Tutch User's Guide

Version 0.5 alpha

Andreas Abel

1 Introduction

Nomen est omen: *Tutch* is an abbreviation of **Tut**orial proof **ch**ecker. Tutch is a checker for the validity of formal proofs in constructive logic. It is still under development, but it is aimed towards handling first-order predicate logic with a few built-in inductive data types and recursive functions over these data types. The strength of the system will be Heyting Arithmetic with inductive data types ($HA\hat{}\backslash omega$).

Tutch is meant to be a tool to assist teaching of logic. It should help students to learn how to formally prove propositions. We believe this is easier if they are allowed to write down the proof like the code of a program and let it be checked afterwards in a compiler-like tool, instead of trying to assemble it with "hands tied" in an interactive system, where they have to search for the appropriate tactics and apply them in the right order.

We try to develop user-friendly error messages. Any feedback on cryptic messages or suggestion of better messages is highly valuable for us. Please sent feedback to `abel@cs.cmu.edu`.

Tutch is **not**:

1. A professional tool for verification
2. An automated theorem prover
3. A logical framework
4. A point-and-click program

About this document

This document is meant as a user's guide. It's neither a reference, where Tutch input is defined formally and all options are listed in alphabetical order, nor a "First Steps Tutch Tour for Dummies", where every keystroke is explained. Instead it tries to be a quick and reasonable introduction along examples, which I believe communicate the basic ideas most quickly and intuitively.

Have fun!

2 How to Run Tutch

The executable files `tutch`, `submit` and `status` you need for the course are installed in `/afs/andrew/scs/cs/15-399/bin`. You have several possibilities to run them:

2.1 Explicitely Specification of Path

You can just type in the complete path every time you run `tutch`, `submit` or `status`. E.g.

```
$ /afs/andrew/scs/cs/15-399/bin/tutch ...
```

Since this is tedious, we recommend one of the following more sophisticated methods.

2.2 Setting Your Global Path

Under Unix, every time a shell is launched a startup script is executed. Usually this script is named '`.`*shellname*`rc`' and is located in your home directory, e.g., '`~/.cshrc`' for the C SHell '`csh`'. To add the Tutch program location to your path, add the line

```
setenv PATH "${PATH}:/afs/andrew/scs/cs/15-399/bin"
```

to '`~/.cshrc`'. If you are using the Bourne Again SHell '`bash`', then add the following line to your script '`~/.bashrc`'.

```
export PATH="${PATH}:/afs/andrew/scs/cs/15-399/bin"
```

To determine whether you run '`csh`' or '`bash`', try `echo $BASH` or `echo $SHELL`. Only one of these should work and give you the shell you are running.

The changes in the startup scripts take effect whenever you open a new shell. To make the new path available immediately within your current shell, run `source` on the startup script, e.g.,

```
$ source ~/.cshrc
```

Sometimes the shell startup script has been created by your system administrator and looks complicated, possibly involving the execution of other scripts during startup. Usually the file then gives you instructions how to modify your path setting.

2.3 Creating Intermediate Scripts

If you do not want to modify your global path and already have a directory in your path where you store your own executables, just add three files into this directory (usually named '`~/bin`'): `tutch`, `submit` and `status`

```
#!/bin/csh
# file: tutch
/afs/andrew/scs/cs/15-399/bin/tutch $*
#EOF
```

```
#!/bin/csh
# file: submit
/afs/andrew/scs/cs/15-399/bin/submit $*
#EOF
```

```
#!/bin/csh
# file: status
/afs/andrew/scs/cs/15-399/bin/status $*
#EOF
```

Then make them executable and add them to the known executables with the following commands (assuming you are in the right directory):

```
$ chmod +x tutch submit status
$ rehash
```

Two more possibilities are aliases and links.

3 Proofs in Propositional Logic

We explain how to code natural deduction style proofs in Tutch, how to check validity of the coded proof and how to submit proofs as homework solutions.

3.1 Propositional Logic I

The proof rules for conjunction & and implication $=>$ in natural deduction are:

AndI	If $Atrue$ and $Btrue$ then $A\&Btrue$
AndEL	If $A\&Btrue$ then $Atrue$
AndER	If $A\&Btrue$ then $Btrue$
ImpI	If $Atrue\| - Btrue$ then $A = >Btrue$
ImpE	If $A = >Btrue$ and $Atrue$ then $Btrue$

The following Tutch file '`prop0.tut`' is a proof of the proposition $A\&(A = >B) = >B$:

```
proof mp: A & (A=>B) => B =
begin
[ A & (A=>B);
  A;
  A=>B;
  B ];
A & (A=>B) => B
end;
```

A proof begins with the keyword '`proof`', followed by a identifier (here '`mp`'), a colon '`:`', then the goal proposition (here '`A & (A=>B) => B`'), an equal symbol '`=`' and then the proof, enclosed between '`begin`' and '`end`'. The last proposition of the proof must match the goal.

Each entry of the proof is either a single assertion (like the last line) or a *frame*, enclosed between brackets '`[`' and '`]`'. The first proposition of a frame (here '`A & (A=>B)`') is a hypothesis that can be used to prove the entries of the frame, but not outside the frame. The entries of a frame can be assertions, separated by semicolons '`;`', or frames again. All entries of a frame are usable only within that frame.

Each entry of a proof must be derivable from previous entries by a single inference step, i.e., by a single proof rule application. In our case '`A`' and '`A=>B`' are immediately derivable from the hypothesis '`A & (A=>B)`' by &-Elimination. '`B`' is the result of eliminating '`A=>B`' using '`A`'.

A frame '`[A; ... C]`' itself stands for the hypothetical judgment "$Atrue| - Ctrue$". Frames can be used in proofs which discharge a hypothesis, like the ImpI-rule. In our example the frame is used to prove the final line '`A & (A=>B) => B`'.

Scoping: Entries within a frame can only be used within that frame. Once a frame is closed, it can be used as a whole as a hypothetical judgment, but its entries are no longer "visible".

To check this small proof, run `tutch prop0`. The extension '`.tut`' is added automatically. The output should look like this:

```
$ tutch prop0
TUTCH Version ...
[Opening file prop0.tut]
Proving mp: A & (A => B) => B ...
QED
[Closing file prop0.tut]
$
```

As in most programming languages, we can supply *comments* for better understandability or informal explanations. There are two kinds of comments:

1. Multi-line comments are enclosed between '`%{`' and '`}%`'.
2. Everything after '`%`' until the end of the line forms a single line comment.

The following file '`prop1.tut`' contains the above proof decorated with justifications and a file header, giving name and informal content.

```
%{ prop1.tut

   Modus ponens.
}%

proof mp: A & (A=>B) => B =
begin
[ A & (A=>B);         % 1 Assumption
  A;                  % 2 by AndE1 1
  A=>B;               % 3 by AndE2 1
  B ];                % 4 by ImpE 3 2
A & (A=>B) => B       % 5 by ImpI 4
end;
```

3.2 Propositional Logic II

Logical equivalence $A< = >B$ is just defined as implication in both directions, made into a conjunction: $A = >B\&B = >A$. Thus, to prove an assertion '`A <=> B`' we have to prove '`A => B`' and '`B => A`' in any order. The goal '`A <=> B`' then follows by AndI. Example: Idempotency of the conjunction.

```
% prop2.tut
% Idempotency of conjunction

proof idemAnd : A & A <=> A =
begin
[ A & A;
  A ];
A & A => A;
A => A & A;
A & A <=> A
end;
```

The file '`prop2.tut`' contains our incomplete proof. Before we finish it, we want to check correctness of this fragment. We run `tutch -v prop2`:

```
$ tutch -v prop2
TUTCH Version ...
[Opening file prop2.tut]
Proving idemAnd: A & A <=> A ...
  1  [ A & A;
  2    A ];                                          by AndE1 1
  3  A & A => A;                                     by ImpI 2
prop2.tut:9.1-9.11 Error:
Unjustified line (A & A |- A); (A & A) => A  |-  A => (A & A)
Assuming this line, checking remainder...
  4  A => A & A;                                     UNJUSTIFIED
  5  A & A <=> A                                     by AndI 3 4
Proof incomplete
[Closing file prop2.tut]
```

Tutch verifies the first three lines and finds line 4 to be unjustified. It prints out all visible judgments, then a turnstile '`|-`' and then the unproven assertion '`A => (A & A)`'. In our case the available judgments are

$A\&A| - A$	This hypothetical judgment is the result of the frame '`[A & A; A]`' (lines 1-2).
$| - (A\&A) = >A$	This non-hypothetical judgment, result of line 3, is printed with out the turnstile. All parenthesis are made explicit.

Now we can continue and add the missing proof of '`A => A & A`'. We leave this as a simple exercise to the reader.

3.3 Propositional Logic III

The remaining constructs of intuitionistic propositional logic are disjunction $|$, truth T and falsehood F. These are their proof rules:

OrI1	If $Atrue$ then $A|Btrue$
OrI2	If $Btrue$ then $A|Btrue$
OrE	If $A|Btrue$, $Atrue| - Ctrue$ and $Btrue| - Ctrue$ then $Ctrue$
TrueI	$Ttrue$
FalseE	If $Ftrue$ then $Ctrue$ for any proposition C

We treat negation as an abbreviation: '`~A`' stands for '`A => F`'.

In the following we present an example making use of '`|`'-elimination. We prove '`~A|B => A=>B`'.

(*An aside on the meaning of this proposition:* In classical logic the implication '`A => B`' can be defined as '`~A | B`'. In intuitionistic logic the disjunction '`~A | B`' is stronger than the implication '`A => B`'. This means '`(A=>B) => ~A|B`' is *not* provable, only the direction given here:)

```
% prop2.tut
% Classical implication definition  ~A|B => A=>B

proof classImpDef : ~A|B => A=>B =
begin
[ ~A|B;
  [ A;
    [ ~A;
      F;
      B ];
    [ B;
      B ];
    B ];
  A=>B ];
~A|B => A=>B
end;
```

We assume '`~A|B`' and '`A`' and have to show '`B`'. Using OrE on the first hypothesis leaves us to show the validity of the hypothetical judgements '`~A |- B`' and '`B |- B`' in our context. While the second is trivial, we proof the first by contradiction after assuming '`~A`' (recall that '`~A`' stands for '`A=>F`').

We obtain the justifications for each proof entry by running `tutch -v prop2` (**v**erbose output):

```
$ tutch -v prop2
TUTCH Version ...
[Opening file prop2.tut]
Proving impDef: ~A | B => A => B ...
  1  [ ~A | B;
  2    [ A;
  3      [ ~A;
  4        F;                    by ImpE 3 2
  5        B ];                  by FalseE 4
  6      [ B;
  7        B ];                  by Hyp 6
  8      B ];                    by OrE 1 5 7
  9    A => B ];                 by ImpI 8
 10  ~A | B => A => B            by ImpI 9
QED
[Closing file prop2.tut]
$
```

Precedence and associativity of the connectives: '`~`' binds strongest, then come '`&`', '`|`' and '`=>`' (weakest). All of these three infix operators are right-associative. Thus '`A => (B => C)`' and '`A => B => C`' denote the same proposition, but '`(A => B) => C`' a different. In the same way from '`A & B & C`' we can infer '`A`' in one step, but neither '`B`' nor '`C`', since we have to prove '`B & C`' first.

3.4 Requirements and Submission

How to complete your homework in five steps:

1. Getting the requirements
2. Completing the proofs
3. Checking against the requirements
4. Submitting
5. Checking status of submission

Let us assume you have to complete assignment '`prop`'.

3.4.1 Getting the requirements

The requirement files for the homework assignments are stored under `/afs/andrew/scs/cs/15-399/req`. First copy the requirements file to obtain a template proof file:

```
$ cp /afs/andrew/scs/cs/15-399/req/prop.req prop.tut
```

3.4.2 Completing the proofs

Edit file '`prop.tut`' in your favored editor (we recommend `xemacs`), fill in the proofs and run Tutch whenever you want to check your accomplishments. You can even check incomplete proofs, as long as the syntax is correct and there are no open frames.

3.4.3 Checking against the requirements

To check whether you fulfilled the requirements run Tutch with option '`-r`'. After this option you specify the requirements file. If your proof file has the same name as the requirements file (except extension), you can save some keystrokes. You will see an output like this:

```
$ tutch -r prop.req prop.tut
TUTCH Version ...
[Opening requirements file /afs/andrew/scs/cs/15-399/req/prop.req]
[Closing requirements file /afs/andrew/scs/cs/15-399/req/prop.req]
[Opening file prop.tut]
Proving mp: A & (A => B) => B ...
QED
Proving impDef: ~A | B => A => B ...
QED
[Closing file prop.tut]

Checking requirements...
[X] proof mp: A & (A => B) => B
[X] proof impDef: ~A | B => A => B
Congratulations! All problems solved!
$
```

A short version of that command is '`tutch -r prop`'. You can also distribute your proofs over several files. To check the two files '`prop1.tut`' and '`prop2.tut`' against '`prop.req`', type

```
$ tutch -r prop prop1 prop2
```

3.4.4 Submitting

You do not get credits for your homework unless you *submit*. You must be registered student of course and have access to the Andrew File System (afs). To submit you type:

```
$ submit -r prop
SUBMIT Version ...
...
[Submitting files: prop.tut ]
[Submission OK]
$
```

This submits file '`prop.tut`' for homework '`prop.req`'. To submit several files, e.g., type

```
$ submit -r prop prop1 prop2
```

3.4.5 Checking status of submission

With the '`status`' command you can check the status of your submission, e.g.,

```
$ status prop
STATUS Version ...
Getting status of submission prop...
Submitted files: prop.tut
Assignment: prop
Student ID: foo
Date:       today

[X] proof mp: A & (A => B) => B
[X] proof impDef: ~A | B => A => B
$
```

4 Proof Terms for Propositional Logic

We support two new methods how to give proofs: *Annotated proofs* and proof *terms*. Annotated proofs are Tutch proofs in which each line is annotated by a proof term. Since proof terms record all information necessary to reconstruct a proof, they alone (without a deduction tree) are sufficient as well. For the syntax see the section **Proof Terms** in the Appendix A [Reference], page 25. Here are two examples, annotating proofs from the last chapter.

```
% prop0-ann.tut
% Modus ponens

annotated proof mp: A & (A=>B) => B =
begin
[ x : A & (A=>B);
  fst x : A;
  snd x : A=>B;
  (snd x) (fst x) : B ];
fn x => (snd x) (fst x) : A & (A=>B) => B
end;


% prop3-ann.tut
% Classical implication definition  ~A|B => A=>B

annotated proof classImpDef : ~A|B => A=>B =
begin
[ x : ~A|B;
  [ a : A;
    [ na : ~A;
      na a : F;
      abort (na a) : B ];
    [ b : B;
      b : B ];
    case x of inl na => abort (na a) | inr b => b end : B ];
  fn a => case x of inl na => abort (na a) | inr b => b end : A=>B ];
fn x => fn a => case x of inl na => abort (na a) | inr b => b end : ~A|B => A=>B█
end;

term classImpDef : ~A|B => A=>B =
  fn x => fn a => case x of inl na => abort (na a) | inr b => b end;
```

5 Types and Programs

Using the Curry-Howard-Isomorphism, we reuse proof terms to write *programs* and we introduce isomorphic constructs to propositions as the *types* of our programs: product '`*`', disjoint sum '`+`', function space '`->`', unit type '`1`' and empty type '`0`'. Note that there are no isomorphic constructs to negation and equivalence.

Furthermore we introduce the three inductive types '`nat`' (Natural numbers), '`bool`' (Booleans) and '`list`' (Polymorphic lists). The constructors and destructors are

`0, s` *t* `nat`: Zero and successor

`rec` *r* `of` *f* `0 =>` *s* `|` *f* `(s` *x*`) =>` *t* `end`
 `nat`: Primitive recursion

`true, false`
 `bool`: The two truth values

`if` *r* `then` *s* `else` *t*
 `bool`: Case distinction

`nil,` *s* `::` *t*
 tau `list`: Empty list and cons.

`rec` *r* `of` *f* `nil =>` *s* `|` *f* `(`*x*`::`*xs*`) =>` *t* `end`
 tau `list`: Primitive recursion

Note that '`0`' can both mean the number zero and the empty type. '`1 + 1`' may look like an arithmetical expression, but denotes a sum type here. Be careful not to mistake '`1`' for a number and to use it in your programs. Tutch will parse it as a type and give you a strange error message.

The two terms for primitive recursion define locally a function f and describe its behaviour for all possible cases of input by the step terms s and t. The use of f within the step terms s and t underlies strong syntactic restrictions that guarantee termination of the function on all inputs: In s no recursive calls are allowed at all. In t, for `nat` a recursive call can *only* look like $f\,x$ (or $f(x)$), where x is the variable defined in the pattern matching *f* `(s` *x*`) => ...`. For `list`, the recursive call must be $f\,xs$ (or with parentheses as above).

Here is one example for primitive recursion over `nat`:

```
val double : nat -> nat
  = fn x => rec x of
        d 0 => 0
      | d (s x') => s (s (d x'))
    end;
```

The keyword 'val' indicates the definition of a program of a given type. Another example is subtraction on nat:

```
val minus : nat -> nat -> nat
  = fn x => rec x of
        m 0 => fn y => 0
      | m (s x') => fn y => rec y of
            p 0 => s x'
          | p (s y') => m x' y'
        end
    end;
```

The types of the locally defined functions are `m : nat -> (nat -> nat)` and `p : nat -> nat`. To be able to apply two arguments to `m` we have used the trick to move the second lambda abstraction `fn y =>` into the outer recursion. Thus `m`, applied to `0` or `s x'`, returns a function from `nat` to `nat`, to which we can apply the second argument.

The inner recursion is just a case distinction, since `p` does not occur on a right hand side.

```
val rev : tau list -> tau list -> tau list
  = fn l => rec l of
        r nil => fn a => a
      | r (x::l') => fn a => r l' (x :: a)
    end;

val reverse : tau list -> tau list
  = fn l => rev l nil;
```

This is an implementation of the reverse function by an auxiliary function with an accumulator argument. We use the same trick as for `minus`.

For more documentation look up the reference.

6 First-Order Logic

We extend our propositions and our linear proof format to include universal and existential quantification. Here is an example demonstrating All-introduction and -elimination and Exists-introduction:

```
proof AllEx : !y:t. (!x:t. A(x)) => ?x:t. A(x) =
begin
    [ c : t;
      [ !x:t. A(x);
        A(c);
        ?x:t. A(x) ];
      (!x:t. A(x)) => ?x:t. A(x)];
    !y:t. (!x:t. A(x)) => ?x:t. A(x)
end;
```

The scope of a quantification starts at the '`.`' and extends as far to the right as syntactically possible. Thus the All-quantified variable '`y`' is bound in the whole proposition, whereas the scope of '`!x:t.`' is limited by parentheses to preserve the intended meaning of the proposition. Invoking the proof checker Tutch, it gives the following justifications

```
Proving AllEx: !y:t. (!x:t. A x) => ?x:t. A x ...
  1  [ c: t;
  2    [ !x:t. A x;
  3      A c;                                     by ForallE 2 1
  4      ?x:t. A x ];                             by ExistsI 1 3
  5    (!x:t. A x) => ?x:t. A x ];                by ImpI 4
  6  !y:t. (!x:t. A x) => ?x:t. A x               by ForallI 5
QED
```

Universal quantification `!y:t. B(y)` can be introduced by a frame `[c:t; ... B(c)]` as we see in the last line: Here `B(y)` is

```
(!x:t. A x) => ?x:t. A x
```

which does not contain any occurrence of `y`. Universal quantification can be eliminated if we have a term of the type over which the quantification is ranging. An example can be seen in line 3: We have an All-quantified assertion `!x:t. A x` in line 2 and a term `c : t`, the parameter introduced in line 1. Thus we can deduce `A x` where all occurrences of `x` are replaced by `c`, which is `A c`.

To introduce an existential quantification `?x:t. A x` (line 4) we need a witness `c : t` (line 1) and a proof for the special instance of the proposition `A c` (line 3). For existential elimination we consider the following example:

```
proof ExNotImpNotAll : (?x:t. ~A(x)) => ~!x:t. A(x) =
begin
[ ?x:t. ~A(x);
  [ !x:t. A(x);
    [ c: t, ~A(c);
      A(c);
      F ];
    F ];
  ~!x:t. A(x) ];
(?x:t. ~A(x)) => ~!x:t. A(x);
end;
```

To prove falsehood F in proof line 6, we eliminate the existential quantification in the first line. This gives us two new hypotheses to show our goal: A witness `c : t` and a proof of `~A(c)`. In principle Exists-elimination is used in the same way as disjunction elimination. Note that we extended our frame syntax to include the introduction of several hypotheses, separated by commas.

For further information consult the reference.

7 Arithmetic

Arithmetic is the first-order theory of natural numbers. This means that to prove properties of propositions and functions over natural numbers, we reason in first-order logic plus induction and the rules for equality on natural numbers. Furthermore we have a relation "less than" on natural numbers.

To prove an assertion `A(x)` for an arbitrary `x:nat`, we can make use of the fact that `x` is a natural number and eliminate it with rule *natE*, what is commonly called *induction*. This only works if we have a proof of `A(0)` and – under the hypotheses `x':nat` and `A(x')` – a proof of `A(s x')`. Example:

```
proof reflEq : !x:nat. (x = x) =
begin
[ x: nat;
  0 = 0;
  [ x': nat, x' = x';
    s x' = s x' ];
  x = x ];
!x:nat. x = x;
end;
```

We use parentheses around the '`x = x`' in the declaration to make clear that the '`=`' does not mean the end of the proposition and the start of the proof block. If we left them out, we would get an error message:

Category mismatch: x is a variable, but a proposition is expected in this place

Details about this ambiguity in the syntax you find in the reference. We are safe if we always put parentheses around equations in all declarations. (Within the proof this is not required!)

Tutch reconstructs the justifications as follows:

```
Proving symEq: !x:nat. x = x ...
  1  [ x: nat;
  2    0 = 0;                  by =NI0
  3    [ x': nat, x' = x';
  4      s x' = s x' ];        by =NIS 3
  5    x = x ];                by NatE 1 2 4
  6  !x:nat. x = x             by ForallI 5
QED
```

Here we clearly see the use of induction in tutch. To prove our goal `!x:nat. x = x` we assume an arbitrary `x: nat` and prove `x = x` for this arbitrary `x`. First we prove the case $x = 0$, which is done in line 2. Then we prove the case $x = sx'$ (line 4), where we assume that the proposition is proven for `x’` (line 3). Using these two subproofs, we can apply induction on `x: nat` and thus prove `x = x` for that arbitrary `x`.

As second example we will define the predecessor function for natural numbers and prove it correct.

```
val pred : nat -> nat = fn x  => rec x
      of f(0) => 0
       | f(s(x’)) => x’
    end;
```

We prove that the successor of the predecessor of a *positive* number x is equal to x. The annotated proof reads like this:

```
Proving verifyPred: !x:nat. ~x = 0 => s (pred x) = x ...
  1  [ x: nat;
  2    [ ~0 = 0;
  3      0 = 0;                                  by =NI0
  4      F;                                      by ImpE 2 3
  5      s (pred 0) = 0 ];                       by FalseE 4
  6    ~0 = 0 => s (pred 0) = 0;                 by ImpI 5
  7    [ x’: nat, ~x’ = 0 => s (pred x’) = x’;
  8      [ ~s x’ = 0;
  9        !z:nat. z = z;                        by Lemma reflEq
 10        s x’ : nat;
 11        s (pred (s x’)) = s x’ ];             by ForallE 9 10
 12      ~s x’ = 0 => s (pred (s x’)) = s x’ ];  by ImpI 11
 13    ~x = 0 => s (pred x) = x ];               by NatE 1 6 12
 14  !x:nat. ~x = 0 => s (pred x) = x            by ForallI 13
QED
```

Again we prove by induction over `x:nat`. At line 6 we have completed the proof for the base case $x = 0$. For the step case $x = sx'$ (lines 7–12) we use the reflexivity lemma that we have proven before. Since we know that `s (pred (s x’))` evaluates to `s x’` by definition of `pred`, we instantiate our lemma with `s x’`. To do so, we need to explicitely have the judgment `s x’ : nat` available. Since it is not a hypothesis, we state it in line 10. No proof is required here, the type-checker ensures that we can only give valid judgments of this form.

7.1 Proof terms

We illustrate how to prove propositions by induction with proof terms by these examples:

```
term refl : !x:nat. (x = x) =
  fn x => rec x of
      f 0 => eq0
    | f (s x') => eqS (f x')
  end;
```

A proof by induction is isomorphic to a primitive recursive function. In this case the corresponding function takes a natural number `x` and returns a proof the `x` equals `x`. In the base case, `eq0` is the proof for `0=0`. In the step case by induction hypothesis `f x'` is a proof of `x' = x'`. If we then apply `eqS` we get the desired proof of `s x' = s x'`.

As a second example we proof transitivity of equality on natural numbers, which shows the proper use of the elimination rules for equality.

```
term trans : !x:nat. !y:nat. !z:nat. (x = y) => (y = z) => (x = z) =
  fn x => rec x of
      f 0 => fn y => rec y of
          g 0 => fn z => fn p => fn q => q
| g (s y') => fn z => fn p => fn q => eqE0S p
      end
    | f (s x') =>  fn y => rec y of
          g 0 => fn z => fn p => fn q => eqES0 p
| g (s y') => fn z => rec z of
              h 0 => fn p => fn q => eqES0 q
    | h (s z') => fn p => fn q => eqS (f x' y' z' (eqESS p) (eqESS q))
          end
      end
  end;
```

UNDER CONSTRUCTION

8 Structural Induction

Similar to reasoning about natural numbers is reasoning about lists. We introduce structural induction and an equality over lists. This equality is denoted by the same symbol '=' as the one for natural numbers. The context makes it clear which of the two relations is meant.

The use of list-induction is analogous to nat-induction. This example should make it clear:

```
proof symList : !l:t list. !k:t list. (l = k) => (k = l) =
begin
[ l:t list;
  [ k: t list;
    [ nil = nil;
      nil = nil ];
    (nil = nil) => (nil = nil);
    [ y: t, ys: t list, nil = ys => ys = nil;
      [ nil = y :: ys;
        y :: ys = nil ];
      nil = y :: ys => y :: ys = nil;
    nil = k => k = nil ];
  !k:t list. nil = k => k = nil;
  [ x: t, xs: t list, !k:t list. xs = k => k = xs;
    [ k: t list.
      [ nil = x::xs;
        x::xs = nil ];
      nil = x::xs => x::xs = nil;

      l = x::xs => x::xs = l ];
    !k:t list. l = x::xs => x::xs = l ];
  !k:t list. l = k => k = l ];
!l:t list. !k:t list. l = k => k = l
end;
```

8.1 Proof terms

UNDER CONSTRUCTION!

We prove properties of these three functions:

```
val app : t list -> t list -> t list =
  fn l => rec l of
      f nil => fn l' => l'
    | f (x :: xs) => fn l' => x :: f xs l'
  end;

val rev : t list -> t list -> t list =
  fn l => rec l of
      f nil => fn k => k
    | f (x::xs) => fn k => f xs (x :: k)
  end;

val reverse : t list -> t list =
  fn l => rev l nil;
```

Here are the proofs:

```
term appnil : !l:t list. app l nil = l =
  fn l => rec l of
      f nil => eqN
    | f (x :: xs) => eqC (f xs)
  end;

term refll : !l:t list. (l = l); % Homework !

term apprev : !l:t list. !k:t list. !m:t list.
    app (rev l k) m = rev l (app k m) =
  fn l => rec l of
      f nil => fn k => fn m => refll (app k m)
    | f (x::xs) => fn k => fn m => f xs (x :: k) m
  end;

term revapp : !l:t list. !k:t list. !m:t list.
    rev (app l k) m = rev k (rev l m) =
  fn l => rec l of
      f nil => fn k => fn m => refll (rev k m)
    | f (x :: xs) => fn k => fn m => f xs k (x :: m)
  end;
```

Appendix A Reference

This part is meant as a brief, but complete reference to Tutch.

A.1 Command Line Syntax

The proof checker *Tutch* is invoked by

```
$ /afs/andrew/scs/cs/15-399/bin/tutch [options] [files]
```

The files are extended by '`.tut`' if they do not have an extension already. Options are:

`-h, --help`
: Print help message.

`-q, --quiet`
: Be quiet, print only version, file access and error messages.

`-Q, --Quiet`
: Be really quiet, print only error messages.

`-r, --requirements file`
: Check Tutch files against requirements in `file` resp. `file.req` (this extension is added if no extension is given) and print out check list. The path '`/afs/andrew/scs/cs/15-399/req`' for `file` is assumed unless `file` starts with a '`.`' or '`/`'. If no files are given, `file.tut` is assumed as input file.

`-v, --verbose`
: Be verbose, print justification for each proof line.

`-V, --Verbose`
: Print every available message.

Assignments submission is possible via

```
$ /afs/andrew/scs/cs/15-399/bin/submit -r file [options] [files]
```

Options are the same as for `tutch`, but `-r` is mandatory. The status of a submission can be checked via

```
$ /afs/andrew/scs/cs/15-399/bin/status file
```

This retrieves the status of the submission `file` handed in via `submit -r file ....`

A.2 Tutch File Syntax

Tutch recognizes a set of *special symbols.* They do not have to be separated by spaces from the remaining code, but serve as separators themselves. These symbols are

```
( ) [ ] ; : = ~ & | => <=>
```

Furthermore there are *reserved words* which cannot be used as identifiers:

```
T F proof begin end
```

Identifiers are made up of letters '`a-zA-Z`', digits '`0-9`' underscores '`_`' and primes '`'`'.

Atoms Q are identifiers that start with capital letter. *Propositions* A, B have the following grammar

```
A, B ::= T              % Truth
       | F              % Falsehood
       | Q              % Atom
       | ~A             % Negation
       | A & B          % Conjunction
       | A | B          % Disjunction
       | A => B         % Implication
       | A <=> B        % Equivalence
       | (A)            % Parentheses
```

The binary operators '`&`', '`|`' and '`=>`' are right associative, '`<=>`' is non-associative. Binding strength decreases in this order:

```
~ & | => <=>
```

A *hypothesis* H is just a proposition. A *proof entry* E is either a line or a frame. A *proof* P is a non-empty list of entries.

```
H      ::= A            % Hypothesis
```

```
E      ::= A                % Line
         | [ H; P ]         % Frame
P      ::= E                % Final step
         | E; P             % Step and remaining proof
```

A *proof name* x is a non-capital identifier. A *Tutch file* F is a sequence of proof declarations. A *proof declaration* D has the following syntax:

```
D      ::= proof x: A = begin P end     % Proof of A with name x
F      ::=                              % Empty file
         | D; F                         % Declaration and remaining file
```

A.2.1 Proof Terms

We extend our list of special symbols and reserved words by the following:

```
, annotated term fst snd inl inr case of fn abort
```

Proof terms for propositional logic are formed according to the following grammar:

```
M, N ::= x                % Variable
       | (M, N)           % Pair
       | fst M            % First projection
       | snd M            % Second projection
       | inl M            % Left injection
       | inr M            % Right injection
       | case M of inl x => N | inr y => O end % Case analysis
       | fn x => M        % Abstraction
       | M N              % Application
       | ()               % Empty tuple (proof of truth)
       | abort M          % Falsehood elimination
       | M : A            % Annotation
```

Application is a "invisible" left-associative infix operator. It has maximal binding strength, along with the prefix operators '`fst`', '`snd`', '`inl`', '`inr`', '`abort`'. This enforces use of parentheses in most cases. E.g.,

```
(fst x) ((snd x) y)
```

has already a minimum amount of parentheses. Abstraction '`fn x =>`' binds less than application and annotation '`:`' least. The following term is syntactically correct.

```
fn y => fn x => x y : A => (A => B) => B
```

Annotated proofs P are proofs as defined above annotated with proof terms. This changes the syntax of hypotheses H and proof entries E.

H	`::=` x `:` A	% Hypothesis
E	`::=` M `:` A	% Line
	`|` `[` H`;` P `]`	% Frame
P	`::=` E	% Final step
	`|` E`;` P	% Step and remaining proof

A *Tutch file* F now can contain proof, *term* and *annotated proof* declarations D:

D	`::=` `proof ...`	
	`|` `annotated proof` x`:` A `= begin` P `end`	% Annotated proof of A with name x
	`|` `term` x`:` A `=` M	% Proof of A with name x
F	`::=`	% Empty file
	`|` D`;` F	% Declaration and remaining file

A.2.2 Types and Programs

New special symbols and keywords are:

```
* + -> :: list 0 s rec true false if then else nil
```

Types T, T' have the following grammar:

T, T'	`::=` `1`	% Unit type
	`|` `0`	% Empty type
	`|` a	% Atom
	`|` `nat`	% Natural numbers
	`|` `bool`	% Booleans
	`|` T `list`	% Lists of element type T
	`|` T `*` T'	% Product
	`|` T `+` T'	% Disjoint sum
	`|` T `->` T'	% Function space
	`|` `(`T`)`	% Parentheses

'`list`' is a postfix operator.

The binary operators '`*`', '`+`' and '`=>`' are right associative. Binding strength decreases in this order:

```
list * + ->
```

We extend the grammar for *terms* by the following constructs:

```
M, N ::= ...
       | 0                    % Zero
       | s M                  % Successor
       | rec M of f 0 => N | f (s x) => O end       % Recursion over nat
       | true                 % True
       | false                % False
       | if M then N else O   % Boolean case distinction

       | nil                  % Empty list
       | M :: N               % List construction
       | rec M of f nil => N | f (x :: xs) => O end % Recursion over list
```

'`0`', '`true`', '`false`' and '`nil`' are constants, '`s`' and '`if` *r* `then` *s* `else`' are prefix operators and '`::`' is an infix operator with lower precedence than the prefix operators or application.

We add one new declaration to the syntax of tutch files:

```
D    ::=  ...
       | val x: T = M                 % Program of type T with name x
F    ::=                              % Empty file
       | D; F                         % Declaration and remaining file
```

A.2.3 First-Order Logic

Reasoning in First-Order Logic (FOL) requires handling of universally and existentially quantified propositions. New special symbols are

```
! ? .
```

We extend our definition of propositions by

```
A, B ::= ...
        | R M1...Mn        % Instantiation
        | !x:T. A          % Universal quantification
        | ?x:T. A          % Existential quantification
```

Relation symbols R are capital identifiers. Only they can be instantiated by terms Mi, e.g. '`A(x)`', which is the same as '`A x`'. Instantiation binds strongest, as strong as application and the prefix operators for terms M.

Quantification '`!x:T`' resp. '`?x:T. A`' is treated as a prefix operator with minimal precedence (like lambda abstraction). Ordered by binding strength, the operators that appear in propositions are:

```
!x:T. resp. ?x:T., <=>, =>, |, &, ~, instantiation
```

The '`proof`' declaration supports now also proofs in FOL. Two judgments can form a statement of the proof: an assertion A (representing '`A true`' and a term declaration $M : T$ expressing "M has type T". In the same way there are now two forms of hypotheses: $x : T$, which introduces a new parameter x into the proof, and A which assumes that A is true. Furthermore one frame can introduce several hypotheses, separated by commas. The grammar for proofs is the following:

```
H    ::= A                 % Hypothesis introduction
       | x : T             % Parameter introduction
Hs   ::= H                 % Last hypothesis
       | H, Hs             % Several hypotheses
E    ::= A                 % Line: Assertion
       | M : T             % Line: Term declaration
       | [ Hs; P ]         % Frame
P    ::= E                 % Final step
       | E; P              % Step and remaining proof
```

All variables in a proposition that appears in a proof must be bound by quantifiers or be (visible) parameters introduced by frames.

A.2.4 Arithmetic

We add the two binary relations "less than" and "equal to" to our definition of propositions

```
A, B ::= ...
        | M < N            % M less than N
        | M = N            % M equal to N
```

These relations allow us to prove properties about them and defined functions by induction.

Unfortunately, '=' introduces a ambiguity into our syntax, e.g. in

```
proof Ex2 : !x:nat. 0 < x => ?y:nat. s(y) = x = ...
```

While parsing the first '=', it is not clear whether it marks the end of the proposition or stands for the equality relation. This ambiguity is resolved by the following rule:

> Whenever the expression before '=' is *definitively* a term, then '=' is parsed as equality.
> In all other cases it is parsed as the end of the declaration.

In our case 's(y)' is definitively a term. At the next '=', the expression on the left of it is 's(y)=x', which is a proposition, not a term. Thus the end of the declaration is correctly recognized. The same happens in the following example:

```
val nth : nat -> tau list -> tau -> tau = ...
```

The expression 'tau' left of the equality symbol is a variable. It could be a term or a type variable. Thus it is not definitely a term, and the parser finished parsing the type of 'nth' here.

Not correctly resolved is the ambiguity in this case:

```
proof refl : !x:nat. x = x = ...
```

Since 'x' is either a term variable or a type variable from the perspective of mere syntax, it is not definitively a term. Thus the parser detects falsely the end of the declaration of 'refl'. The parser will then try to interpret '!x:nat.x' as a proposition, and fail with the following error message:

> Category mismatch: x is a variable, but a proposition is expected in this place

To work around this bug, insert parentheses somewhere around the equality expression, e.g.

```
proof refl : !x:nat. (x = x) = ...
```

A.2.4.1 Proof Terms

The introduction and elimination rules for equality and less-than give rise to the following proof terms. All are reserved words:

```
eq0 eqS eqE0S eqES0 eqESS less0 lessS lessE0 lessES
```

Of these, two are constants: `eq0` and `less0`. All other are prefix operators. For induction we reuse the '`rec`' construct for primitive recursion.

A.2.5 Structural Induction

We reuse '`=`' for equality on lists. The following proof terms represent the introduction and elimination rules for equality on lists. These are new reserved words:

```
eqN eqC eqENC eqECN eqECC
```

Except `eqN`, which is a constants, all are prefix operators. The rule for `eqC` is:

If $M : xs = ys$, then `eqC`$M : x :: xs = x :: ys$ for an arbitrary x.

Here we assume that all these lists $xs, ys, x :: xs, x :: ys$ are well-formed and of the same type.

A.3 Requirements File Syntax

A requirements file F specifies proof and program tasks, but does not give any proofs or implementations. Grammar:

```
S    ::= proof x: A                  % Proof specification
       | annotated proof x: A        % Proof specification
       | term x: A                   % Term specification
       | val x: T                    % Program specification
F    ::=                             % Empty file
       | S; F                        % Specification and remaining file
```

A.4 Proof Checking

We give an inductive definition of the proof checking algorithm implemented in Tutch via two judgments '`step`' and '`valid`'. The definition is given as in Twelf syntax.

```
% Tutch proof checker for propositional logic

% any infinite datatype:
nat : type.
z : nat.
s : nat -> nat.

% Propositions
prop : type. %name prop A.

% Formation rules
true  : prop.
false : prop.
atom  : nat -> prop.
&     : prop -> prop -> prop.  %infix right 14 &.
|     : prop -> prop -> prop.  %infix right 13 |.
=>    : prop -> prop -> prop.  %infix right 12 =>.

% Notational definitions
~     : prop -> prop
      = [A:prop] (A => false).  %prefix 15 ~.
<=>   : prop -> prop -> prop
      = [A:prop][B:prop] (A => B) & (B => A) .  %infix none 11 <=>.

% One-step inference algorithm
step : prop -> type.

nonhyp : prop -> type.          % available non-hypothetical judgment
hyp    : prop -> prop -> type.  % available hypothetical judgment

% immediate tactic
imm    : nonhyp A -> step A.

% introduction tactics
trueI  : step true.
&I     : nonhyp A -> nonhyp B -> step (A & B).
|IL    : nonhyp A -> step (A | B).
|IR    : nonhyp B -> step (A | B).
=>I    : hyp A B -> step (A => B).

% elimination tactics
falseE : nonhyp false -> step A.
```

```
&EL    : nonhyp (A & B) -> step A.
&ER    : nonhyp (A & B) -> step B.
|E     : nonhyp (A | B) -> hyp A C -> hyp B C -> step C.
=>E    : nonhyp (A => C) -> nonhyp A -> step C.

% Proofs
proof : type. %name proof P.

final : prop -> proof.                    % P, Q ::= A
line  : prop -> proof -> proof.           %        | A; P
frame : prop -> proof -> proof -> proof.  %        | [H; P]; Q

% Proof checking
valid : proof -> prop -> type.

vfinal : step A -> valid (final A) A.
vline  : step A -> (nonhyp A -> valid P B) -> valid (line A P) B.
vframe : (nonhyp H -> valid P A) -> (hyp H A -> valid Q B)
         -> valid (frame H P Q) B.
```

A.4.1 Proof Terms

Here we give a new Twelf implementation of Tutch that includes proof terms. The definition and the typing rules are:

```
% Tutch proof checker for propositional logic
% Version 0.2 proof terms

% any infinite datatype:
nat : type.
z : nat.
s : nat -> nat.

% Propositions
prop : type. %name prop A.

% Formation rules
true  : prop.
false : prop.
atom  : nat -> prop.
&     : prop -> prop -> prop.  %infix right 14 &.
|     : prop -> prop -> prop.  %infix right 13 |.
=>    : prop -> prop -> prop.  %infix right 12 =>.

% Notational definitions
```

```
~     : prop -> prop
      = [A:prop] (A => false).  %prefix 15 ~.
<=>   : prop -> prop -> prop
      = [A:prop][B:prop] (A => B) & (B => A) .  %infix none 11 <=>.

% Proof terms
term : type.  %name term M.

fst  : term -> term.
snd  : term -> term.
,    : term -> term -> term.   %infix right 14 ,.
inl  : term -> term.
inr  : term -> term.
case : term -> (term -> term) -> (term -> term) -> term.
\    : (term -> term) -> term. %prefix 11 \.
    : term -> term -> term.   %infix left 20 .
<>   : term.
abort: term -> term.

% Typing judgement
in   : term -> prop -> type.  %infix none 0 in.

% Typing rules
&I    : M in A -> N in B -> (M , N) in A & B.
&EL   : M in A & B -> fst M in A.
&ER   : M in A & B -> snd M in B.
|IL   : M in A -> inl M in A | B.
|IR   : M in B -> inr M in A | B.
|E    : M in A | B -> ({x: term} x in A -> N x in C)
                   -> ({y: term} y in B -> L y in C)
             -> case M N L in C.
=>I   : ({x: term} x in A -> M x in B) -> \ M in A => B.
=>E   : M in A => B -> N in A -> M  N in B.
trueI : <> in true.
falseE: M in false -> abort M in C.
```

We add *annotated proofs*, which need an inference algorithm that respects proof terms:

```
% One-step inference algorithm
step   : term -> prop -> type. % %mode step +A.

j0hyp  : term -> prop -> type.          % available non-hypothetical judgment
j1hyp  : (term -> term) -> prop -> prop -> type.
                                        % available hypothetical judgment
% immediate tactic
imm    : j0hyp M A -> step M A.
```

```
% introduction tactics
bytrueI  : step <> true.
by&I     : j0hyp M A -> j0hyp N B -> step (M , N) (A & B).
by|IL    : j0hyp M A -> step (inl M) (A | B).
by|IR    : j0hyp M B -> step (inr M) (A | B).
by=>I    : j1hyp M A B -> step (\ M) (A => B).

% elimination tactics
byfalseE : j0hyp M false -> step (abort M) A.
by&EL    : j0hyp M (A & B) -> step (fst M) A.
by&ER    : j0hyp M (A & B) -> step (snd M) B.
by|E     : j0hyp M (A | B) -> j1hyp N A C
                           -> j1hyp L B C
           -> step (case M N L) C.
by=>E    : j0hyp M (A => C) -> j0hyp N A -> step (M  N) C.
```

The checking of annotated proofs requires two judgments '`avalid1`' and '`avalid2`': The first returns a proof term and the second the proven proposition.

```
% Annotated proofs
aproof : type. %name aproof P.

afinal : term -> prop -> aproof.                      % P ::= M : A
aline  : term -> prop -> aproof -> aproof.            %     | M : A; P
aframe : prop -> (term -> aproof) -> aproof -> aproof. %     | [x: H; P]; P'

% Annotated proof checking
avalid1 : aproof -> term -> type. % %mode avalid1 +P -M.
avalid2 : aproof -> prop -> type. % %mode avalid2 +P -A.

avfinal1 : step M A -> avalid1 (afinal M A) M.
avfinal2 : step M A -> avalid2 (afinal M A) A.
avline1  : step M A -> (j0hyp M A -> avalid1 P N) ->
              avalid1 (aline M A P) N.
avline2  : step M A -> (j0hyp M A -> avalid2 P B) ->
              avalid2 (aline M A P) B.
avframe1 : ({x:term} j0hyp x H -> avalid1 (P x) (M x)) ->
          ({x:term} j0hyp x H -> avalid2 (P x) A) ->
          (j1hyp M H A -> avalid1 Q N) -> avalid1 (aframe H P Q) N.
avframe2 : ({x:term} j0hyp x H -> avalid1 (P x) (M x)) ->
          ({x:term} j0hyp x H -> avalid2 (P x) A) ->
          (j1hyp M H A -> avalid2 Q B) -> avalid2 (aframe H P Q) B.
```

Appendix B Emacs Quickstart

Emacs is a colossus: It is a plain text editor looking back on a long history, with a version number in the twentieth, a confusing abundance of functions and a stone age internal programming language called Emacs LISP. There are some expansions of the abbreviation "Emacs", e.g., **E**ight **M**egs **A**nd **C**onstantly **S**wapping, or **E**scape-**M**eta-**A**lt-**C**ontrol-**S**hift, making fun of its memory appetite or hotkey combinations.

The complexity of Emacs may deter the novice. But once you made your first steps and learned your first control sequences, you will realize how big the benefit of using Emacs is and how excellent support you can get by its various plug-ins. These packages make Emacs into a highly efficient tool for Java programming, LaTeX type setting or HTML editing.

Break the barrier now! Dare to dive into Emacs!

This short tutorial is meant to guide you in your first steps and give you all the knowledge to beat Notepad users in speed and safety.

B.1 Starting Emacs, File Commands

Emacs exists in different layouts, we use `xemacs`. Just start it from the shell, optionally supply a file name. E.g.,

```
$ xemacs assignments/prop.tut &
```

You are probably used to the reappearing menu pattern from GUIs: File, Edit, View, Help. Emacs has a structure that looks similar in the beginning, but turns out to be badly organized—flooding you with a lot of irrelevant functions, but hiding the most basic editing commands like cut & paste from you. In Emacs you better not use the menus, but learn the control key sequences instead. This actually is to your benefit: You will find it much faster after some time.

The Emacs community has a special tradition how to name the key modifiers Control and Alternate:

`C-x`	Control-x
`M-x`	Meta-x, i.e., Alternate-x or Escape-x

Here is a list of the basic file commands:

`C-x C-f`	Open... (**F**ind file)
`C-x C-s`	Save
`C-x C-w`	Save as... (**W**rite file)
`C-x s`	Save all...
`C-x k`	Close... (**K**ill buffer)
`C-x C-c`	Exit (**C**lose emacs)

Most interactive commands like "find file" or "write file" prompt for input in the so-called *mini buffer*, the last line of the emacs window. The minibuffer supports edit commands like cut & paste and, what is very comfortable, the file name completion known from Unix shells: Just press the `tab` key to complete a filename, and `space` to get a list of possible completions, from which you can select one with the mouse. Navigating through your file structure with the keyboard and completion is much faster than double-clicking through file dialogs. The minibuffer also facilitates a history: repeated `up` brings back the last inputs. Minibuffer commands:

`tab`	Filename completion
`space`	List completions
`up`	History invokation

B.2 Editing Commands

Cursor movement and basic editing is almost identical to Windows editors. Here is a list of useful deleting commands:

`C-d`	Delete (to the right)
`M-d`	Delete word
`C-k`	Delete to the end of the line (**K**ill characters)

Essential for editing is cut & paste. All deleting commands that delete more than one character are indeed cut-commands. They accumulate the killed characters in the cut buffer until you move the cursor with other commands. If you start cutting more text then, the cut buffer will be emptied before accumulating the new characters. Another way to cut is to mark text and killing the selection, *region* in Emacs terminology. Here are the keys for cut & paste:

`C-space`	Start selection
`C-w`	Cut
`M-w`	Copy
`C-y`	Paste (**y**ank)
`C-_`	Undo

Usually you will cut text by a sequence of `C-k` commands or by `C-space`, marking a region and `C-w` and then paste it back with `C-y`.

B.2.1 Find and Replace

`C-s`	Find... (**S**earch)
`C-s C-s`	Find again
`M-%`	Replace...
`C-g`	Abort

Find and replace always start at the cursor position. If you want to move to the next occurence in a search, just hit `C-s` again.

After entering text and replacement you are prompted at each occurrence whether to replace or not. You have the options `y`es, `n`o, `!` to replace all and `C-g` to abort. (Abort and undo work in all contexts.)

B.2.2 Indentation

Many Emacs plug-ins (*modes* in Emacs terminology) provide special indentation modules for the supported programming language. You invoke these modules by:

`tab`	Indent line
`M-C-\`	Indent region

If you do not have a special Emacs mode for you file syntax, you have to indent manually. To indent a region, mark it and indent it with `C-x tab`. The region then becomes *inactive*, but you can reactivate it with `M-C-z` and indent it for one more column. A better way is to supply an *argument* to the indentation command and indent the region as far as you want in one step. You can provide arguments to commands by `C-u <arg>`. Examples:

`C-u 4 C-x tab`
: Indent the region by four columns.

`C-u -2 C-x tab`
: Unindent the region by two columns.

A good default indentation is provided in the "indented text mode". You can switch to it by invoking the command '`indented-text-mode`', which has no key sequence binding. Commands without a key sequence bound to them are launched by `M-x <command>`. Completion with `tab` also works for commands, therefore type:

```
     M-x indented tab
```

B.3 Editing Several Files

Emacs is designed for convenient editing of several files at a time. It has a clean philosophy of "multi editing", by distinguishing *buffers*, *frames* and *windows*.

Buffer	Data structure to hold a file to edit or a process to interact.
Frame	An Emacs "window" (in standard terminology) with title and menu bar. It contains one or more *windows* (in Emacs terminology).
Window	A horizontal part of the frame (usually the whole free area or half of it) with a status bar displaying file name etc. A window displays one buffer. Several windows can display the same buffer.

Here is some commands to change the buffer displayed in the current window:

`M-C-l`	Switch back to the previously displayed buffer.
`C-x b`	Switch to entered buffer (completion with `tab` works)
`C-x C-b`	Display buffer list.
`C-x k`	Kill buffer.

You will find that you often need `M-C-l`. An alternative to `C-x b` is `C-x C-f`, since the latter does not load the file again if it already occupies one buffer. Window and frame commands:

`C-x 2`	Split window into two
`C-x 1`	Unsplit window (keep this)
`C-x 0`	Unsplit window (keep others)
`C-tab`	Switch window
`C-x 5 2`	New frame
`C-x 5 0`	Delete frame

All these commands do not affect the buffers. Deleting a window does not kill the buffer.

B.4 Modes and Customization

This is where the big potential of Emacs is present, and here it gets complicated. We focus on some very basic issues.

An Emacs *mode* is a plug-in written in Emacs LISP that controls the behaviour of the buffer it is attached to. Usually it provides indentation, syntax highlighting and hotkeys for compiling and jumping to error locations. Sophisticated modes turn Emacs into a full-fledged IDE (integrated development environment). Emacs decides on base of the file extension which editing mode is loaded.

Useful modes for editing Tutch files are the indented text, line and column number mode. They can be used in parallel and are invoked by

```
M-x indented-text-mode return
M-x line-number-mode return
M-x column-number-mode return
```

The indented text mode is one of the *major* modes, which are exclusive. The other two are *minor* modes which can be used as add-ons onto any major mode. Keys for the indented text mode are:

`tab`	Indent line like the previous one.

`C-e C-j`	Start new line with indentation (use instead of `return`).

You can configure Emacs to load these modes automatically when loading '`.tut`'-files. Personal customization of Emacs is put into the file '`~/.emacs`'. Here are the necessary lines to add:

```
(command-execute 'line-number-mode)
(command-execute 'column-number-mode)

(setq auto-mode-alist
  (append
    '(("\\.tut$" . indented-text-mode)
      ("\\.req$" . indented-text-mode))
    auto-mode-alist))
```

B.5 More Editing Commands

Here is a random list of some useful commands:

`M-q`	Justify paragraph.
`M-t`	**T**ranspose words.
`M-- M-l`	Make word **l**ower case.
`M-- M-u`	Make word all **u**pper case.
`M-- M-c`	Make word **c**apital.

Here are more deleting commands:

`M-backspace`	Backspace word
`M-k`	Delete paragraph
`M-\`	Delete spaces
`M-space`	Delete spaces, leave one
`M-^`	Join two lines

Table of Contents