# The Minlog Proof Checker: Mpc

Martin Ruckert

December 10, 2016

## Contents

# 1 Lexical Structure

The input of Mpc is a stream of characters that usually comes from a regular text file (see Invocation). Mpc will group these characters together to form whitespace, comments, punctuation, strings, names, keywords, numbers, or indices. Keywords, punctuation, names, numbers and indices are collectively called tokens.

**Whitespace:** You will know whitespace, when you see it. Otherwise, you may look up the function isspace in a scheme revised five report. Whatever this function regards as whitespace, is whitespace.

Whitespace is of no significance to MPC—except between a name and an index (see below). Its only purpose is to separate two tokens and to make files more readable.

**Comments:** Comments are started with `//` and extend until the end of line or the end of file. Like whitespace, comments separate tokens and can improve readability.

**Numbers and Indices** Numbers and Indices are both formed from the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. They form an index, if they follow immediately after a name, or after an underscore `_`, or after a caret `^`. Otherwise, they form a number.

**Punctuation** The following characters are punctuation: `{`, `}`, `[`, `]`, `(`, `)`, `.`, `;`, `,`, and `"`.

**Strings** Strings start with a "`"` character and end with the following "`"` character or the end of the input. Inside a string any character can be escaped by a preceding backslash "`\`". The preceding backslash strips a character of any special meaning and inserts it plainly into the string. This is useful only for the characters that have any special meaning: the quote and the backslash itself. That is, "`\"`" will insert a double-quote into the string without terminating it, and "`\\`" will insert a single backslash into the string.

**Names** Names are formed either from letters only or from special characters only. All characters except whitespace, digits, letters, and punctuation are considered special characters. Names can be indexed by a number. There must be no whitespace between name and index. E.g `f15` is the function $f_{15}$ whereas `f 15` is the function $f$ applied to the number 15.

Examples of names are "`hello`", "`sigma`", "`==>`", and "`|-`". The following strings are **not** names: "`x_i`", "`(;-)`", or "`id3tag`".

Note: All names of types, variables, functions, or predicates must be declared before they can be used. Once an alphabetic name is declared, it provides an infinite sequence of objects (types, variables, functions, predicates) using indexing.

**Keywords** Keywords are predefined names with a fixed, build in meaning. All the keywords that Mpc knows about are explained below.

# 2 Syntax

The syntax of MPC uses as main ingredients formulas and terms. Since MPC is a front-end to the MINLOG system it uses the MINLOG syntax to specify formulas and terms. To find out how to write formulas and terms you can either rely on your intuition or consult the MINLOG manual. Here we describe only the syntax that is particular to MPC.

MPC is designed to process normal text files. Every text file starts with the keyword "`MPC`" followed by a semicolon "`;`". Then a list of commands follows.

## 2.1 Commands

A command can be one of the following:

- `LOAD "`*Filename*`" ;`
  reads in the given file as SCHEME code. Every possible effect can be obtained in this way by writing appropriate SCHEME code.

  If the file does not exist in the current directory, it is searched for in a directory of library files.

- `INCLUDE "`*Filename*`" ;`
  reads in the given file as MPC code. It can be used to store e.g. a collection of definitions and axioms in a file and load them into several proofs.

  If the file does not exist in the current directory, it is searched for in a directory of library files. Currently two libraries exist: nat.mpc gives definitions of natural numbers, and list.mpc gives definitions for lists.

- `SCHEME "`*Scheme Expression*`";`
  reads the given string with the SCHEME function `read` and supplies the result as an argument to the SCHEME function `eval`, in effect evaluating the given string like regular SCHEME code. Note that inside a string doublequotes need to be escaped by a preceding backslash. For example:

  `SCHEME "(display \"hello world!\")";`.

- `PROOF;` starts a proof. This will set the list of known facts to an empty list. MPC will forget all previously proved formulas.

  Two variations exist: `CLASSIC PROOF` starts a proof with the rule of Stability enabled— that is you can conclude $A$ from $\neg\neg A$.

  `INTUITIONISTIC PROOF` starts a proof with the special rule of Ex-Falso-Quodlibet enabled — that is you can conclude any $A$ from $\bot$.

- `END;` terminates a proof (currently optional).

- A declaration,

- an assumption,

- a claim,

- a block, or

- a syntax specification as detailed below.

## 2.2 Declarations

Declarations are used to tell MPC about types, variables, functions, and predicates.

**Types:** Types come as type variables, as simultaneously defined finite algebras, or as composed types. Only variable types and algebra types need to be declared. Type operators (mostly `=>`) can be freely used to construct composed types.

To declare a type variable the keyword "`TYPE`" is followed by a dot "`.`", a list of names, and finally a semicolon "`;`". It will make the given names type variables. For example:

```
TYPE . rho sigma tau;
```

will make `rho`, `sigma`, and `tau` new type variables.

There is one predefined type variable `alpha` that exists, for internal reasons, right from the start.

By adding an index to a variable name, an infinite number of different type variables can be obtained, e.g. `alpha5`, `tau123`, or `sigma1`.

Algebra types are described in the next section.

**Algebra Types:** Objects of a free algebra type are build by applying an appropriate constructor to already existing objects. For example, the free algebra of natural numbers can be defined with the constructors `zero` and `successor`.

`zero` is a constructor of empty arity, that is, it is applied to no object and yields a natural number, and `successor` is applied to a natural number and yields again a natural number. Hence "`successor zero`" is for example a natural number.

To specify an algebra, the name of the algebra together with the name of the constructors and their arity has to be given. The syntax is:

`ALGEBRA` *algebra-name* `{ { {`
*constructor-type* `.` *constructor-name* `;`
*constructor-type* `.` *constructor-name* `;`
...
`} ;`

For example to define the algebra of natural numbers, one can write

```
ALGEBRA nat {
nat .  zero ;
nat => nat .  successor ;
}
```

Algebras can have type parameters. For example a list type, might have the type of the list elements as a parameter. To declare an algebra with type parameters, the number of type parameters has to be given after the algebra name. In this case the type variables `alpha0`, `alpha1`, ... can be used in the *constructor-types*.

For example to define the free algebra of lists of elements of type `alpha0` one can write:

```
ALGEBRA list 1 {
list .  nil ;
```

```
alpha0 => list => list .  cons ;
} ;
```

Finally, several algebras can be defined simultaneously by listing all their names after the keyword `ALGEBRA`.

As a last example we construct an algebra of labeled trees, where labels can be either of type `alpha0`, or of type `alpha1`, or again labeled trees.

```
ALGEBRA tree label 2 {
alpha0 => label .  first ;
alpha1 => label .  second ;
tree => label .  third ;

tree .  empty;
label => tree => tree => tree .  node ;
} ;
```

**Variables:** Variables are declared by specifying their type. The syntax is:

*type* `.` *variablenames* `;`

For example

```
nat => nat .  f g ;
```

defines `f` and `g` as function variables mapping natural numbers to natural numbers. Similarly `f3` or `g2` are such function variables.

**Functions:** Function constants have a name and a type (like function variables) but in addition have fixed computational rules or rewrite rules attached to them. These rules are used automatically by the prover to find out whether two terms are equal.

A function declaration starts with the keyword `FUNCTION` followed by the target type of the function, a dot, the name of the function and a list of argument types. After the name follows a list of rules enclosed in braces. A semicolon terminates the function declaration. As a special convenience, syntax declarations (see below) can be used inside the rule list.

A rule has the form *term* `->` *term* `;` It states that the left term can be replaced by the right term. Of course, the principal operator of the left term has to be the function currently being defined. Furthermore each rule must have a unique left hand side. More rules can be added if prefixed with the keyword `REWRITE`. These additional rules are applied more carefully (avoiding rewrite loops) and hence are more flexible but slower.

The following example illustrates this:

```
SYNTAX ++ PREFIXOP successor ;
FUNCTION nat .  plus (nat nat)
{
  SYNTAX + ADDOP plus ;

          n + zero -> n ;
          n + ++m -> ++(n + m) ;
  REWRITE zero + n -> n;
  REWRITE ++n + m -> ++(n + m) ;
```

```
  REWRITE n + ( m+ k) -> (n + m) + k ;
};
```

Note, that syntax declarations, explained below, are allowed inside a function declaration to enable the use of a more convenient syntax, e.g. infix notation, if so desired.

Functions, by default, are total functions. It is possible to define partial functions by adding the keyword `PARTIAL` in front of the keyword `FUNCTION`.

**Predicates:** To declare a predicate the keyword `PRED` is used followed by a list of types of the arguments (if any), a dot, a list of predicate names, and finally a semicolon.

For example:

`PRED . A B ;`

defines two predicate variables `A` and `B` (as well as `A0`, `A1`, ...). They take no arguments and can be used like propositional variables.

`PRED nat nat .  R ;`

defines a binary predicate over natural numbers (a relation), and

`PRED nat=>nat .  P ;` defines a unary predicate over functions of natural numbers.

## 2.3 Assumptions and Claims

An assumption is a formula followed by a dot "`.`" and a claim is a formula followed by a semicolon "`;`". In the first case the formula is added to the list of known formulas, in the second case an attempt is made to prove the formula form the formulas already known to be true.

Formulas are defined inductively: Every predicate variable `A` is a formula. If `M` and `N` are formulas then `M & N` (conjunction) and `M -> N` (implication) are formulas. If `M` is a formula and `x` is a variable, then `all x M` and `ex x M` are formulas.

Examples of formulas and details of how proofs work can be found below.

For example:

```
PRED . A ; // A is a proposition
A. // we assume A holds
A & A; // we claim that A & A can be proven
```

## 2.4 Blocks

Blocks are used to construct conditional proofs.

They start with `{` and end with `}`. After the opening brace follows either a formula or a variable followed by a dot. This is the local formula or variable of this block.

Next a block contains a non empty sequence of claims or other blocks.

When the block is closed it proves an implication — in case of a local formula — or an all quantified formula — in case of a local variable. The exact usage is explained below. Here an example may suffice:

```
PRED alpha .  P ; // P is a predicate
alpha .  x ; // x is a variable of type alpha
{ x0 .  // assume x0 is given
  { P x0 .  // assume further that P of x0 holds
    P x0 ; // then P of x0 holds
  } // this proves P x0 -> P x0
} // this proves all x0 .  P x0 -> P x0
```

## 2.5 Syntax Declarations

In mathematics, functions are often written in infix, prefix, or postfix notation. Instead of writing `plus x y` — the function `plus` applied to `x` and `y` — we like to write `x + y`. To facilitate this MPC has syntax declarations.

A syntax declaration starts with the keyword `SYNTAX` followed by the name of the new operator, followed by the tokentype, followed by a term, and a semicolon.

A tokentype is one of the following (in order of increasing binding strength): `PAIROP`, `IMPOP`, `OROP`, `ANDOP`, `RELOP`, `ADDOP`, `MULOP`, `PREFIXOP`, `POSTFIXOP`, or `CONST`. A `PAIROP` and `IMPOP` are right associative, a `RELOP` is not associative, and all other infix operators are left associative.

After the syntax declaration, any term containing the new operator as main connective is replaced by the term given in the syntax declaration applied to the arguments of the operator.

For example:

```
boole .  a b; // a and b are booleans.
boole=>boole=>boole .  f ; // f is a function.
SYNTAX | ADDOP f0 ; // we write | as infix operator for f0.

all a .  true|a .  // for all a ((f0 true) a) holds.
all a,b .  a|b -> b|a.  // f0 is commutative.
false|false -> bot.  // ((f0 false) false) implies bot.
```

Syntax declarations might be parameterized with type variables `alpha0`, `alpha1`, ... If the defining term contains one of these type variables, the types of the actual arguments are matched against the type of the operator to instantiate the type variables. If the types match, the syntax declaration is used.

Further, the same operator might be redefined in several syntax declarations as long as all of these declarations use the same tokentype. Multiple declarations are tested in the order declared and the first matching declaration is used.

Note: Syntax declarations are allowed inside a function declaration.

# 3 Proofs

The MINLOG Proof Checker is able to check proofs in "natural deduction" style.

It maintains a list of formulas, called the context, which are assumed or known or proven to be true. Initially this list is empty; using the keyword `PROOF;` it can be reset to an empty list at any time to start a new proof.

There are only two methods to add a formula to the context: First, one can assume the formula by just stating it and putting a dot behind it. This is called an assumption. Example:

```
PRED . A B; // A and B are propositional variables
A. // Let's assume A holds
A -> B. // Let's assume A implies B
```

Second, one can prove a formula from other formulas already known — i.e. fformulas already part of the context — using the rules of natural deduction. The claim that a formula can be proved is expressed by stating the formula followed by semicolon. This is called a claim. MPC then will check whether there are indeed formulas in the context, which prove the new formula using exactly one rule of natural deduction.

If that is not possible, MPC will start a limited proof search trying to obtain a more complicated proof of the claimed formula. If a proof is found this is indicated in an appropriate warning message. It tells the user that the formula is indeed provable, but not with a single step.

If the proof search does not discover a proof, MPC will simply assume the formula and continue. It will output a corresponding error message, and it should be clear that the proof has still a gap at this point.

Whenever a formula is added to the context, it receives a unique number and MPC will use this number later to refer to this formula in its output.

A typical proof will first define the necessary types, variables, functions and predicates to establish the language of the theory, then it states a list of assumptions made (these are the axioms of the theory) and finally it starts to make claims, adding one formula at a time to the pool of knowledge (the context) available for the theory. In the end it will conclude with the final formula, a theorem of the theory.

Often the language of the theory and its axioms are put into include files, to be able to conveniently load them before starting a proof.

In the following sections, we will discuss all the proof rules of natural deduction, one at a time.

## 3.1 Simple Rules

To be applicable, these rules just require certain formulas to be already in the context.

**Trivial Proofs** If a formula, after normalization, is the same as a formula in the context, it is proven by identity. Likewise, if a formula, by normalization, reduces to `True`, it is proven.

**And Elimination:** If a formula of the form $A \wedge B$ is in the context, it is possible to derive either $A$ or $B$ in one step. Example:

```
A & B. // 0 assumed.
A; // OK, 1 proved by and-elim-left from 0
B; // OK, 2 proved by and-elim-right from 0
```

**And Introduction:** If two formulas $A$ and $B$ are part of the context, it is possible to derive $A \wedge B$ in one step. Example:

```
A. // 0 assumed.
B. // 1 assumed.
A & B; // OK, 2 proved by and-intro from 0 and 1
```

**Implication Elimination:** If an implication $A \to B$ and its condition $A$ are part of the context, it is possible to derive the conclusion $B$ in one step. Example:

```
A -> B. // 0 assumed.
A. // 1 assumed.
B; // OK, 2 proved by imp-elim from 0 and 1
```

**All Elimination:** If an all formula $\forall x\, Ax$ is part of the context, it is possible to derive the conclusion $At$ for any term $t$ of the appropriate type in one step. Example:

```
all x A x. // 0 assumed.
A t; // OK, 1 proved by all-elim from 0 using t
```

**Existential Introduction:** If a formula $At$ for some term $t$ is part of the context, it is possible to derive $\exists x\, Ax$ in one step, where $x$ is a variable of the same type as $t$. Example:

```
A t. // 0 assumed.
ex x A x; // OK, 1 proved by ex-intro from 0 using t
```

### 3.2 Block Rules

Sometimes it is necessary in a proof to temporarily make an assumption only to discard it later again. For example, for proving an implication $A \to B$, one would first assume $A$ holds, and then prove $B$ under this assumption. Once this is done, one can conclude that $A \to B$, and this does no longer depend on the assumption $A$.

The assumption $A$ in this example behaves like a local assumption with a limited scope. In programming languages, the usual way to introduce objects with limited scope is a block structure. In MPC, blocks are enclosed in curly braces and introduce exactly one local object, either a formula or a variable. The scope of this local object is its defining block and all blocks nested inside it.

**Implication Introduction:** As said before, an implication $A \to B$ is proved by assuming $A$ and then proving $B$ under this assumption. Once this is done, one can conclude that $A \to B$. In MPC, one opens a block with the local assumption $A$ and proves inside this block the formula $B$. Immediately after the formula $B$ the block is closed again. After the closing brace of the block, MPC will discard all the formulas added to the context during the block (since these may depend on the assumption $A$) and adds the implication $A \to B$ to the context, $A$ being the local formula of the block and $B$ the last formula of the block.

Example:

```
{ A t. // 0 assumed.
  ex x A x; // OK, 1 proved by ex-intro from 0 using t
} OK, 2 A t -> ex x A x proved.
ex x A x; // ERROR: 3 assumed.  Proof not found.
```

**All Introduction:** The proof of a formula with an outer universal quantifier is similar to the proof of an implication: Under the assumption that some $x$ is given, one proves $Ax$. This is sufficient to conclude $\forall x\, Ax$.

For MPC, the proof consists of a block with a local variable $x$ with the last formula being $Ax$. At the end of the block, MPC will discard all the formulas added to the context during the block and adds the formula $\forall x\, Ax$ to the context.

Example:

```
{ x. // x assumed.
  { A x. // 0 assumed.
    A x; // OK, 1 proved by 0
  } OK, 2 A x -> A x proved.
} OK, 3 all x.A x -> A x proved.
```

**Existential Elimination:** A proof of a formula $B$ may use an existentially quantified formula $\exists x\, Ax$. It typically proceeds like this: If we know that $\exists x\, Ax$, let us assume we have such an $x$, call it $x_0$, such that $Ax_0$ holds,... and from this the proof continues to prove the formula B. This then constitutes a proof of $B$ from $\exists x\, Ax$ under the side condition that the $x_0$ is not a free variable of $B$.

This proof can be formulated for MPC in exactly the same fashion as outlined above using two nested blocks. The first block introduces the local variable $x_0$ and the second block the local assumption $Ax_0$. Once the formula $B$ is proved from this, both blocks are closed. This in effect proves the formula $\forall x.Ax \to B$. This, together with the formula $\exists x\, Ax$, can be used to finally prove $B$ using the rule of existential elimination.

Example:

```
ex x.A x & B. // 0 assumed.
{ x0. // x0 assumed.
  { A x0 & B. // 1 assumed.
    B; // OK, 2 proved by and-elim-right from 1
  } OK, 3 A x0 & B -> B proved.
} OK, 4 all x0.A x0 & B -> B proved.
B; // OK, 5 proved by ex-elim from 4 and 0
```

### 3.3 Induction

Induction is used to prove that a formula $Ax$ holds for all objects $x$ of a given algebra type $\tau$. This is done by considering all constructors $C_1, \ldots, C_n$ of the algebra that are capable of producing an object of the type in question and proving for each one of them that the formula $AC_i \ldots$ holds provided that the formula holds already for all arguments of $C_i$ of type $\tau$. Once this is done, all these formulas together, prove by the principle of induction, that $\forall x\, Ax$.

We illustrate this using the standard example of natural numbers.

The free algebra of natural numbers `nat` is generated from two constructors: `Zero` of type `nat` and `Succ` of type `nat` $\to$ `nat`. For convenience we write `0` for `Zero` and $++n$ for `Succ` $n$

To prove $\forall n\, An$ by induction, we need to prove first $A$`0` and $\forall n\, An \to A{+}{+}n$, then we can conclude the desired result. For example lets prove that $\forall n \exists m\, m = n+1$. We proceed like this:

First $1 =$ `0` $+ 1$ and therefore $\exists m\, m =$ `0` $+ 1$.

Second, assume $n$ is given and $\exists m\, m = n+1$ holds. Then there is an $m_0$ with $m_0 = n+1$, and thus $m_0 + 1 = {+}{+}n + 1$. By existential introduction, we conclude $\exists m\, m = {+}{+}n + 1$ and have proved $\forall n\, (\exists m\, m = n+1 \to \exists m\, m = {+}{+}n + 1)$.

From these, by induction, we infer: $\forall n \exists m\, m = n+1$.

The complete proof written for MPC reads:

```
MPC;

INCLUDE "nat.mpc";

PROOF;                  // initializing mpc1
1=0+1;                  // OK, 0 proved trivial
ex m m=0+1;             // OK, 1 proved by ex-intro from 0 using 1
{ n.                    // n assumed.
  { ex m m=n+1.         // 2 assumed.
    { m0.               // m0 assumed.
      { m0=n+1.         // 3 assumed.
        m0+1= ++n+1;    // OK, 4 proved by 3
        ex m m= ++n+1; // OK, 5 proved by ex-intro from 4 using m0+1
      }                 // OK, 6 m0=n+1 -> ex m m= ++n+1 proved.
    }                   // OK, 7 all m0.m0=n+1 -> ex m m= ++n+1 proved.
    ex m m= ++n+1;      // OK, 8 proved by ex-elim from 7 and 2
  }                     // OK, 9 ex m m=n+1 -> ex m m= ++n+1 proved.
}                       // OK, 10 all n.ex m m=n+1 -> ex m m= ++n+1 proved.
all n ex m m=n+1;       // OK, 11 proved by ind from 10 1
```

### 3.4 Proof by Cases

Proof by cases is similar to induction but weaker. Again, we prove that a proposition $Ax$ holds for all objects $x$ of a given algebra type $\tau$. This is done by considering all constructors $C_1, \ldots, C_n$ of the algebra that are capable of producing an object of the type in question and proving for each one of them that the formula $AC_i \ldots$ holds. In contrast to the rule of induction however, no induction hypothesis is available in the proof.

A special case is the proof by cases for objects of type `boole`. Here, the constructors are `True` and `False`. We prove $Ax$ by considering the two cases, proving $A$`True` and $A$`False` to conclude $\forall x\, Ax$. Typically, this formula is then applied to the boolean term $t$ in question to obtaining $At$. Since this process is quite common, in addition to the usual proof by cases rule, an equivalent and more convenient rule is built into MPC: the proof by boolean cases. To prove any formula $A$, you just have to prove $t \to A$ and $(\neg t) \to A$.

### 3.5 Intuitionistic and Classical Logic

MPC provides two keywords `CLASSIC` and `INTUITIONISTIC` to activate proof rules for classic and intuitionistic logic, respectively. If a proof starts with "`INTUITIONISTIC PROOF;`", the proof rule "ex falso quodlibet" is enabled. It allows to conclude from `bot` any formula whatsoever. If a proof starts with "`CLASSIC PROOF;`" in addition the stronger proof rule of "stability" is enabled. It allows to conclude the formula $A$ from a statement of $\neg\neg A$. It is an easy exercise to prove $\bot \to A$ from $\neg\neg A \to A$, and therefore stability alone would be sufficient to have classical logic. It is however convenient to have "ex falso quodlibet" in addition. This weaker rule is always tested first.

As an example, we present a proof of the Pierce Formula $((P \to Q) \to P) \to P$.

```
CLASSIC PROOF;
PRED . P Q;

{ (P -> Q) -> P.      // 0 assumed.
  { P -> bot.         // 1 assumed.
    { P.              // 2 assumed.
      bot;            // OK, 3 proved by imp-elim from 1 and 2
      Q;              // OK, 4 proved by EFQ from 3
    }                 // OK, 5 P -> Q proved.
    P -> Q;           // OK, 6 proved by 5
    P;                // OK, 7 proved by imp-elim from 0 and 6
    bot;              // OK, 8 proved by imp-elim from 1 and 7
  }                   // OK, 9 (P -> bot) -> bot proved.
  (P -> bot) -> bot; // OK, 10 proved by 9
  P;                  // OK, 11 proved by Stability from 10
}                     // OK, 12 ((P -> Q) -> P) -> P proved.
END;
```

# A Library code

This Mpc code may serve as an example to illustrate the concepts of Mpc.

## A.1 Natural Numbers

```
MPC;

ALGEBRA nat  0
{ nat=>nat   . Succ ;
  nat  . Zero  ;
};

nat . n m k;

// to use numbers we have to provide scheme code
// converting numbers to terms using internals of Minlog
SCHEME
"(define (make-numeric-term n)
  (if (= n 0)
      (pt \"Zero\")
      (make-term-in-app-form
       (pt \"Succ\")
       (make-numeric-term (- n 1)))))" ;

SCHEME
"(define (is-numeric-term? term)
  (or
   (and (term-in-const-form? term)
(string=? \"Zero\"
         (const-to-name
   (term-in-const-form-to-const term))))
   (and (term-in-app-form? term)
(let ((op (term-in-app-form-to-op term)))
  (and (term-in-const-form? op)
       (string=? \"Succ\"
                 (const-to-name
  (term-in-const-form-to-const op)))
       (is-numeric-term?
         (term-in-app-form-to-arg term)))))))";

SCHEME
"(define (numeric-term-to-number term)
  (if (equal? term (pt \"Zero\"))
      0
      (+ 1 (numeric-term-to-number
             (term-in-app-form-to-arg term)))))";


SYNTAX ++ PREFIXOP Succ;
```

```
FUNCTION nat . Plus(nat nat)
{ SYNTAX + ADDOP Plus;

          n + 0       -> n;
          n + ++m     -> ++(n + m);
  REWRITE 0 + n       -> n;
  REWRITE ++n + m     -> ++(n + m);
  REWRITE n + (m + k) -> n + m + k;
};


FUNCTION nat . Times(nat nat)
{ SYNTAX * MULOP Times;
          n * 0       -> 0;
          n * ++m     -> (n*m)+n;
  REWRITE 0*n         -> 0;
  REWRITE ++n*m       -> (n*m)+m;
  REWRITE n * (m * k) -> n * m * k;
};

FUNCTION boole . Less(nat nat)
{ SYNTAX < RELOP Less;
  n < 0     -> False;
  0 < ++n   -> True;
  ++n < ++m -> n<m;
};
```

## A.2 Polymorphic Lists

```
MPC;

ALGEBRA list 1 {
list                    . Nil;
alpha1 => list => list . Cons ;
};

// a generic variable of type (list alpha)
(list alpha) . l;

SYNTAX :: PAIROP    (Cons alpha);
SYNTAX :  POSTFIXOP [alpha] alpha ::(Nil alpha);
// example x :: y :: z :


FUNCTION list alpha => list alpha . ListAppend (list alpha)
{ SYNTAX :+: PAIROP (ListAppend alpha);
  (ListAppend alpha)(Nil alpha)   -> [l_2]l_2;
  (ListAppend alpha)(alpha :: l1) -> [l2] alpha::(l1:+:l2);
};


INCLUDE "nat.mpc";

FUNCTION nat . ListLength(list alpha)
{ SYNTAX  lh PREFIXOP (ListLength alpha);
          lh (Nil alpha)  -> 0 ;
          lh (alpha :: l) -> ++ lh l;
  REWRITE lh (l1 :+: l2)  -> lh l1 + lh l2;
};
```