ND: A RULE-BASED IMPLEMENTATION OF NATURAL DEDUCTION

DESIGN OF THE THEOREM-PROVER AND TUTORING SYSTEM

François Dongier
School of Computer Science
McGill University

March 1988

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Masters of Science

ISBN 0-315-46033-4

ABSTRACT

Natural deduction provides an elegant technique for the construction and representation of proofs. This thesis describes ND, a Lisp program implementing a Fitch-style natural deduction theorem-prover for sentential logic. A set of production rules is used to decide, at any stage in the construction of the proof, what rule of inference should be applied next. The extension of ND into STEL, a tutoring system for proofs in natural deduction, is also discussed, as well as some advantages of using natural deduction theorem-proving in artificial intelligence applications.

RÉSUMÉ

La déduction naturelle permet une représentation claire et intuitive de la preuve de la validité d'un raisonnement. Cette thèse décrit l'implémentation en Lisp du système ND, un démonstrateur de théorèmes en logique des propositions basé sur la méthode de déduction naturelle telle que formulée par Frederic B. Fitch. Un ensemble de règles de production est utilisé pour décider, à toute étape de la preuve, quelle règle d'inférence utiliser. Après la description de ce système de règles, on trouvera une discussion de l'extension du démonstrateur de théorèmes au système tutoriel STEL, ainsi que de la pertinence de la déduction naturelle à certains problèmes liés au raisonnement automatique en intelligence artificielle.

ACKNOWLEDGEMENTS

Warm thanks are due to the supervisor of this thesis, Professor G. Hahn. I feel strongly indebted, for their help, comments and encouragement at various stages of the project, to a number of students and Faculty members of the School of Computer Science at McGill, in particular to Jean-François Cloutier, John Kirkpatrick, Professor T. Merrett, Professor C. Paige and Professor S. Whitesides.

The extension of the theorem-prover into a tutoring system was done in collaboration with Michel Paquette and Gaston Gour, who teach philosophy - and in particular, logic - at Collège de Maisonneuve in Montréal. Sincere thanks to both of them for the long discussions that, I hope, are reflected in the final product. I also want to thank Ulrich Aylwin, directeur des services pédagogiques at Collège de Maisonneuve and Claude Seguin, responsable de projets at Direction générale de l'Enseignement Collégial, Ministère de la Science et de l'Enseignement Supérieur, Gouvernement du Québec, who provided financial support to the tutorial project.

TABLE OF CONTENTS

ABSTRACT i

RÉSUMÉ ii

ACKNOWLEDGEMENTS iii

INTRODUCTION 1

I. ELEVEN INFERENCE RULES 4

II. NATURAL DEDUCTION VS OTHER PROOF-TECHNIQUES 8

1. Alternative proof-techniques 8

2. Natural deduction 12

III. OBJECTIVES OF THE THESIS AND JUSTIFICATION OF THE CHOSEN APPROACH (PRODUCTION RULES) AND DOMAIN (SENTENTIAL LOGIC) 16

1. Objectives of the theorem-prover 16

2. Justification of a rule-based system 20

3. Limitation to sentential logic 22

IV. DESIGN OF ND (THE THEOREM-PROVER) 24

1. What is a "good" proof? 24

2. Use of production rules to capture knowledge about elegance of proofs 26

3. Rules of inference vs heuristic rules 27

4. Basic strategy 30

5. Forward and backward reasoning must be implemented ... 32

6. A forward-chaining inference engine generates both forward and backward reasoning 32

7. The main data-structures of ND 34

V. EXTENSION OF ND INTO STEL (A TUTOR) 37

1. Motivation and objectives of the tutor 37

2. STEL in the perspective of current ICAI 38

3. STEL, a modest tutor 43

4. New data-structures
Student-model and bank of exercises 45

5. Explanation system
Enrichment of the rule structure 45

6. Remarks on the implementation of the tutor 47

7. Experimentation of the program
and possible future extensions 49

VI. RELEVANCE OF NATURAL DEDUCTION
TO ARTIFICIAL INTELLIGENCE 50

1. Natural deduction allows an intuitive explanation
of the reasonings embodied in the proofs 50

2. Possibility to give a sketch of the proof
or a partial explanation in case of failure 51

3. A natural way of implementing richer reasonings
than sentential logic reasonings 52

4. Conclusion 60

APPENDIX 1
Rules used in the first version of the theorem-prover ... 61
APPENDIX 2
Listing of a first implementation of the theorem-prover . 63
APPENDIX 3
Rules used in the IQ-Lisp version
of the theorem-prover 83
The vocabulary of rules:
primitive predicates and functions 83
Rules used in forward reasoning 85
Rules used in backward (goal-directed) reasoning 88
Rules used when the goal is to prove a contradiction 95
APPENDIX 4
Screen dumps taken from the Stel tutorial 97

BIBLIOGRAPHY 110

INTRODUCTION

The natural deduction technique is used for constructing proofs in formal systems and for representing these proofs in a clear and intuitive manner. It is typically used to present the derivation of a theorem or, in a system of logic, to show that a sentence is logically true; it can also be applied to establish that an argument is valid or that a set of sentences is inconsistent.

Historically, the method originated in the work of Gentzen [10] and Jaskowski [11] who devised a complete set of rules for natural deduction derivations. The standard form for expressing the rules and formatting the proofs was formulated by Fitch [9].

Any natural deduction system is based on a set of "natural" rules of inference that can be applied in reasonings. These rules specify what inferences are permitted in the domain. It is in fact the "naturalness" of natural deduction that constitutes its main advantage when compared to other, more syntax-oriented, proof-techniques: a natural deduction proof is easier to understand because it is based on an intuitive set of rules of inference; each step of the proof can be seen to follow from one or more earlier steps by the application of a rule of inference.

Natural deduction systems have been constructed for various logics, e.g., sentential logic (the logic of truth-functional connectives), predicate logic (the logic of quantification), and richer sorts of logics (in which, for instance, relevance connectives or modal operators are present).

The present thesis reports on ND, a Lisp implementation of a Fitch-style natural deduction theorem-prover for sentential logic. The rule-based implementation of this theorem-prover allowed the extension of the program into the STEL tutoring system, aimed at teaching students the rudiments of logic and the basics of the art of constructing proofs in natural deduction.

Section I lists the inference rules that define the domain of the proofs that the theorem-prover should be able to construct. Section II compares natural deduction with two other well-known proof-techniques for sentential logic (truth-tables and truth-trees).

Section III presents the objectives of the thesis and discusses the major design choices made in this implementation (limitation to sentential logic, use of production-rules). Section IV focuses on the strategies used by the theorem-prover in the construction of the proofs. Section V describes the design principles adopted in the implementation of the tutor. Section VI discusses the relevance of using natural deduction theorem-proving in AI applications.

The rules used in the first implementation of the theorem-prover (written in Franz Lisp on a Vax) are listed in appendix 1. Appendix 2 contains a partial listing of this program. For the implementation of the tutor, the program was transported to an IBM-PC (first in Mu-Lisp, then in IQ-Lisp). The extended vocabulary that was made available to the rules in this second version of the theorem-prover is described in appendix 3, where a listing of the new set of rules may also be found. Appendix 4 contains a dump of a few screens taken from the tutorial, showing typical interaction between the program and the student.

I. ELEVEN INFERENCE RULES

Formulas of sentential logic are either atomic (e.g., P, Q, ...) or compound: the rules of sentence formation (the grammar of the language) specify how well-formed compound formulas may be generated by grouping simpler ones with connectives. There are 5 connectives : &, V, ¬, =>, <=>, and the rules of sentence formation say that, if X is a well-formed formula, so is (¬ X); if X and Y are well-formed formulas, so are (X & Y), (X V Y), (X => Y) and (X <=> Y). The five connectives are interpreted respectively as AND, OR, NOT, IF...THEN, and IF AND ONLY IF.

A *proof* in natural deduction is a sequence of formulas, each of which "has a justification". A formula can be justified either as a *premiss*, as a *hypothesis*, or as the result of the *application of a rule of inference* on one or more earlier formulas in the proof.

Intuitively, rules of inference may be thought of as truth-preserving tools that guarantee safe deductions. Mathematically speaking, an inference rule is nothing but a mapping that, when applied to a set of theorems, yields a new set of theorems[1]. As such, it is purely syntactic and, therefore, to-

---

[1] In "monotonic logics", the application of inference rules may only *increase* the set of theorems.

tally independent of the interpretation that may be given to the symbols occurring in the formulas on which it is applied.[2]

Two rules of inference are associated with each connective, one for the "elimination" of the connective (e.g., "&Elim") and one for its "introduction" (e.g., "=>Intro"). This gives the following ten rules (m, n, and o stand for line-numbers; vertical dots represent an arbitrary number of intermediary steps that are not relevant to the rule being defined).

INTRODUCTION RULES		ELIMINATION RULES	
m | X ⋮ n | Y ⋮ (X & Y)	&Intro,m,n	m | (X & Y) ⋮ X	&Elim,m
		m | (X & Y) ⋮ Y	&Elim,m
m | X ⋮ (X V Y)	VIntro,m	m | (X V Y) ⋮ n | (¬ X) ⋮ Y	VElim,m,n

[2] However, as we will see in the next section, it is a crucial characteristic of *elegant* natural deduction proofs that they cannot be constructed without reference to the ordinary meaning of the connectives. It is the parallel that such proofs have with ordinary intuitive reasoning that makes them more interesting than other types of proofs.

```
m | Y
  .
  .
  (X V Y)        VIntro,m
```

```
m | (X V Y)
  .
  .
n | (¬ Y)
  .
  .
  X              VElim,m,n
```

```
  .
  .
m |  X
  |  .
  |  .
n |  Y
  |  .
  |  .
o |  (¬ Y)
  (¬ X)          ¬Intro,m,n,o
```

```
  .
  .
m |  (¬ X)
  |  .
  |  .
n |  Y
  |  .
  |  .
o |  (¬ Y)
  X              ¬Elim,m,n,o
```

```
  .
  .
m |  X
  |  .
  |  .
n |  Y
  (X => Y)       =>Intro,m-n
```

```
m | X
  .
  .
n | (X => Y)
  .
  .
  Y              =>Elim,m,n
```

```
m | (X => Y)
  .
  .
n | (Y => X)
  .
  .
  (X <=> Y)      <=>Intro,m,n
```

```
m | (X <=> Y)
  .
  .
n | X
  |
  Y              <=>Elim,m,n (*)
```

---

* See page 13 for a description and use of the notation used here

To be complete, the system needs an additional (eleventh) derivation rule, the rule of *reiteration*, that merely allows the derivation of a sentence from itself[3]:

```
m | .
  | .
  | P
  | .
  | .
  | P        Reit m
```

This gives a total of eleven rules, each of which can clearly be seen (and proved) to be truth-preserving. It is one among several alternative[4] sets of rules of inference for Fitch-style natural deduction proofs in sentential logic: other formulations of the rules also yield sound and complete systems (systems that can prove *all* valid, *and only* valid formulas of sentential logic).

---

[3] This rule may seem trivial. Its presence in Fitch's system is justified by what it excludes: some formulas are reiterable while some are not; restrictions on reiteration forbid, in particular, the reiteration of a formula derived under a hypothesis that has been discarded.

[4] Such alternative systems of natural deduction for sentential logic differ in particular in the expression of the rules for negation and disjunction elimination.

II. NATURAL DEDUCTION VS OTHER PROOF-TECHNIQUES

1. Alternative proof-techniques

Two other reputed theorem-proving techniques, namely truth-tables and truth-trees - it is this second technique, incidentally, that lies at the foundation of the resolution method - have a significant advantage over natural deduction: they are mechanical decision procedures that will always establish, after a finite number of steps, whether an argument is logically valid, whether a set of sentences is logically consistent, or whether a sentence is logically true. No imagination or foresight is required for either of these methods to apply its test. The natural deduction technique does not provide such a guarantee that a proof will be found, if it exists; furthermore, it does not specify in a unique way how the proof is to be constructed. Why, then, do people bother using natural deduction rather than truth-tables or trees? What advantages may compensate for the fact that natural deduction proofs are not mechanical? To answer these questions, let us first look at an example to see how each method solves the validity question.

Consider the following argument, taken from a typical real-life piece of reasoning:

> If the kids didn't clean up their mess, then if Mom saw it, she was not happy.
>
> Surely Mom saw it if the kids didn't clean up their mess.
>
> But she was happy.
>
> So they must have cleaned up their mess.

Using the language of sentential (also called propositional) logic, the argument translates[1] into:

```
¬K => (S => ¬H)      ; if not K, then if S then not H
¬K => S              ; if not K, then S
H                    ; H
--------------       ; therefore
K                    ; K
```

The truth-table[2] gives us a straight-forward way of showing that this argument is valid: after building - very mechanically - the whole table, we observe that there is no

---

[1] In what follows, some liberty is being taken with syntactic correctness (e.g., parentheses) so as to make the formulas easier to read.

[2] Details of the syntax of this language and the different proof-techniques discussed in this section can be found in introduction to logic manuals such as Leblanc & Wisdom [13], or Bergmann, Moor & Nelson [3].

truth-value assignment to the atomic variables K, S, H that makes the premisses of the argument true and the conclusion false at the same time.

K	S	H	¬K	¬H	S=>¬H	¬K=>(S=>¬H)	¬K=>S
T	T	T	F	F	F	T	T
T	T	F	F	T	T	T	T
T	F	T	F	F	T	T	T
T	F	F	F	T	T	T	T
F	T	T	T	F	F	F	T
F	T	F	T	T	T	T	T
F	F	T	T	F	T	T	F
F	F	F	T	T	T	T	F
↑		↑				↑	↑
Concl.		Prem3				Prem1	Prem2

There are 2^n rows in a truth-table, where n is the number of atomic variables occurring in the argument. In the case of the argument that we are now considering, n = 3, so that the truth-table contains only 8 rows - an acceptable size. However, as the size of the table jumps exponentially with n, the standard truth-table technique is a rather uninteresting tool for evaluating real-life arguments.

There exists a short-cut truth-table method, based on the following principle: instead of constructing the whole truth-table, one tries to construct a row which shows the argument invalid, i.e a row in which the premisses are all true and the conclusion false. If this fails, then the argument is valid. In our example, we try to find a truth-value assignment to

K, S, and H that will make the premisses true and the conclusion false:

Starting with the assignments

P1 <=== T
P2 <=== T
P3 <=== T
C <=== F,

we immédiately get the assignments

H <=== T
K <=== F

A simple and mechanical procedure quickly shows that S should be assigned the truth-value false for the first premiss to be true and the truth-value true for the second premiss to be true. This is impossible, hence the argument is valid.

The tree technique applies basically the same strategy as the short-cut truth-table technique: to prove an argument {P1,...,Pn} /:: C valid is to show that the set {P1,...,Pn, ¬C} is inconsistent. A set of sentences is shown to be inconsistent if all branches of the tree representing the set *close* (i.e contain an explicit contradiction).

Here is the tree corresponding to our argument. From the point of view of an intuitive representation of the proof, it does improve over the truth-table (a first step in the right direction).

[Premiss 1] ¬ K => (S => ¬ H)

[Premiss 2] ¬ K => S

[Premiss 3] H

[Negation of conclusion] ¬ K

[Expansion[3] of 1st premiss] K | (S => ¬ H)

[Expansion of 2nd premiss] ¬ S | ¬ H

K | S

(Underlined terminals indicate branches that are "closed", in the sense that they contain an explicit contradiction)

2. Natural deduction

Natural deduction proofs are also called *derivations*: instead of showing that all ways of putting together the premisses and the negation of the conclusion yield inconsistent sets (as is done with the tree method), the ND proof shows how the conclusion can be derived (as it were, constructed) from the premisses via a series of "obvious" steps. Each step consists of a sentence S together with a justification for it.

---

[3] For ¬ K => (S => ¬ H) to be true, either K must be true or (S => ¬ H) must be true.

The justification of S states explicitely

- What rule of inference was used in deriving S (unless of course it is a premiss or hypothesis, in which case it needs no further justification).

- The position in the proof of earlier steps from which S was immediately derived.

Here is the ND version of the messy kids argument:

1	$(\neg K \Rightarrow (S \Rightarrow \neg H))$	Premiss
2	$(\neg K \Rightarrow S)$	Premiss
3	H	Premiss
4	$\underline{\neg K}$	Hypothesis
5	$S \Rightarrow \neg H$	=>Elim 1,4
6	S	=>Elim 2,4
7	$\neg H$	=>Elim 5,6
8	H	Repetition 3
9	K	¬Elim 4,7,8

ND derivations allow the making of hypotheses: you can make a hypothesis anywhere in the proof and see what follows from it. In a Fitch-style ND proof, a vertical line starting at step n indicates that the sentence at n is a hypothesis; the hypothesized sentence is also underlined.

We can see by looking at the proof that

- line 4 is a hypothesis,
- line 5 is justified by the rule "arrow-elimination" (modus ponens) from lines 1 and 4,
- line 8 is a repetition of line 3,
- the conclusion appears as the last line of the proof and is justified by "negation elimination" (a form of reductio ad absurdum).

The position of a sentence with respect to the vertical lines is also very informative, showing precisely under what hypotheses it was derived (here, ¬H was derived under hypotheses 1, 2, 3, 4, while K logically follows from 1, 2, 3 *alone*).

The key advantage a natural deduction proof has to offer is its "naturalness", i.e., its similitude - to a large extent - to our informal, ordinary human way of reasoning. Unlike resolution and truth-tables, natural deduction allows a mixture of top-down and bottom-up reasoning, as well as a mixture of direct and indirect reasoning: just like human reasonings, natural deduction proofs can grow forwards (top-down) or backwards (bottom-up), and use reductio ad absurdum only when no other, more direct way of progressing towards the conclusion is available. A comparison of the resolution and natural deduc-

tion proofs of a relatively very simple argument will illustrate this last point.

Dolphins have lungs and are warm-blooded.

If dolphins have lungs, then they are not fish.

Hence dolphins are warm-blooded and are not fish.

Proof of validity of the argument by the tree method:

(L & W)

(L => ¬ F)

[Negation of conclusion] ¬ (W & ¬ F)

L

W

[Expansion of premiss 2] ¬ L ¬ F

[Expansion of negated conclusion] ¬ W F

Natural deduction proof of the same problem

1	(L & W)	Premiss
2	(L => ¬ F)	Premiss
3	W	&Elim 1
4	L	&Elim 1
5	¬ F	=>Elim 2,4
6	(W & ¬ F)	&Intro 3,5

III: OBJECTIVES OF THESIS AND JUSTIFICATION OF THE CHOSEN APPROACH (PRODUCTION RULES) AND DOMAIN (SENTENTIAL LOGIC)

1. Objectives of the theorem-prover

In the design of this theorem-prover, two antagonistic objectives had to be taken into account: the simplicity of the algorithm on the one hand, and its ability on the other hand to construct sufficiently *elegant* proofs. A choice had to be made between a theorem-prover with relatively simple strategies, and a more sophisticated algorithm capable of writing proofs that would compete in elegance with hand-written ones.

Unlike other proof-techniques, the natural deduction method does *not* provide a mechanical, deterministic procedure specifying *how* proofs are to be constructed. When constructing the truth-table or the tree corresponding to an argument, one never has to make a choice: at any step of the procedure, what comes next is forced. On the other hand, there is always an infinity of legal things to do[1] at any step of a natural deduction proof. This freedom implies the danger that the proof process will follow a wrong direction without ever reaching the goal (the conclusion). But it also allows the

---

[1] From A, one can derive (A V B), (A V C), etc...
From nothing (an empty set of premisses), an infinite number of tautologies can be generated via the rules of inference of natural deduction.

construction of concise proofs (without unnecessary steps) that lead from the premisses to the conclusion in a natural (because *goal-directed*) way.

There are more rules of inference in natural deduction than in the other two methods. There is a direct link between the number of rules of inference and the similarity of the proofs to ordinary human reasoning. However, the more applicable rules of inference there are at a given step in the proof, the more "intelligence" is required in the theorem-prover to decide what is the optimal rule to apply.

The objective of the theorem-prover presented here is not at all to come as close as possible to a decision-procedure. Had this been an objective, then a transformation into some sort of normal form together with a relatively simple strategy would have been appropriate the trick (e.g., putting all premisses and the conclusion in disjunctive normal form and applying repetitively the rule of disjunction elimination). What is wanted on the contrary is loyalty to the spirit of natural deduction proofs, i.e., loyalty to the principle of constructing "nice" proofs that look as similar as possible to proofs written by human provers.

Similarity to human-written proofs was wanted not only in the final output (i.e., the completed proof) but also in the way the proof was *constructed*. It is for instance an empirical fact that, in the construction of a proof, an experienced human prover very rarely has to use backtracking. In-

stead of relying on extensive search and failure-directed backtracking, human provers tend to take the time to "observe" a lot of things in the premisses and the conclusion. They do a lot of pattern analysis, that can be described as "premiss-interpretation" (forward reasoning), "comparison" between premisses (seeing common sub-formulas), "goal-analysis" (backward reasoning, examining the main connective of the conclusion), etc... The same type of approach was expected from the theorem-prover.

Actually, a backtracking procedure was present in the design of an early prototype of the system. It was discarded on the basis that human reasoners only exceptionally rely on backtracking. Hence a program with the appropriate knowledge should also be able to do without it. Backtracking - and extensive search - is probably much easier to do for a machine than it is for a human being, but search is expensive (as far as elegance is concerned) and it is better if possible to avoid it altogether. This decision to avoid backtracking as much as possible made the theorem-prover more complex than it could have been: instead of letting the system try a wrong route and then backtrack, failure of choosing the right route first was interpreted as a symptom of not enough intelligence. Such incorrect behaviour was therefore corrected by the addition of a new rule to the database.

The first and main reason to build a more ambitious theorem-prover was therefore to be faithful to the spirit of

natural deduction proofs. Another reason was that the theorem-prover was meant from the beginning to evolve into a tutoring system able to guide a student in the construction of proofs. Now the aim of teaching natural deduction to students is clearly not to give them a mechanical procedure for constructing proofs, but rather to give them a tool to use in the formalisation of their own reasonings.

This point will be expanded in section IV.4 with a description of the way students are taught to do proofs in natural deduction. Basically, the idea is that it seems preferable, pedagogically speaking, to give the student as much knowledge as possible about the relevant patterns that he/she should be able to expect and detect in the statement of a problem, rather than a universal recipe based on failure-directed backtracking.

We may therefore summarize as follows the objectives of the theorem-prover:

- *construct not only correct but good-looking natural deduction proofs*

An obvious objective in the construction of the proofs is mathematical elegance. Major objective of this prover: minimize the length of proofs.

- *facilitate the implementation of a tutor able to teach natural deduction.*

In other words, the system should be able not only to construct the proofs but also to explain how they were found.

- *minimize backtracking*

More generally, the construction of the proof itself (not just the proof) should be as similar as possible to a human way of doing it.

- *keep as simple as possible the strategies used by the theorem-prover*

This should be done while preserving optimality as far as correctness, elegance of proofs and speed of proof construction are concerned.

2. Justification of a rule-based system

A rule-based system was chosen to implement the theorem-prover's strategies, so as to allow a simple evolution of the system towards more elegance and similitude to human reasoning. In the coming sections, we will see in more details what these heuristic rules do and how they are used. It is sufficient to say at this point that all rules have a pattern similar to:

If the actual goal has such and such property,
then add this or that subgoal to the goal-list.

or

If a formula with such or such property has already been derived,

then apply this or that rule of inference.

The rule-based approach permits an easy experimentation with the content of the knowledge-base available to the main program (whose ~~task,~~ at any step of the proof, is to decide what to do next). The major advantage of production rules is the relative ease of addition, modification and re-ordering of rules. It is this property that makes production rules popular in the construction of "quick prototypes". Another significant advantage of a rule-based program is the ease of generating explanations describing the program's behavior. In the present case, the same rules that are used by the theorem-prover to decide "what to do next" are also used by the tutor to explain a step of the proof or to justify the choice of a specific action.

It remains true that building a system based on heuristic rules, instead of on a clear-cut algorithm, makes it quasi impossible to prove anything about the completeness or the adequacy of the chosen set of rules. Actually, no claim is being made to the effect that an optimal theorem-prover (as far as reliability as well as speed are concerned) should remain rule-based. On the contrary, it seems most likely that a considerably more efficient version of the present theorem-prover

for natural deduction could be rewritten using something like a decision-tree.

3. Limitation to sentential logic

Bledsoe [5] has emphasized the advantages of natural deduction over other (more syntax-oriented) theorem-proving techniques, identifying the crucial feature of non-resolution theorem-proving to be the importance given to heuristic, *domain-dependent knowledge* in the construction of the proof. Proof-checkers and theorem-provers using natural deduction have been applied successfully to various domains such as set-theory, the theory of types, non-standard analysis, elementary number theory[2], and the like.

The scope of the present project has been deliberately restricted to sentential logic proofs, not because natural deduction cannot be applied to other richer logics, but so as to keep the domain of proofs relatively small. The theorem-prover presented here is only "knowledgeable" about the logic of truth-functional connectives. It is not expected, due to this restriction, to invent or even to prove any interesting

---

[2] Significant work in this direction has been pursued at Edinburgh and Cambridge University. See for instance Lawrence C. Paulson [17]. See also Xuhua, L. & Zhan, C. [26].
The system implemented by Xuhua and Zhan constructs natural deduction proofs in elementary number theory. Peano's axioms are expressed in first-order predicate logic. The "relative principle" states that the proof of a theorem should be domain dependent and problem dependent.

new theorem (actually, very few theorems of sentential logic can be viewed as *interesting*).

Again, the focus of this theorem-prover was on the *aesthetics* of proofs rather than on their intrinsic usefulness. The idea was to give the system sufficient information to construct "nice" proofs in a restricted domain. Sentential logic actually turned out to be a sufficiently large domain to experiment with the objectives of elegance and auto-explanation (ability of the system to explain what it does and why). Future enhancements on the theorem-prover may include the addition of more efficient inference-rules for sentential reasoning (e.g., addition of de Morgan's laws, that would drastically shorten the derivations), or extension towards predicate logic or other logics.

IV. DESIGN OF THE THEOREM-PROVER

In section III, the central objective of the theorem-prover was stated as the emulation of proof-construction *as done by human reasoners*. The main difficulty in this task essentially lies in the fact that there is no available theory of what the art of constructing *nice* (or "elegant") proofs amounts to.

1. What is a "good" proof?

Before going into the structure of the program itself, we must therefore ask ourselves, first what is this "niceness", what is "elegance" in the context of proofs, and second how production rules can help reaching it. In the domain of proofs, the concepts of "elegance" and "niceness" are probably as ill-defined and vague as when used in descriptions of art objects. This is not equivalent to say that these concepts are without significative content, but only that their meaning tolerates some imprecision.

There are in fact a few criteria of evaluation of proofs (beyond correctness, which is relatively trivial) that are easy to specify: we are able to tell a good ("elegant") proof from a bad one by checking whether it contains redundancies, or steps that are irrelevant to the goal, and whether the ordering of the steps of the proof makes "good sense" with respect to the overall structure of the proof. Among the most ob-

jectively measurable criteria of elegance is the shortness[1] of the proof (a consequence of the elimination of redundancies and off-the-track attempts to prove the goal)[2]. So we are able to say of a particular proof that it is correct, but non-elegant, by relying to some extent upon such objective criteria. Yet it is important to notice that

a) it is easy to find counter-examples to any proposed set of such criteria: for instance, it is just not true that in the intended sense of elegance the best proof is always the shortest one; in the intended sense, a proof is elegant just in case it parallels "good thinking".

and

b) even if we were satisfied with some set of criteria for evaluating the elegance of proofs, it would not provide us with a mechanical method for the construction of such proofs.

---

[1] When a clear-cut definition of mathematical elegance is needed, people often appeal to a reduction of that concept to that of the size of the proof.

[2] In the present case, the size of the proofs had to be taken seriously, if only for reasons of screen size on micro-computers that sets the maximum length of all proofs to about 20 lines. This is however (most of the time) sufficient for our purposes if proofs are constructed "elegantly".

2. Use of production rules to capture knowledge about elegance

Heuristics are useful when a problem cannot be solved with a deterministic algorithm. The theorem-prover therefore uses a set of heuristics, expressed as "production rules", that are meant to encapsulate specific as well as general knowledge about the art of constructing *not just correct, but elegant proofs*.

Heuristic rules are essentially incomplete and imperfect. They can yield good solutions most of the time, but do not guarantee that such good solutions will always be found. As any rule-based system, the theorem-prover of ND was designed to evolve in time towards more elegance and completeness. The first prototype of the theorem-prover had a relatively limited set of rules and was able to do mostly simple proofs. The current version is much more powerful: for instance, rules have been added that can help in choosing what contradiction to look for in a reductio ad absurdum. Other rules have been added that look at the internal structure of newly derived theorems (e.g., when a hypothesis is made): these rules are likely to be activated if the goal is a sub-formula of one of these new theorems.

Some of the rules were actually written by relying on a priori belief that they would work, but many rules were discovered empirically, by finding problems that the theorem-prover was unable to solve appropriately with the previous set of rules. The evolution of the system was therefore not based

on precise evaluation criteria (such as size, redundancy, etc...), but on a rather subjective evaluation of the quality of proofs. Whenever the program solved a problem differently than what I would have done, it was interpreted as a symptom that a rule was incorrect or missing.

The order in which rules are consulted has been modified many times, again empirically, as a function of the corpus of exercises. Improvement of the system was therefore achieved through the addition, modification and reordering of these heuristic rules[3].

3. Rules of inference vs proof-strategies (heuristic rules): what is permitted vs what should be done.

It is perhaps appropriate at this point to emphasize that the set of heuristic rules should not be confused with the set of inference rules as defined in section I.

The 11 rules described in section I are inference rules: together, they specify the set of valid (legal) inferences that can be made over a given set of sentences. For instance, if sentence A has already been derived, then it is legal

---

[3] In the spirit of such empirical development of an appropriate set of rules, a module for automatic generation of exercises was considered but has so far not been implemented (it is in the agenda for future development). Such a module would be useful to exhaustively test the program's current set of rules and evaluate its ability to solve any proof. This generator could also find interesting applications in the tutorial (e.g., construct an exercise that would take the student model into account, instead of merely selecting an exercise in a pre-established bank of exercises).

to write (A V B) under it and justify it by the rule of V-Introduction. Rules of inference map theorems to theorems. They specify what it is *permitted* to infer. They can be thought of as the *declarative* part of logic and say nothing about *when* a particular rule should be used.

To make the system efficient, and able to produce human-like reasonings, a set of *strategy* rules is used, that specify when and in what order the inference rules should be applied. Instead of mapping theorems to theorems, a strategy rule can most of the time be thought of as mapping goals to subgoals. In other words, strategy rules concern the *control* part of proof-construction. They can be thought of as heuristics or hints about what one should do to find a proof.

Hence the set of strategy rules is an attempt to refine the strategy that would consist in deriving whatever is legally derivable. We can easily imagine how a "dumb" theorem-prover could, without some sort of control from above, spend its time using the inference rules to generate new theorems from the premisses until one of the derived formulas matched the goal formula (the conclusion). This of course would clearly be quite impractical since, as was already mentioned, the inference rules allow an infinity of things to be derived from any number of premisses.

We have seen in section I the conditions of application of each of the 11 rules of inference. Looking a this list, it appears clearly that some of the inference rules (in particular the introduction rules) have very *weak* conditions of application. For instance, the rule of disjunction introduction can be applied on any sentence whatsoever: it says that from any sentence S it is permitted to derive (S V S'), for any S'. The condition of application of the rule of conjunction introduction is also extremely weak: it says that from any pair of sentences S and S', it is permitted to derive the new sentence (S & S'). It would not therefore be feasible to take the rules of inference themselves as production rules and to activate a rule of inference whenever its conditions of application were satisfied: this would allow the derivation of too many sentences, most of which without any relation with the goal.

It is therefore the application of introduction rules that must be most carefully controlled by goal-directed strategies. It appears, on the other hand, that successive applications of elimination rules are not likely to create problems having to do with the undirectedness of the search[4]. But the concern with elegance of the proof forbids the simplest ap-

---

[4] This has to do with the fact that the application of elimination rules may only generate sub-formulas - hence shorter formulas.

proach that would consist in writing on the screen all sentences that are derivable via some elimination rule.

4. Basic strategy

When students learn natural deduction theorem-proving, they are usually told to use the following three principles, that should be applied in order:

a) Work from the bottom up, inasmuch as possible. In other words, try to apply the *introduction rule* for the main connective of the statement you are trying to prove. Determine which sentences, if you could derive them, would enable you to derive the conclusion. Deriving these sentences becomes your new goal.

(Backward reasoning).

b) See what immediately follows from the premisses and what you have derived so far. That is, if you can't work from the bottom up, do some top down thinking.

(Forward reasoning).

c) If nothing else works, use RAA (reductio ad absurdum), i.e., assume the *contrary* of what you are trying to prove, and derive a contradiction.

Although relatively simple and incomplete, this basic strategy is sufficiently strong to lead to a solution in

quite a large number of proofs. Beyond this general advice however, the student is told to "use his/her brain" in the task of determining what should be done in particular situations. In the construction of non-trivial proofs, "using one's brain" essentially amounts to being able to observe certain patterns or properties in the premisses and the conclusion, and to deduce from these observations what subgoal is likely to be a useful intermediary step, what hypothesis is likely to have an interesting consequence that will somehow contribute to the proof, etc... Basically, the student is asked to refine for himself the basic strategy (the three principles listed above) as he/she tries to solve new exercises. To do so, he/she is asked to use intuitions about valid inferences in sentential logic (i.e., about the ordinary interpretation of the connectives used in sentential logic). Indeed, beyond the general strategy described above, the construction of proofs in natural deduction is taught more as an exercise about the meaning of ordinary words used for reasoning (IF, AND, OR, NOT) than as a symbol-pushing mechanical procedure.

In the process of refining the basic strategy, the student is expected to master the technique of goal decomposition into subgoals; in particular the use of subproofs to prove

certain types of goals (the use of subproofs should be restricted to proofs of implications and proofs by RAA).

5. Forward and backward reasoning in the theorem-prover

The theorem-prover mimicks the human way of constructing a proof in natural deduction, using both *forward* and *backward* reasoning: the search is done both in a forward direction from the premisses towards the conclusion (from the top of the proof to the bottom), and in a backward direction (from the bottom up). The strategy rules are therefore divided into two subsets. The first set is concerned with the question of what can be inferred "immediately" from what is already known; the second set with what must be proved in order to reach the conclusion.

6. Both *backward* and *forward reasoning* are implemented in a *forward chaining* rule-based system

Any rule-based system consists of a set of rules of the form IF <X> THEN <Y>, together with an "inference engine" that operates on the set of rules to make deductions. Whatever form the data may take in the system (ordinary data-structures, or sentences), the <X> and <Y> of the last sentence always refer to conditions and actions on the database. It is the inference engine's responsability to specify how the rules are to be used to consult, and eventually modify, the database.

In a forward chaining system, antecedents of rules are considered first and the *firing* (activation) of a particu-

lar rule eventually modifies the context so as to make true the antecedent of another rule. It is therefore appropriate in this case to talk of rules being *chained* in the forward direction.

In a backward chaining system, it is the consequents of the rules that are examined first. This allows the identification of which subgoals should be pursued so as to make the firing of the rule possible.

It is important that forward and backward chaining may be applied on the *same* heuristic rule. They are just two different ways of using the information contained within the same rule. In the present theorem-prover, only *forward chaining* is used (i.e., the rule precondition is always examined first and if satisfied, causes the rule to fire), although certain rules have as objective the implementation of a form of *backward reasoning*.

The implementation of mixed-chaining systems (that use rules sometimes in a forward and sometimes in a backward direction) calls for complex inference engines that are typical of sophisticated rule-based systems. In the present case, the inference engine is extremely simple: rules are consulted in a pre-determined order and the first rule whose pre-condition is satisfied fires.

7. The main data-structures of ND

Backward reasoning - the decomposition of goals into subgoals - is therefore implemented as a set of rules that is used in the same way (forward chaining) as the rules that are meant to be used in forward reasoning - the derivation of the consequences of a sentence or set of sentences. The major difference is that backward reasoning rules act upon a structure that keeps track of the evolution of the *goals* (GOAL_LIST) while the forward reasoning rules update a structure that keeps up to date the set of *available theorems* (AVAILABLE).

A dynamic structure is used to store the output of forward reasoning - the array AVAILABLE. Actual action on the screen is goal-directed. When something is sent to the screen, the structures ONSCREEN and PROOF are also updated.

GOAL_LIST is implemented as a stack of goals. Whenever a "backward-reasoning" rule fires, one or two new new goals are defined and added on top of the stack. Whenever a goal is achieved, it is popped off the stack. A new goal may be

- a formula (e.g., if goal has the form X & Y, add goals X and Y on top of GOAL_LIST),

- a subproof (e.g., if goal has the form X $\Rightarrow$ Y, add as a new goal the proof of Y under the hypothesis X),

- a subproof leading to a contradiction (e.g., if the current goal is the negation of the antecedent of a theorem in AVAILABLE, use *reductio ad absurdum*, i.e. try to derive a contradiction under the hypothesis of the negation of the goal).

Note that GOAL_LIST may contain fully specified goals (e.g., *prove formula X*, or *prove formula X under hypothesis Y*), as well as goals that are only partially determined (e.g., *find a contradiction under hypothesis X*).

In the course of a derivation, whenever a new sentence appears in AVAILABLE, either as a hypothesis or via forward-reasoning, this new sentence itself is "forwarded"[5], i.e., its immediate consequences are derived, stored into AVAILABLE, and forwarded. At all times, the structure AVAILABLE therefore contains the premisses, the hypotheses that are still active, as well as all the sentences that have been derived earlier in the proof and have not yet been discarded.

A sentence occupies only the first part of a slot of AVAILABLE. Actually, AVAILABLE is an array of *lists* each of which consists of

.a sentence,

.the rule of inference used to derive it,

---

[5] Words in bald type are names of functions or predicates used by the theorem-prover. Data-structures are in capital letters.

.the position in AVAILABLE of the sentence(s) from which it was derived

e.g., (AVAILABLE 7) might be (A HYP)

(AVAILABLE 9) might be ((C => A) =>E 4 7)

So, AVAILABLE stores not only sentences but also their origin. Not all sentences that appear at some point or another in AVAILABLE will actually be used in the proof. But when a sentence is needed, it may be fetched from AVAILABLE with all the information needed to justify it. The information stored in AVAILABLE makes it possible to define the function (path x) which returns a sequence of steps from which x was derived.

Care must be taken with the management of AVAILABLE, in particular when a hypothesis is discarded, since whatever was derived on the basis of this hypothesis should no longer be available[6].

The production rules of ND are therefore divided into three sets (one for top-down reasoning, one for bottom-up reasoning, one to guide the search of a contradiction). The context decides what group should be consulted. A detailed listing of these rules is given in appendices 1 and 3.

---

[6] The functions delete and discard are responsible for this elementary "truth-maintenance-system".

V. EXTENDING THE THEOREM-PROVER INTO A TUTOR

1. Motivation and objectives of tutor:

Even at an early stage of development, the theorem-prover appeared to be easily expandable into a computer-assisted logic course (natural deduction constitutes the core of the typical introduction to logic course). The idea was that on top of a rule-based program capable to do the proofs in a natural manner, it should not take too long to build a program able to teach someone how to do them: the strategic rules used by the theorem-prover could be made available to a tutor that would use this "knowledge" to give advice about the construction of proofs.

At the time, no micro-computer program was available on the market to teach natural deduction. A team from Stanford University, headed by Frederic Suppes[1], had developped a computerized logic course (*Berty*), containing among other things natural deduction exercises. But *Berty* ran only on mainframes with lots of virtual memory. Also, it had actually been designed from the start to act as a proof-checker and did not have the ability to construct proofs autonomously. It was

---

[1] Suppes, P. and Sheehan, J., "CAI course in axiomatic set-theory", in [23].

therefore not well-suited to be extended into an "intelligent" tutorial.

2. STEL[2] in the perspective of current ICAI:

The first generation of computer-assisted courses offered relatively little interactivity with the student: the computer's role was essentially limited to presenting the student with questions or exercises for which the correct answer or solution had been prerecorded. The recent evolution in the last few years of computer-assisted instruction has followed two major trends: the first towards sophisticated "micro-worlds", that provide the student with rich environments to explore; the second towards so-called "intelligent tutoring systems".

In such systems, a program (the "tutor" or "coach") attempts to emulate the pedagogical functions of a human teacher. Beyond congratulations and error messages, a tutor is expected to provide intelligent guidance, i.e., to come up with appropriate remarks and explanations that are relevant to the student's progress. The ideal computerized tutor is often pictured as a silent observer who watches patiently over the student's shoulder and occasionally decides to intervene, giving a hint when the student hesitates or ... suggesting further reading when the student seems interested.

---

[2] STEL (Système tutoriel pour l'enseignement de la logique)] is the name of the tutoring system that was developped on top of the theorem-prover.

To be sure, a lot remains to be done in that direction before the dream of a truly human-like tutor is actually implemented. Some programs, however, already exhibit impressive features. At the present time, we are still at an early exploratory stage of the possibilities of computerized tutoring. It is, however, universally accepted that intelligent tutoring is possible only if the tutoring program has enough domain knowledge to solve the exercises by himself. Beyond that, there is also widespread agreement over the idea that intelligent tutoring needs sophisticated student modelling: not surprisingly, it appears that the most difficult part of all ambitious tutors has to do with diagnosing accurately the student's behaviour.

The tutor implemented in *Stel*, as we will see, is relatively modest and shares only a few characteristics with the most ambitious tutorials. To put it in the perspective of the current technology, let us look at two of the most significant tutoring systems, *Buggy* and *Proust*.

Buggy, one of the earlier AI programs in education, has put forward the idea that, whenever a student makes some mistake in attempting to solve a given exercise, this mistake is not to be interpreted as mere "lack of understanding" or irrationality on his/her part, but rather as the consequence of a wrong theory concerning the subject matter. *Buggy* therefore tries to identify the "bug" in the student's head and then to explain to the student what the bug is and how it relates to the correct theory. The domain in which *Buggy* specializes is

subtractions in arithmetic. The domain knowledge contains a set of correct rules as well as a set of "malrules"[3], about the construction of subtractions. The first set represents correct strategies, while the second contains variants (misconceptions) of these strategies that lead to common mistakes. When the student makes a mistake, the program is thereby able to base its diagnosis on the identification of which "malrule" was applied by the student.

The feasibility of this approach depends of course on the possibility to predict a priori all the possible learning difficulties (learning bugs) that any student may encounter. This approach did not seem appropriate in the implementation of a tutor for natural deduction proofs which belong to the class of non-deterministic problems (problems for which there is no mechanical procedure for reaching the solution).

Among other problem-solving domains (in education) belonging to the same class are intuitive geometry, analytic geometry, transformations of equations in elementary calculus, transformations of trigonometric expressions into other expressions, etc... In problems belonging to such domains, it is much harder to diagnose whether what the student is doing should be accepted as progress towards the solution. In natural deduction, identifying strategic mistakes is not straightforward and isolating *the* bug may be rather tricky when a mis-

---

[3] See Sleeman, D., "Assessing aspects of competence in basic algebra", in Sleeman & Brown [22].

take is detected. Sometimes students make mistakes because they have a wrong (i.e., buggy) interpretation of how a rule of inference should be applied. But some students just have wrong logical intuitions, believing for instance that the formula (A or B) implies A in the same way that (A and B) does.

The *Proust* program[4], another classic in ICAI (intelligent computer-assisted instruction) was designed to help students in debugging non-syntactic errors in Pascal programs. Unlike subtracting, programming surely belongs to the category of non-deterministic problem domains (there certainly is more than one way of writing a program that satisfies a given set of specifications). *Proust* incorporates a problem-description language that allows detailed expression of program specifications. The tutor's expertise is based upon specific knowledge about selected problem tokens. It knows enough about programming to write these particular programs, and this knowledge is used by the tutor to decide what errors are to be expected in the student's behaviour.

This makes *Proust* one of the most serious tutoring systems available today. Its knowledge allows it not only to predict what are the most likely mistakes that a student will make in attempting to solve the problem, but also to identify the student's intentions and to base its comments on reasoning

---

[4] PROUST (PROgram Understander for STudents); see Johnson, W. L. and Soloway, E., "Proust: an Automatic Debugger for Pascal Programs", in Kearsley [12], and "Proust: Knowledge-Based Program Understanding, in Rich and Waters [19].

over these intentions. The originality of the program consists in its ability to determine what the student is trying to do.

Proust attempts - successfully, according to its authors - something that does indeed seem very ambitious: to be able not only to decide what the student *should* be attempting to do (this only requires expertise about solving the problems), but also what the student actually *is* trying to do (which requires sophisticated reasoning over intentions).

We may summarize the current state of the art by listing a number of desirable features - many of which seem dictated by common sense - that characterize diagnostic tutors.

a) It is important for the system to have knowledge of the domain it teaches: in order to be able to explain something, the program must be able to solve the problems without external help.

b) In a non-deterministic kind of problem-solving, it is important to give the student the feeling that he is not just following a mechanical procedure but that he or she has to think about what the problem means. Students in fact do tend to expect (wrongly) that all tasks are solvable by following a well-defined algorithm.[5]

c) The program should have the ability to emulate a human

---

[5] See Sleeman, D., "Micro-Search: A 'Shell' for Building Systems to Help Students Solve Non-deterministic Tasks", pp. 69-81, in Kearsley [12].

teacher's language.

d) Students often learn better by being given access to a rich environment that they may explore freely, than by being constantly guided by a tutor. It certainly appears reasonable for the program to vary its teaching strategies.

e) Ideally, the system should know about typical patterns of errors that students are likely to make.

f) Reasoning over the student's intentions is useful: in order to give useful hints, the program must know what the student is trying to do.

3. A modest tutor

Due to time limitations, the ambitions of the *Stel* tutor have been deliberately tied down. In its current state, the student model is nothing but a purely quantitative structure that registers student successes and failures in the completion of proofs. It would certainly be an interesting continuation of the project to work on a reasonably ambitious student-model with qualitative evaluation of student responses, somewhat along the lines suggested by programs such as *Proust*.

The implementation of a tutor on top of the theorem-prover was achieved mainly through the addition of a module able to check the student's proofs and to generate hints when appropriate. The student and the tutor do the proof in parallel. At any step of the proof, the student is given the option to ask for a hint if he/she doesn't know what to do. The tutor attempts no analysis of how the student's strategy dif-

fers from its own, but intervenes only when inference rules are incorrectly applied.

Unlike the tutors of *Buggy* or *Proust*, the tutor in *Stel* does not pretend to be able to diagnose the bug in the student's head. To be sure, a lot of psychology would have to be involved in a sophisticated tutor able to diagnose accurately the foundation of student's mistakes. No analysis of the student's intentions is attempted in *Stel*. On the other hand, while *Buggy* specializes in a deterministic domain, and while *Proust* is able to guide students in only a pre-selected set of programming exercises, *Stel* is able to take any exercise in natural deduction (restricted to sentential logic), complete the proof, explain any step of the proof, and eventually guide a student in the construction of the proof.

Along the lines of the "microworld" concept, a module made available by *Stel* allows the student to explore[6] freely and at his/her own pace the world of concepts that are used in natural deduction proofs. The concepts are structured into five trees that cover:

- the different types of formulas (conjunctions, implications, etc...)
- the names of the parts of compound formulas (antecedent, conjunct, etc...)
- the structure of a proof (premisses, subproofs, justifi-

---

[6] Screen-dumps taken from this module may be found at the end of appendix 4.

cations, etc...)

- the 11 inference rules

- strategy-related concepts (top-down, bottom-up, goal decomposition, the role of hypotheses, etc...)

4. new data-structures: student-model and bank of exercises

The evaluation of the student's progress with respect to the natural deduction technique is recorded in a structure, the "student model". This structure contains slots for each of the eleven inference rules as well as for a number of concepts and techniques that the student is expected to master (e.g., use of hypotheses, restrictions on reiteration, etc...). The student model specifies, for each rule of inference and each major concept, how many mistakes the student has made and how many correct answers he has given. At the end of each session, the student model structure is recorded on the student's floppy-disk to be re-read at the beginning of the next session.

Another module is responsible for choosing the next exercise - tuned to the student's level - from a bank of exercises and submit it to the student. The exercises are divided into subgroups according to their overall difficulty (simple, intermediate, hard) and, within each subgroup, with respect to the rules of inference that must be applied to solve them.

5. explanation system: enrichment of the rule structure

In the theorem-prover, rules were used to decide what to do at any step in the construction of the proof. The

extension of the theorem-prover into a tutorial implied the addition of enough structure to the rules to allow them to be accessed by three new modules:

- an explanation module

  To each rule is attached a "message-schema" that is used to generate a message appropriate to the context[7]

- an error-message generating module

  To the property-list of each rule of inference is attached a condition of application.

  An error-message schema is attached to each rule and used to explain why, in the given context, the condition of application is not satisfied.

- the student model updating module

  To each of the production (strategy) rules is attached an inference rule, and, in some cases, a concept: a correct application of the rule increases the student model's score for that concept as well as for the rule of inference.

  A list of recent errors is also updated constantly and is consulted at the time of selecting the next exercise.

---

[8] See screen-dumps in appendix 4.

6. Remarks on the implementation of the tutor

Altogether, most of the extension of the theorem-prover into a tutorial has been a fascinating project to work on. However, it certainly took *much more time* than was first expected, mostly because of the necessity of a simple to use and clean user interface[8]. The overall quality of user-interfaces has improved greatly in recent years with the multiplication of micro-computer software targeted to naive users. Stel was written in a relatively elementary Lisp that did not include many of the primitive low-level functions available in more popular languages such as Pascal or Basic. But beyond these user-interface functions, I would like to mention briefly two other unexpected "problems" that contributed to take the time devoted to the implementation of this tutor far beyond my first expectations.

First of all, it soon became manifest that the product would not be usable by students without a module explaining all terms used by the tutor. This evolved into a quasi logic-course, defining all the logical concepts used in natural deduction; a lot of time and energy had therefore to be invested in a necessary part of the project that had very little do do with artificial intelligence: deciding what concepts should

---

[8] If I had to get involved again in a program meant to be actually used by students, I would probably think twice about it...

and which should not be included in the directory, classifying them and then finding the right words to explain them.

A second unexpected difficulty in the implementation of the tutorial came from the late realization that there was no way to avoid backtracking[9] when the program was doing the proof in parallel with the student. It is indeed not realistic, for obvious pedagogical reasons, to forbid the students to go in completely wild directions: if a student wants to derive something that is legally obtainable by one of the rules of inference, he or she should be allowed to do it; the program is in fact totally lenient with respect to the choice of all subgoals. Some control is made over the choice of hypotheses[10] because it is part of the natural deduction general strategy that hypotheses should not be made gratuitously, but always with some particular subgoal in mind. This necessitated more complex internal structures (e.g., the new structure ONSCREEN can loose formulas if the student erases them). It also changed significantly the management of the structure AVAILABLE (since backtracking as well as discarding hypotheses could now make a theorem unavailable).

---

[9] Backtracking was considered undesirable in the theorem-prover for reasons discussed in section III.1.

[10] See screen-dumps in appendix 4.

7. Experimentation of the program and possible extensions

An early version of *Stel* was tested successfully with two groups of philosophy students at College de Maisonneuve. The student response was satisfactory although of course not as enthusiastic as we would have hoped. Version 1 of the product is now in the process of being distributed to all Quebec colleges. After spending some time away from the intricacies of user-interfaces, it is very likely that a second improved version will be rewritten in the near future, probably in Common Lisp.

VI. RELEVANCE OF NATURAL DEDUCTION TO ARTIFICIAL INTELLIGENCE

1. It is important for a theorem-prover to be able to explain its own reasoning. This is easier to realize in a natural deduction system.

Leaving aside the realm of education, let us try now to evaluate the potential benefit of using natural deduction as a tool for actual theorem-proving. In some situations, we may be interested to use the deductive ability of a theorem prover only in order to establish whether some beliefs are logically entailed by the set of beliefs stored in a given knowledge-base. In this kind of situation, some version of resolution-based reasoning may be an appropriate choice. If, on the other hand, we are interested in the theorem-prover's ability to explain, in ordinary intuitive terms, how it was able to derive a statement from other statements, then natural deduction presents the obvious advantage that its proofs are constructed in a way that is similar to human proofs.

In mathematics, people often care more for understanding why something is a theorem than for testing whether it is one. In the field of deductive databases, people will not (or should not) accept reasoning-based (i.e., non-trivial) conclusions from a computer without knowing how they were reached. On the one hand, a conclusion is more convincing if

what supports it is made explicit. On the other hand, it is unrealistic to hope for bug-free databases, as soon as they are relatively large: it is clearly a significant advantage for debugging if the system is able not only to deduce things but also to explain how its conclusions were reached. This suggests that sophisticated theorem-provers ought to be complex rule-based systems rather than simple axiomatic generators.

2. Natural deduction allows the possibility to give a sketch of the proof or to give partial explanation in the case of failure in the search of the proof

The structure of a natural deduction proof makes it extremely simple for the system to give a sketch of a proof, instead of the complete proof. This feature could be taken advantage of whenever a quick explanation is wanted. (How would a resolution theorem-prover summarize its proof?)

In the case of failure of finding a proof for a formula, the program could give useful comments. There should be no difficulty in producing reports such as the following:

> "You asked me to examine whether it is true that P => (Q & R). I was able to show that P => Q but, as far as I can see, P does not imply R. Actually, P & ¬ R is consistent with all integrity constraints of the present database.
> From the hypothesis P & ¬ R, I have derived the following: A, B, C, etc..."

3. Richer logics than sentential logic or first-order logic

Resolution theorem-proving uses only one rule of inference. For this reason, it can be praised as a very simple and thereby "elegant"[1] technique. But this simplicity is pathological: because it is so simple, the resolution technique is hard to modify[2]. The naturalness (i.e., inherent complexity and redundancy) of natural deduction makes it more amenable to modifications. Here are a few examples of what modifying a natural deduction system amounts to.

Natural deduction permits a simple definition of new connectives. Indeed, there are a number of applications to which natural deduction can be tuned. This, again, is due to the close resemblance between a natural deduction proof and ordinary human reasoning. The addition of such new connectives to the language makes it possible to express reasonings that were not expressible before. It is important to realize that sentential logic is limited to reasonings over connectives such as AND, OR, NOT and IF THEN. Sentential logic is therefore able to account for the validity of only a very small number of intuitively valid arguments[3]. As soon as we want to go beyond these

---

[1] At least in the mathematical sense, not in the intuitive sense of "elegance".

[2] As far as I know, resolution stops at the level of first-order predicate calculus, in the hierarchy of existing logics.

[3] The fact that the first expert systems (*Mycin*, *Prospector*) used a set of production-rules limited to sentential logic (no variables or quantifiers in the rules) simply shows that these systems were primitive. The fact that they were useful and produced significant results (in such fields as medical diagno-

reasonings, we must enrich the language and the logic that goes with it.

Predicate logic, with the introduction of variables and quantifiers, is able to do significantly better: it adds the expressive power of the words ALL and SOME to the language. To apply the present ND theorem-prover to proofs in predicate logic, the syntax of the language has to be modified (i.e., the function that tests whether a string of symbols is a well-formed formula must of course be changed), and the introduction and elimination rules for the quantifiers have to be added. Quantifiers usually introduce a number of difficulties (substitution is difficult to formalize, free variables must be skolemized, ...). But these difficulties are now relatively well-understood and may be captured by restrictions on the application of rules in natural deduction.

Extension to predicate logic is an *addition* to the language of classical sentential logic. In some cases however, the formalization of certain reasonings calls for *restrictions* to classical logic: there are indeed certain kinds of arguments that classical logic validates and that ordinary language finds unacceptable. It is, for instance, generally accepted that no truth-functional[4] connective can capture the "real" sense of the English "IF..THEN".

---

sis and oil prospection) indicates, however, that very simple logical mechanisms may generate non-trivial reasonings.

[4] A connective is *truth-functional* if the truth-value of the compound formula obtained by using this connective only depends on the truth-values of the formula's components.

Consider for instance the sentence

"If Napoleon had won the battle of Waterloo, England would now be a province of France."

Given your understanding of the meaning of the word IF, you *may* or you *may not* agree with what this sentence says, depending on your beliefs about Napoleon's intentions, the pride of the English people, etc...

However, if we formalize the sentence using classical logic, it turns out to be *true*, since the antecedent of the implication is false.

Classical logic is truth-functional and validates, among other things, sentences such as $\neg A \Rightarrow (A \Rightarrow B)$ and $A \Rightarrow (B \Rightarrow A)$ which are intuitively valid only if the arrow is interpreted in a way that diverges strongly from the ordinary language meaning of "IF...THEN".

If we want the language to be able to express the real if...then relation, i.e, to express not just material implication, but also such things as causal implications and hypothetical or counterfactual conditionals, then a new, non truth-functional connective, must be added to the alphabet. This is precisely what relevance logic is about. In fact, it was designed essentially to forbid inferences such as *A, therefore B implies A* (the so-called "paradoxes of implication") that were inherent to classical logic.

The Pittsburg team of logicians [1], who developped in the 1970's various systems of relevance logics, had adapted natural deduction to do proofs in relevance logic much before they had agreed about an axiomatization for it. The innovation was essentially the attachment of a subscript to each sentence of the proof, together with a number of subscripts-related restrictions on reiteration.

The ability to obey the restrictions of relevance logic could be used in the development of systems able to do different things, such as

- generating comments when the derivation of a theorem violates relevance principles (i.e., when an inference is made that shares something with the paradoxes of implication).

- intelligent reasoning in the presence of contradictory information. Take a distributed database. Someone feeds A at site 1 and someone feeds NOT-A at site 2. If the system uses a theorem-prover that conforms to the rules of classical logic, it will be able to prove just about anything (since, in classical logic, anything follows from a contradiction). In classical logic, as Belnap puts it, a contradiction "pollutes" all the data. Relevance logic, on the contrary, does not validate $(A \& \neg A) \Rightarrow B$.

It seems therefore worth stressing that formal languages - and the logics that are used to evaluate reasonings expressed in those languages - are made to be *extended* or *amended*. When we want to modify a theorem-prover (be it to make it more powerful or to forbid certain inferences that we don't like), we have to *use our intuitions about validity*. Ideally, we should be able to use these intuitions to say:

"If I have this kind of thing and that kind of thing in my premisses, I want the system to be able to derive this." [This corresponds exactly to an elimination rule in natural deduction]

or

"I want to introduce a new connective, or operator, that should behave like this: ..."

If the structure of the theorem-prover consists of a set of *intuitive* rules of inference, the job will be easier.

As was mentioned above, the language of sentential logic may be extended into predicate logic with variables and quantifiers. But there are still a lot of modes of reasoning that are used in natural language and that predicate logic is unable to capture: these are expressed for instance with intensional or modal operators.

Using modalities, one may introduce in the language the concepts of necessity and possibility (with the addition of two operators respectively interpreted as "it is necessary that" and "it is possible that"). Such concepts allow the language to distinguish between contingent and necessary truths. Tense logics (that use operators such as "it will be the case that") also belong to the class of modal logics.

Applications of such logics to real-life computer science are obvious: reasoning over time is typical of systems whose task is to schedule or plan a sequence of operations. Certain sentences, such as, for instance, the integrity-constraints of a database (e.g., "No part *can* be delivered to a company more than 200 miles away from Montreal"), can be given, via modal operators, a particular logical status. The rest of the database would store sentences that just happen to be true, without being necessary (e.g, "All bolts used in part X are red").

A more ambitious - I think not so far-fetched - application of natural deduction is in the implementation of default-reasoning. In everyday thinking, we often infer things for which we do not have an absolutely certain proof. The classical example is Tweety's: knowing that Tweety is a bird, and nothing else about Tweety, we will infer that Tweety can fly, not because we believe that all birds can fly (we of course never believe false things!) but because we know that, if something is a bird, then we may assume that it can fly, unless we

have a good reason to think otherwise. Precisely, the reasoning involved is the following: assume that Tweety can fly and try to derive a contradiction. If you don't find one, then you may assert that Tweety can fly.

If it is known in the database that Tweety happens to be a penguin, and that penguins don't fly, then a contradiction will be derived from the hypothesis that Tweety can fly and the inference that Tweety can fly will be blocked.

Most cases of default-reasoning are auto-epistemic, i.e., refer implicitely to one's knowledge. We reason "auto-epistemically" when we say things like: "I would know it if X were false. Hence X must be true." Here again, an implicit reductio ad absurdum is attempted and failure to find a contradiction suffices to justify a new belief.

This kind of meta-reasoning is extremely powerful, even though its overall validity may be questioned. It could be used to answer queries expressed by sentences that are neither provable nor disprovable by means of standard natural deduction. It is tempting to say, that much default reasoning (which is essentially meta-theoretical, i.e., auto-epistemic) is achieved in people's minds via a failure of finding a contradiction in a RAA (reductio ad absurdum). The natural deduction technique is geared towards an efficient (goal-directed) usage of RAA in ordinary proofs. In the ordinary use of natural deduction, RAA is used only to refute an assumption by showing that it leads to a contradiction. In default-reasoning, people

use RAA for a completely different purpose (i.e., not to *refute* a belief, but to *justify* it by showing that it is consistent with the other beliefs). Natural deduction could, I think without too much work, be tuned so as to make a similar usage of its ability to apply RAA intelligently.

If we want a language with defaults, we may decide to modify the syntax so as to accept sentences (not just rules) of the form[5]

(b : Ma) / a

the intended interpretation being:

If b is true, then if it is consistent to assume a, then assert a.

The elimination rule would look like this:

```
m | b => Ma / a
n | b
o |   | a
  |   |
  |   |  .
  |   |  .
  |   |  .
p |   | No contradiction
  | a   (default elim, m,n,o,p)
```

---

[5] This notation is due to R.Reiter [18].

Theorem-*proving*, whether it is done by a computer or a human being, is a mechanical activity. It has to be formalized in syntactic terms. Natural deduction, like all theorem-proving techniques, must rely on syntactic rules of inference. But it is for semantic reasons that we use theorem-proving. The point of formal systems is to allow the expression of intuitions about what should and what should not be provable in the system. The advantage of intuitive formal systems such as natural deduction is that they make it easy to express such intuitions as unambiguous rules of inference.

APPENDIX 1: RULES USED IN THE FIRST VERSION OF THE THEOREM-PROVER:

[Try strategy 1 (backward reasoning).]

Rule 1: If goal is an implication, then assume antecedent and try to prove consequent. Apply =>I.

Rule 2: If goal is an equivalence statement, prove both corresponding implications and apply <=>I.

Rule 3: If goal is a conjunction, prove both conjuncts and apply &I.

Rule 4: If goal is a disjunction, and if one of the disjuncts is already a theorem, apply VI.

[Try strategy 2 (forward reasoning).]

Rule 5: If goal is a theorem, return the path to the goal.

Rule 6: If a disjunction is a theorem, apply VE, i.e., prove goal from each disjunct separately.

Rule 7: If an implication is a theorem, try to prove antecedent and apply =>E.

Rule 8: If the negation of a disjunction is a theorem, derive the negation of both disjuncts and return (proof goal).

[Try strategy 3 (RAA).]

Rule 9: If no other rule applicable, apply RAA, i.e., assume the negation of what you are trying to prove and try to derive a contradiction.

Rules to find contradictions

Rule 10: If goal is a disjunction, assume negation of goal, derive negation of both disjuncts and look for a contradiction.

Rule 11: When looking for a contradiction, try to prove the negation of a theorem.

Rule 12: Don't use RAA on a goal that you are already trying to prove with RAA.

Rule 13: If sentence is the antecedent of some theorem, apply =>E.

Rule 14: If sentence is atomic (contains no connectives), nothing else can be done with it.

Rule 15: If sentence is a conjunction, apply &E.

Rule 16: If sentence is an implication and some theorem is the antecedent of sentence, apply =>E.

Rule 17: If sentence is an equivalence statement, then if some theorem is the antecedent or the consequent of sentence, then apply <=>E.

Rule 18: If sentence is a double negation, apply ¬E.

APPENDIX 2: LISTING OF THE FIRST IMPLEMENTATION OF THE THEOREM-PROVER (IN FRANZ-LISP)

6.1. Program listing.

```
(def do (lambda (x y)
  (princ '|premisses: |)
  (princ x)
  (terpri)
  (princ '|conclusion: |)
  (princ y)
  (terpri) (terpri)
  (tabulate (prove x y))
  (terpri]


;      (I) FORMATTING FUNCTIONS
;
;print out proof in tabular form: the proof now consists of a
;list,the first element of which is the set of premisses or the
;hypothesis, and the last element the conclusion. Tabulate is a
;formatting routine that converts the proof in readable form.
;Each element of the proof has a level. Subproofs may be
;embedded.

(def tabulate (lambda (x)
    (init)
    (format x]

(defun init ()
   (setq line 1)
  (setq tabnum 1]

(def format
;Input: a proof or a subproof,stored as a list.
;output: a formatted proof.
 (lambda (result)
   (pprms (car result))
              ; print premisses - or, in the case of a
              ; subproof, the hypothesis.
   (underline)
   (printsteps (center result))          ;print proof
   (pconc result)))                        ;print conclusion
```

```
(defun hyp_ify (x)
   (mapcar 'hyp_thisprem x]

(defun hyp_thisprem (x)
   (cons (list x line) hypotheses]


     ;FUNCTIONS PRINTING PREMISSES AND HYPOTHESES
     ;
(def pprms
   (lambda (x)
    (cond
     ((wff x) (printprem x))
     (t (mapcar 'printprem x]

(defun printprem (x)
     (print_line)
     (tabs)
     (princ x)
     (justify 'HYP)
     (terpri)
     (setq line (add1 line]

(defun justify (x)
     (princ '|     |)
     (princ x]


       ;FUNCTIONS PRINTING INTERMEDIARY STEPS OF PROOF

(def center
;returns a list containing all elements of argument except
;first and last. In other words, returns the proof part of the
;argument.
   (lambda (x)
   (cond ((lessp (length x) 3) nil)
      ((equal (length (cdr x)) 2) (list (cadr x)))
      (t (append (list (cadr x)) (center (cdr x)))))))

(defun subproof (x)
  (and
    (listp x)
    (equal (length x) 1)
    (listp (car x))
    (not (wff (car x]   ;(car x) must have at least 2 elements

(defun subprint (x)
  (setq tabnum (add1 tabnum))
  (format (car x))
  (setq tabnum (sub1 tabnum]

(defun printsteps (x)
  (mapcar 'print_step x]
```

```
(defun print_step (x)
        ;a step in the proof is either a wff or a subproof,
        ; or a sequence of steps.
  (cond
    ((null x) nil)
    ((subproof x) (subprint x))
    ((wff x) (print_line)
        (tabs)
        (princ x)
        (setq line (add1 line))
        (terpri))
    (t (printsteps x]


     ;FUNCTION PRINTING CONCLUSION OF A PROOF OR SUBPROOF

(def pconc (lambda (x)
             ;prints conclusion with tabbing corresponding to
             ;the level of the subproof.
  (cond
   ((null (car (last x))) nil)
   (t
   (print_line)
   (tabs)
   (princ (car (last x)))
   (setq line (add1 line))
   (terpri]


      ;FUNCTIONS FOR PRINTING TABS AND UNDERLINING HYPOTHESES

(defun print_line ()
  (cond
    ((lessp line 10) (princ '| |)))
  (princ line]

(def tab (lambda nil (princ '|  |)))

(def tabs
             ;number of tabs printed must correspond to the
             ;depth of the step being printed.
   (lambda nil
   (prog (temp)
     (setq temp tabnum)
    loop (tab)
     (cond ((equal temp 1) (return t))
           (t (setq temp (sub1 temp)) (go loop))))))

(def underline (lambda nil
    (princ '| |)
    (tabs)
    (princ '----)
    (terpri]
```

```
; (II) FUNCTIONS TESTING FOR WELL-FORMEDNESS OF PREMISSES AND
; CONCLUSION

(def wff
      (lambda (x)
           (or (atomic_wff x)
                (compound_wff x)
                (negation x))))

(def atomic_wff
   (lambda (x)
   (member x '(a b c d p q r s x y z]

(def compound_wff
   (lambda (x)
   (and (listp x)
      (member (cadr x) '(& V => <=>))
      (wff (car x))
      (wff (caddr x)))))

(def negation
   (lambda (x)
           (and (listp x) (equal (car x) '¬) (wff (cadr x)))))


;    (III)  GIVEN A LIST OF PREMISSES X AND A CONCLUSION Y,
;     RETURN A PROOF.

  (def prove (lambda (x y)
     (init_lists x)
     (forward)
     (list x (proof y) y]


(defun init_lists (x)
   (setq newinfo nil)
   (setq premlist x)
   (setq disjunctions nil)
   (setq neg_disjs nil)
   (setq hyps nil)
   (setq forbidden nil)
   (setq attempted nil)
   (setq proof_hints nil)
   (setq discarded nil)
   (init_array)
   (setq goals nil)
   (setq last_goals nil]
```

```
(def proof
   (lambda (goal)
      (cond ((null goal) nil)
         ((member goal premlist) nil)
         (t (change goal)))))

(def change
            ;will return a rule to apply immediately to z,
            ;or a hint(about something that can be proved
            ;given the present theorems,or that would be
            ;useful if it were proved, given te goal.

   (lambda (z)
   (setq last_goals (cons z last_goals))
   (cond ((equal (find_main_conn z) '=>) (=>I z) )
      ((equal (find_main_conn z) '&) (&I z))
      ((equal (find_main_conn z) '<=>) (<=>I z))
      ((theorem z)(path (theorem z)))
      ((VI z))          ;if trying to prove (a V b), then if a
                        ;or b is a theorem, you are done.
      ((setq disjunction (car disjunctions))
        (VE disjunction z))
      ((hint1 z))
      ((hint2 z))
      ((not (member z forbidden)) (raa z]

(def find_main_conn
   (lambda (x)
   (cond ((atom x) nil)
      ((equal (car x) '¬) '¬)
      (t (cadr x)))))


;FUNCTIONS THAT DERIVE CONSEQUENCES OF PREMISSES AND HYPOTHESES


(defun init_array ()
  (array available t 30)
  (setq next_slot 1)
  (mapcar 'load_prem premlist]

(defun load_prem (x)
  (setq newinfo (cons next_slot newinfo))
  (insert (list x 'H]

(defun insert (x)
  (store (available next_slot) x)
  (setq next_slot (add1 next_slot]

(def forward
   (lambda ()
    (mapcar 'forwards (reverse newinfo]
```

```
(def forwards   ;x : a slot number
   (lambda (x)
   (setq sent (car (available x)))
   (cond ((member sent (antecedents)) (modus sent)))
   (cond ((atom sent) t)
      ((equal (cadr sent) 'V)
       (setq disjunctions (cons sent disjunctions)))
      ((equal (cadr sent) '&) (&E x))
      ((equal (cadr sent) '=>) (=>E x))
      ((equal (cadr sent) '<=>) (<=>E x))
      ((and (equal (car sent) '¬)
         (not (atomic_wff (cadr sent))))
       (¬E x)))))


(defun modus (thisprem) ;thisprem is in antinfo
            ;so, get all slot-numbers n such that
            ;  1. (theorem (car (available n)))
            ;  2. (caar (available n)) = (caar (available x))
            ;examine consequences of these.
   (mapcar 'store_consequent
         (elim_nil (mapcar 'find_compound (theorems]

(def find_compound
;x = (A n)
;if A = (z => y), then cadar x = =>, cadr x = n
   (lambda (x)
   (cond ((atom (car x)) nil)
      ((and (equal (cadar x) '=>) (equal (caar x) thisprem))
       (cadr x]

(def elim_nil
   (lambda (l)
   (cond ((null l) nil)
      ((null (car l)) (elim_nil (cdr l)))
      ((listp (car l)) (elim_nil (car l)))
      (t (cons (car l) (elim_nil (cdr l]


(def store_consequent  ;z: a slot-number
   (lambda (z)
   (insert (list (caddar (available z)) '=>E z (this_slot)))
   (forwards (this_slot))))


(defun this_slot ()
    (sub1 next_slot]
```

```
; HINTS functions
;
(defun hint1 (x)
;(hints x) checks whether a hint is applicable
;to present situation and, if so, takes appropriate action.
;HINT 1 = try to prove antecedents of available theorems.
;HINT 2 = if a theorem is the negation of a disjunction, derive
;negation of both disjuncts, (using RAA) and then go back to
present goal.

;(hints x) returns nil iff
;no hint is applicable
;and (list (proof z) z (proof x)), for some z
;s.t. 1. ((proof z) z) is in proofhints
;      2. z |- x

     (cond ((and (prove_antecedents) (theorem x))
                 (setq forbidden nil)
                  (setq attempted nil)
                (car proof_hints]


  (defun hint2 (x)
     (cond
          (neg_disjs
               (setq neg_disj
                     (car (available (car neg_disjs))))
           (setq neg_disjs (cdr neg_disjs))
          (list (raa (caadr neg_disj))
                (neg (caadr neg_disj))
                (raa (caddadr neg_disj))
                (neg (caddadr neg_disj))
                (proof x]

(defun prove_antecedents ()
               ;denies permission to use RAA
               ;and stores, for each antecedent
               ;proved, its proof in proof_hints
   (setq forbidden
         (elim_dupl (append (antecedents) forbidden)))
  (elim_nil (mapcar 'prove_ant (antecedents]

(defun elim_dupl (x)
   (cond
     ((null x) nil)
     ((member (car x) (cdr x)) (elim_dupl (cdr x)))
     (t (cons (car x) (elim_dupl (cdr x]
```

```
(defun prove_ant (x)
                ;x is a wff. If x is provable, store
              ;((proof x) x) into proof_hints
              ;else, nil.
   (cond
    ((member x attempted) nil)
    (t
     (setq attempted (cons x attempted))
     (cond
       ((setq thisproof (proof x))
                ;if x is provable
          (setq proof_hints
                (cons (list thisproof x) proof_hints))
         (insert (list x !proofant))
         (forwards (this_slot]

; AVAIL functions

  (defun theorems ()
;should return a list of pairs, e.g., ((a 1) (a 6)...)
    (prog (result)
      (setq result nil)
      (setq position 1)
     loop
      (cond
       ((member position discarded)
                (setq position (add1 position))
                (go loop))
        ((null (available position))
                (return (reverse result)))
        (t
                (setq result
                     (cons (list
                                  (car (available position))
                                position)
                           result))
           (setq position (add1 position))
         (go loop]

(defun theorem (x)
;should return the slot-number corresponding to x
;in AVAILABLE, nil otherwise. x is a wff
   (prog (th)
     (setq th (theorems))
  loop (cond ((null th) nil)
        ((equal x (caar th)) (return (cadar th)))
        (t (setq th (cdr th))
         (go loop]
```

```
(defun avail ()
   (prog (result)
    (setq result nil)
    (setq position 1)
   loop
    (cond
     ((null (available position)) (return (reverse result)))
     (t (setq result (cons (available position) result))
       (setq position (add1 position))
       (go loop]


(defun implications ()
   (mapcar 'implication (theorems)]

(defun implication (x)
   (cond
    ((atomic_wff (car x)) nil)
    ((equal (cadar x) '=>) (cadr x]


  (defun antecedents ()
     (mapcar '(lambda (x)
           (cond
             ((null x) nil)
             (t (caar (available x)))))
         (implications]

(defun consequents ()
   (mapcar '(lambda (x) (caddar (available x))) (implications]


(defun path (x)  ;x should be a slot-number in available
  (setq sentence (available x))
  (mapcar 'get_step (reverse (list_steps (cddr sentence]

(defun get_step (x)
  (car (available x]

(defun list_steps (x)
  (elim_nil (mapcar 'origin x]

(defun origin (x)
  (cond
    ((equal (cadr (available x)) 'H) nil)
    ((equal (cadr (available x)) 'proofant) x)
    (t (cons x (list_steps (cddr (available x]
```

```
;RAA functions


  (def raa
     (lambda (x)
     (prog (A B)
        (setq A nil)
        (setq B nil)
        (newhyp (neg x))
        (cond
         ((disj x)
         (setq A
            (list (raa (neg (car x)))
               (neg (car x))
               (raa (neg (caddr x)))
               (neg (caddr x))))
         (setq B (list (neg (car x)) (neg (caddr x))))
         (insert (list (neg (car x)) 'H))
         (insert (list (neg (caddr x)) 'H))))
        (cond ((setq c (contra (theorems)))
            (return
            (list
             (list (neg x)
                (append A (prove_contra c))
                (discard (car hyps))
             ))))
           (t (setq forbidden (cons x forbidden))
             (cond
             ((setq c (find_contra (append B premlist)))
              (setq forbidden (cdr forbidden))
              (return
              (list
               (list (neg x)
                  (append A (prove_contra c))
                  (discard (car hyps]


(defun disj (x)
  (cond
   ((atom x) nil)
   ((equal (cadr x) 'V) t]


  (def find_contra
     (lambda (x)
     (cond ((null x) nil)
        ((and (not (member (neg (car x)) forbidden))
           (proof (neg (car x))))
         (theorem (car x)))
        (t (find_contra (cdr x))))))
```

```
(defun neg (x)
   (cond
    ((atomic_wff x) (list '¬ x))
    ((equal (car x) '¬) (cadr x))
    (t (list '¬ x]


(def contra
                ;returns slot-number of a theorem the negation
                ;of which is also a theorem.
   (lambda (x)
   (cond ((null x) nil)
      ((theorem (neg (caar x))) (cadar x))
      (t (contra (cdr x))))))

(def prove_contra
                ;should return,for some x
                ; (list (proof x) x (proof (neg x)) (neg x))
                ; So, look if there is a contra available in
                ;theorems. If so, produce it; else, take each
                ;theorem in turn, and try to prove its negation,
                ;without being allowed to use RAA immediately.

    (lambda (x)
     (setq s (car (available x)))
     (list (proof s) s (proof (neg s)) (neg s))))


   ;FUNCTIONS USED TO INTRODUCE AND DISCARD HYPOTHESES

(def newhyp
     (lambda (x)
      (setq hyps (cons next_slot hyps))
      (load_prem x)
      (forwards (this_slot))
      (setq premlist (cons x premlist))
      nil]

(defun discard (x)
        ;x is a slot-number in avail
        ;discarded is a list of discarded slots in AVAILABLE.
   (setq premlist (cdr premlist))
   (setq hyps (cdr hyps))
   (setq discarded (append discarded (consequences x))) nil)

(defun consequences (x)
     (prog (result index)
        (setq index x)
        (setq result nil)
  loop (cond
            ((equal index next_slot) (return result))
            (t (setq result (cons index result))
               (setq index (add1 index))
               (go loop]
```

```
;INTRO FUNCTIONS

    (def =>I
         (lambda (x)
            (setq goals (cons (list x '=>I) goals))
               (newhyp (car x))
               (list (list (car x)
                           (proof (caddr x))
                           (discard (car hyps))
                           (caddr x]

(def <=>I (lambda (x)
     (setq goals (cons (list x '<=>I) goals))
     (list (=>I x) (=>I (reverse x)))))

(def &I (lambda (x)
     (setq goals (cons (list x '&I) goals))
     (list  (proof (car x))
              (car x)
              (proof (caddr x))
              (caddr x))))

(def ¬I (lambda (x)
     (setq goals (cons (list x '¬I) goals))
     (list (list (cadr x) (prove_contra)))))

(defun VI (x)
         (cond
            ((atomic_wff x) nil)
            ((equal (cadr x) 'V)
                 (cond
                   ((theorem (car x))
                            (list (path (theorem (car x)))
                                  (car x)))
                   ((theorem (caddr x))
                            (list (path (theorem (caddr x)))
                                  (caddr x]

;ELIM FUNCTIONS

(defun =>E (x)
    (cond
       ((theorem (caar (available x)))
                (insert (list (caddar (available x))
                              '=>E
                              x
                              (theorem (caar (available x)))))
                (forwards (this_slot]
```

```
(def <=>E
     (lambda (x)
      (cond
         ((theorem (caar (available x)))
             (insert
              (list (caddar (available x))
                    '<=>E
                    x
                    (theorem (caar (available x)))))
             (forwards (this_slot)))
        ((theorem (caddar (available x)))
             (insert
              (list (caar (available x))
                    '<=>E
                    x
                    (theorem (caddar (available x)))))
             (forwards (this_slot))))))

(def &E
     (lambda (x)
      (insert (list (caar (available x)) '&E x))
      (forwards (this_slot))
      (insert (list (caddar (available x)) '&E x))
      (forwards (this_slot]

(def ¬E
     (lambda (x)
      (cond
          ((equal (cadar (available x)) '¬)
           (insert (list (cddar (available x)) '¬E x))
           (forwards (this_slot)))
          ((equal (car (cdadar (available x))) 'V)
           (setq neg_disjs (cons x neg_disjs]

(def VE
     (lambda (d x)
      (setq disjunctions (cdr disjunctions))
      (list (path (theorem d))
            d
            (list
             (list (dis1 d)
                   (newhyp (dis1 d))
                   (proof x)
                   (discard (car hyps))
                   x))
            (list
             (list (dis2 d)
                   (newhyp (dis2 d))
                   (proof x)
                   (discard (car hyps))
                   x)))))
```

```
(defun dis1 (x) (car x))
(defun dis2 (x) (caddr x))

    ;TESTS

(setq p nil)

(setq p1 '(a))
(setq c1 'a)

(setq p2 '(a))
(setq c2 '(a V b))

(setq p3 '(a (a => b)))
(setq c3 '(a => (b => a)))

(setq p4 '(b (b => a)))
(setq c4 '(a & b))

(setq p5 '((a => b) a ((a & b) => c)))
(setq c5 '(b => c))

(setq p6 '(((a V b) => c) b]
(setq c6 'c)

(setq p7 nil)
(setq c7 '(a => (b => (c => (d => a]

(setq p8 '((a V b) (a => c) (b => c]
(setq c8 'c)

(setq p9 '((a => b]
(setq c9 '((¬ b) => (¬ a]

(setq p10 '((a => b]
(setq c10 '(¬ (a & (¬ b]

(setq p11 '((a & b) (b => c]
(setq c11 '(c V d]

(setq p12 '((a V b) (c => (¬ a)) c]
(setq c12 'b)

(setq p13 '((a V b) (¬ a]
(setq c13 'b)

(setq p14 '((p => q) (r V (¬ q)) (¬ r]
(setq c14 '(¬ p]

(setq p15 '((p => q) (q => (r V s]
(setq c15 '(p => (r V s]
```

```
(setq p16 '((p V (q => r)) (q V (¬ p)) (¬ q]
(setq c16 '(q => r]

(setq p17 '((r => q) (q => p) (p => s) (¬ s]
(setq c17 '(¬ r]

(setq p18 '((p => q) (q => r) (p V (¬ s)) (¬ r]
(setq c18 '(¬ s]

(setq c20 '((p => q) => ((¬ q) => (¬ p]

(setq c21 '((p => (q & (¬ q))) => (¬ p]

(setq c22 '((p V (¬ p)) V (r & s]
(setq c23 '(p V (¬ p]
```

6.2.
Tests of the program

6.2.1.
Command file

```
(load 'project)
(do p c3)
(do p4 c4)
(do p5 c5)
(do p7 c7)
(do p9 c9)
(do p10 c10)
(do p12 c12)
(do p14 c14)
(do p16 c16)
(do p17 c17)
(do p18 c18)
```

6.2.2.
Output of test

```
Franz Lisp, Opus 36
-> t
-> premisses: nil
conclusion: (a => (b => a))

        |----
 1      |   |a          HYP
        |   |----
 2      |   |   |b          HYP
        |   |   |----
 3      |   |   |a
 4      |   |(b => a)
 5      |(a => (b => a))
nil
```

```
->
nil
-> premisses: (b (b => a))
conclusion: (a & b)

 1     |b                HYP
 2     |(b => a)                HYP
       |----
 3     |a
 4     |b
 5     |(a & b)
nil
->
nil
```

```
-> premisses: ((a => b) a ((a & b) => c))
conclusion: (b => c)

 1     |(a => b)                HYP
 2     |a                HYP
 3     |((a & b) => c)                HYP
       |----
 4     |     |b                HYP
       |     |----
 5     |     |a
 6     |     |b
 7     |     |(a & b)
 8     |     |c
 9     |(b => c)

nil
->
nil
```

```
-> premisses: nil
conclusion: (a => (b => (c => (d => a))))

       |----
 1     |     |a                HYP
       |     |----
 2     |     |     |b                HYP
       |     |     |----
 3     |     |     |     |c                HYP
       |     |     |     |----
 4     |     |     |     |     |d                HYP
       |     |     |     |     |----
 5     |     |     |     |     |a
 6     |     |     |     |(d => a)
 7     |     |     |(c => (d => a))
 8     |     |(b => (c => (d => a)))
 9     |(a => (b => (c => (d => a))))

nil
```

```
->
nil
```

```
-> premisses: ((a => b))
conclusion: ((¬ b) => (¬ a))

 1    |(a => b)                    HYP
      |----
 2    |    |(¬ b)                  HYP
      |    |----
 3    |    |   |a                  HYP
      |    |   |----
 4    |    |   |(¬ b)
 5    |    |   |b
 6    |    |(¬ a)
 7    |((¬ b) => (¬ a))

nil
->
nil
```

```
-> premisses: ((a => b))
conclusion: (¬ (a & (¬ b)))

 1    |(a => b)                    HYP
      |----
 2    |    |(a & (¬ b))                HYP
      |    |----
 3    |    |a
 4    |    |b
 5    |    |(¬ b)
 6    |(¬ (a & (¬ b)))

nil
->
nil
```

```
-> premisses: ((a V b) (c => (¬ a)) c)
conclusion: b

 1    |(a V b)                    HYP
 2    |(c => (¬ a))                    HYP
 3    |c                  HYP
      |----
 4    |(a V b)
 5    |    |a                   HYP
      |    |----
 6    |    |    |(¬ b)                 HYP
      |    |    |----
 7    |    |    |(¬ a)
 8    |    |    |a
 9    |    |b
10    |    |b                   HYP
      |    |----
11    |    |b
12    |b

nil
->
nil

-> premisses: ((p => q) (r V (¬ q)) (¬ r))
conclusion: (¬ p)

 1    |(p => q)                    HYP
 2    |(r V (¬ q))                    HYP
 3    |(¬ r)                  HYP
      |----
 4    |(r V (¬ q))
 5    |    |r                   HYP
      |    |----
 6    |    |    |p                   HYP
      |    |    |----
 7    |    |    |(¬ r)
 8    |    |    |r
 9    |    |(¬ p)
10    |    |(¬ q)                 HYP
      |    |----
11    |    |    |p                   HYP
      |    |    |----
12    |    |    |(¬ q)
13    |    |    |q
14    |    |(¬ p)
15    |(¬ p)

nil
->
nil
```

```
-> premisses: ((p V (q => r)) (q V (¬ p)) (¬ q))
conclusion: (q => r)

 1  |(p V (q => r))                    HYP
 2  |(q V (¬ p))                  HYP
 3  |(¬ q)                  HYP
    |----
 4  |   |q                   HYP
    |   |----
 5  |   |(q V (¬ p))
 6  |   |   |q                   HYP
    |   |   |----
 7  |   |   |(p V (q => r))
 8  |   |   |   |p                        HYP
    |   |   |   |----
 9  |   |   |   |   |(¬ r)                      HYP
    |   |   |   |   |----
10  |   |   |   |   |(¬ q)
11  |   |   |   |   |q
12  |   |   |   |r
13  |   |   |   |(q => r)                 HYP
    |   |   |   |----
14  |   |   |   |r
15  |   |   |r
16  |   |   |(¬ p)                    HYP
    |   |   |----
17  |   |   |   |(¬ r)                    HYP
    |   |   |   |----
18  |   |   |   |(¬ q)
19  |   |   |   |q
20  |   |   |r
21  |   |r
22  |(q => r)

nil
->
nil
-> premisses: ((r => q) (q => p) (p => s) (¬ s))
conclusion: (¬ r)

 1  |(r => q)                    HYP
 2  |(q => p)                    HYP
 3  |(p => s)                    HYP
 4  |(¬ s)                  HYP
    |----
 5  |   |r                   HYP
    |   |----
 6  |   |(¬ s)
 7  |   |q
 8  |   |p
 9  |   |s
10  |(¬ r)

nil
```

```
->
nil

   -> premisses: ((p => q) (q => r) (p V (¬ s)) (¬ r))
   conclusion: (¬ s)

    1   |(p => q)              HYP
    2   |(q => r)              HYP
    3   |(p V (¬ s))                HYP
    4   |(¬ r)              HYP
        |----
    5   |(p V (¬ s))
    6   |   |p                HYP
        |   |---
    7   |   |   |s                HYP
        |   |   |----
    8   |   |   |(¬ r)
    9   |   |   |q
   10   |   |   |r
   11   |   |(¬ s)
   12   |   |(¬ s)              HYP
        |   |----
   13   |   |(¬ s)
   14   |(¬ s)
   nil
```

APPENDIX 3:
RULES USED IN THE IQ-LISP VERSION OF THE THEOREM-PROVER:

The vocabulary of rules: primitive predicates and functions

A number of primitive functions and predicates have been defined and used in rules. Beyond a number of syntactic predicates and functions (that check whether a sentence is a conjunction, a well-formed formula, etc...), there are a number of functions for:

. examining a goal to decide whether it is likely to be provable, before actually trying to prove it (**attempt**[1])

. deciding when a goal ought to be proved by RAA or by a more direct method

. selecting the contradiction that is most likely to be provable in a reductio ad absurdum

. deciding what to print on the screen and where to print it

- goals: **printgoal**

- paths leading to formulas: **printpath**

---

[1] Function and predicate names are in bold while data structures are in capital letters

- subproofs (vertical lines, hypotheses, subgoals)

. checking what is available and identifying justifications

Theorem: checks whether something is available

Justified: the PROOF structure keeps track of whatever has been proved so far

Column, Above, above+: verify that a sentence really belongs to the PROOF and that it is repeatable

Origin: find from what line a sentence is derived

Repetable: verify that a sentence satisfies the restrictions on reiteration

. adding facts to the database

Inserta, insert, ...: maintain AVAILABLE (truth-maintenance system)

Inserthyp: Hypotheses are to be inserted into the structure AVAILABLE, even if this is only temporary

. removing hypotheses and their consequences

A crucial feature of natural deduction is the possibility to make hypotheses. And hypotheses, at some point,

have to be discarded. Whenever a hypothesis is discarded, it is attached, as well as all of its consequences (i.e., all sentences in AVAILABLE following it) to the list DISCARDED.

Something is a theorem iff it is in AVAILABLE, but not in discarded. (For programming convenience, the function (theorem sentence) does not return a boolean value but rather a number indicating the position of sentence in AVAILABLE.

Discard: discarding a hypothesis

Delete: Procedure for discarding hypothesis

Validation of student's proposal

Validhyp: Validate hypothesis

Validbut: Verify that a goal is a legitimate one

Rules used in forward reasoning (application of elimination rules of inference)

These rules are consulted whenever forward reasoning is necessary, i.e., in the beginning of the proof construction over the set of premisses; and then whenever a new sentence is derived or when a hypothesis is made.

The variable FOCUS is set to the formula that is being currently considered for forward reasoning. By successive

application of relevant elimination rules, the consequences of FOCUS are derived.

If a conjunction is under focus,

insert its first conjunct in AVAILABLE

If a conjunction is under focus,

insert its second conjunct in AVAILABLE

If an implication is under focus

check if antecedent is available and if so, add consequent in AVAILABLE, with the justification of E=>; add focus to the list IMPLICATIONS

If an equivalence is under focus and its antecedent is a theorem,

insert its consequent in AVAILABLE

If an equivalence is under focus and its consequent is a theorem,

insert its antecedent in AVAILABLE

If a disjunction is under focus, and the negation of its first disjunct is a theorem,

insert the second disjunct in AVAILABLE

If a disjunction is under focus, and the negation of its second disjunct is a theorem,

insert the first disjunct in AVAILABLE

If a disjunction is under focus

add it to the list DISJONCTIONS

If the formula under focus is the antecedent of some implications in the list IMPLICATIONS

add the appropriate consequents to AVAILABLE

If the formula under focus is the antecedent of some equivalences in EQUIV

add the appropriate consequents to AVAILABLE

If the formula under focus is the consequent of an equivalence in EQUIV

add the appropriate antecedents to AVAILABLE

If the formula under focus is the negation of the first disjunct of some disjunctions in DISJONCTIONS

add the corresponding second disjuncts to AVAILABLE

If the formula under focus is the negation of the second disjunct of a disjunction in DISJONCTIONS

add the corresponding first disjuncts to AVAILABLE

Rules used in backward (goal-directed) reasoning

If goal already has a justification

Write this justification and suppress this goal from GOAL_LIST.

If goal is an implication

Create appropriate subproof, write justification for goal and add subproof to list of goals.

If goal is repeatable

Justify goal with rule REP and suppress the goal from GOAL_LIST.

If a contradiction is available

print justification for goal (appropriate RAA rule, i.e., ¬Intro or ¬Elim) and add the appropriate subproof to the list of goals (i.e., prove contradiction under the negation of the goal).

If goal is a conjunction

Print appropriate subgoals above the goal (i.e., each conjunct that is not already printed; justify the goal with I&; and add appropriate subgoals to GOAL_LIST (one or two disjuncts).

If goal is an equivalence

print both appropriate implications above goal; add them to GOAL_LLST; and justify goal with IEquiv

If goal is a theorem,

print the path that leads to it, as well as the goal, fetching the appropriate justifications from the truth-maintenance monitor.

If goal is a disjunction and one of disjuncts is already proved

Insert goal in AVAILABLE; justify goal with IV

Construct INTS = list of newly available formulas (via elimination rules)
and INT = list of formulas in INTS that have the goal as a sub-formula

If nothing is in INT

add (RAAGOAL) to GOAL_LIST

If goal is the consequent of an implication in INT

justify the goal with E-implic and add the antecedent of this implication to GOAL_LIST.

If goal is a conjunct in (a conjunction that is) the consequent of an implication in INT

Prove this implication; then try to prove the antecedent of this implication (so as to get the conjunction by E-implic and the goal by E&).

If goal is the first disjunct of a newly available disjunction

print the proof of this disjunction, and then try to prove the goal by EV, that is by adding to GOAL_LIST the negation of the second disjunct.

If goal is the second disjunct of a newly available disjunction

prove it by EV, adding the negation of the first disjunct to GOAL_LIST.

If goal is the consequent of an equivalence in INT

add to GOAL_LIST the antecedent of this equivalence and use E-equiv to justify the goal.

If goal is the antecedent of an equivalence in INT

add to GOAL_LIST the consequent of this equivalence and use E-equiv to justify the goal.

If goal is the second conjunct in the consequent of an equivalence in INT

add to GOAL_LIST the antecedent of this equivalence; (it will then be possible to use E-equiv to get the conjunction that will lead to the goal.

If goal is the first conjunct in the consequent of an equivalence in INT

add to GOAL_LIST the consequent of this equivalence; (it will then be possible to use E-equiv to get the conjunction that will lead to the goal.

If goal is a disjunction and one of its disjuncts is a theorem

Insert the goal in AVAILABLE, print the proof to the provable disjunct; justify the goal with IV.

If a disjunction is available as a theorem

prove goal by RAA

If the goal is a disjunction and one of its disjuncts is provable (via ATTEMPT)

add this disjunct to GOAL_LIST, intending to prove the goal by the rule IV.

If the negation of the goal is the antecedent of an available implication

Prove the goal by RAA

If the goal is the consequent of an available implication

Try to prove it by E=>: add to GOAL_LIST the antecedent of this implication

If there is in available an implication of which the antecedent has not yet been proved

Try to prove this antecedent so as to get the consequent

If the negation of the goal is not a theorem

try to prove the goal by RAA.

If there is an available equivalence such that

- the negation of its antecedent is a theorem

- the negation of its consequent is not already a goal or a theorem.

Add to GOAL_LIST the negation of the consequent

If there is an available equivalence such that

- the negation of its consequent is a theorem

- the negation of its antecedent is not already a goal or a theorem

Add to GOAL_LIST the negation of the antecedent.

If the goal is a disjunction and the negation of the goal is a theorem

Add to GOAL_LIST the negation of the disjunction's first disjunct.

Otherwise,

Give up with the proof; print failure message

Rules used when the goal is to prove a contradiction

If a contradiction has been found, but not yet been printed in a sub-proof by reductio

Print the contradiction; justify the goal with the appropriate RAA rule (I¬ or E¬)

If the first disjunct of an available disjunction is the negation of a theorem

Add the negation of the first disjunct to GOAL_LIST

If the negation of a theorem is provable via attempt

add this negation to GOAL_LIST

If the negation of a disjunction is a theorem and the first disjunct of this disjunction is not already a goal

Add the negation of this disjunct to GOAL_LIST

If the negation of a disjunction is a theorem and the negation of the first disjunct of this disjunction is a theorem but the negation of the second disjunct of the disjunction is not a goal

Add the negation of the second disjunct to GOAL_LIST

Construct CANDIS, containing all available implications such that the negation of their consequent is a theorem, while the negation of their antecedent is not a theorem.

If an implication is a member of CANDIS and the negation of its consequent is not just a theorem but already proved above the goal, select that one; otherwise, select the first element of CANDIS

and add to GOAL_LIST the negation of the antecedent of this implication

If it is possible to select the first likely candidate for reductio via GETCANDIS.

Add this candidate on top of GOAL_LIST

If an equivalence is available

add the negation of the antecedent to GOAL_LIST

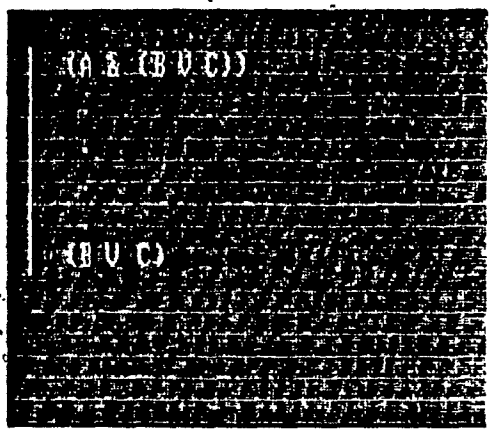
Otherwise

give up and print failure message

APPENDIX 4: SCREEN DUMPS TAKEN FROM THE STEL TUTORIAL.

EXERCICE N° 29 —(facile)— Règle d'ELIMINATION DE LA CONJONCTION.

Schéma à compléter



Interprétation de ce schéma

A, et B ou C

∴

DONC,
B ou C

Faire cet exercice | Changer d' exercice | Explorer les concepts

1	(B ⇒ A)	PREM	
2	((¬ B) ⇒ (¬ C))	PREM	
3	C	PREM	
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19	A	???	
20	(A & C)	I& 3,19	

On veut prouver
A à la ligne 19 .

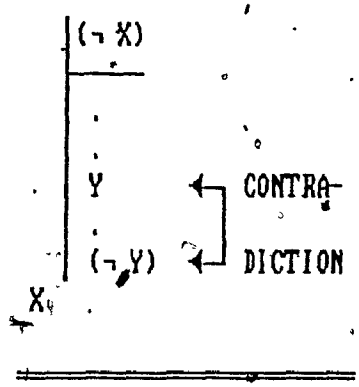
On peut noter que A
est le CONSEQUENT de
la prémisse 1 ..

↵ = INDICE SUPPLEMENTAIRE

Esc = RETOUR AU MENU PRECEDENT

1	(B ⇒ A)	PREM
2	((¬ B) ⇒ (¬ C))	PREM
3	C	PREM
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18	B	???
19	A	E⇒ 1,18
20	(A & C)	I& 3,19

PREUVE PAR L'ABSURDE
ELIMINATION DE LA NEGATION



1	(A ⇒ B)	PREM	
2	(¬ ((¬ A) ∨ B))	HYP	On veut prouver une contradiction sous l'hypothèse (¬ ((¬ A) ∨ B)) .
3	(¬ A)	HYP	
4			
5			
6	(¬ ((¬ A) ∨ B))	REP 2	
7	((¬ A) ∨ B)	IV 3	On peut prouver ((¬ A) ∨ B) et sa NEGATION.
8			
9	A	E¬ 3,6,7	
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20	((¬ A) ∨ B)	E¬?	

Désirez-vous voir l'étape suivante? [O/N]

1	(A ⇒ B)	PREM	
2	¬ ((¬ A) ∨ B))	HYP	On veut prouver une DISJONCTION à la ligne 18.
3	(¬ A)	HYP	
4			
5			
6	(¬ ((¬ A) ∨ B))	REP 2	Dans le cas présent, on peut prouver un des deux DISJOINTS.
7	((¬ A) ∨ B)	IV 3	
8			
9	A	E¬ 3,6,7	
10			On utilisera ensuite la règle d'INTRODUCTION de la DISJONCTION (IV).
11			
12			
13			
14			
15			
16			
17			
18	((¬ A) ∨ B)	???	
19			
20	((¬ A) ∨ B)	E¬?	

↵ = INDICE SUPPLEMENTAIRE

Esc = RETOUR AU MENU PRECEDENT

```
1  | A                          PREM
2  | (C & (¬ B))                PREM
3  |    | B                     HYP
4  |    |
5  |    |
6  |    |
7  |    |
8  |    |
9  |    |
10 |    |
11 |    |
12 |    |
13 |    |
14 |    |
15 |    |
16 |    |
17 |    |
18 |    |
19 |    | <<CONTRADICTION>>
20 | (B ⇒ D)                    ???
```

Non: dans ce cas-ci, il ne faut pas chercher à prouver que l'hypothèse B est absurde.

On veut prouver que (B ⇒ D)
Il faut donc prouver D sous l'hypothèse B

Appuyer sur Esc pour annuler cette hypothèse

1	A	PREM	
2	(C & (¬ B))	PREM	Bonne stratégie
3	B	HYP	(règle I⇒)
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19	D		
20	(B ⇒ D)	I⇒ 3,19	

Appuyer sur ↵ pour continuer

1	A	PREM	On veut prouver
2	(C & (¬ B))	PREM	D à la ligne 19 .
3	B	HYP	
4			Il y a une CONTRADICTION
5			(explicite ou cachée)
6			dans ce qui précède.
7			
8			On peut donc utiliser
9			une preuve par L'ABSURDE
10			
11			
12			
13			
14			
15			
16			
17			
18			
19	D	???	
20	(B ⇒ D)	I⇒ 3,19	

↵ = INDICE SUPPLEMENTAIRE

Esc = RETOUR AU MENU PRECEDENT

1	A	PREM	
2	(C & (¬ B))	PREM	Il n'y a pas de raison
3	B	HYP	apparente de faire
4	(¬ (B ⇒ D))	HYP	l'hypothèse (¬ (B ⇒ D))
5			à ce stade-ci de
6			la preuve.
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18	<<CONTRADICTION>>		
19	D	???	
20	(B ⇒ D)	I⇒ 3,19	

Appuyer sur Esc pour annuler cette hypothèse

LES CONCEPTS DE LA DEDUCTION NATURELLE

Les différents types de formules

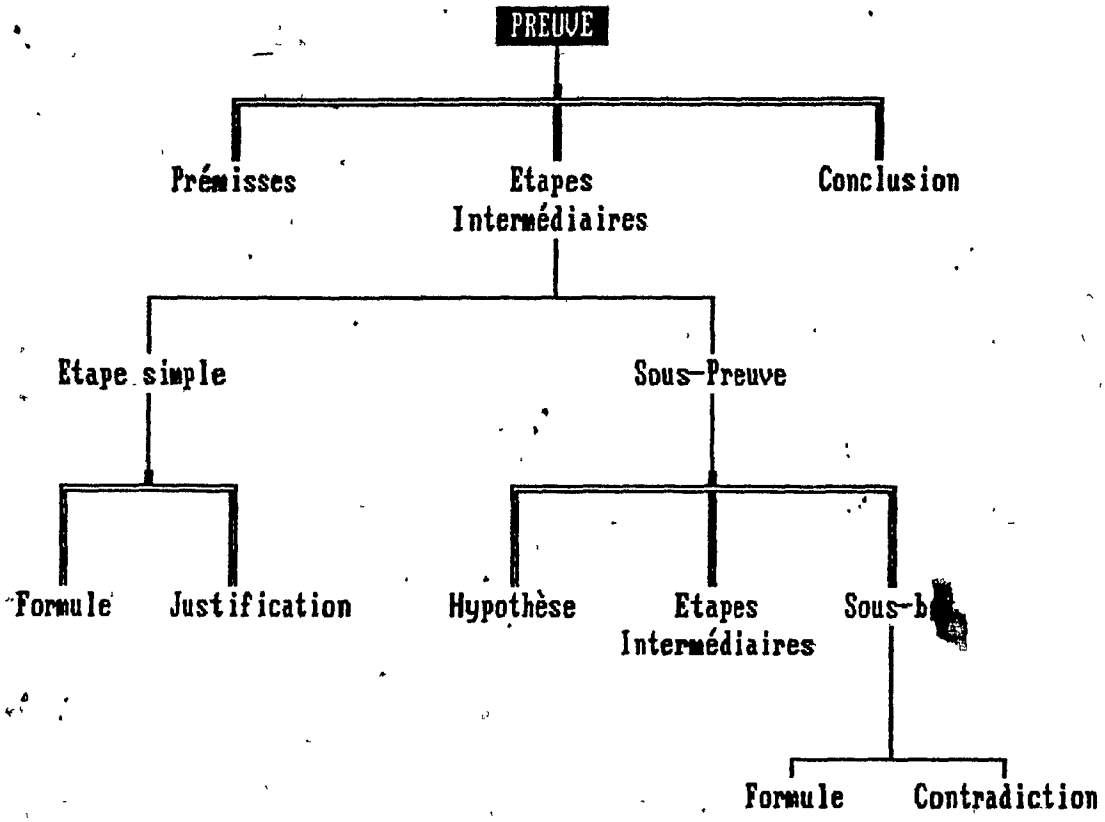
Nom des parties d'une formule composée

Structure d'une preuve

11 règles d'inférence

Stratégie: Comment construire une preuve

Esc = Retour aux exercices



Ésc = Table des matières

Étapes intermédiaires

Les étapes intermédiaires d'une preuve sont des étapes simples ou des sous-preuves .

1.	$P \Rightarrow Q$	Prém.
2.	$(\neg Q)$	Prém.
3.	$\vert\ P$	Hyp.
4.	$\vert\ P \Rightarrow Q$	Rép. 1
5.	$\vert\ Q$	E⇒ 3,4
6.	$\vert\ (\neg Q)$	Rép. 2
7.	$(\neg P)$	I¬ 3,5,6

La sous-preuve qui occupe les lignes 3 à 6 est une étape intermédiaire .

A l'intérieur de cette sous-preuve, les lignes 4 et 5 sont aussi des étapes intermédiaires .

Détails: ↓

BIBLIOGRAPHY:

[1] Anderson, A. and Belnap, N., *Entailment*, Pittsburg University Press, 1972.

[2] Barr, A. and Feigenbaum, E. A., eds., *The Handbook of Artificial Intelligence*, Vols 1,2,3, William Kaufmann, Los Altos, Calif., 1981.

[3] Bergmann, M., Moor, J. and Nelson, J., *The Logic Book*, New York: Random House, 1980.

[4] Bledsoe, W. W. and Loveland, D. W., *Automated Theorem Proving: After 25 Years*, American Mathematical Society Press, 1984.

[5] Bledsoe, W. W., Non-resolution Theorem Proving, in *Artificial Intelligence* 9, 1977, pp.1-35.

[6] Bledsoe, W. W., Splitting and reduction heuristics in automatic theorem proving, in *Artificial Intelligence*, 2, 1971, pp. 55-77.

[7] Boyer, R. S. and Moore, J. S., *A Computational Logic*, Academic Press, New York, 1979.

[8] Chang, C. and Lee, R. C., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[9] Fitch, F. B., *Symbolic Logic: An Introduction*, Ronald, New-York, 1952.

[10] Gentzen, G., "Untersuchungen uber das logische Schliessen", *Mathematische Zeitschrift*, 39 (1934-1935), 176-210, 405-431.

[11] Jaskowski, S., "On the Rules of Supposition in Formal Logic", *Studia Logica*, 1 (1934), pp.5-32.

[12] G. Kearsley, ed, *Artificial Intelligence & Instruction: Applications and Methods*, Addison-Wesley, 1987.

[13] Leblanc, H. and Wisdom, W., *Deductive Logic*, Boston: Allyn & Bacon, 1972.

[14] Loveland, D. W., *Automatic Theorem Proving: A Logical Basis*, North-Holland, 1977.

[15] Manna, Z. and Waldinger, R., *The Logical Basis for Computer Programming*, Volume 1, Addison-Wesley, 1985.

[16] Paulson, L. C., Lessons learned from LCF: A Survey of Natural Deduction Proofs, in *The Computer Journal*, vol 28, no 5, 1985, pp. 474-479.

[17] Paulson, C. L., Natural deduction as higher-order resolution, in *The Journal of Logic Programming*, 1986, no 3, pp. 237-258.

[18] Reiter, R., "A logic for default reasoning", *Artificial Intelligence* 13 (1980), 81-132.

[19] Rich, C., and Waters, R. C., eds, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, Los Altos, California, 1986.

[20] Robinson, J.A., *Logic: Form and Function*, Edinburgh U.P. 1979.

[21] Schank, R., and Abelson, R., *Scripts, Plans, Goals, and Understanding*, Erlbaum, Hillsdale, NJ, 1977.

[22] Sleeman, D. and Brown, J. S., eds, *Intelligent Tutoring Systems*, Academic Press, New York, 1982.

[23] Suppes, P., ed, *University-level computer-assisted instruction at Stanford: 1968-1980*, Stanford University, 1981.

[24] Tennant, N., *Natural Logic*, Edinburgh U.P., 1978.

[25] Weyrauch, R., Prolegomena to a mechanized theory of formal reasoning, in *Artificial Intelligence*, 13, 1980, pp. 133-170.

[26] Xuhua, L. and Zhan, C., A natural deduction theorem proving system -- An implementation for elementary number theory, in *Proceedings of the International Symposium on New Directions in Computing*, IEEE, 1985, pp.66-71.