# Ev3: A Library for Symbolic Computation in C++ using $n$-ary Trees

Leo Liberti

*LIX, École Polytechnique, 91128 Palaiseau, France*
(`liberti@lix.polytechnique.fr`)

3 october 2003

**Abstract**

Ev3 is a callable C++ library for performing symbolic computation (calculation of symbolic derivatives and various expression simplification). The purpose of this library is to furnish a fast means to use symbolic derivatives to third-party scientific software (e.g. nonlinear optimization, solution of nonlinear equations). It is small, easy to interface, even reasonably easy to change; it is written in C++ and the source code is available. One feature that makes Ev3 very efficient in algebraic manipulation is that the data structures are based on $n$-ary trees.

## 1 Introduction

Computer Algebra Systems (CAS) have helped mathematicians, physicists, engineers and other scientists enormously since their appearance. CASes are extremely useful in performing complex symbolic calculations quickly. They do not generally require particular programming skills, yet they manage to carry out numerical analysis, symbolic manipulation and graph plotting in 2 and 3 dimensions. The range of existing CASes is impressive, both commercially available (e.g. Mathematica, Maple, Matlab, Mathcad, Reduce, Mupad etc.) and as free software (e.g. Maxima, GAP, yacas, Singular, GiNaC etc.); however most of them (with the notable exception of GiNaC, see below) fall short on the external Application Programming Interface (API), i.e. it is difficult, or impossible in many cases, to create a program in a compiled language, say in C/C++ or Fortran, which can use the external CAS facilities. This is a serious hindrance for user applications with programming requirements that go beyond those offered by the CAS proprietary language, or for those applications that need to be coded in a compiled language for efficiency, but would still benefit from the CAS facilities. The need for CAS-type libraries designed to be used from an API arises as the complexity or the commercial viability of the application gets higher. Thus, having experienced the benefits of CASes from the command line, so to speak, users now want to incorporate these functionalities in their own programs.

This paper presents Ev3, a callable C++ library designed to perform simple symbolic manipulation. "Simple" in this context means symbolic differentiation and various expression simplifications. Thus, Ev3 is very useful for those scientific applications that need to compute derivatives, like for example continuous nonlinear programming or nonlinear equation solving. The reason why we have chosen to keep Ev3 "simple", i.e. to limit the range of symbolic algorithms included in Ev3, is that we expect advanced users to take the source code and modify it to suit their own needs. For example, if an application needs a symbolic integration routine, Ev3's data structures and basic methods can be used as a starting point. We did our best in keeping Ev3's source code small, readable, well structured and easily extendable. Ev3 can be downloaded from `http://www.lix.polytechnique.fr/ liberti/Ev3-1.0.tar.gz`.

Ev3 records mathematical expressions in a tree-like data structure (called *n-ary tree*) where each node has an arbitrary number of direct subnodes. Monomials of all degrees (including constants) are represented by the leaf nodes of the tree, and mathematical operators by the intermediate nodes. Entire subexpressions can be easily changed, substituted or deleted. New expressions can be formed in a number of different ways. A "standard form" for expressions is defined in order to make simplification more effective.

These are some of Ev3's features:

- based on $n$-ary trees;
- every node in a tree can have a numeric multiplicative coefficient;
- every leaf node in a tree can have a numeric exponent;
- basic form of garbage collection via reference counter;
- basic parser transforms strings in $n$-ary trees;
- basic functionality: function evaluation, expression simplification, symbolic differentiation.

The rest of this paper is organized as follows. In section 2 we review the current situation as regards some existing CASes and their ability to be used as a callable library for external applications. In section 3 we describe the most common data structures used for symbolic computation (namely, lists and binary and $n$-ary trees). In section 4 we describe Ev3's object-oriented software architecture and implementation of the abstract data structure describing the $n$-ary tree. Finally, section 5 is a tutorial on the use of Ev3.

## 2 Review of Current Situation

As has been mentioned in the introduction, it is difficult, or impossible in many cases, to use CAS functionality within a C++ program. In this section we describe the current situation with some existing symbolic computation packages.

### 2.1 Commercial CASes

This list encompasses some of the most widespread commercial CASes. Although some of these CASes are distributed for free for non-commercial purposes (e.g. MuPad), they appear in this list because they do not abide by the "free software" rules involving distribution of the source code and copyrighting via the GNU Public License (GPL).

- **Mathematica**. Mathematica offers some support for external programs wanting to use its features via the *MathLink* [Mathematica(2001)] mechanism. However, a kernel needs to be running on the local machine in order to use it. This has a number of disadvantages:
  - the produced executables are not portable, for they need to find a running Mathematica kernel in order to run;
  - there is some computational overhead involved in runtime linking with a running Mathematica kernel;
  - there is no significant computational speed to be gained from this exercise, as all the Mathematica instructions are carried out by the Mathematica kernel, not by the user executable.
- **Maple**. Maple [Maple(2001)] is capable of translating its procedures and functions in C or Fortran, however it is not possible to export symbolic manipulation code, and there is no way to call a Maple library from C/C++/Fortran.

- **Mathlab**. Strictly speaking, MatLab does not do symbolic computation. However there are some symbolic computation extensions available for MatLab [Matlab(2001)] so it makes sense to include it in this list. MatLab external API works in much the same was as Mathematica's does: on initialization, the external C/C++/Fortran program spawns a running MatLab process on the local machine, and sets up input/output pipes from it to the program [Matlab API(2001)]. The same comments given above for Mathematica also apply here.

- **Mathcad**. Unfortunately, publically available documentation about MathCad is scarce. MathCad can link to Microsoft-compliant dynamic link libraries (DLL) at runtime, and it would seem like it is also possible to do the reverse, i.e. use an external C++ driver program to use MathCad facilities; however the exact mechanism by which this is implemented seems to be a proprietary piece of information, and hence we cannot comment on it.

- **Reduce**. The commercial distribution of Reduce includes the full source code (in standard Lisp), so it would seem feasible to try to compile the Lisp source code to an object file that would then be linked against a C/C++/Fortran driver program; however, this would fall into a "hacking trick" category more than into established use; moreover, this is true for each CAS that comes with source code. In practice, Reduce has no native mechanism for allowing easy linking with C/C++/Fortran programs, although it does have a facility for writing out its functions/procedure to C/Fortran code (with no symbolic computation).

- **MuPad**. MuPad allows linking of external compiled binary modules via an API, however it is not possible to do the reverse, i.e. use MuPad's facilities within an external C/C++/Fortran driver program [MuPad(2001), MuPad Documentation(2001)]. MuPad can export its functions/procedures to C/Fortran code (with no symbolic computation).

## 2.2 Free Software CASes

This list comprises some of the most widespread general-purpose "free software" CASes. In all cases, the source code is available. Packages like GAP [GAP(2001)] or Singular [Singular(2001)] do not appear here because they are too specific in nature. SciLab [SciLab(2001)] does not appear because although it offers some runtime dynamic linking from C/C++/Fortran compiled code, its symbolic capabilities are only present when interfacing through Maple, i.e. symbolic computation within SciLab is actually carried out by Maple. Octave [Octave(2001)], like SciLab, does not have built-in symbolic computation facilities, although it does have a library that external program can link against.

- **Maxima**. Maxima [Maxima(2001)] is a version of the MIT-developed MACSYMA system, modifed to run under GNU Common LISP. It is a wholly interactive system, and it has no support for external linking.

- **JACAL**. JACAL [JACAL(2001)] is a symbolic mathematics system for the simplification and manipulation of equations and single and multiple valued algebraic expressions, written in Scheme. It has no support for external linking.

- **yacas**. Yacas (Yet Another CAS, [yacas(2001)]) is a small computer algebra system based on a lisp interpreter. It can run in "web server mode", so that any client can connect remotely and perform calculations. Data are passed back and forth via the `http` protocol. In this sense, it would be possible to implement a C/C++/Fortran client code. However, no local binary linking is possible.

- **GiNaC**. The acronym stands for *GiNaC is Not a CAS* [GiNaC(2001)], and indeed this package keeps its promise. It is a library for doing symbolic computation (see [BFK01]); it is coded in C++ and designed to be used from C++. Symbolic computation performed by GiNaC extends to symbolic manipulation and symbolic differentiation, but it does not include symbolic integration or an algorithm for automatic simplification of symbolic expression (the authors argue that there is no such thing as a "standard for simplifying expressions", so the way to carry it out is best left to the user). GiNaC is not limited to symbolic computation, as it is based on the CLN library for arbitrary size numerical computation. Furthermore, matrices, vectors, tensors can all be formed as arrays of symbolic expressions indexed by virtually any type of object, be it numeric or symbolic (it is even possible to index arrays by symbolic expressions, although the authors warn that it may be of little use).

As has been shown in the last two sections, embedding symbolic computation in a compiled language (C++) is only possible using GiNaC. However, as has been said, Ev3 only implements symbolic differentiation and simplification routines and is easy to extend; GiNaC, on the other hand, implements a lot of algorithms (not all of these of a symbolic computational nature) *except* for expression simplification ones, and, consisting of a large source code base, is difficult to extend and adapt. In other words, GiNaC is more useful as a finished product than as a starting point for new symbolic algorithms. Whereas Ev3 only consists of 7 source files for a total of around 6000 lines of code (including comments and headers), against GiNaC's nearly 37000 lines of code shared in over 90 source files.

## 3 Data Structures for Symbolic Computation

Symbolic computation relies on a machine representation of mathematical operations on some numbers or literal symbols (constants, variables, or expressions involving constants and variables). Usually, one of the following techniques is employed to represent these operations:

- binary trees;
- lists;
- $n$-ary trees.

### 3.1 Binary Trees

Binary trees have been proposed as a way of representing mathematical expressions by Knuth (see [Knu81]) and made their way in computational engineering and other fields of scientific computing (just to cite an example, see [BP88]). This representation is based on the idea that operators, variables and constants are nodes of a digraph; binary operators have two outcoming edges and unary operators have only one; leaf nodes have no outcoming edge (for graph-related terminology and definitions, see [Har71], [KV00]). One disadvantage is that binary tree representation makes it cumbersome to implement associativity. For example, the expression $y + x + 2x + 3x$ is represented as $(((y + x) + 2x) + 3x)$, so it would require three recursive steps to lower tree ranks to find out that it is possible to write it as $(y + 6x)$. Another disadvantage is that different parsers may have different representations for the same expressions. With the example above, a "left-hand-side-first" parser would create $(y + (x + (2x + 3x)))$ instead of the "right-hand-side-first" $(((y + x) + 2x) + 3x)$.

Octave's internal algebraic parser is based on binary trees, as can be seen in the file `pt-exp.h`, inside the `src` subdirectory of the octave source distribution directory. One reason for this is that Octave has no symbolic manipulation abilities, thus its trees are mostly used for fast evaluation of mathematical expressions.

Where symbolic manipulation is only desired to compute symbolic derivatives and performing little or no symbolic manipulation, this approach may be the best, as it is simpler to implement than the other techniques and generally performs very efficiently ([Pan88], [LTKP01]).

## 3.2 Lists

The representation of algebraic expressions by lists dates back to the AI-type languages Prolog and Lisp. Lisp, in particular, was so successful at the task that a lot of CASes, today, are still based on Lisp's list manipulation abilities (e.g. Maxima, JACAL, yacas, see above). Prolog has some interesting features in conjunction with symbolic computation, in particular the "computation-reversing" ability, by which if you compute a symbolic derivative and do not bother to simplify it, Prolog lets you integrate it simbolically performing virtually no calculation at all.

Any CAS library written in Prolog/Lisp faces the hard problem of implementing an API which can be used by procedural languages like Fortran, C or C++. Whilst technically not impossible, the architectures and OSes offering stable and compatible Lisp and C/C++ compilers are few. GNU/Linux actually has object-compatible Lisp and C/C++ compilers; however, the GNU Lisp compiler uses an array of internal data structures which are very difficult to read from a C/C++ program, making data interchange between the different modules very hard to implement.

There are two other problems faced by Prolog/Lisp programs: portability (many Prolog/Lisp compilers implement different dialects of the languages) and a reduced user base.

## 3.3 $n$-ary Trees

Expression representation by $n$-ary trees can be seen as a combination of the previous two techniques. Both GiNaC and Ev3 make use of this representation, albeit in slightly different ways. In order to characterize this representation formally, we need some definitions.

An *operator* is a node in a directed tree-like graph. Let $L$ be the set

$$\{+, -, \times, /, \hat{}, (-1)\times, \log, \exp, \sin, \cos, \tan, \cot, \text{VAR}, \text{CONST}\}$$

of *operator labels*. An operator with label VAR is a *variable*, an operator with label CONST is a *constant*. Operator nodes may generally have any number of outcoming edges; variables and constants have no outcoming edges and are called *leaf nodes*. A variable is also characterized by a non-negative integer index $i$, and a constant by a value which is an element of a number field $F$. We shall assume $F = \mathbb{R}$ (or at least, a machine representation of $\mathbb{R}$) in what follows, but this can vary (e.g. because GiNaC is based on the CLN library, it can implement complex numbers as well). Let $V$ be the set of all variable-type operator nodes. Let $T_0 = V \cup \mathbb{R}$. This is the set of the terminal (or leaf) nodes, i.e. the variables and constants. Now for each positive integer $i$, define recursively $T_i = L \times (T_{i-1} \cup T_0)^{<\omega}$. Elements of $T_i$ are operator nodes having *rank* $i$. Basically, an element of $T_i$ is made up of an operator label $l \in L$ and a finite number of subnodes. A *subnode* $s$ of $n$ is a node $s$ in the digraph so that there is an edge leaving node $n$ and entering $s$.

The biggest advantage of $n$-ary tree representation is that it makes it very fast and easy to perform expression simplification. Another advantage is that expression evaluation on $n$-ary trees is faster than that obtained with a binary tree structure [Lib03].

# 4 Ev3 Architecture

Ev3 software architecture is based on 5 classes. Two of them, `Tree` and `Pointer`, are generic templates that provide the basic tree structure and a no-frills garbage collection based on reference count. Each object has a reference counter which increases every time a reference of that object is taken; the object destructor decreases the counter while it is positive, only actually deleting the object when the counter reaches zero. This type of garbage collecting is due to Collins, 1960 (see [Kal00]). Other two classes, `Operand` and `BasicExpression`, implement the actual semantics of an algebraic expression. The last class, `ExpressionParser`, implements a simple parser (based on the ideas given in [Str99]) which reads in a string containing a valid mathematical expression and produces the corresponding $n$-ary tree.

Notice that Ev3 uses some of the STL generic container classes (like, e.g., `vector`).

## 4.1 The `Pointer` Class

This is a template class defined as

```
template<class NodeType> class Pointer {
  NodeType* node;
  int* ncount;
  // methods
};
```

The constructor of this class allocates a new integer for the reference counter `ncount` and a new `NodeType` object, and the copy constructor increases the counter. The destructor deletes the reference counter and invokes the delete method on the `NodeType` object. In order to access the data and methods of the `NodeType` object pointed to by `node`, the `->` operator in the `Pointer` class is overloaded to return `node`.

A mathematical expression, in Ev3, is defined as a *pointer to a* `BasicExpression` *object* (see below for the definition of a `BasicExpression` object):

```
typedef Pointer<BasicExpression> Expression;
```

## 4.2 The `Tree` Class

This is a template class defined as

```
template<class NodeType> class Tree {
  vector<Pointer<NodeType> > nodes;
  // methods
};
```

This is the class implementing the $n$-ary tree (subnodes are contained in the `nodes` vector). Notice that, being a template, the whole implementation is kept independent of the semantics of a `NodeType`. Notice also that because pointers to objects are pushed on the vector, algebraic substitution is very easy: just replace one pointer with another one. This differs from the implementation of GiNaC ([GiNaC Group(2001)], p. 36) where it appears that algebraic substitution is a more convoluted operation.

## 4.3 The `Operand` Class

This class holds the information relative to each expression term, be they constants, variables or operators.

```
class Operand {
  int oplabel;          // operator label
  double value;         // if constant, value of constant
  long varindex;        // if variable, the variable index
```

```
    string varname;      // if variable, the variable name
    double coefficient;  // terms can be multiplied by a number
    double exponent;     // leaf terms can be raised to a number
    // methods
  };
```

Notice that:

- `oplabel` can be one of the following labels (the meaning of which should be clear):

  ```
  enum OperatorType {
    SUM, DIFFERENCE, PRODUCT, FRACTION, POWER,
    PLUS, MINUS, LOG, EXP, SIN,
    COS, TAN, COT, SINH, COSH,
    TANH, COTH, SQRT, VAR, CONST,
    ERROR
  };
  ```

- `value`, the value of a constant numeric term, only has meaning if `oplabel` is `CONST`;
- `varindex`, the variable index, only has meaning if `oplabel` is `VAR`;
- every term, (variables, constants and operators), can be multiplied by a numeric coefficient. This makes it easy to perform symbolic manipulation on like terms (e.g. $x + 2x = 3x$).
- every leaf term (variables and constants) can be raised to a numeric power. This makes it easy to perform symbolic manipulation of polynomials.

Introducing numeric coefficients and exponents is a choice that has advantages as well as disadvantages. GiNaC, for example, does not explicitely account for numeric coefficients. The advantages are obvious: it makes symbolic manipulation very efficient for certain classes of basic operations (operations on like terms). The disadvantage is that the programmer has to explicitely account for the case where terms are assigned coefficients: whereas with a pure tree structure recursive algorithms can be formulated as "for each node, do something", this becomes more complex when numeric coefficients are introduced. Checks for non-zero or non-identity have to be performed prior to carrying out certain operations, as well as having to manually account for cases where coefficients have to be used. However, by setting both multiplicative and exponent coefficients to 1, the mechanism can to a certain extent be ignored and a pure tree structure can be recovered.

## 4.4 The `BasicExpression` Class

This class is defined as follows:

```
class BasicExpression :
public Operand, public Tree<BasicExpression> {
  // methods
};
```

It includes no data of its own, but it inherits its semantic data from class `Operand` and its tree structure from template class `Tree` with itself (`BasicExpression`) as a base type. This gives `BasicExpression` an $n$-ary tree structure.

Note that an object of class `BasicExpression` is *not* a `Pointer`, only its subnodes (if any) are stored as `Pointer`s to other `BasicExpression`s. This is the reason why the client code should never explicitely use `BasicExpression`; instead, it should use objects `Expression`, which are defined as `Pointer<BasicExpression>`. This allows the automatic garbage collector embedded in `Pointer` to work.

## 4.5 The `ExpressionParser` Class

This parser originates from the example parser found in [Str99]. The original code has been extensively modified to support:

- exponentiation;
- unary functions in the form $f(x)$;
- creation of $n$-ary trees of type `Expression`.

For an example of usage, see section 5 below.

## 4.6 Application Programming Interface

The Ev3 API consists in a number of *internal methods* (i.e., methods belonging to classes) and *external methods* (functions whose declaration is outside the classes). Objects of type `class Expression` can be built from strings containing infix-format expressions (like, e.g. `"log(2*x*y)+sin(z)"`) by using the built-in parser. However, they may also be built from scratch using the supplied construction methods (see section 5 for examples).

Since the fundamental type `Expression` is an alias for `Pointer<BasicExpression>`, and `BasicExpression` is in turn a mix of different classes (including a `Tree` with itself as a template type), calling internal methods of an `Expression` object may be confusing. Thus, for each class name involved in the definition of `Expression`, we have listed the calling procedure explicitly.

For further explanations about these methods, consult the Ev3 source files.

**Class `Operand`.** Call: `ret = (Expression e)->MethodName(args)`.

| Method name | Purpose |
|---|---|
| int GetOpType(void) | returns the operator label |
| double GetValue(void) | returns the value of the constant leaf (takes multiplicative coefficient and exponent into account) |
| double GetSimpleValue(void) | returns the value (takes no notice of coefficient and exponent) |
| long GetVarIndex(void) | returns the variable index of the variable leaf |
| string GetVarName(void) | returns the name of the variable leaf |
| double GetCoeff(void) | returns the value of the multiplicative coefficient |
| double GetExponent(void) | returns the value of the exponent (for leaves) |
| void SetOpType(int) | sets the operator label |
| void SetValue(double) | sets the numeric value of the constant leaf |
| void SetVarIndex(long) | sets the variable index of the variable leaf |
| void SetVarName(string) | sets the name of the variable leaf |
| void SetExponent(double) | sets the exponent (for leaves) |
| void SetCoeff(double) | sets the multiplicative coefficient |
| bool IsConstant(void) | is the node a constant? |
| bool IsVariable(void) | is the node a variable? |
| bool IsLeaf(void) | is the node a leaf? |
| bool HasValue(double v) | is the node a constant with value $v$? |
| bool IsLessThan(double v) | is the node a constant with value $\leq v$? |
| void ConsolidateValue(void) | set value to `coeff*value*exponent` and set `coeff` to 1 and `exponent` to 1 |
| void SubstituteVariableWithConstant(long int varindex, double c) | substitute a variable with a constant `c` |

**Template Class** `Pointer<NodeType>`. Call: `ret = (Expression e).MethodName(args)`.

| METHOD NAME | PURPOSE |
|---|---|
| Pointer<NodeType> Copy(void) | returns a copy of this node |
| void SetTo(Pointer<NodeType>& t) | this is a reference of t |
| void SetToCopyOf(Pointer<NodeType>& t) | this is a copy of t |
| Pointer<NodeType> operator=(Pointer<NodeType> t) | assigns a reference of t to this |
| void Destroy(void) | destroys the node (collects garbage) |

**Template Class** `Tree<NodeType>`. Call: `ret = (Expression e)->MethodName(args)`.

| METHOD NAME | PURPOSE |
|---|---|
| void AddNode(Pointer<NodeType>) | pushes a node at the end of the node vector |
| void AddCopyOfNode(Pointer<NodeType> n) | pushes a copy of node $n$ at the end of the node vector |
| bool DeleteNode(long i) | deletes the $i$-th node, returns true if successful |
| void DeleteAllNodes(void) | empties the node vector |
| Pointer<NodeType> GetNode(long i) | returns a reference to the $i$-th subnode |
| Pointer<NodeType> $*$ GetNodeRef(long i) | returns a pointer to the $i$-th subnode |
| Pointer<NodeType> GetCopyOfNode(long i) | returns a copy of the $i$-th subnode |
| long GetSize(void) | returns the length of the node vector |

**Class** `BasicExpression` (inherits from `Operand`, `Tree<BasicExpression>`).
Call: `ret = (Expression e)->MethodName(args)`.

| METHOD NAME | PURPOSE |
|---|---|
| string ToString(void) | returns a string with the expression in infix notation |
| void Zero(void) | sets this to zero |
| void One(void) | sets this to one |
| bool IsEqualTo(Expression&) | is this equal to the argument? |
| bool IsEqualToNoCoeff(Expression&) | [like above, ignoring multiplicative coefficient] |
| int NumberOfVariables(void) | number of variables in the expression |
| double Eval(double* v, long vsize) | evaluate; `v[i]` contains the value for variable with index $i$, `v` has length `vsize` |
| bool DependsOnVariable(long i) | does this depend on variable $i$? |
| int DependsLinearlyOnVariable(long i) | does this depend linearly on variable $i$? (0=depends nonlinearly, 1=linearly, 2=no dependency) |
| void ConsolidateProductCoeffs(void) | if node is a product, move product of all coefficients as coefficient of node |
| void DistributeCoeffOverSum(void) | if coeff. of a sum operand is not 1, distribute it over the summands |
| void VariableToConstant(long varindex, double c) | substitute a variable with a constant `c` |
| void ReplaceVariable(long vi1, long vi2, string vn2) | replace occurrences of variable `vi1` with variable `vi2` having name `vn2` |
| string FindVariableName(long vi) | find name of variable `vi` |
| bool IsLinear(void) | is this expression linear? |
| bool GetLinearInfo(...) | returns info about the linear part |
| Expression Get[Pure]LinearPart(void) | returns the linear part |
| Expression Get[Pure]NonlinearPart(void) | returns the nonlinear part |
| double RemoveAdditiveConstant(void) | returns any additive constant and removes it |
| void Interval(...) | performs interval arithmetics on the expression |

**Class** `ExpressionParser`.

| METHOD NAME | PURPOSE |
|---|---|
| void SetVariableID(string x, long i) | assign index $i$ to variable $x$; var. indices start from 1 and increase by 1 |
| long GetVariableID(string x) | return index of variable $x$ |
| Expression Parse(char* buf, int& errors) | parse `buf` and return an `Expression` `errors` is the number of parsing errors occcurred |

**Methods outside classes.**

| Method name | Purpose |
|---|---|
| Expression operator+(Expression a, Expression b) | returns symbolic sum of $a, b$ |
| Expression operator-(Expression a, Expression b) | returns symbolic difference of $a, b$ |
| Expression operator*(Expression a, Expression b) | returns symbolic product of $a, b$ |
| Expression operator/(Expression a, Expression b) | returns symbolic fraction of $a, b$ |
| Expression operatorˆ(Expression a, Expression b) | returns symbolic power of $a, b$ |
| Expression operator-(Expression a) | returns symbolic form of $-a$ |
| Expression Log(Expression a) | returns symbolic $\log(a)$ |
| Expression Exp(Expression a) | returns symbolic $\exp(a)$ |
| Expression Sin(Expression a) | returns symbolic $\sin(a)$ |
| Expression Cos(Expression a) | returns symbolic $\cos(a)$ |
| Expression Tan(Expression a) | returns symbolic $\tan(a)$ |
| Expression Sinh(Expression a) | returns symbolic $\sinh(a)$ |
| Expression Cosh(Expression a) | returns symbolic $\cosh(a)$ |
| Expression Tanh(Expression a) | returns symbolic $\tanh(a)$ |
| Expression Coth(Expression a) | returns symbolic $\coth(a)$ |
| Expression SumLink(Expression a, Expression b) | returns symbolic sum of $a, b$ |
| Expression DifferenceLink(Expression a, Expression b) | returns symbolic difference of $a, b$ |
| Expression ProductLink(Expression a, Expression b) | returns symbolic product of $a, b$ |
| Expression FractionLink(Expression a, Expression b) | returns symbolic fraction of $a, b$ |
| Expression PowerLink(Expression a, Expression b) | returns symbolic power of $a, b$ |
| Expression MinusLink(Expression a) | returns symbolic form of $-a$ |
| Expression LogLink(Expression a) | returns symbolic $\log(a)$ |
| Expression ExpLink(Expression a) | returns symbolic $\exp(a)$ |
| Expression SinLink(Expression a) | returns symbolic $\sin(a)$ |
| Expression CosLink(Expression a) | returns symbolic $\cos(a)$ |
| Expression TanLink(Expression a) | returns symbolic $\tan(a)$ |
| Expression SinhLink(Expression a) | returns symbolic $\sinh(a)$ |
| Expression CoshLink(Expression a) | returns symbolic $\cosh(a)$ |
| Expression TanhLink(Expression a) | returns symbolic $\tanh(a)$ |
| Expression CothLink(Expression a) | returns symbolic $\coth(a)$ |
| Expression Diff(const Expression& a, long i) | returns derivative of $a$ w.r.t variable $i$ |
| Expression DiffNoSimplify(const Expression& a, long i) | returns unsimplified derivative of $a$ w.r.t variable $i$ |
| bool Simplify(Expression* a) | apply all simplification rules |
| Expression SimplifyCopy(Expression* a, bool& has_changed) | simplify a copy of the expression |
| void RecursiveDestroy(Expression* a) | destroys the whole tree and all nodes |

## Notes

- The lists given above only include the most important methods. For the complete lists, see the files `expression.h`, `tree.cxx`, `parser.h` in the source code distribution.
- There exist a considerable number of different constructors for `Expression`. See their purpose and syntax in files `expression.h`, `tree.cxx`. See examples of their usage in file `expression.cxx`.
- Internal class methods usually return or set atomic information inside the object, or perform limited symbolic manipulation. Construction and extended manipulation of symbolic expressions have been confined to external methods. Furthermore, external methods may have any of the following characteristics:
  - they combine *references* of their arguments;
  - they may change their arguments;
  - they may change the order of the subnodes where the operations are commutative;
  - they may return one of the arguments.

  Thus, it is advisable to perform the operations on copies of the arguments when the expression being built is required to be independent of its subnodes. In particular, all the expression building functions (e.g. `operator+()`, . . . , `Log()`, . . . ) do *not* change their arguments, whereas their `-Link` counterparts do.

- The built-in parser (`ExpressionParser`) uses linking and not copying (also see section 4.7) of nodes when building up the expression.
- The symbolic derivative routine `Diff()` uses copying and not linking of nodes when building up the derivative.
- The method `BasicExpression::IsEqualToNoCoeff()` returns true if two expressions are equal apart from the multiplicative coefficient of the root node only. I.e., $2(x + y)$ would be deemed "equal" to $x + y$ (if 2 is a multiplicative coefficient, *not* an operand in a product) but $x + 2y$ would *not* be deemed "equal" to $x + y$.
- The `Simplify()` method applies all simplification rules known to Ev3 to the expression and puts it in standard form.
- The methods `GetLinearInfo()`, `GetLinearPart()`, `GetPureLinearPart()`, `GetNonlinearPart()`, `GetPureNonlinearPart()` return various types of linear and nonlinear information from the expression. Details concerning these methods can be found in the Ev3 source code files `expression.h`, `expression.cxx`.
- The method `Interval()` performs interval arithmetic on the expression. Details concerning this method can be found in the Ev3 source code files `expression.h`, `expression.cxx`.
- Variables are identified by a variable index, but they also know their variable name. Variable indices are usually assigned within the `ExpressionParser` object, with the `SetVariableID()` method. It is important that variable indices should start from 1 and increase monotonically by 1, as variable indices are used to index the array of values passed to the `Eval()` method.

## 4.7 Copying vs. Linking

One thing that is immediately noticeable is that this architecture gives a very fine-grained control over the construction of expressions. Subnodes can be copied or "linked" (i.e., a reference to the object is put in place, instead of a copy of the object — this automatically uses the garbage collection mechanism, so the client code does not need to worry about these details). Copying an expression tree entails a set of advantages/disadvantages compared to linking. When an expression is constructed by means of a copy to some other existing expression tree, the two expressions are thereafter completely independent. Manipulation one expression does not change the other. This is the required behaviour in many cases. The symbolic differentiation routine has been designed using copies because a derivative, in general, exists independently of its integral.

Linking, however, allows for things like "cascaded simplification", where some symbolic manipulation on an expression changes all the expressions having the manipulated expression tree as a subnode. This may be useful but calls for extra care. The built-in parser has been designed using linking because the "building blocks" of a parsed expression (i.e. its subnodes of all ranks) will not be used independently outside the parser.

## 4.8 Simplification Strategy

The routine for simplifying an expression repeatedly calls a set of simplification rules acting on the expression. These rules are applied to the expression as long as at least one of them manages to further simplify it.

Simplifications can be *horizontal*, meaning that they are carried out on the same list of subnodes (like e.g. $x + y + y = x + 2y$), or *vertical*, meaning that the simplification involves changing of node level (like e.g. application of associativity: $((x + y) + z) = (x + y + z)$).

The order of the simplification rules applied to an object `Expression e` is the following:

1. `e->ConsolidateProductCoeffs()`: in a product having $n$ subnodes, collect all multiplicative coefficients, multiply them together, and set the result as the multiplicative coefficient of the whole product:
$$\prod_{i=1}^{n}(c_i f_i) = (\prod_{i=1}^{n} c_i)(\prod_{i=1}^{n} f_i).$$
2. `e->DistributeCoeffOverSum()`: in a sum with $n$ subnodes and a non-unit multiplicative coefficient, distribute this coefficient over all subnodes in the sum:
$$c\sum_{i=1}^{n} f_i = \sum_{i=1}^{n} cf_i.$$
3. `DifferenceToSum(e)`: replace all differences and unary minus with sums, multiplying the coefficient of the operands by `-1`.
4. `SimplifyConstant(e)`: simplify operations on constant terms by replacing the `value` of the node with the result of the operation.
5. `CompactProducts(e)`: associate products; e.g. $((xy)z) = (xyz)$.
6. `CompactLinearPart(e)`: this is a composite simplification consisting of the following routines:
   (a) `CompactLinearPartRecursive(e)`: recursively search all sums in the expression and perform horizontal and vertical simplifications on the coefficients of like terms.
   (b) `ReorderNodes(e)`: puts each list of subnodes in an expression in *standard form*:
   $$\textit{constant} + \textit{monomials in rising degree} + \textit{complicated operands}$$
   (where *complicated operands* are sublists of subnodes).
7. `SimplifyRecursive(e)`: deals with the most common simplification rules, i.e.:
   - try to simplify like terms in fractions where numerator and denominator are both products;
   - $x \pm 0 = 0 + x = x$;
   - $x \times 1 = 1 \times x = x$;
   - $x \times 0 = 0 \times x = 0$;
   - $x^0 = 1$;
   - $x^1 = x$;
   - $0^x = 0$;
   - $1^x = 1$.

## 4.9 Differentiation

Derivative rules are the usual ones; the rule for multiplication is expressed in a way that allows for $n$-ary trees to be derived correctly:
$$\frac{\partial}{\partial x}\prod_{i=1}^{n} f_i = \sum_{i=1}^{n}\left(\frac{\partial f_i}{\partial x}\prod_{j\neq i} f_j\right).$$

### 4.10 Files Organization

Ev3 is organized in seven source files:

- `expression.h`: main header file; it contains the declaration for classes `Operand` and `Basic-Expression`, and for all the external symbolic manipulation functions;
- `auxiliary.h`: header file containing declarations for a few auxiliary functions;
- `expression.cxx`: main source file: it contains the definition of all methods (internal and external) relative to classes `Operand` and `BasicExpression`, as well as the definitions of the auxiliary functions;
- `tree.cxx`: source file containing declarations and definitions for class templates `Pointer` and `Tree`. This file, although it has a `.cxx` extension, is included in `expression.h`;
- `parser.h`: header file for the string to $n$-ary tree expression parser;
- `parser.cxx`: source file for the string to $n$-ary tree expression parser.

New user-defined symbolic manipulation functions should be added in files `expression.h` (declaration) and `expression.cxx` (definition).

### 4.11 Improvements to Ev3

The possible improvements to Ev3 are countless. Besides adding more functionality to the library, which would be useful in a way but would otherwise make it more complicated for people to adapt it to their needs, the following issues need to be addressed.

1. Testing. However much one tests a software, it is always likely that bugs will crop up.
2. A consistent error reporting mechanism based on exceptions. Some work is under way in this sense.
3. Linking with some arbitrary-length arithmetic library.

## 5 Tutorial

The example in this section explains the usage of the methods which represent the core, high-level functionality of Ev3: fast evaluation, symbolic simplification and differentiation of mathematical expressions.

The following C++ code is a simple driver program that uses the Ev3 library. Its instructions should be self-explanatory. First, we create a "parser object" of type `ExpressionParser`. We then set the mapping variable names / variable indices, and we parse a string containing the mathematical expression $\log(2xy) + sin(z)$. We print the expression, evaluate it at the point $(2, 3, 1)$, and finally calculate its symbolic derivatives w.r.t. $x$, $y$, $z$, and print them.

```
#include "expression.h"
#include "parser.h"
int main(int argc, char** argv) {
  ExpressionParser p;      // create the parser object
  p.SetVariableID("x", 1) // map between symbols and variable indices
  p.SetVariableID("y", 2) // x --> 0, y --> 1, z --> 2
  p.SetVariableID("z", 3)
  int parsererrors = 0;   // number of parser errors
  /* call the parser's Parse method, which returns an Expression
     which is then used to initialize Expression e     */
  Expression e(p.Parse("log(2*x*y)+sin(z)", parsererrors));
  cout << "parsing errors: " << parsererrors << endl;
  cout << "f = " << e->ToString() << endl; // print the expression
  double val[3] = {2, 3, 1};
  cout << "eval(2,3,1):   " << e->Eval(val, 3) << endl; // evaluate the expr.
  cout << "numeric check: " << ::log(2*2*3)+::sin(1) << endl; // check result
  // test diff
  Expression de1 = Diff(e, 1);  // calculate derivative w.r.t. x
  cout << "df/dx = " << de1->ToString() << endl; // print derivative
  Expression de2 = Diff(e, 2);  // calculate derivative w.r.t. y
  cout << "df/dy = " << de2->ToString() << endl; // print derivative
  Expression de3 = Diff(e, 3);  // calculate derivative w.r.t. z
  cout << "df/dz = " << de3->ToString() << endl; // print derivative
  return 0;
}
```

The corresponding output is

```
parsing errors: 0
f = (log((2*x)*(y)))+(sin(z))
eval(2,3,1):   3.32638
numeric check: 3.32638
df/dx = (1)/(x)
df/dy = (1)/(y)
df/dz = cos(z)
```

**Notes**

- In order to evaluate a mathematical expression $f(x_1, x_2, \ldots, x_n)$, where $x_i$ are the variables and $i$ are the variable indices (starting from 1 and increasing by 1), we use the `Eval()` internal method, whose complete declaration is as follows:

  ```
  double Expression::Eval(double* varvalues, int size) const;
  ```

  The array of doubles `varvalues` contains `size` real constants, where `size` $>= n$. The variable indices are used to address this array (the value assigned to $x_i$ during the evaluation is `varvalues[i-1]`), so it is important that the order of the constants in `varvalues` reflects the order of the variables. This method does not change the `expression` object being evaluated.

- The core simplification method is an external method with declaration

  ```
  bool Simplify(Expression* e);
  ```

  It consists of a number of different simplifications, as explained in section 4.8. It takes a *pointer* to `Expression` as an argument, and it returns `true` if some simplification has taken place, and `false` otherwise. This method changes its input argument.

- The symbolic differentiation procedure is an external method:

  ```
  Expression Diff(const Expression& e, int varindex);
  ```

  It returns a simplified expression which is the derivative of the expression in the argument with respect to variable `varindex`. This method does not change its input arguments.

- External class methods take `Expressions` as their arguments. According as to whether they need to change their input argument or not, the `Expression` is passed by value, by reference, or as a pointer. This may be a little confusing at first, especially when using the overloaded `->` operator on `Expression` objects. Consider an `Expression e` object and a pointer `Expression* ePtr = &e`. The following calls are possibile:
  - `e->MethodName(args); (*ePtr)->MethodName(args);`
    Call a method in the `BasicExpression`, `Operand` or `Tree<>` classes.
  - `e.MethodName(args); (*ePtr).MethodName(args); ePtr->MethodName(args);`
    Call a method in the `Pointer<>` class.

  In particular, care must be taken between the two forms `e->MethodName()` and `ePtr->MethodName()` as they are syntactically very similar but semantically very different.

## 5.1 Algorithms on $n$-ary Trees

We store mathematical expressions in a tree structure so that we can apply recursive algorithms to them. Most of these algorithms are based on the following model.

```
if expression is a leaf node
  do something
else
  recurse on all subnodes
  do something else
end if
```

In particular, when using Ev3, the most common methods used in the design of recursive algorithms are the following:

- `IsLeaf()`: is the node a leaf node (variable or constant)?
- `GetSize()`: find the number of subnodes of any given node.
- `GetOpType()`: return the type of operator node.
- `GetNode(int i)`: return the $i$-th subnode of this node (nodes are numbered starting from 0).
- `DeleteNode(int i)`: delete the $i$-th subnode of this node (care must be taken to deal with cases where all the subnodes have been deleted — Ev3 allows the creation of operators with 0 subnodes, although this is very likely to lead to subsequent errors, as it has no mathematical meaning).
- Use of the operators for manipulation of nodes: supposing *Expression e, f* contain valid mathematical expressions, the following are all valid expressions (the new expressions are created using copies of the old ones).

  ```
  Expression e1 = e + f;
  Expression e2 = e * Log(Sqrt(e^2 - f^2));
  Expression e3 = e + f - f; // this is automatically by simplified to e
  ```

See section 5.2 for an example of usage of the methods above. Furthermore, most algorithms in the source code file `expression.cxx` are recursive algorithms, and good examples of usage for these methods.

## 5.2 Extending the Library: A Trigonometric Simplification

In order to illustrate how to extend Ev3's capabilities, we are going to add a method, `TrigSimp()`, that simplifies the term $\sin^2(f) + \cos^2(f)$ to 1, where $f$ is an expression node[1].

Because we want to change an expression node, we want to make sure we operate on an `Expression`, not on a `BasicExpression`; i.e. we want the garbage collection to be active. Thus, we shall not add a method to the `BasicExpression` class, but our method `TrigSimp` will live outside the class definitions. Hence we add its prototype to the end of the header file `expression.h`.

```
  long TrigSimp(Expression a);
```

The method will return the number of simplifications it has managed to carry out.

In the definition of the method (to be appended to the file `expression.cxx`), we recursively test all nodes in the $n$-ary tree; if any node is equal to $\sin^2(f) + \cos^2(g)$, where $f$ and $g$ are identical sub-trees, we replace that node with 1, and we see whether some further simplification involving the new constant node is possible.

```
long TrigSimp(Expression a) {
  long i = 0;
  long ret = 0;
  for(i = 0; i < a->GetSize(); i++) {
    // recurse over subnodes of a
    ret += TrigSimp(a->GetNode(i));
  }
  // now try simplification on a
  if (a->GetOpType() == SUM && a->GetSize() > 1) {
    // try to look for a sin^2 and a cos^2
    long sinpos = -1;
    long cospos = -1;
    for (i = 0; i < a->GetSize(); i++) {
      // cycle over subnodes
      if (a->GetNode(i)->GetOpType() == POWER) {
        if (a->GetNode(i)->GetNode(0)->GetOpType() == SIN &&
            a->GetNode(i)->GetNode(1)->HasValue(2))
          // found sin^2
          sinpos = i;
      }
      if (a->GetNode(i)->GetOpType() == POWER) {
        if (a->GetNode(i)->GetNode(0)->GetOpType() == COS &&
            a->GetNode(i)->GetNode(1)->HasValue(2))
          // found cos^2
          cospos = i;
      }
    }
    if (sinpos != -1 && cospos != -1) {
      // found both, check their arguments
      if (a->GetNode(sinpos)->GetNode(0)->GetNode(0)->IsEqualTo
          (a->GetNode(cospos)->GetNode(0)->GetNode(0))) {
        ret++; // augment simplification counter
        // arguments are equal, can simplify
        long f = sinpos < cospos ? sinpos : cospos; // first to delete
        long l = sinpos > cospos ? sinpos : cospos; // last to delete
        a->DeleteNode(f);     // delete first subnode - this makes all
```

[1]In fact, the `TrigSimp()` method has been implemented and is present in the current Ev3 distribution.

```
        a->DeleteNode(l - 1); // subsequent subnodes be indexed at -1
        // verify that there are still some nodes left
        if (a->GetSize() == 0) {
          // there aren't any, set a to one
          a->One();
        } else {
          // there are some, check whether there is a constant part
          // we can add the 1 to
          bool addflag = false;
          for (i = 0; i < a->GetSize(); i++) {
            if (a->GetNode(i)->IsConstant) {
              // yes there is
              a->GetNode(i)->SetValue(a->GetNode(i)->GetSimpleValue() + 1);
              addflag = true;
              break;
            }
          }
          if (!addflag) {
            // no there wasn't, add it as a symbolic node
            Expression one(1.0);
            a->AddNode(one);
          }
          // check that there is more than just one node
          if (a->GetSize() == 1) {
            // only one node, shift everything one rank level up
            a = a->GetNode(0);
          }
        }
      }
    }
  }
  return ret;
}
```

### 5.3 Source Code Building Instructions

The distribution can be unpacked in any directory. Building the source should be straightforward, by typing "`make`" at the shell prompt. Ev3 has been written for and tested on the GNU C++ compiler v. 3.2; there might be some glitches in porting it to a different compiler, but it should be mostly straightforward. Be warned that Ev3 uses some advanced C++ and STL features, so very old compilers might not be up to the task.

## 6 Conclusion

We have reviewed current options to perform symbolic computation within a C++ environment. We have presented Ev3, a C++ library for symbolic computation. We have analysed the Ev3 system architecture and shown how to expand the library functionalities by describing a trigonometric simplification method. Ev3 has very fast evaluation routines, and can do symbolic differentiation and simplification of mathematical expressions. It is well suited for integration into a third-party scientific software package that needs to perform a substantial number of derivative evaluations.

## References

[Mathematica(2001)] Mathematica. http://support.wolfram.com/MathLink/WhatIs.html. MathLink Documentation, Mathematica website, 2001.

[Maple(2001)] Maple. http://www.maplesoft.com/products/Maple7/functionality.html. Maple Documentation, Maple website, 2001.

[Matlab(2001)] Matlab. http://www.mathworks.com/access/helpdesk/help/toolbox/symbolic/symbolic.shtml. Matlab Toolbox, Matlab website, 2001.

[Matlab API(2001)] Matlab. http://www.mathworks.com/access/helpdesk/help/tech-doc/apiref/apiref.shtml. Matlab API Documentation, Matlab website, 2001.

[MuPad(2001)] MuPad. http://www.sciface.com/news/features.shtml. MuPad Features, MuPad website, 2001.

[MuPad Documentation(2001)] MuPad. ftp://ftp.uni-paderborn.de/pub/MuPAD/doc/. MuPad Documentation, MuPad website, 2001.

[GAP(2001)] GAP. http://www-groups.dcs.st-andrews.ac.uk/~gap/. GAP Home Page, GAP website, 2001.

[Singular(2001)] Singular. http://www.singular.uni-kl.de/. Singular Home Page, Singular website, 2001.

[SciLab(2001)] SciLab. http://www-rocq.inria.fr/scilab/. SciLab Home Page, SciLab website, 2001.

[Octave(2001)] Octave. http://www.octave.org. Octave Home Page, Octave website, 2001.

[Maxima(2001)] Maxima. http://www.maxima.org. Maxima Home Page, Maxima website, 2001.

[JACAL(2001)] JACAL. http://www-swiss.ai.mit.edu/~jaffer/JACAL.html. JACAL Home Page, JACAL website, 2001.

[GiNaC(2001)] GiNaC. http://www.ginac.de. GiNaC Home Page, GiNaC website, 2001.

[yacas(2001)] yacas. http://www.xs4all.nl/ apinkus/yacas.html. yacas Home Page, yacas website, 2001.

[GiNaC Group(2001)] The GiNaC Group. *GiNaC 0.9.1, an Open Framework for Symbolic Computation within the C++ Programming Language.* http://www.ginac.de/tutorial, 27 June 2001.

[BFK01] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *Journal of Symbolic Computation*, to appear, 2001.

[BP88] I.D.L. Bogle and C.C. Pantelides. Sparse Nonlinear Systems in Chemical Process Simulation. In A.J. Osiadacz, editor, *Simulation and Optimization of Large Systems*, Oxford, 1988. Clarendon Press.

[Har71] F. Harary. *Graph Theory.* Addison-Wesley, Reading, MA, second edition, 1971.

[Kal00] E. Kaltofen. Challenges of Symbolic Computation: My favorite Open Problems. *Journal of Symbolic Computation*, 29:891–919, 2000.

[Knu81] D.E. Knuth. *The Art of Computer Programming, Part II: Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 1981.

[KV00] B. Korte and J. Vygen. *Combinatorial Optimization, Theory and Algorithms.* Springer-Verlag, Berlin, 2000.

[LTKP01] L. Liberti, P. Tsiakis, B. Keeping, and C.C. Pantelides. *oo$\mathcal{OPS}$*. Centre for Process Systems Engineering, Chemical Engineering Department, Imperial College, London, UK, 1.24 edition, jan 2001.

[Lib03] L. Liberti. *Performance Comparison of Function Evaluation Methods.* In "Progress in Computer Science Research", Nova Science, 2003 (to appear).

[Pan88] C.C. Pantelides. *Symbolic and Numerical Techniques for the Solution of Large Systems of Nonlinear Algebraic Equations.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, May 1988.

[SSH00] T.K. Shi, W.H. Steeb, and Y. Hardy. *An Introduction to Computer Algebra Using Object-Oriented Programming.* Springer-Verlag, Berlin, second edition, 2000.

[Str99] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, third edition, 1999.