A Logic Evaluator

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Wolfgang.Schreiner@risc.uni-linz.ac.at

August 2, 1999

Abstract

The Logic Evaluator is an interpreter that understands an executable subset of first order logic with natural numbers, sets, and tuples. The user may define predicates and functions and determine the values of formulas and terms; a simple visualization feature allows to plot the graphs of relations and functions. The evaluator is implemented in Java; instances of it can be embedded as applets into HTML documents such that readers may interactively experiment with basic mathematical concepts. The executable software and its source code can be downloaded from

`http://www.risc.uni-linz.ac.at/software/formal`

Contents

1 Examples

The following example demonstrates the use of the evaluator on some set-theoretic notions. We define the product of two sets A and B as

$$A \times B := \{\langle a, b\rangle : a \in A \wedge b \in B\}.$$

This is written in the language of the evaluator as

```
fun x(A, B) = set(a in A, b in B: true, tuple(a, b));
```

(where "true" is a placeholder for additional restrictions on a and b).

This function definition is stored in a file `set.txt` (listed in the appendix) which is loaded in the evaluator as shown below. When pressing the `Enter` key, the product of the sets $\{1, 2, 3\}$ and $\{0, 1\}$ is computed (you may also experiment with other examples).

```
read set;
fun A = set(x in nat(1, 3): true, x);
fun B = set(x in nat(0, 1): true, x);
> file 'set.txt' read.
> function A/0.
> function B/0.
term x(A, B);
> {<1, 0>, <1, 1>, <2, 0>, <2, 1>, <3, 0>, <3, 1>}.
```

```
term x(A, B);
```

File `set.txt` also includes a definition of the unary function `Powerset` that returns the set of all subsets of a given set

$$\text{Powerset}(A) := \{S : S \subseteq A\}$$

where the subset relationship

$$A \subseteq B :\Leftrightarrow (\forall x \in A : x \in B)$$

is defined in the evaluator as

```
pred isSubset(A, B) <=> forall(x in A: in(x, B));
```

For instance, if we take S as $\{1, 2, 3, 4, 5\}$, then its powerset is computed as

```
read set;
fun S = set(x in nat(1, 5): true, x);
> file 'set.txt' read.
> function S/0.
term Powerset(S);
> {{1, 2, 3, 4, 5}, {2, 3, 4, 5}, {1, 3, 4, 5}, {3, 4, 5}, {1, 2, 4,
5}, {2, 4, 5}, {1, 4, 5}, {4, 5}, {1, 2, 3, 5}, {2, 3, 5}, {1, 3,
5}, {3, 5}, {1, 2, 5}, {2, 5}, {1, 5}, {5}, {1, 2, 3, 4}, {2, 3, 4},
{1, 3, 4}, {3, 4}, {1, 2, 4}, {2, 4}, {1, 4}, {4}, {1, 2, 3}, {2,
3}, {1, 3}, {3}, {1, 2}, {2}, {1}, {}}.
```

```
term Powerset(S);
```

We now want to check whether every element in this powerset is indeed a subset of the input, i.e.,

$$(\forall A \in \text{Powerset}(S) : A \subseteq S).$$

The test is computed by the following program:

```
read set;
fun S = set(x in nat(1, 5): true, x);
> file 'set.txt' read.
```

Now for some geometry: a tuple $p = \langle p_x, p_y \rangle$ of numbers may be interpreted as a point in the plane with horizontal coordinate p_x and vertical coordinate p_y, respectively. The relation

$$\begin{aligned} p \sim q :\Leftrightarrow\ & r = s \text{ where} \\ & (r = (p_x - c_x)^2 + (p_y - c_y)^2, \\ & \ s = (q_x - c_x)^2 + (q_y - c_y)^2) \end{aligned}$$

expresses the fact that two points p and q have the same distance from a given point c. The set of points

$$\text{circle}(p) := \{q : p \sim q\}$$

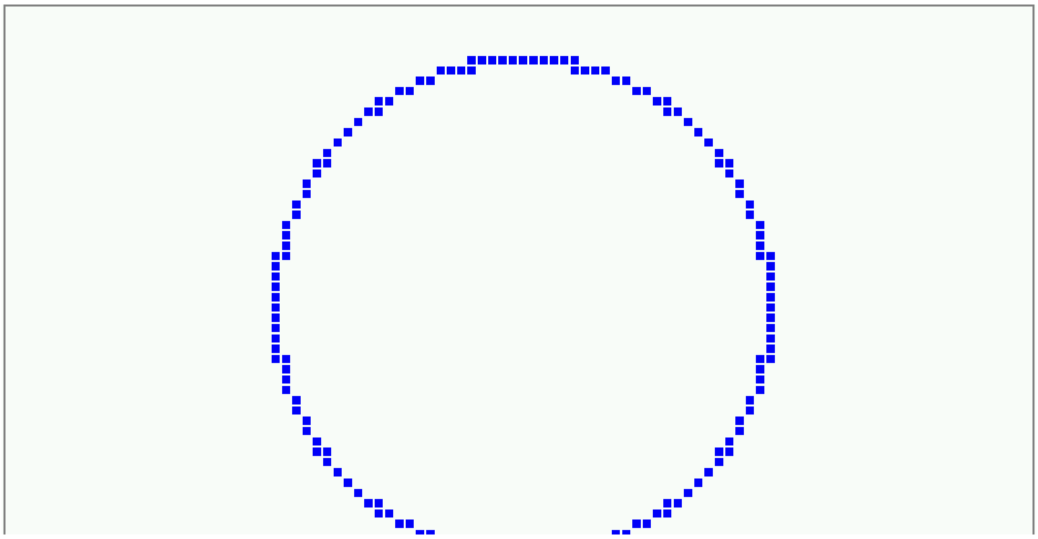
therefore describes the circle with center c that goes through p.

Using the corresponding definitions

```
pred ~(p: Point, q: Point) <=>
  let (r = +(-^2(.0(p), .0(c)), -^2(.1(p), .1(c))),
       s = +(-^2(.0(q), .0(c)), -^2(.1(q), .1(c))):
  ~=(r, s));

fun circle(p: Point) =
  set(x in rangeX, y in rangeY, q = tuple(x, y): ~(p, q), q);
```

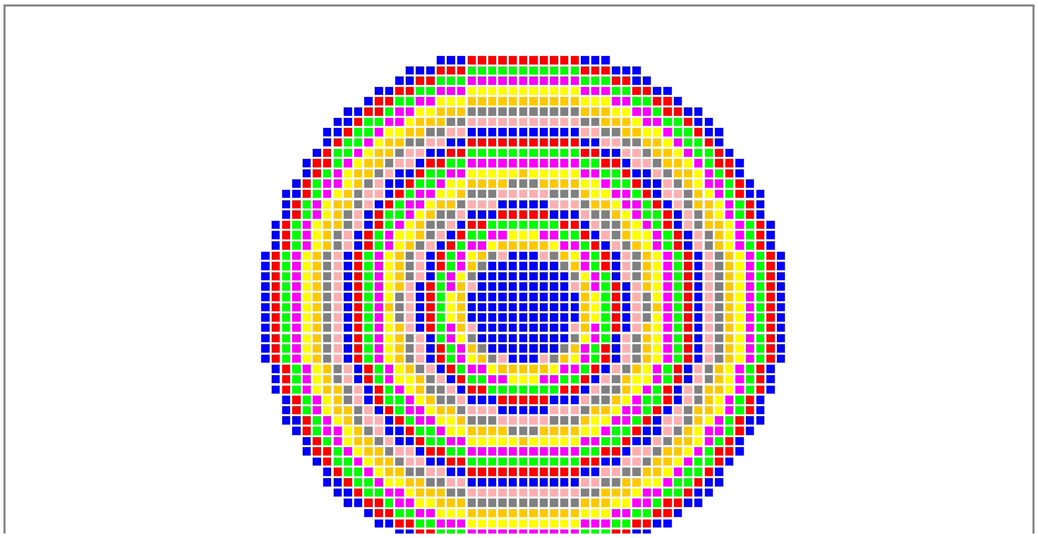
stored in file `circle.txt` (listed in the appendix) we can visualize such a circle as shown below.



Likewise, we may partition (a subset of) the plane into the set of all such circles

$$\{\text{circle}(p) : p = \langle x, y\rangle, x \in \text{rangeX}, y \in \text{rangeY}\}$$

In the executable version, we take a fixed value for y in order to speed up the computation (the result is the same):



2 Language

The Logic Evaluator reads a sequence of input lines; it continues with parsing and executing after every line read (i.e., in interactive mode, when the user has terminated a line with the `Enter` key). However, when executing as an applet embedded into an HTML page, the evaluator only runs while the focus is on the evaluator's input field such that this field is gray; if this field is not gray, execution is suspended. Applets may be reinitialized by pressing the keys `Shift`+`Reload` (Netscape Communicator) respectively `Refresh` (Microsoft Internet Explorer).

All responses of the evaluator are prefixed by the token '`>`':

```
fun f(x) = x;
> function f/1.
```

Any text starting with the token '`//`' is ignored until the end of a line is encountered:

```
fun f(x) = x; // this is the identity function
> function f/1.
```

Apart from that, the separation of input into lines has no significance:

```
fun f(x) = // this is the identity function
  x;
> function f/1.
```

We describe the grammar of the evaluator's language by an extended version of BNF where "{ *Phrase* }" denotes zero or more repetitions of *Phrase* and "[*Phrase*]" denotes zero or one occurrence of *Phrase*. Do not confuse the meta-symbols '{', '}', '[', ']' with the language symbols '`{`', '`}`', '`[`', '`]`' (which are typeset in a different font).

2.1 Commands

The evaluator executes a sequence of commands:

{ [*Command*] `;` }

The input consists of a sequence of commands separated by one or more semicolons. The semicolon serves as a "boundary marker": in the case of a syntax error, the parser skips all input until it encounters the next semicolon.

```
pred p(x,y) <=> <=(*(2, y), x);
> predicate p/2.
fun f(x, y) = if(p(x, y), x, y);
> function f/2.
term f(2 3);
> ERROR: unexpected '3'.
term f(2, 3);
> 3.
```

A *Command* is one of the following statements executed by the evaluator.

`pred` *Name* [`(` *ConstrainedVariables* `)`] [`recursive` *Term*]
`<=>` *Formula*

Defines a predicate with a certain name and arity (number of parameters). Different predicates may have the same name if they have different arities. The symbol '`<=>`' should be surrounded by whitespace, because it can be also part of a name.

```
pred isSubset(A: Set, B: Set) <=> // A is subset of B
  forall(x in A: in(x, B));
> predicate isSubset/2.
pred positive(x: Nat)<=>>(x, 0);
> ERROR: unexpected '<=>>'.
```

If the `recursive` clause is given, the definition may refer to the function being defined. In this case, however, a *termination term* must be given that denotes a natural number which is decreased in every recursive invocation of the predicate (which is checked by the interpreter, if execution checking is switched on).

```
pred <(m: Nat, n: Nat) recursive m =
  if (=(m, 0), not(=(n, 0)),
    and(not(=(n, 0)), <(-(m, 1), -(n, 1))));
> function </2.
```

`fun` *`Name`* [`(` *`ConstrainedVariables`* `)`] [`recursive` *`Term`*] `=` *`Term`*

Defines a function with a certain name and arity (number of parameters). Different functions may have the same name if they have different arities. The symbol `=` should be surrounded by whitespace, because it can be also part of a name.

```
fun **(A: Set, B: Set) = // the intersection of A and B
  set(x in A: in(x, B), x);
> function **/2.
fun inc(x: Nat)=+(x, 1);
> ERROR: unexpected '=+'.
```

If the `recursive` clause is given, the definition may refer to the function being defined. In this case, however, a *termination term* must be given that denotes a natural number which is decreased in every recursive invocation of the function (which is checked by the interpreter, if execution checking is switched on).

```
fun sum(S: Set) recursive #(S) =
  if (=(S, {}), 0,
    let(e = element(S):
      +(e, sum(-(S, {}(e)))));
> function sum/1.
```

`formula` *`Formula`*

Evaluates a formula to true or false.

```
formula isSubset(**(A, B), A);
> true.
```

`term` *Term*

Evaluates a term to a value.

```
term **(A, **(B, C));
> {1, 2, 3}.
```

`plot` *Term*

Plots a set of points. The value of the given term must be a set of tuples of length 2 which are interpreted as points in the plane. The elements of each tuple must be natural numbers that are interpreted as the horizontal respectively vertical coordinate of the point. The coordinate system has point (0,0) in the lower left corner and extends `swidth` units to the right and `sheight` units to the top. The values `swidth` and `sheight` are defined as startup parameters.

```
plot set(x in nat(1,50), y in nat(1, x): <=(x,*(y,y)),
  tuple(x, y));
> done.
```

`plots` *Term*

Plots a set of point sets. The value of the given term must be a set of point sets as described for the command `plot`. The point sets are printed in various colors.

```
plots set(r in nat(1,50): true,
  set(x in nat(1, r), y in nat(1, x): <=(x,*(y,y)),
    tuple(x, y)));
> done.
```

`read` *Name*

Reads the file *Name* `.txt` and executes the commands contained in the file.

```
read theory/set;
> predicate Subset/2.
> function **/2.
> file 'theory/set.txt' read.
```

`option` ***`Option`***

Sets ***`Option`*** which can be one of the following:

`silent = ( true | false )`

Switches on respectively off the messages that are printed after the definition of every predicate respectively function (initially on). This is the only command to which the evaluator does *not* respond with a message:

```
fun f(x) = x;
> function f/1.
set silent = true;
fun inc(x: Nat) = +(x, 1);
fun dec(x: Nat) = -(x, 1);
```

`check = ( true | false )`

Switches on respectively off the checking of guard formulas, parameter types, and termination terms in the evaluation of formulas and terms (initially on).

```
fun f(x: Nat) = x;
> function f/1.
term f({});
> ERROR: value {} does not satisfy predicate Nat/1
  for parameter x.
option check = false;
> execution checks are off.
term f({});
> {}.
```

`universe =` *`Term`*

Sets the *universe of discourse* to the domain denoted by the given term (initially undefined). The evaluation of this term must denote a domain (an interval or a set). All quantified variables without domain constraints are assumed to range over this domain.

```
option universe = nat(1,100);
> universe of discourse set.
formula forall(x: <=(x, 1000));
> true.
```

`help`

Prints a short help message with a reference to the home page of the evaluator.

```
help;
> commands (terminated by ';'):
>   pred <name>(<parameters>) <=> <formula>;
>   fun <name>(<parameters>) = <term>;
>   formula <formula>;
>   term <term>;
>   plot <term>;
>   plots <term>;
>   read <name>;
>   option <name> = <value>;
>   help;
> see http://www.risc.uni-linz.ac.at/software/formal.
```

2.2 Formulas

A *Formula* is one of the following phrases denoting a Boolean value (true or false):

Name [(*Terms*)]

An *atomic formula*: the value of the formula is the value of the denoted predicate where the parameters are bound to the values of the corresponding argument terms.

```
pred p(x) <=> <=(x, 5);
> predicate p/1.
formula p(4);
> true.
```

if (*Formula* , *Formula* [, *Formula*])

A *conditional* formula: if the first formula (the *guard*) evaluates to true, the result is the value of the second formula, otherwise it is the result of the third formula. If the guard evaluates to false and the third formula is not given, the result is undefined (evaluation aborts, if guard checking is switched on).

```
formula if(not(true), false, or(true, false));
> true;
formula if(not(true), false);
> ERROR: guard condition is false.
```

`let` (*BoundVariables* : *Formula*)

A formula with some *locally bound variables*; the value of the formula is the value of the base formula for the denoted variable bindings.

```
formula let(x = 1, y = 2: <=(x, y));
> true;
```

`not` (*Formula*)

The *negation* of a formula: the negation is true if and only if the base formula is false.

```
formula not(true);
> false.
```

`and` ({ *Formulas* })

The *conjunction* of a sequence of formulas; the conjunction is true if and only if every formula in the sequence is true.

```
formula and(true, not(true), true);
> false.
```

`or` ({ *Formulas* })

The *disjunction* of a sequence of formulas; the disjunction is true if and only if some formula in the sequence is true.

```
formula or(false, and(false, true), not(false));
> true.
```

`implies` (*Formula* , *Formula*)

The *implication* of two formulas: the implication is false if and only if the first formula is true and the second one is false.

```
formula implies(not(false), false);
> false.
```

`equiv` (*Formula* , *Formula*)

The *equivalence* of two formulas: the equivalence is true if and only if both formulas are true or both are false.

```
formula equiv(true, or(false, true));
> true;
```

`forall` (*IteratorVariables* : *Formula*)

A formula that is *universally quantified* over a sequence of variables: the universal quantification is true if the base formula is true for every binding of the variables denoted by the iterators.

```
formula forall(x in nat(1,10), y in nat(1, x): <=(y, x));
> true;
```

`exists` (*IteratorVariables* : *Formula*)

A formula that is *existentially quantified* over a sequence of typed variables; the existential quantification is true if the base formula is true for some binding of the variables denoted by the iterators.

```
formula exists(x in nat(1, 10), y in nat(x, 10):
  =(y, *(2, x)));
> true;
```

2.3 Terms

A *Term* is one of the following phrases denoting a *value*values:

Variable

A *variable*: the value of a variable is determined by the context in which the variable is evaluated.

```
formula forall(x in Nat(1, 10): <=(x, 10));
> true.
fun f(x: Nat) = +(x, 1)
> function f/1.
```

Name [(*Terms*)]

A *function application*: the value of the application is the result of the function denoted by the given name and arity when applied to the values of the given argument terms.

```
fun sumprod(x, y) = +(*(x, y), y);
> function sumprod/2.
term sumprod(2, 3);
> 9.
```

`if ( `*`Formula`*` , `*`Term`*` [ , `*`Term`*` ] )`

A conditional term: if the formula (the *guard*) evaluates to true, the result is the value of the first term, otherwise it is the result of the second term. If the guard evaluates to false and the second term is not given, the result is undefined (evaluation aborts, if guard checking is switched on).

```
term if(>(2, 3), {}, join(1, {}));
> {1}.
```

`let ( `*`BoundVariables`*` : `*`Term`*` )`

A term with some locally bound variables: the value of this term is the value of the base term for the denoted variable bindings. The token '`=`' must be surrounded by whitespace because it can also appear as part of a name.

```
term let(x = 1, y = 2: x+y);
> 3.
```

`such ( `*`IteratorVariables`*` : `*`Formula`*` , `*`Term`*` )`

An *implicit definition*: the value of this term is the value of the base term for some binding of the quantified variables that satisfies the given formula.

```
term such(x in nat(1, 10): =(12, *(2, x)), x);
> 6.
term such(x in nat(1, 10): =(24, *(2, x)), x);
> ERROR: no such value.
```

`0-9{0-9}`

A natural number is represented by a non-empty sequence of decimal digits.

```
term 17;
> 17.
```

`set ( `*`IteratorVariables`*` : `*`Formula`*` , `*`Term`*` )`

The set of values of the base term for all bindings of the quantified variables that satisfy the given formula.

```
term set(x in nat(1, 6), y in nat(1, 6): =(6, +(x, y)),
  tuple(x,y));
> {<1, 5>, <2, 4>, <3, 3>, <4, 2>, <5, 1>}.
```

`reduce` (*Name* , *Term* , *Term*)

The reduction of a set by a binary function. The name must denote a binary function f which is applied to each element of the set s denoted by the first term and the base value b of the last term such that

$$\texttt{reduce}(f,\ \{\},\ b) = b$$
$$\texttt{reduce}(f,\ v \cup s,\ b) = f(v, \mathrm{reduce}(f, s, b))$$

where v is an element not contained in s. An application `reduce`(f, s, b) only yields a unique result if $f(p_0, f(p_1, \ldots, f(p_{n-1}, b)))$ denotes the same value for every permutation p of the elements of s.

```
term reduce(+, nat(1, 6), 0);
> 21.
term reduce(join, nat(1, 6), {});
> {1, 2, 3, 4, 5, 6}.
```

`tuple` (*Terms*)

A tuple, i.e., a sequence of values.

```
term tuple(2, 3, 4);
> <2, 3, 4>.
```

.*Number* (*Term*)

The element at the denoted position in the denoted tuple; *Number* is a natural number in the interval 0 to the length of the tuple minus one.

```
term let(t = tuple(2, 3): +(.0(t), .1(t)));
> 5.
```

2.4 Miscellaneous

Formulas = [*Formula* { , *Formula* }]

A (possibly empty) sequence of formulas separated by commas.

```
pred areOrdered(a, b, c, d) <=>
  and(<=(a, b), <=(b, c), <=(c, d));
> predicate areOrdered/4.
```

Terms = [*Term* { , *Term* }]

A (possibly empty) sequence of formulas separated by commas.

```
fun sumSquares(a, b) =
  +(*(a, a), *(b, b)).
> function sumSquares/2.
```

IteratorVariables = [*IteratorVariable* { , *IteratorVariable* }]

A (possibly empty) sequence of iterator variables separated by commas.

```
option universe = nat(1,10);
> universe of discourse set.
formula forall(x, y in nat(1, x), z = y: <=(y, z));
> true.
```

ConstrainedVariables = [*ConstrainedVariable* { , *ConstrainedVariable* }]

A (possibly empty) sequence of constrained variables separated by commas.

```
fun minus(x: Nat, y in nat(1, x)) = -(x, y);
> function minus/2.
```

BoundVariables = [*BoundVariable* { , *BoundVariable* }]

A (possibly empty) sequence of bound variables separated by commas.

```
term let(x = 1, y = 2: +(x, y));
> 3.
```

IteratorVariable = *Variable* | *TypedVariable* | *BoundVariable*

An *iterator variable* is a variable that iterates over a sequence of values: a plain variable (iterating over the universe of discourse), a typed variable (iterating over some domain), or a bound variable (iterating over a single value).

```
option universe = nat(1, 3);
> universe of discourse set.
term set(x, y in nat(1, x), p = tuple(x, y): true, p);
> > <1, 1>, <2, 1>, <2, 2>, <3, 1>, <3, 2>, <3, 3>.
```

ConstrainedVariable = *Variable* | *TypedVariable* | *CheckedVariable*

A *constrained variable* is a variable whose domain may be restricted: a plain variable (not constrained), a typed variable (constrained by a domain), or a checked variable (constrained by a predicate).

```
pred isMultiple(x: Nat, y in nat(1, x)) <=>
  exists(z in nat(1, x): =(x, *(y, z)));
> predicate isMultiple/2.
```

Variable = *Name*

A name represents a variable if it occurs in the scope of a corresponding variable declaration of some quantifier or if it is declared as a parameter in a predicate/function definition.

```
fun a = 0;
> function a/0.
formula forall(x in nat(2,10): not(=(a, -(x, 1))));
> true.
fun f(x) = x;
> fun f/1.
```

TypedVariable = *Name* `in` *Term*

A typed variable is a name whose value range is constrained by a *domain* (an interval or a set). It is an error to attempt to bind to the variable a value that is not from the specified domain (such an attempt is detected if parameter checking is switched on).

```
fun -1(x in nat(1, 100)) = -(x, 1);
> function -1/1.
```

BoundVariable = *Name* `=` *Term*

A *bound variable* receives its value from the denoted term. The token '`=`' should be surrounded by whitespace because it can be also part of a a name.

```
term let(x = 7: *(2, x));
> 14.
term let(x=7: *(2, x));
> ERROR: unexpected ':'.
```

CheckedVariable = *Name* : *Name*

A checked variable is a name whose value range is constrained by a predicate. The second name must denote a unary predicate; it is an error to attempt to bind a value to the variable that is not satisfy the predicate (such an attempt is detected if parameter checking is switched on).

```
pred isEven(n: Nat) <=>
  exists(m in nat(0, n): =(n, *(2, m)));
> predicate isEven/2.
fun half(n: isEven) =
  such(m in nat(0, n): =(n, *(2, m)));
> function half/2.
```

Name

A name is any sequence of non-whitespace characters that does not start with a decimal digit and that does not include any of the characters '(', ')', ',', ':', ';'.

```
term +3(xVal, f_0(y, #));
```

2.5 Values

Every *term* denotes one of the following types of values:

A Natural Number One of the values 0, 1, 2, ...

```
term +(1, 1);
> 2.
```

A Set An unordered (and in the evaluator finite) collection of values.

```
term join(tuple(1, 2), join({}, join(1, {})));
> {1, {}, <1, 2>}.
```

A Tuple A finite sequence of values.

```
term tuple(2, tuple(1, 4), {}, 3);
> <2, <1, 4>, {}, 3>.
```

An Interval A subrange of the natural numbers.

```
term nat(2, 7);
> 2..7.
```

Both sets and intervals are *domains* that may be used to constrain the value range of a variable.

2.6 Predefined Predicates and Functions

The following predicates are predefined (/n denotes arity n):

`true/0` The predicate denoting true.

```
formula true;
> true.
```

`false/0` The predicate denoting false.

```
formula false;
> false.
```

`Set/1` The predicate that returns true if and only if its argument is a set.

```
formula Set({});
> true.
```

`Tuple/1` The predicate that returns true if its argument is a tuple.

```
formula Tuple(tuple(2, 3));
> true.
```

`Nat/1` The predicate that returns true if its argument is a natural number.

```
formula Nat(1);
> true.
```

`=/2` The equality of values.

```
formula =(2, +(1, 1));
> true.
```

`in/2` The inclusion of an element (the first argument) in a set (the second argument).

```
formula in(1, join(1, {}));
> true.
```

`<=/2` The predicate on natural numbers that returns true if and only if its first argument is not larger than the second one.

```
formula <=(1, 2);
> true.
```

Likewise the following functions are predefined:

`length/1` The number of elements in a tuple.

```
term length(tuple(2, 3));
> 2.
```

`{}/0` The empty set.

```
term {};
> {}.
```

`join/2` The function that returns the set that results from joining an element (the first argument) to a set (the second argument).

```
term join (2, join(1, {}));
> {1, 2}.
```

`nat/2` The interval of natural numbers from a lower bound (the first argument) up to and including an upper bound (the second argument).

```
term nat(1, 10);
> 1..10.
term set(x in nat(1, 10): true, x);
> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.
```

`+/2` Addition on natural numbers.

```
term +(7, 8);
> 15.
```

`*/2` Multiplication on natural numbers.

```
term *(2, 3);
> 6.
```

`-/2` Subtraction of a natural number (the second argument) from another natural number (the first argument); the result is zero, if the second argument is larger than the first one.

```
term -(7, 8);
> 0.
```

3 Installation and Use

The Logic Evaluator has been developed under Linux with the Sun Java Development Toolkit. It is written in Java 1.1 as a package `Logic` with main class `Main` and is distributed as the Java archive `Logic.jar`.

3.1 Text Mode

Having retrieved `Logic.jar`, unpack it in the current directory or in a directory of the `CLASSPATH` as

```
jar xf Logic.jar
```

and start the evaluator with

```
java Logic.Main
> Logic Evaluator (c) 1999, Wolfgang Schreiner.
> Type 'help;' for help.
```

Execution is terminated by the end of the input stream (which is triggered by pressing the keys `Control+D` on most Unix systems).

Please note that in text mode the commands `plot` and `plots` do *not* work.

3.2 Window Mode

Having retrieved `Logic.jar`, unpack it in the current directory or in a directory of the `CLASSPATH` as

```
jar xf Logic.jar
```

and start the evaluator with

```
java Logic.Main -window parameters
```

A window pops up in which input is entered and the output is displayed. The optional list of *parameters* is the same as for applets where the name of each parameter is prefixed with the token '-', e.g.,

```
java Logic.Main -window -screen true -lines 16
```

Execution is terminated by closing the window (menu entry `Close` in most window managers).

3.3 Applet Mode

The evaluator may be embedded as an applet into an HTML page, e.g., as

```
<applet
  code="Logic/Main.class"
  archive="Logic.jar,Input.jar"
  width=500 height=150>
<param name="exec" value="
read set;\n
fun A = set(x in nat(1, 3): true, x);\n
fun B = set(x in nat(0, 1): true, x);">
<param name="input" value="term x(A, B);">
<param name="screen" value="true">
<param name="swidth" value="50">
<param name="sheight" value="50">
<param name="lines" value="6">
</applet>
```

In this example, the archive `Logic.jar` is located in the same directory as the HTML page embedding the applet. The file `Input.jar` is an archive of files `Logic/*.txt` that contains all files that can be read by the interpreter, e.g., if there exists a file `Logic/set.txt`, we may say in the interpreter

```
read set;
> file 'set.txt' read.
```

The evaluator applet understands the following parameters:

exec A string that represents a command to be executed by the evaluator after the applet has been loaded. The token "\n" may be used to decompose the string into multiple lines.

input A string that appears in the input field of the evaluator after the applet has been loaded.

screen A flag; if this parameter appears with any value, the applet exhibits a screen that allows to use the commands `plot` and `plots`.

swidth If `screen` is given, this parameter denotes the number of (logical) pixels of this screen in the horizontal direction (default 50).

sheight If `screen` is given, this parameter denotes the number of (logical) pixels of this screen in the vertical direction (default 50).

lines If `screen` is given, this parameter denotes the number of lines of the output field of the interpreter (default 6).

Please note that in applet mode the interpreter only executes while the focus is on the input field such that the background is gray. Otherwise, execution of the interpreter is deliberately suspended. This implies, that the interpreter does not execute when the embedding Web page is not visible or the browser is iconified.

3.4 Known Problems

There are no problems known when the evaluator runs in text mode or in window mode or in applet mode with the JDK appletviewer.

As for execution as an applet in a Web browser, we have tested the evaluator with various versions of Netscape Communicator on Linux and Windows and with Microsoft Internet Explorer on Windows. There are several problems caused by Java/AWT bugs in these browsers.

Communicator: No Focus When leaving a page embedding the evaluator applet (by following a link or pressing the buttons `Back` or `Forward`), on return to this page the applet may not be able to gain the focus again.

This seems to be due to an error in AWT. Do not leave the page (use the middle mouse button to open a link in a new frame) or press the keys `Shift+Reload` to reload the applet.

Internet Explorer: Applets Reloaded When leaving a page embedding the evaluator applet (by following a link or pressing the buttons `Back` or `Forward`), the applets are restarted (i.e., they do not preserve their previous state).

This is apparently a feature of the Internet Explorer. Do not leave the page if you want to have the applet state preserved.

Communicator (some versions): Browser Frozen When starting the execution of a command in the evaluator, the browser freezes until the execution of the evaluator has terminated.

This seems to be due to an error in multi-threading. Wait until the corresponding execution has terminated or kill the browser.

Communicator: Console Messages Sometimes error messages appear in the Java console when (re)loading a page that embeds the evaluator.

This behavior seems to be due a Java internal problem; apparently it does not indicate any significant troubles. Ignore the messages.

Communicator and Explorer: Java Bugs Several Java bugs have required workarounds in the evaluator (most notably security violations when using inner classes and wrong handling of end of line conventions).

Since we have found workarounds, one should not encounter these problems any more.

A Sample Files

A.1 `set.txt`

```
// ----------------------------------------------------------------
// $Id: set.txt,v 1.7 1999/07/13 06:39:49 schreine Exp $
// some set-theoretic notions
```

```
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

option silent = true;

// singleton set
fun {}(x) =
  join(x, {});

// A is a subset of B iff every element of A is also in B
pred isSubset(A: Set, B: Set) <=>
  forall(x in A: in(x, B));

// two sets are equal if each is a subset of the other
pred equals(A: Set, B: Set) <=>
  and(isSubset(A, B), isSubset(B, A));

// the intersection of two sets
fun **(A: Set, B: Set) =
  set(x in A: in(x, B), x);

// the difference of two sets
fun --(A: Set, B: Set) =
  set(x in A: not(in(x, B)), x);

// the union of two sets
fun ++(A: Set, B: Set) =
  reduce(join, A, B);

// the product of two sets A and B
fun x(A: Set, B: Set) =
  set(a in A, b in B: true, tuple(a, b));

// cardinality of S is determined by iteration over the elements
fun count(e, i: Nat) = +(i, 1);
fun #(S: Set) = reduce(count, S, 0);

// the union of S and of e joined to all elements of S
fun combine(e, S: Set) =
  ++(S, set(x in S: true, join(e, x)));
```

```
// the set of all subsets of S
fun Powerset(S: Set) =
  reduce(combine, S, {}({}));

// alternative recursive definition
fun PowersetR(S: Set) recursive #(S) =
  if (=(S, {}), join({}, {}),
    let(e = such(x in S: true, x):
      combine(e, Powerset(--(S, {}(e))))));

option silent = false;

// ----------------------------------------------------------------
// $Id: set.txt,v 1.7 1999/07/13 06:39:49 schreine Exp $
// ----------------------------------------------------------------
```

A.2 circle.txt

```
// ----------------------------------------------------------------
// $Id: circle.txt,v 1.4 1999/07/13 06:39:49 schreine Exp $
// circles in the plane
//
// (c) 1999, Wolfgang Schreiner, see file COPYRIGHT
// http://www.risc.uni-linz.ac.at/software/formal
// ----------------------------------------------------------------

option silent = true;

// p is a point
pred Point(p) <=>
  and(Tuple(p), =(2, length(p)), Nat(.0(p)), Nat(.1(p)));

// the center and the image range
fun c = tuple(50, 25);
fun rangeX = nat(-(.0(c), .1(c)), +(.0(c), .1(c)));
fun rangeY = nat(0, *(2, .1(c)));

// the square of the difference of a and b
fun -^2(a: Nat, b: Nat) =
  let (d = if (<=(a, b), -(b, a), -(a, b)):
    *(d, d));
```

```
// approximate equality of a and b
pred ~=(a: Nat, b: Nat) <=>
  let (diff = 25:
    if (<=(a, b), <=(-(b, a), diff), <=(-(a, b), diff)));

// p and q are approximately on same circle
pred ~(p: Point, q: Point) <=>
  let (r = +(-^2(.0(p), .0(c)), -^2(.1(p), .1(c))),
       s = +(-^2(.0(q), .0(c)), -^2(.1(q), .1(c))):
  ~=(r, s));

// the circle that goes through p
fun circle(p: Point) =
  set(x in rangeX, y in rangeY, q = tuple(x, y): ~(p, q), q);

option silent = false;

// ------------------------------------------------------------------
// $Id: circle.txt,v 1.4 1999/07/13 06:39:49 schreine Exp $
// ------------------------------------------------------------------
```