Your title here

Generated by Doxygen 1.8.16

1 CArL

This is the documentation of CArL, an Open Source C++ Library for Computer Arithmetic and Logic. On this page, you can find introductory information on how to obtain and compile CArL, discussion of some core features of CArL as well as traditional doxygen API documentation.

If you are new to CArL and want to have a look around, we recommend reading the User Documentation. This section gives a gentle introduction to basic concepts like number types, polynomials and alike.

If you want to use CArL and want to know how to get and install it, have a look at Getting Started. It covers the most important steps including obtaining the actual source code, obtaining dependencies, building the library and running our test suite.

If you already use CArL and want to dig deeper or submit new code, you can read the Developers' Guide. It contains information about supplementary features like our logging framework and some basic guidelines for our code like how we use doxygen.

Note that this documentation is, and will probably always be, work in progress. If you feel that some topic that is important to you is missing or some explanation is unclear, please let us know!

1.0.1 Contact

- github: `https://github.com/ths-rwth/carl`

2 Developers' Guide

2.1 Documentation

On this page, we refer to some internal documentation rules. We use doxygen to generate our documentation and code reference. The most important conventions for documentation in CArL are collected here.

Note that some of the documentation may be incomplete or rendered incorrectly, especially if you use an old version of doxygen. Here is a list of known problems:

- Comments in code blocks (see below) may not work correctly (e.g. with doxygen 1.8.1.2). See `here` for a workaround. This will however look ugly for newer doxygen versions, hence we do not use it.
- Files with `static_assert` statements will be incomplete. A `patch` is pending and will hopefully make it into doxygen 1.8.9.
- Member groups (usually used to group operators) may or may not work. There still seem to be a few cases where doxygen `messes up`.
- Documenting unnamed parameters is not possible. A corresponding `ticket` exists for several years.

2.1.1 Modules In order to structure the reference, we use the concept of `Doxygen modules`. Such modules are best thought of as a hierarchical set of tags, called groups. We define those groups in `/doc/markdown/codedocs/groups.dox`. Please make sure to put new files and classes in the appropriate groups.

2.1.2 Literature references Literature references should be provided when appropriate.

We use a bibtex database located at `/doc/literature.bib` with the following conventions:

- Label for one author: `LastnameYY`, for example `Ducos00` for **?** .
- Label for multiple authors: `ABCYY` where `ABC` are the first letters of the authors last names. For example `GCL92` for **?** .
- Order the bibtex entrys by label.

These references can be used with `@cite label`, for example like this:

```
/**
 * Checks whether the polynomial is unit normal
 * @see @cite GCL92, page 39
 * @return If polynomial is normal.
 */
bool is_normal() const;
```

2.1.3 Code comments

2.1.3.1 File headers

```
/**
 * @file <filename>
 * @ingroup <groupid1>
 * @ingroup <groupid2>
 * @author <author1>
 * @author <author2>
 *
 * [ Short description ]
 */
```

Descriptions may be omitted when the file contains a single class, either implementation or declaration.

2.1.3.2 Namespaces Namespaces are documented in a separate file, found at '/doc/markdown/codedocs/namespaces.dox'

2.1.3.3 Class headers

```
/**
 * @ingroup <groupid>
 * [ Description ]
 * @see <reference>
 * @see <OtherClass>
 */
```

2.1.3.4 Method headers

```
/**
 * [ Usage Description ]
 * @param <p1> [ Short description for first parameter ]
 * @param <p2> [ Short description for second parameter ]
 * @return [ Short description of return value ]
 * @see <reference>
 * @see <otherMethod>
 */
```

These method headers are written directly above the method declaration. Comments about the implementation are written above the or inside the implementation.

The `see` command is used likewise as for classes.

2.1.3.5 Method groups There are some cases when documenting each method is tedious and meaningless, for example operators. In this case, we use doxygen method groups.

For member operators (for example `operator+=`), this works as follows:

```
/// @name In-place addition operators
/// @{
/**
 * Add something to this polynomial and return the changed polynomial.
 * @param rhs Right hand side.
 * @return Changed polynomial.
 */
MultivariatePolynomial& operator+=(const MultivariatePolynomial& rhs);
MultivariatePolynomial& operator+=(const Term<Coeff>& rhs);
MultivariatePolynomial& operator+=(const Monomial& rhs);
MultivariatePolynomial& operator+=(Variable::Arg rhs);
MultivariatePolynomial& operator+=(const Coeff& rhs);
/// @}
```

2.1.4 Writing out-of-source documentation Documentation not directly related to the source code is written in Markdown format, and is located in `/doc/markdown/`.

2.2 Logging

2.2.1 Logging frontend The frontend for logging is defined in logging.h.

It provides the following macros for logging:

- LOGMSG_TRACE(channel, msg)
- LOGMSG_DEBUG(channel, msg)
- LOGMSG_INFO(channel, msg)
- LOGMSG_WARN(channel, msg)
- LOGMSG_ERROR(channel, msg)
- LOGMSG_FATAL(channel, msg)
- LOG_FUNC(channel, args)
- LOG_FUNC(channel, args, msg)
- LOG_ASSERT(channel, condition, msg)
- LOG_NOTIMPLEMENTED()
- LOG_INEFFICIENT()

Where the arguments mean the following:

- `channel`: A string describing the context. For example `"carl.core"`.
- `msg`: The actual message as an expression that can be sent to a std::stringstream. For example `"foo: " << foo`.
- `args`: A description of the function arguments as an expression like `msg`.
- `condition`: A boolean expression that can be passed to `assert()`.

Typically, logging looks like this:

```
bool checkStuff(Object o, bool flag) {
    LOG_FUNC("carl", o << ", " << flag);
    bool result = o.property(flag);
    LOGMSG_TRACE("carl", "Result: " << result);
    return result;
}
```

Logging is enabled (or disabled) by the `LOGGING` macro in CMake.

2.2.2 Logging configuration As of now, there is no frontend interface to configure logging. Hence, configuration is performed directly on the backend.

2.2.3 Logging backends As of now, only two logging backends exist.

2.2.3.1 CArL logging CArL provides a custom logging mechanism defined in carl::logging.

2.2.3.2 Fallback logging If logging is enabled, but no real logging backend is selected, all logging of level `WARN` or above goes to `std::cerr`.

2.3 Finding and Reporting Bugs

This page is meant as a guide for the case that you find a bug or any unexpected behaviour. We consider any of the following events a (potential) bug:

- CArL crashes.
- A library used through CArL crashes.
- CArL gives incorrect results.
- CArL does not terminate (for reasonably sized inputs).
- CArL does not provide a method or functionality that should be available according to this documentation.
- CArL does not provide a method or functionality that you consider crucial or trivial for some of the datastructures.
- Compiling the CArL library fails.
- Compiling your code using CArL fails and you are pretty sure that you use CArL according to this documentation.

In any of the above cases, make sure that:

- You have installed all necessary Dependencies in the required versions.
- You work on something that is similar to a system listed as supported platform at Getting Started.
- You can (somewhat reliably) reproduce the error with a (somewhat) clean build of CArL. (i.e., you did not screw up the CMake flags, see Building with CMake for more information)

- You compile either with `CMAKE_BUILD_TYPE=DEBUG` or `DEVELOPER=ON`. This will give additional warnings during compilation and enable assertions during runtime. This will slow down CArL significantly, but detect errors before an actual crash happens and give a meaningful error message in many cases.

If you are unable to solve issue yourself or you find the issue to be an actual bug in CArL, please do not hesitate to contact us. You can either contact us via email (if you suspect a configuration or usage issue on your side) or create a ticket in our bug tracker (if you suspect an error that is to be fixed by us). We use the github bug tracker at https://github.com/ths-rwth/carl/issues.

When sending us a mail or creating a ticket, please provide us with:

- Your system specifications, including versions of compilers and libraries listed in the dependencies.
- The CArL version (release version or git commit id).
- A minimal working example.
- A description of what you would expect to happen.
- A description of what actually happens.

2.4 Code style

Please follow these guidelines for new code. We are migrating old code over the time.

2.4.0.1 Code formatting

`ClangFormat` allows to define code style rules and format source files automatically. A `.clang-format` file is provided with the repository. Please use this file to format all sources.

2.4.0.2 Naming conventions

For all new code, the following rules apply.

- type names and template parameter: `CamelCase`
- variable and function names: `snake_case`
- compiler macros and defines: `ALL_UPPERCASE`
- enum values: `UPPERCASE`
- (private) class members: start with `m_` respectively `mp_` for pointers and `mr_` for references
- type traits: `snake_case` and end with `_type`
- namespace: `snake_case`

2.4.0.3 Use of classes, structs and functions

We follow a Rust-style approach where we define data structures and attach basic operations that as methods to it. All functionality that can be considered optional is realized via free functions.

2.4.0.4 Directory structure and namespaces

- Libraries
  - Dependencies between libraries are acyclic!
- Folders and files
  - A folder represents a module.
  - A file contains either of the following:
    - a data structure, a collection of related data structures and basic functionality,
    - free functions that operate on data structures.
  - Dependencies between folders on the same level need to be acyclic.
  - Either all files in a directory depend on subdirectories in the same folder or subdirectories depend on files from the parent directory, but not both.
- Namespaces
  - CArL lives within the `carl` namespace.
  - Each library has its own sub-namespace, except `carl-common`, `carl-arithmetic`, `carl-formula`, `carl-extpolys`.
  - Auxiliary functions are in an appropriate sub-namespace.

2.4.0.5 C++ features

- As of now, please stick to C++17 features.
- Use `enum class` instead of `enum`.

3 Getting Started

3.1 Download

We mirror our master branch to github.com. If you want to use the newest bleeding edge version, you can checkout from `https://github.com/ths-rwth/carl`. Although we try to keep the master branch stable, there is a chance that the current revision is broken. You can check `here` if the current revision compiles and all the unit tests work.

We regularly tag reasonably stable versions. You can find them at `https://github.com/ths-rwth/carl/releases`.

3.2 Quick installation guide

- Make sure all dependencies are available.
- Download the latest release or clone the git repository from `https://github.com/ths-rwth/carl`.
- Prepare the build.
  ```
  $ mkdir build && cd build && cmake ../
  ```
- Build carl (with tests and documentation).
  ```
  $ make
  $ make test doc
  ```

3.3 Using CArL

CArL registers itself in the CMake system, hence to include CArL in any other CMake project, just use `find_package(carl)`.

To use CArL in other projects, link against the shared or static library created in `build/`.

3.4 Supported platforms

We test carl on the following platforms:
- Ubuntu 22.04 LTS with several compilers

We usually support at least all `clang` and `gcc` versions starting from those shipped with the latest Ubuntu LTS or Debian stable releases. As of now, this is `clang-13` and newer and `gcc-11` and newer.

3.5 Advanced building topics

- Building with CMake

3.6 Troubleshooting

If you're experiencing problems, take a look at our Troubleshooting section. If that doesn't help you, feel free to contact us.

3.7 Dependencies

To build and use CArL, you need the following other software:
- `git` to checkout the git repository.
- `cmake` to generate the make files.
- `g++` or `clang` to compile.

We use C++17 and thus need at least `g++` 7 or `clang` 5.

Optional dependencies
- `ccmake` to set cmake flags.
- `doxygen` and `doxygen-latex` to build the documentation.
- `gtest` to build the test cases.

Additionally, CArL requires a few external libraries, which are installed automatically by CMake if no local version is available:
- `boost` for several additional libraries.
- `gmp` for calculations with large numbers.
- `Eigen3` for numerical computations.

To simplify the installation process, all these libraries can be built by CArL automatically if it is not available on your system. You can do this manually by running
```
make resources
```

3.8 Building with CMake

We use `CMake` to support the building process. CMake is a command line tool available for all major platforms. To simplify the building process on Unix, we suggest using `CCMake`.

CMake generates a Makefile likewise to Autotools' configure. We suggest initiating this procedure from a separate build directory, called 'out-of-source' building. This keeps the source directory free from files created during the building process.

3.8.1 CMake Options for building CArL. Run `ccmake` to obtain a list of all available options or change them.

```
$ cd build/
$ ccmake ../
```

Using `[t]`, you can enable the *advanced mode* that shows all options. Most of these should not be changed by the average user.

3.8.1.1 General

- *CMAKE_BUILD_TYPE* [Release, Debug]
  - *Release*
  - *Debug*
- *CMAKE_CXX_COMPILER* <compiler command>
  - `/usr/bin/c++`: Default for most linux distributions, will probably be an alias for `g++`.
  - `/usr/bin/g++`: Uses `g++`.
  - `/usr/bin/clang++`: Uses `clang`.
- *USE_CLN_NUMBERS* [ON, OFF]
  If set to *ON*, CLN number types can be used in addition to GMP number types.
- *USE_COCOA* [ON, OFF]
  If set to *ON*, CoCoALib can be used for advanced polynomial operations, for example multivariate gcd or factorization.
- *USE_GINAC* [ON, OFF]
  If set to *ON*, GiNaC can be used for some polynomial operations. Note that this implies *USE_CLN_NUMBERS = ON*.

3.8.1.2 Debugging

- *DEVELOPER*
  Enables additional compiler warnings.
- *LOGGING* [ON, OFF]
  Setting *LOGGING* to *OFF* disables all logging output. It is recommended if the performance should be maximized, but notice that this also prevents important warnings and error messages to be generated.

3.8.2 CMake Targets There are a few important targets in the CArL CMakeLists:

- `doc`: Builds the doxygen documentation.
- `libs`: Builds all libraries.
- `runXTests`: Builds the tests for the `X` module.
- `test`: Build and run all tests.

3.9 Troubleshooting

3.9.1 General CArL tries to make use of modern C++ features. Though we try to be compatible with the stock versions of all dependencies of Debian stable and the latest Ubuntu LTS, this does not always work out.

4 User Documentation

This is the introductory user documentation of CArL. It explains the basic concepts and classes that CArL provides.

- CArL module structure

4.1 Basic concepts

- Numbers
- Polynomials
- Numbers

4.2 Tutorial

There are some introductory code examples how CArL can be used. You find them at Tutorial .

4.3 CArL module structure

CArL is separated into several libraries implementing functionality on different abstraction levels.

4.3.1 General utilities

- carl-common: Basic data structures, helper methods, etc.
- carl-logging: Logging funtionality. *Depends on carl-common.*
- carl-statistics: Collect statistics about a run of a program. *Depends on carl-common.*
- carl-checkpoints: Collect the trace of the run of a programm and compare it with certain checkpoints. *Depends on carl-common and carl-logging.*
- carl-settings: Runtime settings infrastructure.

4.3.2 Core libraries

- carl-arithmetic: Arithmetic package. Does not do sophisticated memory management. *Depends on carl-common and carl-logging.*
- carl-formula: Logical formulas with support for arithmetic, bitvector and uninterpreted function constraints. Does pooling of some types for memory efficiency. *Depends on carl-arithmetic.*
- carl-vs: Implements virtual substitution. For legecy reasons, this depends on carl-formula.
- carl-extpolys: Extended polynomial types: factorized polynomials, rational functions. *Depends on carl-arithmetic.*

4.3.3 Higher level

- carl-io: Input/output functionality for CArL types from/to different file formats. *Depends on carl-formula.*
- carl-covering: Data structures and heuristics for computing coverings. *Depends on carl-common and carl-logging.*

4.4 Numbers

The higher-level datastructures in CArL are templated with respect to their underlying number type and can therefore be used with any number type that fulfills some common requirements. This is the case, for example, for carl::Term, carl::MultivariatePolynomial, carl::UnivariatePolynomial or carl::Interval objects.

Everything related to number types resides in the /carl/numbers/ directory. For each group of supported number types `T`, a folder `adaption_T` exists that contains the following:

- Include of the library (if necessary)
- Type traits according to typetraits.
- Static constants for zero and one.
- Operations to fulfill our common interface.

From the outside, that is also the rest of the CArL library, only the central numbers/numbers.h shall be included. This file includes all available adaptions and takes care of disabling adaptions if the respective library is unavailable.

4.4.1 Adaptions As of now, we provide adaptions of the following types:

- CLN (cln::cl_I and cln::cl_RA).
- FLOAT_T<mpfr_t>, our own wrapper for mpfr_t
- GMPxx, the C++ interface of GMP.
- Native datatypes as defined by **?**

Note that these adaptions may not fully implement all methods described below, but only to some extend that is used. Finishing these adaptions is work in progress.

4.4.2 Interface The following interface should be implemented for every number type `T`.

- typetraits if applicable.
- `carl::constant_zero<T>` and `carl::constant_one<T>` if the generic definition from carl/numbers/constants.h does not fit.
- Specialization of `std::hash<T>`
- Arithmetic operators:
  - `T operator+(const T&, const T&)` and `T& operator+=(const T&, const T&)`
  - `T operator-(const T&, const T&)` and `T& operator-=(const T&, const T&)`
  - `T operator-(const T&)`
  - `T operator*(const T&, const T&)` and `T& operator*=(const T&, const T&)`
  - `T& operator=(const T&)`
- `bool carl::is_zero(const T&)` and `bool carl::is_one(const T&)`
- If `carl::is_rational_type<T>::value`:
  - `carl::get_num(const T&)` and `carl::get_denom(const T&)`
  - `T carl::rationalize(double)`
- `bool carl::is_integer(const T&)`
- `std::size_t carl::bitsize(const T&)`
- `double carl::to_double(const T&)` and `I carl::to_int<I>(const T&)` for some integer types `I`.
- `T carl::abs(const T&)`
- `T carl::floor(const T&)` and `T carl::ceil(const T&)`
- If `carl::is_integer_type<T>::value`:
  - `T carl::gcd(const T&, const T&)` and `T carl::lcm(const T&, const T&)`
  - `T carl::mod(const T&, const T&)`
- `T carl::pow(const T&, unsigned)`
- `std::pair<T,T> carl::sqrt(const T&)` where the result represents an interval containing the exact result.
- `T carl::div(const T&, const T&)` asserting that exact division is possible.
- `T carl::quotient(const T&, const T&)` and `T carl::remainder(const T&, const T&)`

4.5 Polynomials

In order to represent polynomials, we define the following hierarchy of classes:

- Coefficient: Represents the numeric coefficient..
- Variable: Represents a variable.
- Monomial: Represents a product of variables.
- Term: Represents a product of a constant factor and a Monomial.
- MultivariatePolynomial: Represents a polynomial in multiple variables with numeric coefficients.

We consider these types to be embedded in a hierarchy like this:

- MultivariatePolynomial
  - Term
    - Monomial
      - Variable
    - Coefficient

We will abbreviate these types as C, V, M, T, MP.

4.5.1 UnivariatePolynomial Additionally, we define a UnivariatePolynomial class. It is meant to represent either a univariate polynomial in a single variable, or a multivariate polynomial with a distinguished main variable.

In the former case, a number type is used as template argument. We call this a *univariate polynomial*.

In the latter case, the template argument is instantiated with a multivariate polynomial. We call this a *univariately represented polynomial*.

A UnivariatePolynomial, regardless if univariate or univariately represented, is mostly compatible to the above types.

Operators

4.5.2 Operators The classes used to build polynomials are (almost) fully compatible with respect to the following operators, that means that any two objects of these types can be combined if there is a directed path between them within the class hierarchy. The exception are shown and explained below. All the operators have the usual meaning.

- Comparison operators
  - `operator==(lhs, rhs)`
  - `operator!=(lhs, rhs)`
  - `operator<(lhs, rhs)`
  - `operator<=(lhs, rhs)`
  - `operator>(lhs, rhs)`
  - `operator>=(lhs, rhs)`
- Arithmetic operators
  - `operator+(lhs, rhs)`
  - `operator+=(lhs, rhs)`
  - `operator-(lhs, rhs)`
  - `operator-(rhs)`
  - `operator-=(lhs, rhs)`
  - `operator*(lhs, rhs)`
  - `operator*=(lhs, rhs)`

4.5.2.1 Comparison operators All of these operators are defined for all combination of types. We use the following ordering:

- For two variables `x` and `y`, `x < y` if the id of `x` is smaller then the id of `y`. The id is generated automatically by the VariablePool.
- For two monomials `a` and `b`, we use a lexicographical ordering with total degree, that is `a < b` if
  - the total degree of `a` is smaller than the total degree of `b`, or
  - the total degrees are the same and
    - the exponent of some variable `v` in `a` is greater than in `b` and
    - the exponents of all variables smaller than `v` are the same in `a` and in `b`.
  - The intuition is that the monomials are considered as a sorted product of plain variables.
- For two terms `a` and `b`, `a < b` if
  - the monomial of `a` is smaller than the monomial of `b`, or
  - the monomials of `a` and `b` are the same and the coefficient of `a` is smaller than the coefficient of `b`.
- For two polynomials `a` and `b`, we use a lexicographical ordering, that is `a < b` if
  - `term(a,i) < term(b,i)` and
  - `term(a,j) = term(b,j)` for all `j=0, ..., i-1`, where `term(a,0)` is the leading term of `a`, that is the largest term with respect to the term ordering.

4.5.2.2 Arithmetic operators We now give a table for all (classes of) operators with the result type or a reason why it is not implemented for any combination of these types.

+	C	V	M	T	MP
C	C	MP	MP	MP	MP
V	MP	1)	1)	MP	MP
M	MP	1)	1)	MP	MP
T	MP	MP	MP	MP	MP
MP	MP	MP	MP	MP	MP

4.5.2.2.1 <tt>operator+(lhs, rhs)</tt>, <tt>operator-(lhs, rhs)</tt>

-	C	V	M	T	MP
-	C	1)	1)	T	MP

4.5.2.2.2 <tt>operator-(lhs)</tt> (unary minus)

*	C	V	M	T	MP
C	C	T	T	T	MP
V	T	M	M	T	MP
M	T	M	M	T	MP
T	T	T	T	T	MP
MP	MP	MP	MP	MP	MP

4.5.2.2.3 operator*(lhs, rhs)

+=	C	V	M	T	MP
C	C	2)	2)	2)	2)
V	2)	2)	2)	2)	2)
M	2)	2)	2)	2)	2)
T	2)	2)	2)	2)	2)
MP	MP	MP	MP	MP	MP

4.5.2.2.4 <tt>operator+=(rhs)</tt>, <tt>operator-=(rhs)</tt>

*=	C	V	M	T	MP
C	C	3)	3)	3)	3)
V	3)	3)	3)	3)	3)
M	3)	M	M	3)	3)
T	T	T	T	T	3)
MP	MP	MP	MP	MP	MP

4.5.2.2.5 <tt>operator*=(rhs)</tt>

1. A coefficient type is needed to construct the desired result type, but none can be extracted from the argument types.
2. The type of the left hand side can not represent sums of these objects.
3. The type of the left hand side can not represent products of these objects.

4.5.2.3 UnivariatePolynomial operators

4.5.2.4 Implementation We follow a few rules when implementing these operators:

- Of the comparison operators, only `operator==` and `operator<` contain a real implementation. The others are implemented like this:
  - `operator!=(lhs, rhs):!(lhs == rhs)`
  - `operator<=(lhs, rhs):!(rhs < lhs)`
  - `operator>(lhs, rhs):rhs < lhs`
  - `operator>=(lhs, rhs):rhs <= lhs`
- Of all `operator==`, only those where `lhs` is the most general type contain a real implementation. The others are implemented like this:
  - `operator==(lhs, rhs):rhs == lhs`
- They are ordered like in the list above.
- Operators are implemented in the file of the most general type involved (either an argument or the return type).

- Operators are not implemented as friend methods. Those are usually only found by the compiler due to ADL, but as we need to declare `operator+(Term, Term) -> MultivariatePolynomial` next to the MultivariatePolynomial, this will not work. If a friend declaration is necessary, it will be done as a forward declaration.
- Overloaded versions of the same operator are ordered in decreasing lexicographical order, like in this example:
  - `operator(Term,Term)`
  - `operator(Term,Monomial)`
  - `operator(Term,Variable)`
  - `operator(Term,Coefficient)`
  - `operator(Monomial,Term)`
  - `operator(Variable,Term)`
  - `operator(Coefficient,Term)`
- Other versions are below those.

4.5.2.5 Testing the operators There are two stages for testing these operators: a syntactical check that these operators exist and have the correct signature and a semantical check that they actually work as expected.

4.5.2.5.1 Syntactical checks The syntactical check for all operators specified here is done in `tests/core/Test_Operators.cpp`. We use `boost::concept_check` to check the existence of the operators. There are the following concepts:

- `Comparison`: Checks for all comparison operators. (==, !=, <, <=, >, >=)
- `Addition`: Checks for out-of-place addition operators. (+, −)
- `UnaryMinus`: Checks for unary minus operators. (−)
- `Multiplication`: Checks for out-of-place multiplication operators. (∗)
- `InplaceAddition`: Checks for all in-place addition operators. (+=, −=)
- `InplaceMultiplication`: Checks for all in-place multiplication operators. (∗=)

4.5.2.5.2 Semantical checks Semantical checking is done within the test for each class.

4.6 Numbers

4.7 Tutorial

As a tutorial, we have a number of small programs that show certain features of CArL. The code is explained using normal comments and can be compiled using `make tutorial`.

Whenever we want to state that a certain property holds at some point, we will use `assert()` to do so.

- Creating Variables
- Creating Monomials
- Creating Polynomials