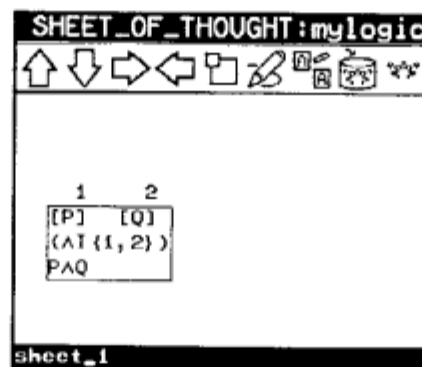


Figure 72: The whole proof fragment is selected.

Here we make double left clicks or press the return key. We have to be careful that we are in the pencil mode. The fourth icon from the right in the icons area indicates if we are in the pencil mode or the eraser mode. We have the command menu window as shown in Figure 73.

The items in the command menu mean as follows.

- save_as_theorem/derived_rule:** Save the proof fragment as either a theorem or a derived rule. If the root expression does not depend on any assumptions it would be saved as a theorem, otherwise as a derived rule.
- save_as_proof_fragment:** Save the proof fragment in the proof group area. We can choose a proof group name where the proof fragment is saved.
- instantiate:** Instantiate the metavariables that appear in the specified box; the proof fragment or the expression.

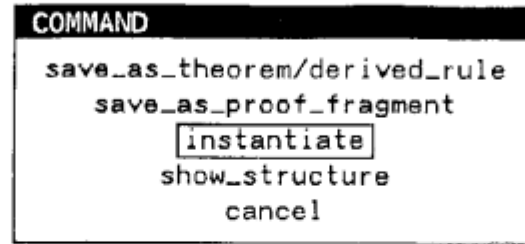


Figure 73: Command menu for a proof in a group

show_structure: Display the structure of the expression we have selected in a tree form.

cancel: Terminate the menu without doing anything.

Let us select the “*instantiate*” item in the menu. Then the system searches the whole expression and collect the metasymbols used in the expression(or sub-proof). We need to answer to the each enquiry and specify what expression is to be substituted for each metasymbol. Figure 74 shows how this enquiry looks like.

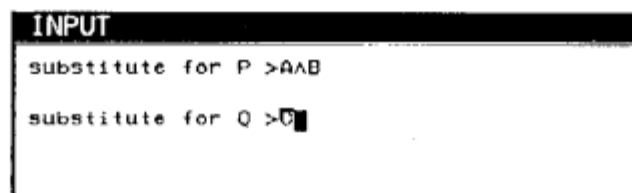


Figure 74: Enquiring the expressions for substituting the metavariables

In this example, we have two metasymbols in the proof fragment, say “*P*” and “*Q*”. Figure Enquiring the expressions for substituting the metavariables says we are substituting “ $A \wedge B$ ” for “*P*” and “*C*” for “*Q*”. As the result of these substitutions we have Figure 75.

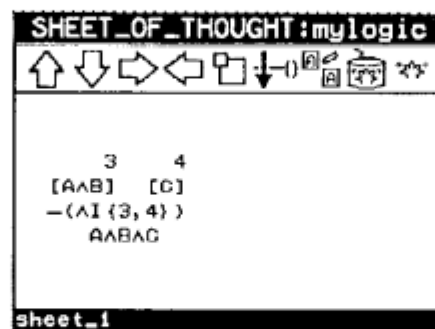




Figure 75: Expressions $A \wedge B$ and C are substituted.

If we select an expression for instantiation the metavariables appearing only in the formula are instantiated.

9.4 Editing Proof Fragments

We have learned some basic features of sheet of thought so far, such as how to put assumptions and how to apply rules to get new results. In this section we will see other editing facilities sheet of thought provides.

9.4.1 Deleting Proof Fragment and Node Expression

The first thing we do for deleting an object is to select it with the left mouse click at the root of the subfragment to be deleted. By the first click the subfragment (or the whole proof fragment if we give a click on the root of the fragment) is highlighted with an ordinary marking box. If we want to delete only the root formula then give another right click on the root formula. Then the box surrounds only the formula we have chosen. The third icon from the right indicates the mode either insertion (the pencil icon “”) or deletion (the eraser icon “”). Finally give a double left clicks for doing the action. Figure 76 shows that an assumption is selected for deletion.

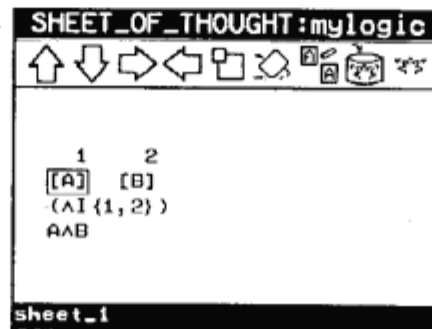


Figure 76: An assumption is selected to be erased.

If we delete the assumption “[A]” then it is impossible the remaining parts of the proof fragment form a single valid proof fragment. So the remaining parts, “[B]” and “A & B” make separated proof fragments (Figure 77).

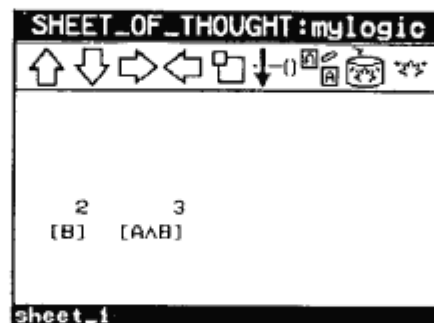




Figure 77: Assumption “[A]” is erased and the rest are separated in two.

9.4.2 Copying Proof Data

We specify both “which one to be copied” and “where will it be copied” for copying. The third icon from the right indicates which one of *copy* () or *move* () mode we are in. By making a middle click on the icon we can change it from one to the other. Select the tree to be copied by a right click and make a box marker where the tree will be copied into by a left click (Figure 78).

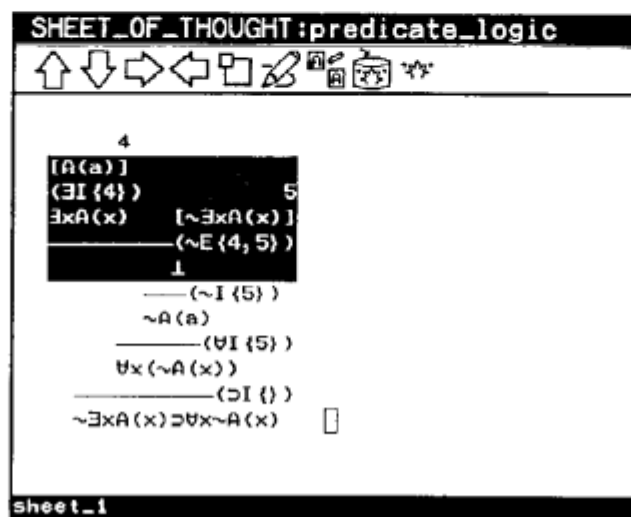


Figure 78: A subproof is going to be copied.

As we make double left clicks or press the return key and give the command *execute* we get the result as is shown in Figure 79, where the subproof that was selected has been copied to the next of the original proof fragment.

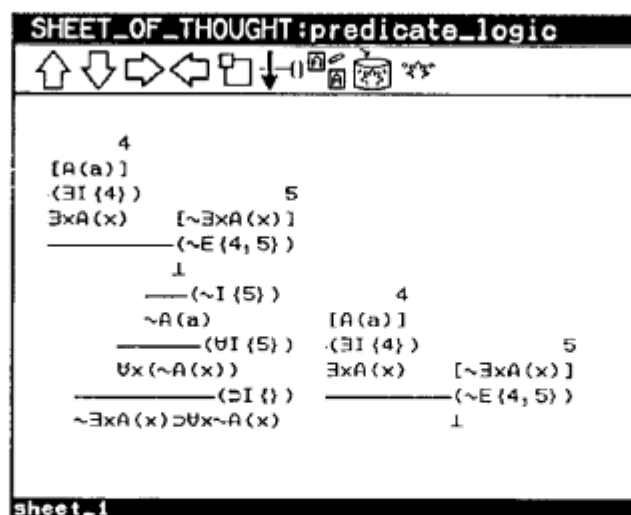


Figure 79: The subproof has been copied.

9.4.3 Moving Proof Data

Moving a data is almost the same as copying. The only difference in the process is that after the new data is created the original data will be erased in the move mode whereas it will remain unchanged in the copy mode. Select the tree to be moved by right click and position the place where the tree will be moved into by a left click. Figure 80 shows where we are.

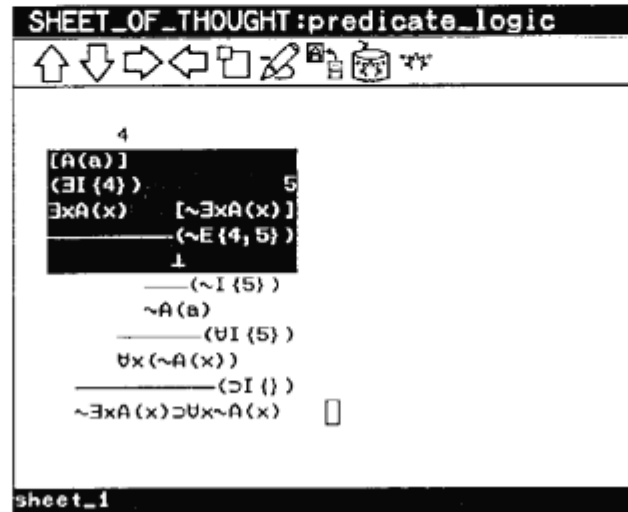


Figure 80: The subproof having “ \perp ” as the root is selected for moving.

When we make double left clicks or press the return key while we are in the pencil and move mode we will get Figure 81.

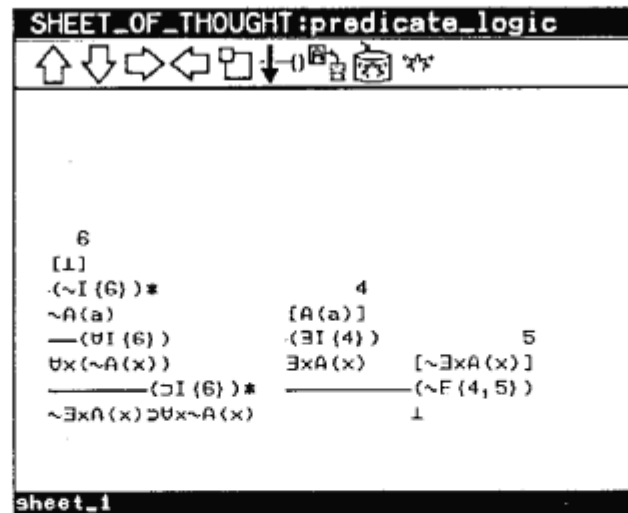


Figure 81: The subproof has been moved.

9.4.4 Connecting Proof Fragments

Connecting two proof fragments by attaching a root node and a leaf node can be achieved by copying or moving. We select a whole proof tree by clicking the right mouse button on the root expression of the tree. We also select a leaf of a tree by clicking the left mouse button on the assumption (Figure 82).

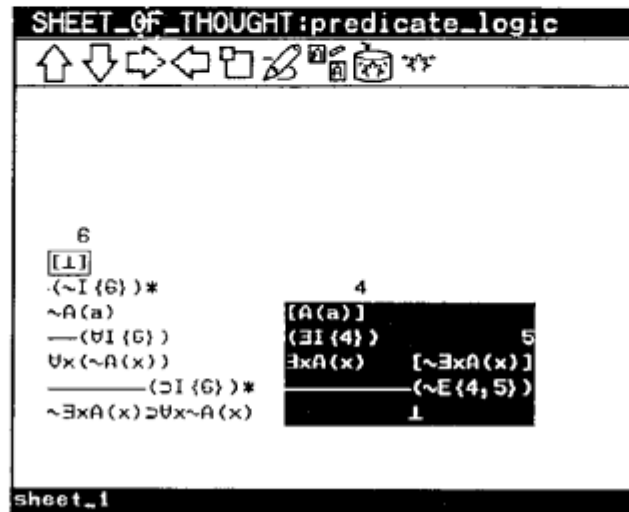


Figure 82: A root and a leaf are selected for connection.

Making double left clicks under the pencil mode induces the result as shown in Figure 83.

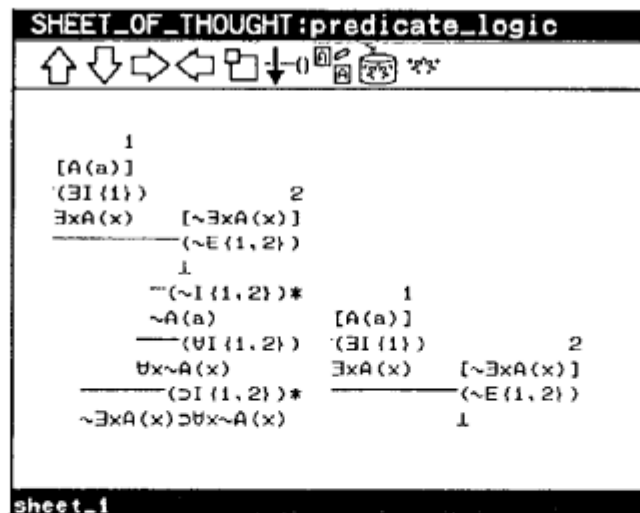


Figure 83: Connection has done in the copy mode.

The tree selected by right click remains after connecting if we are in the copy mode and it will be erased if we are in the move mode.

In Figure 83 we see two “*”’s in the proof. These “*”’s indicate that the applications of the rules have not been completed (i.e. not yet discharged). Therefore we need to discharge these assumptions for completing the proof. In order to discharge these assumption, first we need to specify that they are the assumptions to be discharged by making a middle click on each of the assumptions. Now we have Figure 84.

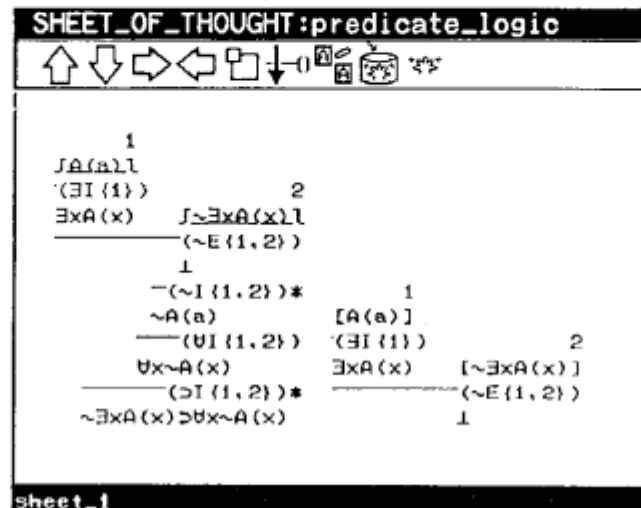


Figure 84: Assumptions going to be discharged are selected.

Let us make a double left clicks for discharging the specified assumptions.

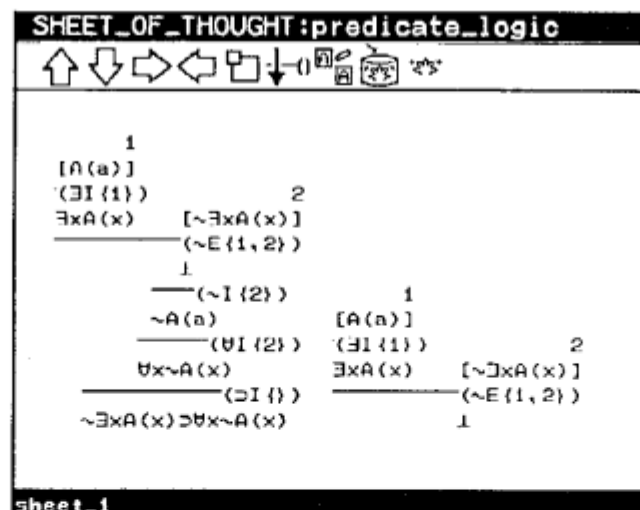


Figure 85: Discharging has done and the proof completes.

Here, in Figure 85 the stars are erased. Since the conclusion formula does not depend on any assumptions we know that the proof of the conclusion has been completed and the conclusion is a theorem.

9.5 Other Facilities of Sheet of Thought

We see some other useful facilities of sheet of thought that have not been explained. They are the facilities of the structure displaying, changing an assumption into an axiom or a theorem, and saving proof fragment as theorems or derived rules.

9.5.1 Displaying the Structure of an Expression

If we find something wrong, such as an application does not work that should be good as far as we see, then one possible reason can be that the expression we are dealing with is parsed differently from that what we are expecting. So providing the facility of displaying the structure of an expression in a proof is helpful for us to find out what is wrong. For displaying the structure, first we select a root of a proof tree by giving a left click two times while the mouse cursor is on the root expression (Figure 86).

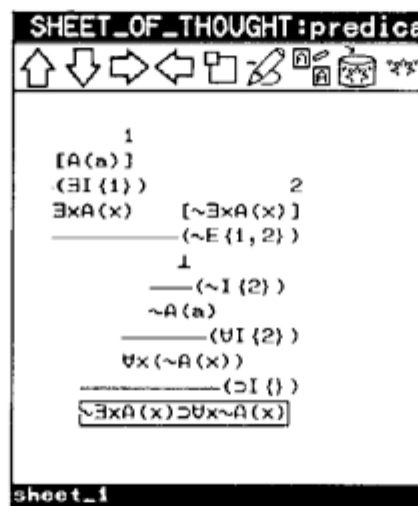


Figure 86: The root formula is selected for structure displaying.

Then make double left clicks or press the return key while we are in the pencil mode so that we can get the command menu as in Figure 87.

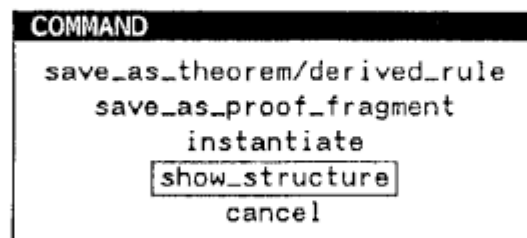


Figure 87: The “show_structure” is going to be selected in the command menu.

Select the “show_structure” item. We will get the structure displaying window as in Figure 88.

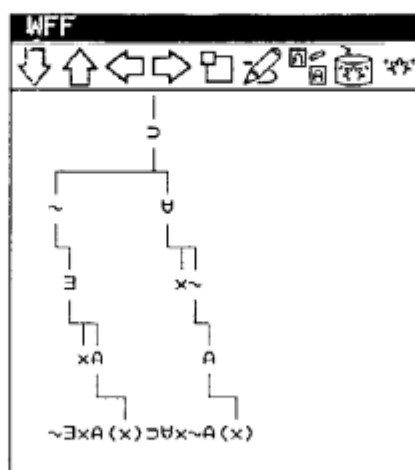


Figure 88: Structure displaying

9.5.2 is_axiom/theorem

It sometimes happens that we have an logical expression which has been put on a sheet of thought as an assumption but we begin to wonder and want to check if it might be an axiom or a theorem. EUODHILOS has a facility to help users to check if an assumption if either an axiom or a theorem. We are supposed to use the function “is_axiom/theorem” in such situation.

What we do for invoking this function is first we select the assumption by giving a left click while the mouse cursor is on the assumption. Figure 89 shows that the assumption is surrounded by a box marker.

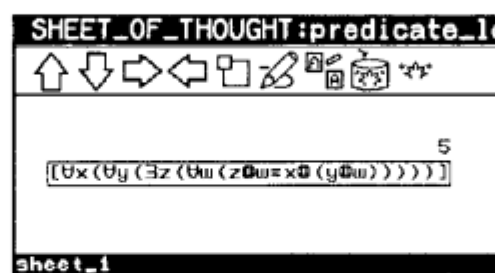


Figure 89: An assumption is selected for checking if it is an axiom or not.

Next let us make a double left click or press the return key while we are in the pencil mode so that we get the command menu for this assumption. Figure 90 shows the command menu for a stand-alone assumption.

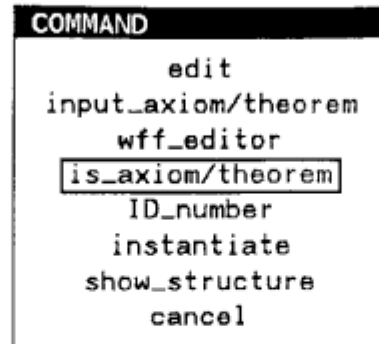


Figure 90: Selecting “is_axiom/theorem” item in the menu.

Next let us select the “is_axiom/theorem” among the items in the menu. If the assumption actually is either an axiom or a theorem the brackets and the assumption ID would disappear as is shown in Figure 91.

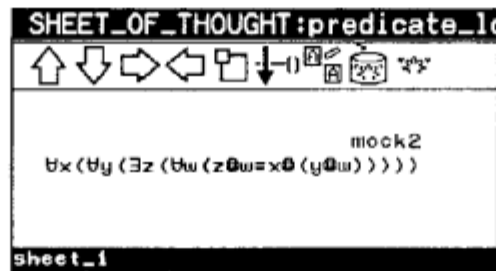


Figure 91: The assumption has changed to axiom.

In this example, the axiom (or theorem) has a name. The name is displayed at the upper right corner of the expression. If the axiom (or the theorem) has no name attached to it, only the axiom (respectively the theorem) body is displayed.

9.5.3 Saving as Theorems or Derived Rules

It is possible to see a proof fragment as a package that can be used in a big proof. So it would be quite useful to save proof fragments and use them as if they are axioms or inference rules. EUODHILOS provides the facility to save and use a proof fragment so that it can be used as a part of other proof. How can we discriminate theorems and derived rules, then? If the root formula of the proof fragment depends on one or more assumptions then it can be used as a derived rule, otherwise, i.e. it depends on no assumptions, as a theorem.

In order to save a proof fragment and let it be used as a theorem or a derived rule, first we need to give a left click at the root formula of the fragment and have a box marker surrounding the whole proof fragment. Next we make a double left clicks to get the command menu as is shown in Figure 92.

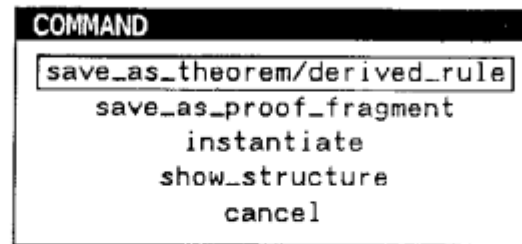


Figure 92: Command menu for saving as theorem or derived rule.

Select the “`save_as_theorem/derived_rule`” item. Then we get a prompting window for the name of the theorem or the derived rule. We do not have to give a name for a theorem. If we just type the return key then the theorem has no name. The theorem body itself will be displayed in a list of theorems when needed. Once defined, the theorem can be used just like the same way as an axiom.

Acknowledgement

The authors would like to express out thanks to Ms. Kyoko Ohashi and Ms. Kaoru Yokota for their contributions to the research and development of EUODHILOS. The most part of this document was written while the first author was staying at the Center for Information Science Research of Australian National Univerisity. He would like to thank Professor M. A. McRobbie for giving him the comfortable environment for writing. The development of EUODHILOS was a part of a major research and development of the Fifth Generation Computer System project conducted under a program set up by the MITI.

References

- [Anderson 75] A.R. Anderson and N.D. Belnap, Jr.: Entailment-The Logic of Relevance and Necessity Vol. I, Princeton University Press, 1975.
- [Hoare 69] C.A.R. Hoare: An Axiomatic Basis for Computer Programming, CACM, Vol. 12, No. 10, 1969.
- [Langer 25] S.K. Langer: A Set of Postulates for the Logical Structure of Music, *Monist* **39**, pp. 561-570, 1925.
- [MacLane 71] S. MacLane: Categories for the Working Mathematician, Springer-Verlag, 1971.
- [Matsumoto 83] Y. Matsumoto et al.: BUP: A Bottom-Up Parser Embedded in Prolog, *New Generation Computing*, Vol 1, pp. 145-158, 1983.
- [Minami 88] T. Minami, H. Sawamura, K. Satoh and K. Tsuchiya: EUODHILOS: A General-Purpose Reasoning Assistant System - Concepts and Implementation, Lecture Notes in Artificial Intelligence **383** (Eds. K. Furukawa, H. Tanaka, and T. Fujisaki), Springer-Verlag, 1989.
- [Minami 92] T. Minami, K. Ohashi, H. Sawamura, and T. Ohtani: General-Purpose Reasoning Assistant System EUODHILOS Visual Manual, *Fujitsu IIAS Research Report*, No. 92-18J, 1992 (in Japanese).
- [Ohashi 90] K. Ohashi, K. Yokota, T. Minami, H. Sawamura, and T. Ohtani: An Automatic Generation of a Parser and an Unparser in the Definite Clause Grammar, *Transactions of Information Processing Society of Japan*, Vol. 31, No. 11, pp. 1616-1626, 1990 (in Japanese).
- [Ohtani 93] T. Ohtani, H. Sawamur and T. Minami: Implementing Constructive Type Theory on EUODHILOS, *ISIS Research Report*, ISIS-RR-93-14E, Fujitsu Laboratories Limited, 1993.
- [Pereira 80] F.C.N. Pereira and D.H.D. Warren: Definite Clause Grammars for Language Analysis-A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artificial Intelligence*, Vol. 13, pp. 231-278, 1980.

- [Prawitz 65] D. Prawitz: Natural Deduction, Almqvist & Wiksell, 1965.
- [Sawamura 91a] H. Sawamura, T. Minami, K. Yokota, and K. Ohashi: Potential of General-Purpose Reasoning Assistant System EUODHILOS, *Software Science and Engineering: Selected Papers from the Kyoto Symposia*, World Scientific Pub. Co., 1991.
- [Sawamura 91b] H. Sawamura, T. Minami, T. Ohtani, K. Yokota, and K. Ohashi: A Collection of Logical Systems and Proofs Implemented in EUODHILOS I, *Fujitsu IIAS Research Report* IAS-RR-91-13E, 1991.
- [Sawamura 92a] H. Sawamura, T. Minami, and K. Ohashi: Proof Methods Based on Sheet of Thought in EUODHILOS, *Fujitsu IIAS Research Report*, No. 92-6E, 1992.
- [Sawamura 92b] H. Sawamura, T. Minami, and K. Ohashi: System Description of EUODHILOS: A General Reasoning System for a Variety of Logics, *Lecture Notes in Artificial Intelligence*, Vol. 624, 1992.
- [Sawamura 93] H. Sawamura, T. Minami, and T. Ohtani: Application and Evaluation of General-Purpose Reasoning Assistant System EUODHILOS, *Trans. IPS Japan*, Vol. 34 No. 5, 1993 (in Japanese).

A Appendix

Here are some samples of logics stored in EUODHILOS. See [Sawamura 91b] for more samples.

A.1 First-Order Logic

The first-order logic formulated with natural deduction[Prawitz 65] is the first example. The sample problem is a proof of the unsolvability of halting problem. Ordinary logics are well defined as is shown in the following figure.

halting_problem	INFORMATION	INFORMATION	SYNTAX : halting_problem
INFORMATION SOFT_KEYBOARD SYNTAX INFERENCE_RULE REWRITING_RULE AXIOM PROVER DERIVED_RULE AXIOM : halting_problem save wff_editor reshape exit	** new ** ** exit ** AEP AEQ AI VE VIP VIQ DE	save make test structure print reshape exit formula --> formula, equivalence, formula1; formula --> formula1; formula1 --> formula1, imply, formula2; formula1 --> formula2; formula2 --> formula2, or, formula3; formula2 --> formula3;	
$\exists v (C(v) \wedge \forall y ((C(y) \wedge H(y, y) \supset H(v, y) \wedge O(v, g)) \wedge (C(y) \wedge \neg H(y, y) \supset H(v, y) \wedge O(v, b))) \supset \exists u (\neg H(u, y) \wedge (C(y) \wedge \neg H(y, y) \supset H(u, y) \wedge O(u, b))))$ $\forall w (C(w) \wedge \forall y (C(y) \supset \exists z D(w, y, z)) \supset \forall y (\forall z ((C(y) \wedge H(y, z) \supset H(w, y, z) \wedge O(w, g)) \wedge (C(y) \wedge \neg H(y, z) \supset H(w, y, z) \wedge O(w, b))))$ $\exists x (A(x) \wedge \forall y (C(y) \supset \exists z D(x, y, z))) \supset \exists w (C(w) \wedge \forall y (C(y) \supset \exists z D(w, y, z)))$ $\exists w (C(w) \wedge \forall y ((C(y) \wedge H(y, y) \supset H(w, y, y) \wedge O(w, g)) \wedge (C(y) \wedge \neg H(y, y) \supset H(w, y, y) \wedge O(w, b))) \wedge \forall y (H(w, y, y) \wedge O(w, g) \wedge (C(y) \wedge \neg H(y, y) \supset H(w, y, y) \wedge O(w, b))))$ \square $H(y, y) \supset H(w, y, y) \wedge O(w, g) \wedge (C(y) \wedge \neg H(y, y) \supset H(w, y, y) \wedge O(w, b))) \supset \exists v (C(v) \wedge \forall y ((C(y) \wedge H(y, y) \supset H(v, y) \wedge O(v, g)) \wedge (C(y) \wedge \neg H(y, y) \supset H(v, y) \wedge O(v, b))))$ $\supset \supset \supset$ $\supset \supset \supset (DE(1))$ $\frac{1}{\neg \exists x (A(x) \wedge \forall y (C(y) \supset \exists z D(x, y, z)))}$			
sheet_1			

INFORMATION

name: ~I

{P}

1

~P

**** Side condition ****

**** Define ****

INFORMATION

name: DE

P \supset Q

Q

**** Side condition ****

**** Define ****

A.2 Category Theory



This is a formulation of elementary category theory⁹ mostly using inference rules. Mathematical reasoning is also well formulated in the EUODIHILOS framework of defining logics.

The diagram illustrates the evolution of a theorem prover through several stages:

- category_theory**: A window showing a list of inference rules (e.g., `coeq-E(1)`, `coeq-E(=)`, `coeq-E(cod)`, etc.) and a list of axioms (e.g., `coeq-E(1)`, `coeq-E(=)`, `coeq-E(cod)`, etc.).
- REWRITING_RULE:cat**: A window showing a specific rule for `F:obj`, with a side condition `** Side condition **` and a definition `** Define **`.
- SHEET_OF_THOUGHT:category_theory**: A window showing a complex proof structure with multiple inference rules and a final goal. The proof structure includes a list of inference rules (e.g., `coeq-E(1)`, `coeq-E(=)`, `coeq-E(cod)`, etc.) and a list of axioms (e.g., `coeq-E(1)`, `coeq-E(=)`, `coeq-E(cod)`, etc.).

⁹For category theory see [MacLane 71]

Hoare logic[Hoare 69] is a kind of programming logic on which we can prove, for example, the partial correctness of a program.
























Hoare_logic INFORMATION SOFT_KEYBOARD SYNTAX INFERENCE_RULE REWRITING_RULE AXIOM PROVER DERIVED_RULE THEOREM PROOF ** EXIT **	AXIOM : Hoare_ SYNTAX : Hoare_logic save wff_editor save make test structure print reshape exit z=y!Ay=xZ=x! (x=y!Ay=xZ=x!)A(xZ=x!) PAQDP P(T/X){X:=1}P(X) ged_prop1 (X>Y)⊃(ged(X,Y)=g ged_prop2 ged(X,Y)=ged(V,X) ged_prop3 ged(X,X)=X □	INFERENCE_RULE:Hoare  name:repetition F&G(A)F F{whileGdoAod}FA~G ** Side condition ** ** Define **
THEOREM:Hoare		REWRITING_RULE:Hoare
SHEET_OF_THOUGHT:Hoare_logic		 name:arith z=y! z*(y+1)=(y+1)! z=y!A~(y=x)z=y! (arith()) z=y!A~y=xZ*(y+1)=(y+1)! z*(y+1)=(y+1)! {y:=y+1} z=y! (conseq()) z=y!A~y=x{y:=y+1} z=y+y! z=y!A~y=x{y:=y+1; z:=z*y} z z=y! {while~y=xdo y:=y+1; z:=z*yod} z=y! {while~y=xdo y:=y+1; z:=z*yod} true {z:=1; y:=0; (while~y=xdo y:=y+1; z:=z*yod)} z=y!Ay=x true {z:=1; y:=0; (while~y=xdo y:=y+1; z:=z*yod)} z=y!Ay=x
true⊃1=0! 1=0! {z:=1} z=0! (conseq1()) true {z:=1} z=0! z=0! {y:=0} z=y! (comp()) true {z:=1; y:=0} z=y!		
sheet_1		

A.4 Relevant Logic







Relevant logic[Anderson 75] is the logic which allows only the “relevant” proofs. The treatment of dependency relations is different from the logics shown above. In our formulation, the dependency relations are represented in the “tag” part of the formula.

relevant_logic_Rir	INFERENC	INFERENC	INFERENC	INFERENC
INFORMATION SOFT_KEYBOARD SYNTAX INFERENC_RULE REWRITING_RULE AXIOM PROVER DERIVED_RULE THEOREM PROOF ** EXIT **	 name: ->E $\frac{T1 \Rightarrow P \Rightarrow Q \quad T2 \Rightarrow P}{T1 * T2 \Rightarrow Q}$ ** Side conditor ** Define **	 name: Tag_rule_B $\frac{T1 * (T2 * T3) \Rightarrow P}{T1 * T2 * T3 \Rightarrow P}$ ** Side conditor ** Define **	 name: Tag_rule_O $\frac{\langle T1 * T2 \rangle * T3 \Rightarrow P}{\langle T1 * T3 \rangle * T2 \Rightarrow P}$ **	 name: Tag_rule_W $\frac{T1 * T2 * T2 \Rightarrow P}{T1 * T2 \Rightarrow P}$ **
SYNTAX : relevant_logic_Rimp save make test structure print reshape exit tag_formula --> tag, "=>", formula "=>", formula; formula --> formula, ">", formula ; "(", formula, ")" "p" "q" "r" meta_formula; meta_formula --> "p" "q" "r"; tag --> tag, "*", tag ; "(", tag, ")" "a" "b" "c" "d" "K" "I" "O" "B" "CB" "W"; meta_tag; meta_tag --> "T1" "T2" "T3". operator (">", ">"); "=>".				
SHEET_OF_THOUGHT:relevant_log $\frac{[a \Rightarrow P \Rightarrow (P \Rightarrow Q)] \quad [b \Rightarrow P]}{a * b \Rightarrow P \Rightarrow Q} \quad \frac{[b \Rightarrow P]}{b \Rightarrow P} \quad \frac{[a * b \Rightarrow P \Rightarrow Q]}{a * b \Rightarrow P \Rightarrow Q}$ $\frac{[a * b \Rightarrow P \Rightarrow Q] \quad \langle Tag_rule_W \{4, 5\} \rangle}{a * b \Rightarrow Q} \quad \frac{a * b \Rightarrow Q}{a \Rightarrow P \Rightarrow Q} \quad \frac{a \Rightarrow P \Rightarrow Q}{(P \Rightarrow (P \Rightarrow Q)) \Rightarrow (P \Rightarrow Q)}$				
SHEET_OF_THOUGHT:relevant_logic $\frac{[a \Rightarrow P \Rightarrow (Q \Rightarrow R)] \quad [c \Rightarrow P]}{a * c \Rightarrow Q \Rightarrow R} \quad \frac{[c \Rightarrow P]}{c \Rightarrow P} \quad \frac{a * c \Rightarrow Q \Rightarrow R}{(a * c) * b \Rightarrow R} \quad \frac{(a * c) * b \Rightarrow R}{\langle Tag_rule_O \{1, 2, 3\} \rangle} \quad \frac{\langle Tag_rule_O \{1, 2, 3\} \rangle}{(a * b) * c \Rightarrow R} \quad \frac{(a * b) * c \Rightarrow R}{a * b \Rightarrow P \Rightarrow R} \quad \frac{a * b \Rightarrow P \Rightarrow R}{a \Rightarrow Q \Rightarrow (P \Rightarrow R)} \quad \frac{a \Rightarrow Q \Rightarrow (P \Rightarrow R)}{(P \Rightarrow (Q \Rightarrow R)) \Rightarrow (Q \Rightarrow (P \Rightarrow R))}$				
PROOF:relevant_1 ** new ** ** exit ** ** work ** Contraction Permutation Prefixing Self_implication all				

Index

-  inference rule, 12
-  sheet of thought, 18, 42
-  inference rule, 12
-  sheet of thought, 17
-  sheet of thought, 17
-  sheet of thought, 17
-  inference rule, 12
-  sheet of thought, 17
-  inference rule, 12
-  sheet of thought, 17
-  inference rule, 12
-  sheet of thought, 17
-  inference rule, 11
-  sheet of thought, 17
-  sheet of thought, 18, 23
-  sheet of thought, 17
-  inference rule, 11
-  sheet of thought, 17
-  inference rule, 11
-  sheet of thought, 17
-  inference rule, 12, 14
-  sheet of thought, 42
-  sheet of thought, 17
- " , 34
- ** Define ****, 37
- ** EXIT ****
- functions menu, 9
- ** calculator ****, 6
- ** exit ****
- EUODHILOS, 7, 23
- inference rule, 11, 35
- logic, 23
- logic menu, 7
- proof group, 40
- proof manipulation, 16
- ** font_editor ****, 6, 28
- ** new ****
- copying proof, 41
- inference rule, 11, 35
- logic menu, 6
- proof group, 40
- proof group name for saving, 42
- proof manipulation, 16, 18
- ** work ****, 16
- proof group, 40
- proof group name for saving, 42
- *, 57
- ., 34
- >, 32
- , 33
- ., 34
- :left, 34
- :right, 34
- ;, 32, 34
- <, 30
- >, 30
- AXIOM, 9, 39
- DERIVED_RULE, 9
- ID_number, 44
- INFERENCE_RULE, 9, 11
- INFORMATION, 8
- NOT SAVE
- proof, 43
- OK
- logic copying, 27
- PROOF, 40
- PROVER, 9
- REWRITING_RULE, 9
- SAVE ALL
- proof, 43

- SOFT_KEYBOARD, 8, 28
- SYNTAX, 9, 30
- THEOREM, 9
- [,], 43
- activate, 7, 26
- all_occurrence, 38
- assign, 29
- bind_op, 34
- call, 33
- cancel
 - assumption, 44
 - command for proof, 52
 - derivation, 46
 - side condition, 38
- copy_to_sheet, 41
- copy
 - logic, 8, 26
 - rule, 36
- delete_all
 - side condition, 38
- delete
 - logic, 8, 26
 - proof, 41
 - rule, 36
 - side condition, 38
- discharge, 46
- edit, 44
 - proof group, 41
 - rule, 36
- exit
 - axiom, 39
 - side condition, 38
 - software keyboard, 29, 30
 - syntax, 10
- failure, 31
- information, 7, 26
- input_assumption, 18
- input_axiom/theorem, 44
- input, 38
- instantiate, 44, 51
- is_axiom/theorem, 44, 59
- make, 10, 31
- menu, 38
- meta_, 10, 50
- operator, 33
- predicate, 34
- print, 10
- release, 30
- rename
 - logic, 7, 26
 - rule, 36
- reshape
 - axiom, 39
 - syntax, 10
- save_as_proof_fragment, 42, 51
- save_as_theorem/derived_rule, 22, 51, 61
- saved
 - inference rule, 14
- save
 - axiom, 39
 - software keyboard, 30
 - syntax, 10
- search_rule(input), 46
- show_structure, 44, 59
 - proof, 52
- structure, 31
 - syntax, 10
- success, 31
- test, 10, 31
- v, 30
- wff_editor, 44
 - axiom, 39
- call prover--, 47
- ~, 30
- assumption, 13, 18, 43
- axiom, 38
 - closing, 39
 - creation, 39
 - modification, 39
- backward derivation, 3
- character range specification, 33
- conclusion, 13
- constructor declaration, 9, 25
- DCG, 3, 25
- DCG part, 9, 32
- delimiter, 32
- dependency, 20

- derivation, 19, 44
 - backward, 49
 - forward, 45
- derivation system, 3
- derived rule, 60
- discharging an assumption, 57
- discharging assumption, 48
- ESP program, 33
- EUODHILOS, 3, 5
 - closing, 7
 - installation, 5
 - opening, 5, 6
 - process of using, 24
- forward derivation, 3
- ID number, 43
- inference rule, 35
 - closing, 11, 35
 - copying, 36
 - creating, 35
 - definition, 11
 - deleting, 36
 - editing, 36
 - manipulation, 36
 - menu, 11
 - opening, 11, 35
 - rule body
 - creation, 13
 - rule name
 - creation, 12
 - side condition, 37
- input
 - assumption, 18
 - expression, 13
 - logic name, 7
 - rule name, 12
 - target logic for copying, 27
 - theorem name, 22
- input assumption menu, 18
- instantiation, 51
- language system, 3
- logic
 - copying, 26
 - creation, 7, 25, 26
 - functions menu, 8
 - manipulation, 7, 26
- logic definition, 24
- logic menu, 6
- logics experimented, 4
- menu
 - application, 19
 - closing sheet of thought, 43
 - command for proof in group, 52
 - commands for assumption, 44
 - commands for proof, 41
 - commands for proof group, 41
 - functions for logic, 8
 - functions on rules, 36
 - inference rule, 11, 35
 - items for copying, 27
 - logic manipulation, 7, 26
 - logic menu, 6
 - proof, 42
 - proof group name, 40
 - rule for derivation, 46
 - sheet of thought
 - saving, 22
 - system menu, 5
- metavariable, 50
- mode
 - changing, 54
 - copy() / move() , 12
 - forward() / backward() , 45
 - pencil() / eraser() , 12
- mouse clicking
 - double right, 6
 - inference rule
 - double left, 13, 15
 - left, 13-15
 - middle, 14
 - proof
 - double left, 42, 46, 48
 - left, 51, 53, 58
 - middle, 47, 48, 54
 - right, 45
 - sheet of thought
 - double left, 19

- left, 18
 - right, 19, 20
- or-notation, 32
- parser, 10
- premise, 13
- proof
 - is_axiom/theorem, 59
 - commands for derivation, 46
 - commands for proof in group, 52
 - connecting, 56
 - construction, 24, 40
 - copying, 20, 54
 - creation, 42
 - deleting, 53
 - derived rule creation, 60
 - discharging application, 20
 - editing, 53
 - formula item, 41
 - fragment, 3
 - group, 41
 - manipulation menu, 16
 - proof group, 41
 - theorem creation, 60
- prover, 49
- rewriting rule, 35, 38
 - opening, 35
- rule, 35
- rule application, 19, 44
- sheet of thought, 3, 16, 40
 - closing, 42
 - confirmation, 43
 - opening, 16, 40
 - proof
 - moving, 55
- software keyboard, 25
- special symbol, 25, 27
 - editing, 28
 - software keyboard, 29
- structure displaying
 - proof, 58
 - syntax definition, 31
- syntax
 - constructor, 33
 - DCG, 32
 - declaration, 33
 - definition, 9
 - description, 3
 - editing, 30
 - making parser and unparser, 30
 - testing, 31
 - window, 9
 - window items, 10
- system
 - derivation, 3
 - language, 3
- system menu, 6
- theorem
 - creation, 22
 - using, 9
- unparser, 10
- window
 - axiom, 39
 - font editor, 28
 - inference rule, 11, 36
 - logic menu, 6
 - sheet of thought, 17
 - side condition, 37
 - software keyboard, 29
 - structure displaying, 32
 - syntax definition, 9, 30
 - syntax testing, 31