Contents

1 PerlMol

Perl modules for molecular chemistry

SYNOPSIS

```
# This is a bundle containing all of the modules
# of the PerlMol Project and their dependencies.
# This is not a real module; it is the main index
# to the documentation of the PerlMol modules.
```

DESCRIPTION

PerlMol is a collection of Perl modules for chemoinformatics and computational chemistry with the philosophy that "simple things should be simple". It should be possible to write one-liners that use this toolkit to do meaningful "molecular munging". The PerlMol toolkit provides objects and methods for representing molecules, atoms, and bonds in Perl; doing substructure matching; and reading and writing files in various formats.

DOCUMENTATION

What follows is an index of the relevant documentation.

Tutorial

***Chemistry::Tutorial* - A good place to start reading the documentation.**

Object oriented modules

The following modules are indented according to the class hierarchy:

Chemistry::Obj

- ***Chemistry::Mol***
  - ***Chemistry::Pattern***
    - ***Chemistry::FormulaPattern***
    - ***Chemistry::MidasPattern***
  - ***Chemistry::Domain***
  - ***Chemistry::MacroMol***
  - ***Chemistry::Ring***
- ***Chemistry::Atom***
  - ***Chemistry::Pattern::Atom***
- ***Chemistry::Bond***
  - ***Chemistry::Pattern::Bond***

Chemistry::File

- ***Chemistry::File::Formula***
- ***Chemistry::File::FormulaPattern***
- ***Chemistry::File::MDLMol***
- ***Chemistry::File::MidasPattern***
- ***Chemistry::File::Mopac***
- ***Chemistry::File::PDB***
- ***Chemistry::File::SDF***
- ***Chemistry::File::SLN***
- ***Chemistry::File::SMARTS***
- ***Chemistry::File::SMILES***
- ***Chemistry::File::VRML***
- ***Chemistry::File::XYZ***

Chemistry::InternalCoords

Chemistry::Mok

Procedural modules

These are auxiliary modules for which object classes seemed overkill

Chemistry::3DBuilder

Chemistry::Bond::Find

Chemistry::InternalCoords::Builder

Chemistry::Ring::Find

Chemistry::Canonicalize

Programs (scripts)

mok **- an AWK for molecules**

PerlMol bundle description and contents

PerlMol **- This document**

Publications

- Tubert-Brohman, I. Perl and Chemistry. The Perl Journal 2004-06 (subscription required)
- Cozens, S. Molecular Biology With Perl. The Perl Journal 2004, 8[8], 15-19 (http://www.tpj.com/documents/s=7618/tpj0408/; requires subscription)
- F. Rosselló, G. Valiente. Chemical Graphs, Chemical Reaction Graphs, and Chemical Graph Transformation. 2nd Int. Workshop on Graph-Based Tools, Electronic Notes in Computer Science 2005, 127, 157-166. (abstract: http://www.lsi.upc.es/%7Evaliente/abs-grabats-2004.html; preprint full text: http://tfs.cs.tu-berlin.de/grabats/Final04/valiente.pdf; published version: http://dx.doi.org/10.1016/j.entcs.2004.12.033).
- Rosselló, F.; Valiente, G. Graph Transformation in Molecular Biology. (Full text: http://bioinfo.uib.es/˜cesc/recerca/he-paper.pdf).

EXAMPLES

The "examples" directory in the PerlMol distribution file has several sample scripts with lots of comments and a few input and output files that show how one can use PerlMol for common tasks. They can also be browsed online at http://www.perlmol.org/examples/. Some of the examples are:

```
* combinatorial_enumeration
* file_conversion
* molgrep
* pdb_viewer
* peptide_builder
* polar_surface_area
```

VERSION INFORMATION

This is the PerlMol bundled release version 0.3500. It includes the following distributions:

```
Chemistry-3DBuilder             0.10
Chemistry-Bond-Find             0.21
Chemistry-Canonicalize          0.10
Chemistry-File-MDLMol           0.20
Chemistry-File-Mopac            0.15
Chemistry-File-PDB              0.21
Chemistry-File-SLN              0.10
Chemistry-File-SMARTS           0.22
Chemistry-File-SMILES           0.44
Chemistry-File-VRML             0.10
Chemistry-File-XYZ              0.11
Chemistry-FormulaPattern        0.10
```

```
Chemistry-InternalCoords        0.18
Chemistry-Isotope               0.11
Chemistry-MacroMol              0.06
Chemistry-MidasPattern          0.11
Chemistry-Mok                   0.25
Chemistry-Mol                   0.35
Chemistry-Pattern               0.26
Chemistry-Reaction              0.02
Chemistry-Ring                  0.18
Math-VectorReal                 1.02
Parse-Yapp                      1.05
Statistics-Regression           0.15
```

The version number of a PerlMol bundle is always the same as the version number of the included Chemistry-Mol distribution, plus two extra digits that distinguish between different bundles based on the same Chemistry-Mol distribution.

SEE ALSO

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

2 Chemistry::Tutorial

PerlMol Quick Tutorial

Introduction

The modules in the PerlMol toolkit are designed to simplify the handling of molecules from Perl programs in a general and extensible way. These modules are object-oriented; however, this tries to assume little or no knowledge of object-oriented programming in Perl. For a general introduction about how to use object-oriented modules, see *HTML::Tree::AboutObjects*.

This document shows some of the more common methods included in the PerlMol toolkit, in a reasonable order for a quick introduction. For more details see the perldoc pages for each module.

How to read a molecule from a file

The following code will read a PDB file:

```
use Chemistry::Mol;
use Chemistry::File::PDB;
my $mol = Chemistry::Mol->read("test.pdb");
```

The first two lines (which only need to be used once in a given program) tell Perl that you want to `use` the specified modules The third line reads the file and returns a molecule object.

To read other formats such as MDL molfiles, you need to `use` the corresponding module, such as *Chemistry::File::MDLMol*. Readers for several formats are under development.

The molecule object

`Chemistry::Mol->read` returns a *Chemistry::Mol* object. An *object* is a data structure of a given *class* that has *methods* (i.e. subroutines) associated with it. To access or modify an object's properties, you call the methods on the object through "arrow syntax":

```
my $name = $mol->name; # return the name of the molecule
$mol->name("water");   # set the name of the molecule to "water"
```

Note that these so-called accessor methods return the molecule object when they are used to set a property. A consequence of that if you want, you can "chain" several methods to set several options in one line:

```
$mol->name("water")->type("wet");
```

A *Chemistry::Mol* object contains essentially a list of atoms, a list of bonds, and a few generic properties such as name, type, and id. The atoms and bonds themselves are also objects.

Writing a molecule file

To write a molecule to a file, just use the `write` method:

```
$mol->write("test.pdb");
```

Make sure you `use`d the right file I/O module. If you want to load all the available file I/O modules, you can do it with

```
use Chemistry::File ':auto';
```

Selecting atoms in a molecule

You can get an array of all the atoms by calling the atoms method without parameters, or a specific atom by giving its index:

```
@all_atoms = $mol->atoms;
$atom3 = $mol->atoms(3);
```

Note: Atom and bond indices are counted from 1, not from 0. This deviation from common Perl usage was made to be consistent with the way atoms are numbered in most common file formats.

You can select atoms that match an arbitrary expression by using Perl's built-in `grep` function:

```
# get all oxygen atoms within 3.0 Angstroms of atom 37
@close_oxygens = grep {
    $_->symbol eq 'O'
    and $_->distance($mol->atoms(37)) < 3.0
} $mol->atoms;
```

The `grep` function loops through all the atoms returned by `$mol->atoms`, aliasing each to `$_` at each iteration, and returns only those for which the expression in braces is true.

Using `grep` is a general way of finding atoms; however, since finding atoms by name is common, a convenience method is available for that purpose.

```
$HB1     = $mol->atoms_by_name('HB1');
@H_atoms = $mol->atoms_by_name('H.*'); # name treated as a regex
```

Since the atom name is not generally unique, even the first example above might match more than one atom. In that case, only the first one found is returned. In the second case, since you are assigning to an array, all matching atoms are returned.

The atom object

Atoms are usually the most interesting objects in a molecule. Some of their main properties are Z, symbol, and coords.

```
$atom->Z(8); # set atomic number to 8
$symbol = $atom->symbol;
$coords = $atom->coords;
```

Atom coordinates

The coordinates returned by `$atom->coords` are a *Math::VectorReal* object. You can print these objects and use them to do vector algebra:

```
$c1            = $atom1->coords;
$c2            = $atom2->coords;
$dot_product   = $c1 . $c2;       # returns a scalar
$cross_product = $c1 x $c2;       # returns a vector
$delta         = $c2 - $c1;       # returns a vector
$distance      = $delta->length;  # returns a scalar
($x, $y, $z)   = $c1->array;      # get the components of $c1
print $c1;     # prints something like "[ 1.0E0  2.0E0  3.0E0 ]"
```

Since one is very often interested in calculating the distance between atoms, Atom objects provide a `distance` method to save some typing:

```
$d  = $atom1->distance($atom2);
$d2 = $atom1->distance($molecule2);
```

In the second case, the value obtained is the minimum distance between the atom and the molecule. This can be useful for things such as finding the water molecules closest to a given atom.

Atoms may also have internal coordinates, which define the position of an atom relative to the positions of other atoms by means of a distance, an angle, and a dihedral angle. Those coordinates can be accessed through the $atom->internal_coords method, which uses *Chemistry::InternalCoords* objects.

The Bond object

A *Chemistry::Bond* object is a list of atoms with an associated bond order. In most cases, a bond has exactly two atoms, but we don't want to exclude possibilities such as three-center bonds. You can get the list of atoms in a bond by using the `atoms` method; the bond order is accessed trough the `order` method;

```
@atoms_in_bond = $bond->atoms;
$bond_order    = $bond->order;
```

The other interesting method for Bond objects is `length`, which returns the distance between the two atoms in a bond (this method requires that the bond have two atoms).

```
my $bondlength = $bond->length;
```

In addition to these properties, Bond objects have the generic properties described below. The most important of these, as far as bonds are concerned, is `type`.

Generic properties

There are three generic properties that all PerlMol objects have:

id

Each object must have a unique ID. In most cases you don't have to worry about it, because it is assigned automatically unless you specify it. You can use the `by_id` method to select an object contained in a molecule:

```
$atom = $mol->by_id("a42");
```

In general, ids are preferable to indices because they don't change if you delete or move atoms or other objects.

name

The name of the object does not have any meaning from the point of view of the core modules, but most file types have the concept of molecule name, and some (such as PDB) have the concept of atom names.

type

Again, the meaning of type is not universally defined, but it would likely be used to specify atom types and bond orders.

Besides these, the user can specify arbitrary attributes, as discussed in the next section.

User-specified attributes

The core PerlMol classes define very few, very generic properties for atoms and molecules. This was chosen as a "minimum common denominator" because every file format and program has different ideas about the names, values and meaning of these properties. For example, some programs only allow bond orders of 1, 2, and 3; some also have "aromatic" bonds; some use calculated non-integer bond orders. PerlMol tries not to commit to any particular convention, but it allows you to specify whatever attributes you want for any object (be it a molecule, an atom, or a bond). This is done through the `attr` method.

```
$mol->attr("melting point", "273.15"); # set m.p.
$color = $atom->attr("color"); # get atom color
```

The core modules store these values but they don't know what they mean and they don't care about them. Attributes can have whatever name you want, and they can be of any type. However, by convention, non-core modules that need additional attributes should prefix their name with a *namespace*, followed by a slash. (This is done to avoid modules fighting over the same attribute name.) For example, atoms created by the PDB reader module (Chemistry::File::PDB) have the "pdb/residue" attribute.

```
$mol  = Chemistry::Mol->read("test.pdb");
$atom = $mol->atoms(1234);
print $atom->attr("pdb/residue_name"); # prints "ALA123"
```

Molecule subclasses

You can do lots of interesting thing with plain molecules. However, for some applications you may want to extend the features of the main Chemistry::Mol class. There are several subclasses of Chemistry::Mol available already:

Chemistry::MacroMol

Used for macromolecules.

Chemistry::Pattern

Used for substructure matching.

Chemistry::Ring

Used for representing rings (cycles) in molecules.

Chemistry::Reaction

Used for representing and applying chemical transformations.

As an example we'll discuss macromolecules. Future versions of this tutorial may also include a discussion about patterns and rings.

Macromolecules

So far we have assumed that we are dealing with molecules of the *Chemistry::Mol* class. However, one of the interesting things about object-oriented programming is that classes can be extended. For dealing with macromolecules, we have the MacroMol class, which extends the *Chemistry::Mol* class. This means that in practice you can use a *Chemistry::MacroMol* object exactly as you would use a *Chemistry::Mol* object, but with some added functionality. In fact, the PDB reader can return *Chemistry::MacroMol* instead of *Chemistry::Mol* objects just by changing the first example like this:

```
use Chemistry::MacroMol;
use Chemistry::File::PDB;
my $macromol = Chemistry::MacroMol->read("test.pdb");
```

Now the question is, what is the "added functionality" that MacroMol objects have on top of the original Chemistry::Mol object?

The MacroMol object

For the purposes of this module, a macromolecule is considered to be a big molecule where atoms are divided in *Domains*. A domain is just a subset of the atoms in the molecule; in a protein, a domain would be just a residue.

You can select domains in a molecule in a way similar to that used for atoms and bonds, in this case through the `domains` method:

```
my @all_domains = $macromol->domains;
my $domain      = $macromol->domains(57);
```

The Domain object

A domain is a substructure of a larger molecule. Other than having a *parent* molecule, a domain is just like a molecule. In other words, the Domain class extends the Chemistry::Mol class; it is basically a collection of atoms and bonds.

```
my @atoms_in_domain = $domain->atoms;
my $atom5_in_domain = $domain->atoms(5);
```

If you want to get at a given atom in a given domain in a macromolecule, you can "chain" the method calls without having to save the Domain object in a temporary variable:

```
my $domain57_atom5 = $macromol->domains(57)->atoms(5);
my $res233_HA = $macromol->domains(233)->atoms_by_name('HA');
```

The second example is a good way of selecting an atom from a PDB file when you know the residue number and atom name.

VERSION

0.36

SEE ALSO

Chemistry::Mol, *Chemistry::Atom*, *Chemistry::Bond*, *Chemistry::File*, *Chemistry::MacroMol*, *Chemistry::Domain*.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

3 Chemistry::Obj

Abstract chemistry object

SYNOPSIS

```
package MyObj;
use base "Chemistry::Obj";
Chemistry::Obj::accessor('color', 'flavor');

package main;
my $obj = MyObj->new(name => 'bob', color => 'red');
$obj->attr(size => 42);
$obj->color('blue');
my $color = $obj->color;
my $size = $obj->attr('size');
```

DESCRIPTION

This module implements some generic methods that are used by *Chemistry::Mol*, *Chemistry::Atom*, *Chemistry::Bond*, *Chemistry::File*, etc.

Common Attributes

There are some common attributes that may be found in molecules, bonds, and atoms, such as id, name, and type. They are all accessed through the methods of the same name. For example, to get the id, call `$obj->id`; to set the id, call `$obj->id('new_id')`.

id

Objects should have a unique ID. The user has the responsibility for uniqueness if he assigns ids; otherwise a unique ID is assigned sequentially.

name

An arbitrary name for an object. The name doesn't need to be unique.

type

The interpretation of this attribute is not specified here, but it's typically used for bond orders and atom types.

attr

A space where the user can store any kind of information about the object. The accessor method for attr expects the attribute name as the first parameter, and (optionally) the new value as the second parameter. It can also take a hash or hashref with several attributes. Examples:

```
$color = $obj->attr('color');
$obj->attr(color => 'red');
$obj->attr(color => 'red', flavor => 'cherry');
$obj->attr({color => 'red', flavor => 'cherry'});
```

OTHER METHODS

$obj->del_attr($attr_name)

Delete an attribute.

$class->new(name => value, name => value...)

Generic object constructor. It will automatically call each "name" method with the parameter "value". For example,

```
$bob = Chemistry::Obj->new(name => 'bob', attr => {size => 42});
```

is equivalent to

```
$bob = Chemistry::Obj->new;
$bob->name('bob');
$bob->attr({size => 42});
```

OPERATOR OVERLOADING

Chemistry::Obj overloads a couple of operators for convenience.

""

The stringification operator. Stringify an object as its id. For example, If an object $obj has the id 'a1', print "$obj" will print 'a1' instead of something like 'Chemistry::Obj=HASH(0x810bbdc)'. If you really want to get the latter, you can call `overload::StrVal($obj)`. See *overload* for details.

cmp

Compare objects by ID. This automatically overloads `eq`, `ne`, `lt`, `le`, `gt`, and `ge` as well. For example, `$obj1 eq $obj2` returns true if both objects have the same id, even if they are different objects with different memory addresses. In contrast, `$obj1 == $obj2` will return true only if `$obj1` and `$obj2` point to the same object, with the same memory address.

VERSION

0.37

SEE ALSO

Chemistry::Atom, *Chemistry::Bond*, *Chemistry::Mol*

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

4 Chemistry::Mol

Molecule object toolkit

SYNOPSIS

```
use Chemistry::Mol;

$mol = Chemistry::Mol->new(id => "mol_id", name => "my molecule");
$c = $mol->new_atom(symbol => "C", coords => [0,0,0]);
$o = $mol->new_atom(symbol => "O", coords => [0,0,1.23]);
$mol->new_bond(atoms => [$c, $o], order => 3);

print $mol->print;
```

DESCRIPTION

This package, along with Chemistry::Atom and Chemistry::Bond, includes basic objects and methods to describe molecules.

The core methods try not to enforce a particular convention. This means that only a minimal set of attributes is provided by default, and some attributes have very loosely defined meaning. This is because each program and file type has different idea of what each concept (such as bond and atom type) means. Bonds are defined as a list of atoms (typically two) with an arbitrary type. Atoms are defined by a symbol and a Z, and may have 3D and internal coordinates (2D coming soon).

METHODS

See also *Chemistry::Obj* for generic attributes.

Chemistry::Mol->new(name => value, ...)

Create a new Mol object with the specified attributes.

```
$mol = Chemistry::Mol->new(id => 'm123', name => 'my mol')
```

is the same as

```
Chemistry::Mol->new()
$mol->id('m123')
$mol->name('my mol')
```

$mol->add_atom($atom, ...)

Add one or more Atom objects to the molecule. Returns the last atom added.

$mol->atom_class

Returns the atom class that a molecule or molecule class expects to use by default. *Chemistry::Mol* objects return "Chemistry::Atom", but subclasses will likely override this method.

$mol->new_atom(name => value, ...)

Shorthand for `$mol->add_atom($mol->atom_class->new(name => value, ...))`.

$mol->delete_atom($atom, ...)

Deletes an atom from the molecule. It automatically deletes all the bonds in which the atom participates as well. $atom should be a Chemistry::Atom reference. This method also accepts the atom index, but this use is deprecated (and buggy if multiple indices are given, unless they are in descending order).

$mol->add_bond($bond, ...)

Add one or more Bond objects to the molecule. Returns the last bond added.

$mol->bond_class

Returns the bond class that a molecule or molecule class expects to use by default. *Chemistry::Mol* objects return "Chemistry::Bond", but subclasses will likely override this method.

$mol->new_bond(name => value, ...)

Shorthand for `$mol->add_bond($mol->bond_class->new(name => value, ...))`.

$mol->delete_bond($bond, ...)

Deletes a bond from the molecule. $bond should be a *Chemistry::Bond* object.

$mol->by_id($id)

Return the atom or bond object with the corresponding id.

$mol->atoms($n1, ...)

Returns the atoms with the given indices, or all by default. Indices start from one, not from zero.

$mol->atoms_by_name($name)

Returns the atoms with the given name (treated as an anchored regular expression).

$mol->sort_atoms($sub_ref)

Sort the atoms in the molecule by using the comparison function given in $sub_ref. This function should take two atoms as parameters and return -1, 0, or 1 depending on whether the first atom should go before, same, or after the second atom. For example, to sort by atomic number, you could use the following:

```
$mol->sort_atoms( sub { $_[0]->Z <=> $_[1]->Z } );
```

Note that the atoms are passed as parameters and not as the package variables $a and $b like the core sort function does. This is because $mol->sort will likely be called from another package and we don't want to play with another package's symbol table.

$mol->bonds($n1, ...)

Returns the bonds with the given indices, or all by default. Indices start from one, not from zero.

$mol->print(option => value...)

Convert the molecule to a string representation. If no options are given, a default YAML-like format is used (this may change in the future). Otherwise, the format should be specified by using the `format` option.

$s = $mol->sprintf($format)

Format interesting molecular information in a concise way, as specified by a printf-like format.

```
%n - name
%f - formula
%f{formula with format} - (note: right braces within
    the format should be escaped with a backslash)
%s - SMILES representation
%S - canonical SMILES representation
%m - mass
%8.3m - mass, formatted as %8.3f with core sprintf
%q - formal charge
%a - atom count
%b - bond count
%t - type
%i - id
%% - %
```

For example, if you want just about everything:

```
$mol->sprintf("%s - %n (%f). %a atoms, %b bonds; "
    . "mass=%m; charge =%q; type=%t; id=%i");
```

Note that you have to `use Chemistry::File::SMILES` before using `%s` or `%S` on `$mol->sprintf`.

$mol->printf($format)

Same as `$mol->sprintf`, but prints to standard output automatically. Used for quick and dirty molecular information dumping.

Chemistry::Mol->parse($string, option => value...)

Parse the molecule encoded in `$string`. The format should be specified with the the `format` option; otherwise, it will be guessed.

Chemistry::Mol->read($fname, option => value ...)

Read a file and return a list of Mol objects, or croaks if there was a problem. The type of file will be guessed if not specified via the `format` option.

Note that only registered file readers will be used. Readers may be registered using `register_format()`; modules that include readers (such as *Chemistry::File::PDB*) usually register them automatically when they are loaded.

Automatic decompression of gzipped files is supported if the *Compress::Zlib* module is installed. Files ending in .gz are assumed to be compressed; otherwise it is possible to force decompression by passing the gzip => 1 option (or no decompression with gzip => 0).

$mol->write($fname, option => value ...)

Write a molecule file, or croak if there was a problem. The type of file will be guessed if not specified via the `format` option.

Note that only registered file formats will be used.

Automatic gzip compression is supported if the IO::Zlib module is installed. Files ending in .gz are assumed to be compressed; otherwise it is possible to force compression by passing the gzip => 1 option (or no compression with gzip => 0). Specific compression levels between 2 (fastest) and 9 (most compressed) may also be used (e.g., gzip => 9).

Chemistry::Mol->file($file, option => value ...)

Create a *Chemistry::File*-derived object for reading or writing to a file. The object can then be used to read the molecules or other information in the file.

This has more flexibility than calling `Chemistry::Mol->read` when dealing with multi-molecule files or files that have higher structure or that have information that does not belong to the molecules themselves. For example, a reaction file may have a list of molecules, but also general information like the reaction name, yield, etc. as well as the classification of the molecules as reactants or products. The exact information that is available will depend on the file reader class that is being used. The following is a hypothetical example for reading MDL rxnfiles.

```
# assuming this module existed...
use Chemistry::File::Rxn;

my $rxn = Chemistry::Mol->file('test.rxn');
$rxn->read;
$name      = $rxn->name;
@reactants = $rxn->reactants; # mol objects
```

```
@products  = $rxn->products;
$yield     = $rxn->yield;     # a number
```

Note that only registered file readers will be used. Readers may be registered using register_format(); modules that include readers (such as Chemistry::File::PDB) usually register them automatically.

Chemistry::Mol->register_format($name, $ref)

Register a file type. The identifier $name must be unique. $ref is either a class name (a package) or an object that complies with the *Chemistry::File* interface (e.g., a subclass of Chemistry::File). If $ref is omitted, the calling package is used automatically. More than one format can be registered at a time, but then $ref must be included for each format (e.g., Chemistry::Mol->register_format(format1 => "package1", format2 => package2).

The typical user doesn't have to care about this function. It is used automatically by molecule file I/O modules.

Chemistry::Mol->formats

Returns a list of the file formats that have been installed by register_format()

$mol->mass

Return the molar mass. This is just the sum of the masses of the atoms. See *Chemistry::Atom*::mass for details such as the handling of isotopes.

$mol->charge

Return the charge of the molecule. By default it returns the sum of the formal charges of the atoms. However, it is possible to set an arbitrary charge by calling `$mol->charge($new_charge)`

$mol->formula_hash

Returns a hash reference describing the molecular formula. For methane it would return { C => 1, H => 4 }.

$mol->formula($format)

Returns a string with the formula. The format can be specified as a printf-like string with the control sequences specified in the *Chemistry::File::Formula* documentation.

my $mol2 = $mol->clone;

Makes a copy of a molecule. Note that this is a **deep** copy; if your molecule has a pointer to the rest of the universe, the entire universe will be cloned!

my $mol2 = $mol->safe_clone;

Like clone, it makes a deep copy of a molecule. The difference is that the copy is not "exact" in that new molecule and its atoms and bonds get assigned new IDs. This makes it safe to combine cloned molecules. For example, this is an error:

```
# XXX don't try this at home!
my $mol2 = Chemistry::Mol->combine($mol1, $mol1);
# the atoms in $mol1 will clash
```

But this is ok:

```
# the "safe clone" of $mol1 will have new IDs
my $mol2 = Chemistry::Mol->combine($mol1, $mol1->safe_clone);
```

($distance, $atom_here, $atom_there) = $mol->distance($obj)

Returns the minimum distance to $obj, which can be an atom, a molecule, or a vector. In scalar context it returns only the distance; in list context it also returns the atoms involved. The current implementation for calculating the minimum distance between two molecules compares every possible pair of atoms, so it's not efficient for large molecules.

my $bigmol = Chemistry::Mol->combine($mol1, $mol2, ...)

$mol1->combine($mol2, $mol3, ...)

Combines several molecules in one bigger molecule. If called as a class method, as in the first example, it returns a new combined molecule without altering any of the parameters. If called as an instance method, as in the second example, all molecules are combined into $mol1 (but $mol2, $mol3, ...) are not altered. **Note**: Make sure you don't combine molecules which contain atoms with duplicate IDs (for example, if they were cloned).

my @mols = $mol->separate

Separates a molecule into "connected fragments". The original object is not modified; the fragments are clones of the original ones. Example: if you have ethane (H3CCH3) and you delete the C-C bond, you have two CH3 radicals within one molecule object ($mol). When you call $mol->separate you get two molecules, each one with a CH3.

$mol->sprout_hydrogens

Convert all the implicit hydrogen atoms in the molecule to explicit atoms. It does **not** generate coordinates for the atoms.

$mol->collapse_hydrogens

Convert all the explicit hydrogen atoms in the molecule to implicit hydrogens. (Exception: hydrogen atoms that are adjacent to a hydrogen atom are not collapsed.)

$mol->add_implicit_hydrogens

Use heuristics to figure out how many implicit hydrogens should each atom in the molecule have to satisfy its normal "organic" valence.

Chemistry::Mol->register_descriptor($name => $sub_ref)

Adds a callback that can be used to add functionality to the molecule class (originally meant to add custom molecule descriptors.) A descriptor is a function that takes a molecule object as its only argument and returns a value or values. For example, to add a descriptor function that computes the number of atoms:

```
Chemistry::Mol->register_descriptor(
    number_of_atoms => sub {
        my $mol = shift;
        return scalar $mol->atoms;
    }
);
```

The descriptor is accessed by name via the `descriptor` instance method:

```
my $n = $mol->descriptor('number_of_atoms');
```

my $value = $mol->descriptor($descriptor_name)

Calls a previously registered descriptor function giving it $mol as an argument, as shown above for `register_descriptor`.

VERSION

0.37

SEE ALSO

Chemistry::Atom, *Chemistry::Bond*, *Chemistry::File*, *Chemistry::Tutorial*
The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

5 Chemistry::Pattern

Chemical substructure pattern matching

SYNOPSIS

```
use Chemistry::Pattern;
use Chemistry::Mol;
use Chemistry::File::SMILES;

# Create a pattern and a molecule from SMILES strings
my $mol_str = "C1CCCC1C(Cl)=O";
my $patt_str = "C(=O)Cl";
my $mol = Chemistry::Mol->parse($mol_str, format => 'smiles');
my $patt = Chemistry::Pattern->parse($patt_str, format => 'smiles');

# try to match the pattern
while ($patt->match($mol)) {
    @matched_atoms = $patt->atom_map;
    print "Matched: (@matched_atoms)\n";
    # should print something like "Matched: (a6 a8 a7)"
}
```

DESCRIPTION

This module implements basic pattern matching for molecules. The Chemistry::Pattern class is a subclass of Chemistry::Mol, so patterns have all the properties of molecules and can come from reading the same file formats. Of course there are certain formats (such as SMARTS) that are exclusively used to describe patterns.

To perform a pattern matching operation on a molecule, follow these steps.

1) Create a pattern object, either by parsing a file or string, or by adding atoms and bonds by hand by using Chemistry::Mol methods. Note that atoms and bonds in a pattern should be Chemistry::Pattern::Atom and Chemistry::Patern::Bond objects. Let's assume that the pattern object is stored in $patt and that the molecule is $mol.

2) Execute the pattern on the molecule by calling $patt->match($mol).

3) If $patt->match() returns true, extract the "map" that relates the pattern to the molecule by calling $patt->atom_map or $patt->bond_map. These methods return a list of the atoms or bonds in the molecule that are matched by the corresponding atoms in the pattern. Thus $patt->atom_map(1) would be analogous to the $1 special variable used for regular expresion matching. The difference between Chemistry::Pattern and Perl regular expressions is that atoms and bonds are always captured.

4) If more than one match for the molecule is desired, repeat from step (2) until match() returns false.

METHODS

Chemistry::Pattern->new(name => value, ...)

Create a new empty pattern. This is just like the Chemistry::Mol constructor, with one additional option: "options", which expects a hash reference (the options themselves are described under the options() method).

$pattern->options(option => value,...)

Available options:

overlap

If true, matches may overlap. For example, the CC pattern could match twice on propane if this option is true, but only once if it is false. This option is true by default.

permute

Sometimes there is more than one way of matching the same set of pattern atoms on the same set of molecule atoms. If true, return these "redundant" matches. For example, the CC pattern could match ethane with two different permutations (forwards and backwards). This option is false by default.

$patt->reset

Reset the state of the pattern matching object, so that it begins the next match from scratch instead of where it left off after the last one.

$pattern->atom_map

Returns the list of atoms that matched the last time $pattern->match was called.

$pattern->bond_map

Returns the list of bonds that matched the last time $pattern->match was called.

$pattern->match($mol, %options)

Returns true if the pattern matches the molecule. If called again for the same molecule, continues matching where it left off (in a way similar to global regular expressions under scalar context). When there are no matches left, returns false. To force the match to always start from scratch instead of continuing where it left off, the `reset` option may be used.

```
$pattern->match($mol, atom => $atom)
```

If atom => $atom is given as an option, match will only look for matches that start at $atom (which should be an atom in $mol, of course). This is somewhat analog to anchored regular expressions.

To find out which atoms and bonds matched, use the atom_map and bond_map methods.

VERSION

0.27

SEE ALSO

Chemistry::Pattern::Atom, *Chemistry::Pattern::Bond*, *Chemistry::Mol*, *Chemistry::File*, *Chemistry::File::SMARTS*.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

6 Chemistry::FormulaPattern

Match molecule by formula

SYNOPSIS

```
use Chemistry::FormulaPattern;

# somehow get a bunch of molecules...
use Chemistry::File::SDF;
my @mols = Chemistry::Mol->read("file.sdf");

# we want molecules with six carbons and 8 or more hydrogens
my $patt = Chemistry::FormulaPattern->new("C6H8-");

for my $mol (@mols) {
    if ($patt->match($mol)) {
        print $mol->name, " has a nice formula!\n";
    }
}

# a concise way of selecting molecules with grep
my @matches = grep { $patt->match($mol) } @mols;
```

DESCRIPTION

This module implements a simple language for describing a range of molecular formulas and allows one to find out whether a molecule matches the formula specification. It can be used for searching for molecules by formula, in a way similar to the NIST WebBook formula search (http://webbook.nist.gov/chemistry/form-ser.html). Note however that the language used by this module is different from the one used by the WebBook!

Chemistry::FormulaPattern shares the same interface as *Chemistry::Pattern*. To perform a pattern matching operation on a molecule, follow these steps.

1) Create a pattern object, by parsing a string. Let's assume that the pattern object is stored in $patt and that the molecule is $mol.

2) Execute the pattern on the molecule by calling $patt->match($mol).

If $patt->match returns true, there was a match. If $patt->match is called two consecutive times with the same molecule, it returns false; then true (if there is a match), then false, etc. This is because the Chemistry::Pattern interface is designed to allow multiple matches for a given molecule, and then returns false when there are no further matches; in the case of a formula pattern, there is only one possible match.

```
$patt->match($mol); # may return true
$patt->match($mol); # always false
$patt->match($mol); # may return true
$patt->match($mol); # always false
# ...
```

This allows one two use the standard while loop for all kinds of patterns without having to worry about endless loops:

```
# $patt might be a Chemistry::Pattern, Chemistry::FormulaPattern,
# or Chemistry::MidasPattern object
while ($patt->match($mol)) {
    # do something
}
```

Also note that formula patterns don't really have the concept of an atom map, so $patt->atom_map and $patt->bond_map always return the empty list.

FORMULA PATTERN LANGUAGE

In the simplest case, a formula pattern may be just a regular formula, as used by the *Chemistry::File::Formula* module. For example, the pattern "C6H6" will only match molecules with six carbons, six hydrogens, and no other atoms.

The interesting thing is that one can also specify ranges for the elements, as two hyphen-separated numbers. "C6H8-10" will match molecules with six carbons and eight to ten hydrogens.

Ranges may also be open, by omitting the upper part of the range. "C6H0-" will match molecules with six carbons and any number of hydrogens (i.e., zero or more).

A formula pattern may also allow for unspecified elements by means of the asterisk special character, which can be placed anywhere in the formula pattern. For example, "C2H6*" (or "C2*H6, etc.) will match C2H6, and also C2H6O, C2H6S, C2H6SO, etc.

Ranges can also be used after a subformula in parentheses: "(CH2)1-2" will match molecules with one or two carbons and two to four hydrogens. Note, however, that the "structure" of the bracketed part of the formula is forgotten, i.e., the multiplier applies to each element individually and does not have to be an integer. That is, the above pattern will match CH2, CH3, CH4, C2H2, C2H3, and C2H4.

VERSION

0.10

SEE ALSO

Chemistry::Pattern, *Chemistry::File::FormulaPattern*

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

7 Chemistry::MidasPattern

Select atoms in macromolecules

SYNOPSIS

```
use Chemistry::MidasPattern;
use Chemistry::File::PDB;

# read a molecule
my $mol = Chemistry::MacroMol->read("test.pdb");

# define a pattern matching carbons alpha and beta
# in all valine residues
my $str  = ':VAL@CA,CB';
my $patt = Chemistry::MidasPattern->new($str);

# apply the pattern to the molecule
$patt->match($mol);

# extract the results
for my $atom ($patt->atom_map) {
    printf "%s\t%s\n",  $atom->attr("pdb/residue_name"), $atom->name;
}
printf "FOUND %d atoms\n", scalar($patt->atom_map);
```

DESCRIPTION

This module partially implements a pattern matching engine for selecting atoms in macromolecules by using Midas/Chimera patterns. See http://www.cmpharm.ucsf.edu/~troyer/troff2html/midas/Midas-uh-3.html#sh-2.1 for a detailed description of this language.

This module shares the same interface as *Chemistry::Pattern*; to perform a pattern matching operation on a molecule, follow these steps.

1) Create a pattern object, by parsing a string. Let's assume that the pattern object is stored in $patt and that the molecule is $mol.

2) Execute the pattern on the molecule by calling $patt->match($mol).

3) If $patt->match() returns true, extract the "map" that relates the pattern to the molecule by calling $patt->atom_map. These method returns a list of the atoms in the molecule that are matched by the pattern. Thus $patt->atom_map(1) would be analogous to the $1 special variable used for regular expresion matching. The difference between Chemistry::Pattern and Perl regular expressions is that atoms are always captured, and that each atom always uses one "slot".

MIDAS ATOM SPECIFICATION LANGUAGE QUICK SUMMARY

The current implementation does not have the concept of a model, only of residues and atoms.

What follows is not exactly a formal grammar specification, but it should give a general idea:

SELECTOR = ((:RESIDUE(.CHAIN)?)*(@ATOM)*)*

The star here means "zero or more", the question mark means "zero or one", and the parentheses are used to delimit the effect of the star. All other characters are used verbatim.

RESIDUE can be a name (e.g., LYS), a sequence number (e.g., 108), a range (e.g., 1-10), or a comma-separated list of RESIDUEs (e.g. 1-10,6,LYS).

ATOM is an atom name, a serial number (this is a non-standard extension) or a comma-separated list of ATOMs.

Names can have wildcards: * matches the whole name; ? matches one character; and = matches zero or more characters. An @ATOM specification is asociated with the closest preceding residue specification.

DISTANCE_SELECTOR = SELECTOR za< DISTANCE

Atoms within a certain distance of those that are matched by a selector can be selected by using the za< operator, where DISTANCE is a number in Angstroms.

EXPR = (SELECTOR | DISTANCE_SELECTOR) (& (SELECTOR | DISTANCE_SELECTOR))*

The result of two or more selectors can be intersected using the & operator.

EXAMPLES

```
:ARG                All arginine atoms
:ARG.A              All arginine atoms in chain 'A'
:ARG@*              All arginine atoms
@CA                 All alpha carbons
:*@CA               All alpha carbons
:ARG@CA             Arginine alpha carbons
:VAL@C=             Valine carbons
:VAL@C?             Valine carbons with two-letter names
:ARG,VAL@CA         Arginine and valine alpha carbons
:ARG:VAL@CA         All arginine atoms and valine alpha carbons
:ARG@CA,CB          Arginine alpha and beta carbons
:ARG@CA@CB          Arginine alpha and beta carbons
:1-10               Atoms in residues 1 to 10
:48-*               Atoms in residues 11 to the last one
:30-40@CA & :ARG    Alpha carbons in residues 1-10 which are
                    also arginines.
@123                Atom 123
@123 za<5.0         Atoms within 5.0 Angstroms of atom 123
@123 za>30.0        Atoms not within 30.0 Angstroms of atom 123
@CA & @123 za<5.0   Alpha carbons within 5.0 Angstroms of atom 123
```

CAVEATS

If a feature does not appear in any of the examples, it is probably not implemented. For example, the zr< zone specifier, atom properties, and various Chimera extensions.

The zone specifiers (selection by distance) currently use a brute-force N^2 algorithm. You can optimize an & expression by putting the most unlikely selectors first; for example

```
:1-20 zr<10.0 & :38         atoms in residue 38 within 10 A of atoms
                            in residues 1-20 (slow)

:38 & :1-20 zr<10.0         atoms in residue 38 within 10 A of atoms
                            in residues 1-20 (not so slow)
```

In the first case, the N^2 search measures the distance between every atom in the molecule and every atom in residues 1-20, and then intersects the results with the atom list of residue 28; the second case only measures the distance between every atom in residue 38 with every atom in residues 1-20. The second way is much, much faster for large systems.

Some day, a future version may implement a smarter algorithm...

VERSION

0.11

SEE ALSO

Chemistry::File::MidasPattern, *Chemistry::Pattern*

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert <itub@cpan.org>

COPYRIGHT

8 Chemistry::Reaction

Explicit chemical reactions

SYNOPSIS

```
use Chemistry::Reaction;
use Chemistry::File::SMILES;

my $s = Chemistry::Pattern->parse('C=CC=C.C=C', format=>'smiles');
my $p = Chemistry::Pattern->parse('C1=CCCCC1', format=>'smiles');
my %m;
for (my $i = 1; $i <= $s->atoms; $i++) {
  $m{$s->atoms($i)} = $p->atoms($i);
}
my $r = Chemistry::Reaction->new($s, $p, \%m);
```

DESCRIPTION

This package, along with Chemistry::Pattern, provides an implementation of explicit chemical reactions.

An explicit chemical reaction is a representation of the transformation that takes place in a given chemical reaction. In an explicit chemical reaction, a substrate molecule is transformed into a product molecule by breaking existing bonds and creating new bonds between atoms.

The representation of an explicit chemical reaction is a molecule in which the order of a bond before the chemical reaction is distinguished from the order of the bond after the chemical reaction. Thus, the breaking of an existing bond is represented by one of the following before/after pairs:

```
3/2, 2/1, 1/0 (breaking of a single bond or reduce order by one)
     3/1, 2/0 (breaking of a double bond or reduce order by two)
          3/0 (breaking of a triple bond)
```

The creation of a new bond is represented by one of the following before/after pairs:

```
0/1, 1/2, 2/3 (creation of a single bond or increase order by one)
     0/2, 1/3 (creation of a double bond or increase order by two)
          0/3 (creation of a triple bond)
```

An explicit chemical reaction $react can be forward or reverse applied once to a molecule $mol at the first subgraph of $mol found which is isomorphic to the substrate or product of $react:

```
my $subst = $react->substrate;
if ($subst->match($mol)) {
  $react->forward($mol, $subst->atom_map);
}
```

Also, an explicit chemical reaction $react can be forward or reverse applied once to a molecule $mol at each subgraph of $mol which is isomorphic to the substrate or product of $react:

```
my $subst = $react->substrate;
my @products;
while ($subst->match($mol)) {
  my $new_mol = $mol->clone; # start from a fresh molecule
  my @map = $subst->atom_map;
  # translate atom map to the clone
  my @m = map { $new_mol->by_id($_->id) } @map;
  $react->forward($new_mol, @m);
  push @products, $new_mol;
}
```

Furthermore, an explicit chemical reaction $react can be forward or reverse applied as long as possible to a molecule $mol at the first subgraph of $mol found which is isomorphic to the substrate or product of $react:

```
my $subst = $react->substrate;
while ($subst->match($mol)) {
  $react->forward($mol, $subst->atom_map);
}
```

METHODS

Chemistry::Reaction->new($subst, $prod, \%map)

Create a new Reaction object that describes the transformation of the $subst substrate into the $prod product, according to the %map mapping of substrate atoms to product atoms.

$react->substrate

Return a Chemistry::Pattern object that represents the substrate molecules of the explicit chemical reaction $react.

$react->product

Return a Chemistry::Pattern object that represents the product molecules of the explicit chemical reaction $react.

$react->forward($mol, @map)

Forward application of the explicit chemical reaction $react to the molecule $mol, according to the mapping @map of substrate atoms to $mol atoms. The substrate of the explicit chemical reaction $react must be a subgraph of the molecule $mol. Return the modified molecule $mol.

$react->reverse($mol, @map)

Reverse application of the explicit chemical reaction $react to the molecule $mol, according to the mapping @map of product atoms to $mol atoms. The product of the explicit chemical reaction $react must be a subgraph of the molecule $mol. Return the modified molecule $mol.

VERSION

0.02

SEE ALSO

Chemistry::Mol, *Chemistry::Pattern*, *Chemistry::Tutorial*

Rosselló, F. and G. Valiente, Analysis of metabolic pathways by graph transformation, in: Proc. 2nd Int. Conf. Graph Transformation, Lecture Notes in Computer Science 3256 (2004), pp. 73–85.

Rosselló, F. and G. Valiente, Chemical graphs, chemical reaction graphs, and chemical graph transformation, in: Proc. 2nd Int. Workshop on Graph-Based Tools, Electronic Notes in Theoretical Computer Science (2004), in press.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org> and Gabriel Valiente <valiente@lsi.upc.es>

COPYRIGHT

9 Chemistry::Domain

Class for domains in macromolecules

SYNOPSIS

```
use Chemistry::Domain;
my $domain = Chemistry::Domain->new(parent => $bigmol);
```

DESCRIPTION

A domain is a substructure of a larger molecule. It is typically used to represent aminoacid residues within a protein, or bases within a nucleic acid, but you could use it for any arbitrary substructure such as functional groups and rings. A domain has all the properties of a molecule, plus a "parent". The parent is the larger molecule that contains the domain. In other words, the Chemistry::Domain class inherits from Chemistry::Mol.

METHODS

Note: the methods that are inherited from Chemistry::Mol are not repeated here.

Chemistry::Domain->new(parent => $mol, name => value, ...)

Create a new Domain object with the specified attributes. You can use the same attributes as for Chemistry::Mol->new, plus the parent attribute, which is required.

$domain->parent

Returns the parent of the domain.

$domain->add_atom($atom, ...)

Add one or more Atom objects to the domain. Returns the last atom added. It also automatically adds the atoms to the atom table of the parent molecule.

$domain->add_bond($bond, ...)

Add one or more Bond objects to the domain. Returns the last bond added. It also automatically adds the bond to the bond table of the parent molecule.

VERSION

0.06

SEE ALSO

Chemistry::MacroMol, Chemistry::Mol, Chemistry::Atom, Chemistry::Bond

AUTHOR

Ivan Tubert, <itub@cpan.org>

COPYRIGHT AND LICENSE

10 Chemistry::MacroMol

Perl module for macromolecules

SYNOPSIS

```
use Chemistry::MacroMol;

my $mol = Chemistry::MacroMol->new(name => 'my big molecule');
$mol->new_domain(name => "ASP"); # see Chemistry::Domain for details
my @domains = $mol->domains;
```

DESCRIPTION

For the purposes of this module, a macromolecule is just a molecule that consists of several "domains". For example, a protein consists of aminoacid residues, or a nucleic acid consists of bases. Therefore Chemistry::MacroMol is derived from Chemistry::Mol, with additional methods to handle the domains.

The way things are currently structured, an atom in a macromolecule "belong" both to the MacroMol object and to a Domain object. This way you can get all the atoms in $protein via $protein->atoms, or to the atoms in residue 123 via $protein->domain(123)->atoms.

METHODS

Remember that this class inherits all the methods from Chemistry::Mol. They won't be repeated here.

Chemistry::MacroMol->new(name => value, ...)

Create a new MacroMol object with the specified attributes. You can use the same attributes as for Chemistry::Mol->new.

$mol->add_domain($domain, ...)

Add one or more Domain objects to the molecule. Returns the last domain added.

$mol->domain_class

Returns the domain class that a macromolecule class expects to use by default. Chemistry::MacroMol objects return "Chemistry::Domain", but subclasses will likely override this method.

$mol->new_domain(name => value, ...)

Shorthand for $mol->add_domain($mol->domain_class->new(parent => $mol, name => value, ...));

$mol->domains($n1, ...)

Returns the domains with the given indices, or all by default. NOTE: the indices start from one (1), not from zero.

VERSION

0.06

SEE ALSO

Chemistry::Domain, *Chemistry::Mol*

AUTHOR

Ivan Tubert, <itub@cpan.org>

COPYRIGHT AND LICENSE

11 Chemistry::Ring

Represent a ring as a substructure of a molecule

SYNOPSIS

```
use Chemistry::Ring;

# already have a molecule in $mol...
# create a ring with the first six atoms in $mol
my $ring = Chemistry::Ring->new;
$ring->add_atom($_) for $mol->atoms(1 .. 6);

# find the centroid
my $vector = $ring->centroid;

# find the plane that fits the ring
my ($normal, $distance) = $ring->plane;

# is the ring aromatic?
print "is aromatic!\n" if $ring->is_aromatic;

# "aromatize" a molecule
Chemistry::Ring::aromatize_mol($mol);

# get the rings involving an atom (after aromatizing)
my $rings = $mol->atoms(3)->attr('ring/rings');
```

DESCRIPTION

This module provides some basic methods for representing a ring. A ring is a subclass of molecule, because it has atoms and bonds. Besides that, it has some useful geometric methods for finding the centroid and the ring plane, and methods for aromaticity detection.

This module does not detect the rings by itself; for that, look at *Chemistry::Ring::Find*.

This module is part of the PerlMol project, http://www.perlmol.org/.

METHODS

Chemistry::Ring->new(name => value, ...)

Create a new Ring object with the specified attributes. Same as `Chemistry::Mol->new`.

$ring->centroid

Returs a vector with the centroid, defined as the average of the coordinates of all the atoms in the ring. The vecotr is a *Math::VectorReal* object.

my ($norm, $d) = $ring->plane

Returns the normal and distance to the origin that define the plane that best fits the atoms in the ring, by using multivariate regression. The normal vector is a *Math::VectorReal* object.

$ring->is_aromatic

Naively guess whether ring is aromatic from the molecular graph, with a method based on Huckel's rule. This method is not very accurate, but works for simple molecules. Returns true or false.

EXPORTABLE SUBROUTINES

Nothing is exported by default, but you can export these subroutines explicitly, or all of them by using the ':all' tag.

aromatize_mol($mol)

Finds all the aromatic rings in the molecule and marks all the atoms and bonds in those rings as aromatic.

It also adds the 'ring/rings' attribute to the molecule and to all ring atoms and bonds; this attribute is an array reference containing the list of rings that involve that atom or bond (or all the rings in the case of the molecule). NOTE (the ring/rings attribute is experimental and might change in future versions).

VERSION

0.20

SEE ALSO

Chemistry::Mol, *Chemistry::Atom*, *Chemistry::Ring::Find*, *Math::VectorReal*.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

12 Chemistry::Atom

Chemical atoms as objects in molecules

SYNOPSIS

```
use Chemistry::Atom;

my $atom = new Chemistry::Atom(
    id => 'a1',
    coords => [$x, $y, $z],
    symbol => 'Br'
);

print $atom->print;
```

DESCRIPTION

This module includes objects to describe chemical atoms. An atom is defined by its symbol and its coordinates, among other attributes. Atomic coordinates are described by a Math::VectorReal object, so that they can be easily used in vector operations.

Atom Attributes

In addition to common attributes such as id, name, and type, atoms have the following attributes, which are accessed or modified through methods defined below: bonds, coords, internal_coords, Z, symbol, etc.

In general, to get the value of a property, use $atom->method without any parameters. To set the value, use $atom->method($new_value). When setting an attribute, the accessor returns the atom object, so that accessors can be chained:

```
$atom->symbol("C")->name("CA")->coords(1,2,3);
```

METHODS

Chemistry::Atom->new(name => value, ...)

Create a new Atom object with the specified attributes.

$atom->Z($new_Z)

Sets and returns the atomic number (Z). If the symbol of the atom doesn't correspond to a known element, Z = undef.

$atom->symbol($new_symbol)

Sets and returns the atomic symbol.

$atom->mass($new_mass)

Sets and returns the atomic mass in atomic mass units. Any arbitrary mass may be set explicitly by using this method. However, if no mass is set explicitly and this method is called as an accessor, the return value is the following:

1) If the mass number is undefined (see the mass_number method below), the relative atomic mass from the 1995 IUPAC recommendation is used. (Table stolen from the Chemistry::MolecularMass module by Maksim A. Khrapov).

2) If the mass number is defined and the *Chemistry::Isotope* module is available and it knows the mass for the isotope, the exact mass of the isotope is used; otherwise, the mass number is returned.

$atom->mass_number($new_mass_number)

Sets or gets the mass number. The mass number is undefined unless is set explicitly (this module does not try to guess a default mass number based on the natural occurring isotope distribution).

$atom->coords

```
my $vector = $atom->coords;  # get a Math::VectorReal object
$atom->coords($vector);      # set a Math::VectorReal object
$atom->coords([$x, $y, $z]); # also accepts array refs
$atom->coords($x, $y, $z);   # also accepts lists
```

Sets or gets the atom's coordinates. It can take as a parameter a Math::VectorReal object, a reference to an array, or the list of coordinates.

$atom->internal_coords

```
# get a Chemistry::InternalCoords object
my $ic = $atom->internal_coords;

# set a Chemistry::InternalCoords object
$atom->internal_coords($vic);

# also accepts array refs
$atom->internal_coords([8, 1.54, 7, 109.47, 6, 120.0]);

# also accepts lists
$atom->internal_coords(8, 1.54, 7, 109.47, 6, 120.0);
```

Sets or gets the atom's internal coordinates. It can take as a parameter a Chemistry::InternalCoords object, a reference to an array, or the list of coordinates. In the last two cases, the list elements are the following: atom number or reference for distance, distance, atom number or reference for angle, angle in degrees, atom number or reference for dihedral, dihedral in degrees.

$atom->x3, $atom->y3, $atom->z3

Get the x, y or z 3D coordinate of the atom. This methods are just accessors that don't change the coordinates. $atom->x3 is short for ($atom->coords->array)[0], and so on.

$atom->formal_charge($charge)

Set or get the formal charge of the atom.

$atom->formal_radical($radical)

Set or get the formal radical multiplicity for the atom.

$atom->implicit_hydrogens($h_count)

Set or get the number of implicit hydrogen atoms bonded to the atom.

$atom->hydrogens($h_count)

Set or get the number of implicit hydrogen atoms bonded to the atom (DEPRECATED: USE `implicit_hydrogens` INSTEAD).

$atom->total_hydrogens($h_count)

Get the total number of hydrogen atoms bonded to the atom (implicit + explicit).

$atom->sprout_hydrogens

Convert all the implicit hydrogens for this atom to explicit hydrogens. Note: it does **not** generate coordinates for the new atoms.

$atom->collapse_hydrogens

Delete neighboring hydrogen atoms and add them as implicit hydrogens for this atom.

$atom->calc_implicit_hydrogens

Use heuristics to figure out how many implicit hydrogens should the atom have to satisfy its normal "organic" valence. Returns the number of hydrogens but does not affect the atom object.

$atom->add_implicit_hydrogens

Similar to calc_implicit_hydrogens, but it also sets the number of implicit hydrogens in the atom to the new calculated value. Equivalent to

```
$atom->implicit_hydrogens($atom->calc_implicit_hydrogens);
```

It returns the atom object.

$atom->aromatic($bool)

Set or get whether the atom is considered to be aromatic. This property may be set arbitrarily, it doesn't imply any kind of "intelligent aromaticity detection"! (For that, look at the *Chemistry::Ring* module).

$atom->valence

Returns the sum of the bond orders of the bonds in which the atom participates, including implicit hydrogens (which are assumed to have bond orders of one).

$atom->explicit_valence

Like `valence`, but excluding implicit hydrogen atoms. To get the raw number of bonds, without counting bond orders, call $atom->bonds in scalar context.

$atom->delete

Calls $mol->delete_atom($atom) on the atom's parent molecule.

$atom->parent

Returns the atom's containing object (the molecule to which the atom belongs). An atom can only have one parent.

$atom->neighbors($from)

Return a list of neighbors. If an atom object $from is specified, it will be excluded from the list (this is useful if an atom wants to know who its neighbor's neighbors are, without counting itself).

$atom->bonds($from)

Return a list of bonds. If an atom object $from is specified, bonds to that atom will be excluded from the list.

$atom->bonds_neighbors($from)

Return a list of hash references, representing the bonds and neighbors from the atom. If an atom object $from is specified, it will be excluded from the list. The elements of the hash are 'to', and atom reference, and 'bond', a bond reference. For example,

```
for my $bn ($atom->bonds_neighbors) {
    print "bond $bn->{bond} point to atom $bn->{to}\n";
}
```

($distance, $closest_atom) = $atom->distance($obj)

Returns the minimum distance to $obj, which can be an atom, a molecule, or a vector. In scalar context it returns only the distance; in list context it also returns the closest atom found. It can also be called as a function, Chemistry::Atom::distance (which can be exported).

$atom->angle($atom2, $atom3)

Returns the angle in radians between the atoms involved. $atom2 is the atom in the middle. Can also be called as Chemistry::Atom::angle($atom1, $atom2, $atom3). This function can be exported. Note: if you override this method, you may also need to override angle_deg or strange things may happen.

$atom->angle_deg($atom2, $atom3)

Same as angle(), but returns the value in degrees. May be exported.

$atom->dihedral($atom2, $atom3, $atom4)

Returns the dihedral angle in radians between the atoms involved. Can also be called as Chemistry::Atom::dihedral($atom1, $atom2, $atom3, $atom4). May be exported. Note: if you override this method, you may also need to override dihedral_deg and angle or strange things may happen.

$atom->dihedral_deg($atom2, $atom3, $atom4)

Same as dihedral(), but returns the value in degrees. May be exported.

$atom->print

Convert the atom to a string representation (used for debugging).

my $info = $atom->sprintf($format)

Format interesting atomic information in a concise way, as specified by a printf-like format.

```
%s - symbol
%Z - atomic number
%n - name
%q - formal charge
%h - implicit hydrogen count
%v - valence
%i - id
%8.3m - mass, formatted as %8.3f with core sprintf
%8.3x - x coordinate, formatted as %8.3f with core sprintf
%8.3y - y coordinate, formatted as %8.3f with core sprintf
%8.3z - z coordinate, formatted as %8.3f with core sprintf
%% - %
```

$atom->printf($format)

Same as $atom->sprintf, but prints to standard output automatically. Used for quick and dirty atomic information dumping.

VERSION

0.37

SEE ALSO

Chemistry::Mol, *Chemistry::Bond*, *Math::VectorReal*, *Chemistry::Tutorial*, *Chemistry::InternalCoords*
The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

13 Chemistry::Pattern::Atom

An atom that knows how to match

SYNOPSIS

```
my $patt_atom = Chemistry::Pattern::Atom->new(symbol => C);
$patt_atom->test_sub( sub {
    my ($what, $where) = @_;
    $where->bonds == 3 ? 1 : 0; # only match atoms with three bonds
});
```

DESCRIPTION

Objects of this class represent atoms in a pattern. This is a subclass of Chemistry::Atom. In addition to the properties of regular atoms, pattern atoms have a method for testing if they match an atom in a molecule. By default, a pattern atom matches an atom if they have the same symbol. It is possible to substitute this by an arbitrary criterion by providing a custom test subroutine.

METHODS

$patt_atom->test($atom)

Tests if the pattern atom matches the atom given by $atom. Returns true or false.

$patt_atom->test_sub(\&my_test_sub)

Specify an arbitrary test subroutine to be used instead of the default one. &my_test_sub must take two parameters; the first one is the pattern atom and the second is the atom to match. It must return true if there is a match.

$patt_atom->map_to([$atom])

Returns or sets the atom that is considered to be matched by $patt_atom.

VERSION

0.27

SEE ALSO

Chemistry::Pattern

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

14 Chemistry::Bond

Chemical bonds as objects in molecules

SYNOPSIS

```
use Chemistry::Bond;

# assuming we have molecule $mol with atoms $a1 and $a2
$bond = Chemistry::Bond->new(
    id => "b1",
    type => '=',
    atoms => [$a1, $a2]
    order => '2',
);
$mol->add_bond($bond);

# simpler way of doing the same:
$mol->new_bond(
    id => "b1",
    type => '=',
    atoms => [$a1, $a2]
    order => '2',
);
```

DESCRIPTION

This module includes objects to describe chemical bonds. A bond is defined as a list of atoms (typically two), with some associated properties.

Bond Attributes

In addition to common attributes such as id, name, and type, bonds have the order attribute. The bond order is a number, typically the integer 1, 2, 3, or 4.

METHODS

Chemistry::Bond->new(name => value, ...)

Create a new Bond object with the specified attributes. Sensible defaults are used when possible.

$bond->order()

Sets or gets the bond order.

$bond->length

Returns the length of the bond, i.e., the distance between the two atom objects in the bond. Returns zero if the bond does not have exactly two atoms.

$bond->aromatic($bool)

Set or get whether the bond is considered to be aromatic.

$bond->print

Convert the bond to a string representation.

$bond->atoms()

If called with no parameters, return a list of atoms in the bond. If called with a list (or a reference to an array) of atom objects, define the atoms in the bond and call $atom->add_bond for each atom in the list. Note: changing the atoms in a bond may have strange side effects; it is safer to treat bonds as immutable except with respect to properties such as name and type.

$bond->delete

Calls $mol->delete_bond($bond) on the bond's parent molecule. Note that a bond should belong to only one molecule or strange things may happen.

VERSION

0.37

SEE ALSO

Chemistry::Mol, *Chemistry::Atom*, *Chemistry::Tutorial*
The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

15 Chemistry::Pattern::Bond

A bond that knows how to match

SYNOPSIS

```
my $patt_bond = Chemistry::Pattern::Bond->new(order => 2);
$patt_bond->test_sub( sub {
    my ($what, $where) = @_;
    $where->type eq 'purple' ? 1 : 0; # only match purple bonds
});
```

DESCRIPTION

Objects of this class represent bonds in a pattern. This is a subclass of Chemistry::Bond. In addition to the properties of regular bonds, pattern bonds have a method for testing if they match an bond in a molecule. By default, a pattern bond matches an bond if they have the same bond order or both are aromatic. It is possible to substitute this by an arbitrary criterion by providing a custom test subroutine.

METHODS

$patt_bond->test($bond)

Tests if the pattern bond matches the bond given by $bond. Returns true or false.

$patt_bond->test_sub(\&my_test_sub)

Specify an arbitrary test subroutine to be used instead of the default one. &my_test_sub must take two parameters; the first one is the pattern bond and the second is the bond to match. It must return true if there is a match.

$patt_bond->map_to([$bond])

Returns or sets the bond that is considered to be matched by $patt_bond.

VERSION

0.27

SEE ALSO

Chemistry::Pattern

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

16 Chemistry::File

Molecule file I/O base class

SYNOPSIS

```
# As a convenient interface for several mol readers:
use Chemistry::File qw(PDB MDLMol); # load PDB and MDL modules

# or try to use every file I/O module installed in the system:
use Chemistry::File ':auto';

my $mol1 = Chemistry::Mol->read("file.pdb");
my $mol2 = Chemistry::Mol->read("file.mol");

# as a base for a mol reader:

package Chemistry::File::Myfile;
use base qw(Chemistry::File);
Chemistry::Mol->register_type("myfile", __PACKAGE__);

# override the read_mol method
sub read_mol {
    my ($self, $fh, %opts) = shift;
    my $mol_class = $opts{mol_class} || "Chemistry::Mol";
    my $mol = $mol_class->new;
    # ... do some stuff with $fh and $mol ...
    return $mol;
}

# override the write_mol method
sub write_mol {
    my ($self, $fh, $mol, %opts) = shift;
    print $fh $mol->name, "\n";
    # ... do some stuff with $fh and $mol ...
}
```

DESCRIPTION

The main use of this module is as a base class for other molecule file I/O modules (for example, Chemistry::File::PDB). Such modules should override and extend the Chemistry::File methods as needed. You only need to care about the methods here if if you are writing a file I/O module or if you want a finer degree of control than what is offered by the simple read and write methods in the Chemistry::Mol class.

From the user's point of view, this module can also be used as shorthand for using several Chemistry::File modules at the same time.

```
use Chemistry::File qw(PDB MDLMol);
```

is exactly equivalent to

```
use Chemistry::File::PDB;
use Chemistry::File::MDLMol;
```

If you use the :auto keyword, Chemistry::File will autodetect and load all the Chemistry::File::* modules installed in your system.

```
use Chemistry::File ':auto';
```

FILE I/O MODEL

Before version 0.30, file I/O modules typically used only parse_string, write_string, parse_file, and write_file, and they were generally used as class methods. A file could contain one or more molecules and only be read or written whole; reading it would return every molecule on the file. This was problematic when dealing with large multi-molecule files (such as SDF files), because all the molecules would have to be loaded into memory at the same time.

While version 0.30 retains backward compatibility with that simple model, it also allows a more flexible interface that allows reading one molecule at a time, skipping molecules, and reading and writing file-level information that is not associated with specific molecules. The following diagram shows the global structure of a file according to the new model:

```
+-----------+
| header    |
+-----------+
| molecule  |
+-----------+
| molecule  |
+-----------+
| ...       |
+-----------+
| footer    |
+-----------+
```

In cases where the header and the footer are empty, the model reduces to the pre-0.30 version. The low-level steps to read a file are the following:

```
$file = Chemistry::File::MyFormat->new(file => 'xyz.mol');
$file->open('<');
$file->read_header;
while (my $mol = $self->read_mol($file->fh, %opts)) {
    # do something with $mol...
}
$self->read_footer;
```

The `read` method does all the above automatically, and it stores all the molecules read in the mols property.

STANDARD OPTIONS

All the methods below include a list of options %opts at the end of the parameter list. Each class implementing this interface may have its own particular options. However, the following options should be recognized by all classes:

mol_class

A class or object with a `new` method that constructs a molecule. This is needed when the user want to specify a molecule subclass different from the default. When this option is not defined, the module may use Chemistry::Mol or whichever class is appropriate for that file format.

format

The name of the file format being used, as registered by Chemistry::Mol->register_format.

fatal

If true, parsing errors should throw an exception; if false, they should just try to recover if possible. True by default.

CLASS METHODS

The class methods in this class (or rather, its derived classes) are usually not called directly. Instead, use Chemistry::Mol->read, write, print, parse, and file. These methods also work if called as instance methods.

$class->parse_string($s, %options)

Parse a string $s and return one or mole molecule objects. This is an abstract method, so it should be provided by all derived classes.

$class->write_string($mol, %options)

Convert a molecule to a string. This is an abstract method, so it should be provided by all derived classes.

$class->parse_file($file, %options)

Reads the file $file and returns one or more molecules. The default method slurps the whole file and then calls parse_string, but derived classes may choose to override it. $file can be a filehandle, a filename, or a scalar reference. See `new` for details.

$class->write_file($mol, $file, %options)

Writes a file $file containing the molecule $mol. The default method calls write_string first and then saves the string to a file, but derived classes may choose to override it. $file can be either a filehandle or a filename.

$class->name_is($fname, %options)

Returns true if a filename is of the format corresponding to the class. It should look at the filename only, because it may be called with non-existent files. It is used to determine with which format to save a file. For example, the Chemistry::File::PDB returns true if the file ends in .pdb.

$class->string_is($s, %options)

Examines the string $s and returns true if it has the format of the class.

$class->file_is($file, %options)

Examines the file $file and returns true if it has the format of the class. The default method slurps the whole file and then calls string_is, but derived classes may choose to override it.

$class->slurp

Reads a file into a scalar. Automatic decompression of gzipped files is supported if the Compress::Zlib module is installed. Files ending in .gz are assumed to be compressed; otherwise it is possible to force decompression by passing the gzip => 1 option (or no decompression with gzip => 0).

$class->new(file => $file, opts => \%opts)

Create a new file object. This method is usually called indirectly via the Chemistry::Mol->file method. $file may be a scalar with a filename, an open filehandle, or a reference to a scalar. If a reference to a scalar is used, the string contained in the scalar is used as an in-memory file.

INSTANCE METHODS

Accessors

Chemistry::File objects are derived from Chemistry::Obj and have the same properties (name, id, and type), as well as the following ones:

file

The "file" as described above under `new`.

fh

The filehandle used for reading and writing molecules. It is opened by `open`.

opts

A hashref containing the options that are passed through to the old-style class methods. They are also passed to the instance method to keep a similar interface, but they could access them via $self->opts anyway.

mode

'>' if the file is open for writing, '<' for reading, and false if not open.

mols

`read` stores all the molecules that were read in this property as an array reference. `write` gets the molecules to write from here.

Abstract methods

These methods should be overridden, because they don't really do much by default.

$file->read_header

Read whatever information is available in the file before the first molecule. Does nothing by default.

$file->read_footer

Read whatever information is available in the file after the last molecule. Does nothing by default.

$self->slurp_mol($fh)

Reads from the input string until the end of the current molecule and returns the "slurped" string. It does not parse the string. It returns undefined if there are no more molecules in the file. This method should be overridden if needed; by default, it slurps until the end of the file.

$self->skip_mol($fh)

Similar to slurp_mol, but it doesn't need to return anything except true or false. It should also be overridden if needed; by default, it just calls slurp_mol.

$file->read_mol($fh, %opts)

Read the next molecule in the input stream. It returns false if there are no more molecules in the file. This method should be overridden by derived classes; otherwise it will call slurp_mol and parse_string (for backwards compatibility; it is recommended to override read_mol directly in new modules).

Note: some old file I/O modules (written before the 0.30 interface) may return more than one molecule anyway, so it is recommended to call read_mol in list context to be safe:

```
($mol) = $file->read_mol($fh, %opts);
```

$file->write_footer

Write whatever information is needed after the last molecule. Does nothing by default.

$self->write_mol($fh, $mol, %opts)

Write one molecule to $fh. By default and for backward compatibility, it just calls `write_string` and prints its return value to $self->fh. New classes should override it.

Other methods

$self->open($mode)

Opens the file (held in $self->file) for reading by default, or for writing if $mode eq '>'. This method sets $self->fh transparently regardless of whether $self->file is a filename (compressed or not), a scalar reference, or a filehandle.

$self->close

Close the file. For regular files this just closes the filehandle, but for gzipped files it does some additional postprocessing. This method is called automatically on object destruction, so it is not mandatory to call it explicitly.

$file->read

Read the whole file. This calls open, read_header, read_mol until there are no more molecules left, read_footer, and close. Returns a list of molecules if called in list context, or the first molecule in scalar context.

$self->write

Write all the molecules in $self->mols. It just calls open, write_header, write_mol (per each molecule), write_footer, and close.

CAVEATS

The :auto feature may not be entirely portable, but it is known to work under Unix and Windows (either Cygwin or ActiveState).

VERSION

0.37

SEE ALSO

Chemistry::Mol
The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman-Brohman <itub@cpan.org>

COPYRIGHT

17 Chemistry::File::Dumper

Read and write molecules via Data::Dumper

SYNOPSIS

```
use Chemistry::File::Dumper;

my $mol = Chemistry::Mol->read("mol.pl");
print $mol->print(format => dumper);
$mol->write("mol.pl", format => "dumper");
```

DESCRIPTION

This module hooks the Data::Dumper Perl core module to the Chemistry::File API, allowing you to dump and undump Chemistry::Mol objects easily. This module automatically registers the "dumper" format with Chemistry::Mol.

For purposes of automatic file type guessing, this module assumes that dumped files end in `.pl`.

This module is useful mainly for debugging purposes, as it dumps *all* the information available in an object, in a reproducible way (so you can use it to compare molecule objects). However, it wouldn't be a good idea to use it to read untrusted files, because they may contain arbitrary Perl code.

OPTIONS

The following options can be used when writing a molecule either as a file or as a string.

dumper_indent

Value to give to Data::Dumper::Indent. Default is 1.

dumper_purity

Value to give to Data::Dumper::Purity. Default is 1.

There are no special options for reading.

VERSION

0.37

SEE ALSO

Chemistry::Mol, *Chemistry::File*, *Data::Dumper*

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

18 Chemistry::File::Formula

Molecular formula reader/formatter

SYNOPSIS

```
use Chemistry::File::Formula;

my $mol = Chemistry::Mol->parse("H2O");
print $mol->print(format => formula);
print $mol->formula;    # this is a shorthand for the above
print $mol->print(format => formula,
    formula_format => "%s%d{<sub>%d</sub>});
```

DESCRIPTION

This module converts a molecule object to a string with the formula and back. It registers the 'formula' format with Chemistry::Mol. Besides its obvious use, it is included in the Chemistry::Mol distribution because it is a very simple example of a Chemistry::File derived I/O module.

Writing formulas

The format can be specified as a printf-like string with the following control sequences, which are specified with the formula_format parameter to $mol->print or $mol->write.

%s symbol

%D number of atoms

%d number of atoms, included only when it is greater than one

%d{substr} substr is only included when number of atoms is greater than one

%j{substr} substr is inserted between the formatted string for each element. (The 'j' stands for 'joiner'.) The format should have only one joiner, but its location in the format string doesn't matter.

%% a percent sign

If no format is specified, the default is "%s%d". Some examples follow. Let's assume that the formula is C2H6O, as it would be formatted by default.

%s%D

Like the default, but include explicit indices for all atoms. The formula would be formatted as "C2H6O1"

%s%d{_{%d}}

HTML format. The output would be "C₂H₆O".

%D %s%j{, }

Use a comma followed by a space as a joiner. The output would be "2 C, 6 H, 1 O".

Symbol Sort Order The elements in the formula are sorted by default in the "Hill order", which means that:

1) if the formula contains carbon, C goes first, followed by H, and the rest of the symbols in alphabetical order. For example, "CH2BrF".

2) if there is no carbon, all the symbols (including H) are listed alphabetically. For example, "BrH".

It is possible to supply a custom sorting subroutine with the 'formula_sort' option. It expects a subroutine reference that takes a hash reference describing the formula (similar to what is returned by parse_formula, discussed below), and that returns a list of symbols in the desired order.

For example, this will sort the symbols in reverse asciibetical order:

```
my $formula = $mol->print(
    format          => 'formula',
    formula_sort    => sub {
        my $formula_hash = shift;
        return reverse sort keys %$formula_hash;
    }
);
```

Parsing Formulas

Formulas can also be parsed back into Chemistry::Mol objects. The formula may have parentheses and square or triangular brackets, and it may have the following abbreviations:

```
Me => '(CH3)',
Et => '(CH3CH2)',
Bu => '(C4H9)',
Bn => '(C6H5CH2)',
Cp => '(C5H5)',
Ph => '(C6H5)',
Bz => '(C6H5CO)',
```

The formula may also be preceded by a number, which multiplies the whole formula. Some examples of valid formulas:

```
Formula              Equivalent to
-------------------------------------------------------------
CH3(CH2)3CH3         C5H12
C6H3Me3              C9H12
2Cu[NH3]4(NO3)2      Cu2H24N12O12
```

```
2C(C[C<C>5]4)3       C152
2C(C(C(C)5)4)3       C152
C 1 0 H 2 2          C10H22 (whitespace is completely ignored)
```

When a formula is parsed, a molecule object is created which consists of the set of the atoms in the formula (no bonds or coordinates, of course). The atoms are created in alphabetical order, so the molecule object for C2H5Br would have the atoms in the following sequence: Br, C, C, H, H, H, H, H.

If you don't want to create a molecule object, but would rather have a simple hash with the number of atoms for each element, use the `parse_formula` method:

```
my %formula = Chemistry::File::Formula->parse_formula("C2H6O");
use Data::Dumper;
print Dumper \%formula;
```

which prints something like

```
$VAR1 = {
          'H' => 6,
          'O' => 1,
          'C' => 2
        };
```

The `parse_formula` method is called internally by the `parse_string` method.

Non-integer numbers in formulas The `parse_formula` method can also accept formulas that contain floating-point numbers, such as H1.5N0.5. The numbers must be positive, and numbers smaller than one should include a leading zero (e.g., 0.9, not .9).

When formulas with non-integer numbers of atoms are turned into molecule objects as described in the previous section, the number of atoms is always **rounded up**. For example, H1.5N0.5 will produce a molecule object with two hydrogen atoms and one nitrogen atom.

There is currently no way of *producing* formulas with non-integer numbers; perhaps a future version will include an "occupancy" property for atoms that will result in non-integer formulas.

VERSION

0.37

SEE ALSO

Chemistry::Mol, *Chemistry::File*

For discussion about Hill order, just search the web for `formula "hill order"`. The original reference is *J. Am. Chem. Soc.* **1900**, *22*, 478-494. http://dx.doi.org/10.1021/ja02046a005.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>.
Formula parsing code contributed by Brent Gregersen.
Patch for non-integer formulas by Daniel Scott.

COPYRIGHT

19 Chemistry::File::FormulaPattern

Wrapper Chemistry::File class for Formula patterns

SYNOPSIS

```
use Chemistry::File::FormulaPattern;

# somehow get a bunch of molecules...
use Chemistry::File::SDF;
my @mols = Chemistry::Mol->read("file.sdf");

# we want molecules with six carbons and 8 or more hydrogens
my $patt = Chemistry::Pattern->new("C6H8-", format => "formula_pattern");

for my $mol (@mols) {
    if ($patt->match($mol)) {
        print $mol->name, " has a nice formula!\n";
    }
}

# a concise way of selecting molecules with grep
my @matches = grep { $patt->match($mol) } @mols;
```

DESCRIPTION

This is a wrapper class for reading Formula Patterns using the standard Chemistry::Mol I/O interface. This allows Formula patterns to be used interchangeably with other pattern languages such as SMARTS in the context of programs such as *mok*. All of the real work is done by *Chemistry::FormulaPattern*.

This module register the 'formula_pattern' format with *Chemistry::Mol*.

VERSION

0.10

SEE ALSO

Chemistry::FormulaPattern, *Chemistry::Pattern*, *Chemistry::File*, *Chemistry::Mol*, *mok*.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

20 Chemistry::File::MDLMol

MDL molfile reader/writer

SYNOPSIS

```
use Chemistry::File::MDLMol;

# read a molecule
my $mol = Chemistry::Mol->read('myfile.mol');

# write a molecule
$mol->write("myfile.mol");

# use a molecule as a query for substructure matching
use Chemistry::Pattern;
use Chemistry::Ring;
Chemistry::Ring::aromatize_mol($mol);

my $patt = Chemistry::Pattern->read('query.mol');
if ($patt->match($mol)) {
    print "it matches!\n";
}
```

DESCRIPTION

MDL Molfile (V2000) reader/writer.

This module automatically registers the 'mdl' format with Chemistry::Mol.

The first three lines of the molfile are stored as $mol->name, $mol->attr("mdlmol/line2"), and $mol->attr("mdlmol/comment").

This version only reads and writes some of the information available in a molfile: it reads coordinats, atom and bond types, charges, radicals, and atom lists. It does not read other things such as stereochemistry, 3d properties, isotopes, etc.

This module is part of the PerlMol project, http://www.perlmol.org.

Query properties

The MDL molfile format supports query properties such as atom lists, and special bond types such as "single or double", "single or aromatic", "double or aromatic", "ring bond", or "any". These properties are supported by this module in conjunction with *Chemistry::Pattern*. However, support for query properies is currently read-only, and the other properties listed in the specification are not supported yet.

So that atom and bond objects can use these special query options, the conditions are represented as Perl subroutines. The generated code can be read from the 'mdlmol/test_sub' attribute:

```
$atom->attr('mdlmol/test_sub');
$bond->attr('mdlmol/test_sub');
```

This may be useful for debugging, such as when an atom doesn't seem to match as expected.

Aromatic Queries

To be able to search for aromatic substructures are represented by Kekule structures, molfiles that are read as patterns (with `Chemistry::Pattern`-read) are aromatized automatically by using the *Chemistry::Ring* module. The default bond test from Chemistry::Pattern::Bond is overriden by one that checks the aromaticity in addition to the bond order. The test is,

```
$patt->aromatic ?  $bond->aromatic
    : (!$bond->aromatic && $patt->order == $bond->order);
```

That is, aromatic pattern bonds match aromatic bonds, and aliphatic pattern bonds match aliphatic bonds with the same bond order.

VERSION

0.21

SEE ALSO

Chemistry::Mol

The MDL file format specification. http://www.mdl.com/downloads/public/ctfile/ctfile.pdf or Arthur Dalby et al., J. Chem. Inf. Comput. Sci, 1992, 32, 244-255.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

21 Chemistry::File::MidasPattern

Wrapper Chemistry::File class for Midas patterns

SYNOPSIS

```
use Chemistry::File::MidasPattern;
use Chemistry::File::PDB;

# read a molecule
my $mol = Chemistry::MacroMol->read("test.pdb");

# define a pattern matching carbons alpha and beta
# in all valine residues
my $str  = ':VAL@CA,CB';
my $patt = Chemistry::MidasPattern->parse($str, format => 'midas');
# Chemistry::Mol->parse($str, format => 'midas') also works

# apply the pattern to the molecule
$patt->match($mol);

# extract the results
for my $atom ($patt->atom_map) {
    printf "%s\t%s\n",  $atom->attr("pdb/residue_name"), $atom->name;
}
printf "FOUND %d atoms\n", scalar($patt->atom_map);
```

DESCRIPTION

This is a wrapper class for reading Midas Patterns using the standard Chemistry::Mol I/O interface. This allows Midas patterns to be used interchangeably with other pattern languages such as SMARTS in the context of programs such as *mok*. All of the real work is done by *Chemistry::MidasPattern*.

This module register the 'midas' format with Chemistry::Mol.

VERSION

0.11

SEE ALSO

Chemistry::MidasPattern, *Chemistry::File*, *Chemistry::Mol*, *Chemistry::MacroMol*, *mok*.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert <itub@cpan.org>

COPYRIGHT

22 Chemistry::File::Mopac

MOPAC 6 input file reader/writer

SYNOPSIS

```
use Chemistry::File::Mopac;

# read a MOPAC file
my $mol = Chemistry::Mol->read('file.mop');

# write a MOPAC file using cartesian coordinates
$mol->write('file.mop', coords => 'cartesian');

# now with internal coordinates
$mol->write('file.mop', coords => 'internal');

# rebuild the Z-matrix from scratch while we are at it
$mol->write('file.mop', rebuild => 1);
```

DESCRIPTION

This module reads and writes MOPAC 6 input files. It can handle both internal coordinates and cartesian coordinates. It also extracts molecules from summary files, defined as those files that match /SUMMARY OF/ in the third line. Perhaps a future version will extract additional information such as the energy and dipole from the summary file.

This module registers the `mop` format with Chemistry::Mol. For detection purposes, it assumes that filenames ending in .mop or .zt have the Mopac format, as well as files whose first line matches /am1|pm3|mndo|mdg|pdg/i (this may change in the future).

When the module reads an input file into $mol, it puts the keywords (usually the first line of the file) in $mol->attr("mopac/keywords"), the comments (usually everything else on the first three lines) in $mol->attr("mopac/comments") and $mol->name, and the internal coordinates for each atom in $atom->internal_coords.

When writing, the kind of coordinates used depend on the `coords` option, as shown in the SYNOPSIS. Internal coordinates are used by default. If the molecule has no internal coordinates defined or the rebuild option is set, the build_zmat function from Chemistry::InternalCoords::Builder is used to renumber the atoms and build the Z-matrix from scratch.

TO DO

When writing a Mopac file, this version marks all coordinates as variable (for the purpose of geometry optimization by Mopac). A future version should have more flexibility.

VERSION

0.15

SEE ALSO

Chemistry::Mol, *Chemistry::File*, *Chemistry::InternalCoords*, *Chemistry::InternalCoords::Builder*, http://www.perlmol.org/.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

23 Chemistry::File::PDB

Protein Data Bank file format reader/writer

SYNOPSIS

```
use Chemistry::File::PDB;

# read a PDB file
my $macro_mol = Chemistry::MacroMol->read("myfile.pdb");

# write a PDB file
$macro_mol->write("out.pdb");

# read all models in a multi-model file
my @mols = Chemistry::MacroMol->read("models.pdb");

# read one model at a time
my $file = Chemistry::MacroMol->file("models.pdb");
$file->open;
while (my $mol = $file->read_mol($file->fh)) {
    # do something with $mol
}
```

DESCRIPTION

This module reads and writes PDB files. The PDB file format is commonly used to describe proteins, particularly those stored in the Protein Data Bank (http://www.rcsb.org/pdb/). The current version of this module only reads the following record types, ignoring everything else:

```
ATOM
HETATM
ENDMDL
END
```

This module automatically registers the 'pdb' format with Chemistry::Mol, so that PDB files may be identified and read by Chemistry::Mol->read(). For autodetection purpuses, it assumes that files ending in .pdb or having a line matching /^(ATOM|HETATM)/ are PDB files.

The PDB reader and writer is designed for dealing with Chemistry::MacroMol objects, but it can also create and use Chemistry::Mol objects by throwing some information away.

Properties

When reading and writing files, this module stores or gets some of the information in the following places:

$domain->type

The residue type, such as "ARG".

$domain->name

The type and sequence number, such as "ARG114".

$domain->attr("pdb/sequence_number")

The residue sequence number as given in the PDB file.

$domain->attr("pdb/chain_id")

The chain to which this residue belongs (one character).

$domain->attr("pdb/insertion_code")

The residue insertion code (see the PDB specification for details).

$atom->name

The PDB atom name, such as "CA".

$atom->attr("pdb/residue_name")

The name of the residue, as discussed above.

$atom->attr("pdb/serial_number")

The serial number for the atom, as given in the PDB file.

If some of this information is not available when writing a PDB file, this module tries to make it up (by counting the atoms or residues, for example). The default residue name for writing is UNK (unknown). Atom names are just the atomic symbols.

Multi-model files

If a PDB file has multiple models (separated by END or ENDMDL records), each call to read_mol will return one model.

Output features

On writing Chemistry::Mol objects, which don't have macromolecule information and usually don't have atom names, the atom names are made up by concatenating the atomic symbol with a unique ID (up to 1296 atoms are possible). The ID can be disabled by setting the option 'noid':

```
$mol->write("out.pdb", noid => 1);
```

The molecule's name is written as a HEADER record; A REMARK record is added listing the version of Chemistry::File::PDB that was used.

VERSION

0.22

SEE ALSO

Chemistry::MacroMol, *Chemistry::Mol*, *Chemistry::File*, http://www.perlmol.org/.

The PDB format description at http://www.rcsb.org/pdb/docs/format/pdbguide2.2/guide2.2_frame.html

There is another PDB reader in Perl, as part of the BioPerl project: *Bio::Structure::IO::pdb*.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

24 Chemistry::File::QChemOut

Q-Chem ouput molecule format reader

SYNOPSIS

```
use Chemistry::File::QChemOut;

# read an QChemOut file
my $mol = Chemistry::Mol->read("myfile.out", format => 'qchemout');

# read all the intermediate structures (e.g., optimization steps)
my $mol = Chemistry::Mol->read("myfile.out",
    format => 'chemout', all => 1);
```

DESCRIPTION

This module reads Q-Chem output files. It automatically registers the 'qchemout' format with Chemistry::Mol, so that Q-Chem outuput files may be identified and read using Chemistry::Mol->read().

The current version of this reader simply extracts the cartesian coordinates and symbols from the Q-Chem outuput file. All other information is ignored.

INPUT OPTIONS

all

If true, read all the intermediate structures, as in a structure optimization. This causes $mol->read to return an array instead of a single molecule. Default: false.

VERSION

0.10

SEE ALSO

Chemistry::Mol, http://www.perlmol.org/.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

25 Chemistry::File::SDF

MDL Structure Data File reader/writer

SYNOPSIS

```
use Chemistry::File::SDF;

# Simple interface (all at once)
# read all the molecules in the file
my @mols = Chemistry::Mol->read('myfile.sdf');

# assuming that the file includes a <PKA> data item...
print $mols[0]->attr("sdf/data")->{PKA};

# write a bunch of molecules to an SDF file
Chemistry::Mol->write('myfile.sdf', mols => \@mols);

# or write just one molecule
$mol->write('myfile.sdf');

# Low level interface (one at a time)
# create reader
my $reader = Chemistry::Mol->file('myfile.sdf');
$reader->open('<');
while (my $mol = $reader->read_mol($reader->fh)) {
    # do something with $mol
}
```

DESCRIPTION

MDL SDF (V2000) reader.

This module automatically registers the 'sdf' format with Chemistry::Mol.

The parser returns a list of Chemistry::Mol objects. SDF data can be accessed by the $mol->attr method. Attribute names are stored as a hash ref at the "sdf/data" attribute, as shown in the synopsis. When a data item has a single line in the SDF file, the attribute is stored as a string; when there's more than one line, they are stored as an array reference. The rest of the information on the line that holds the field name is ignored.

This module is part of the PerlMol project, http://www.perlmol.org.

CAVEATS

Note that by storing the SDF data as a hash, there can be only one field with a given name. The SDF format description is not entirely clear in this regard. Also note that SDF data field names are considered to be case-sensitive.

VERSION

0.21

SEE ALSO

Chemistry::Mol

The MDL file format specification. http://www.mdl.com/downloads/public/ctfile/ctfile.pdf or Arthur Dalby et al., J. Chem. Inf. Comput. Sci, 1992, 32, 244-255.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

26 Chemistry::File::SLN

SLN linear notation parser/writer

SYNOPSYS

```
#!/usr/bin/perl
use Chemistry::File::SLN;

# parse a SLN string for benzene
my $s = 'C[1]H:CH:CH:CH:CH:CH@1';
my $mol = Chemistry::Mol->parse($s, format => 'sln');

# print a SLN string
print $mol->print(format => 'sln');

# print a unique (canonical) SLN string
print $mol->print(format => 'sln', unique => 1);

# parse a multiline SLN file
my @mols = Chemistry::Mol->read("file.sln", format => 'sln');

# write a multiline SLN file
Chemistry::Mol->write("file.sln", mols => [@mols]);
```

DESCRIPTION

This module parses a SLN (Sybyl Line Notation) string. This is a File I/O driver for the PerlMol project. http://www.perlmol.org/. It registers the 'sln' format with Chemistry::Mol, and recognizes filenames ending in '.sln'.

Optional attributes for atoms, bonds, and molecules are stored as $atom->attr("sln/attr"), $bond->attr("sln/attr"), and $mol->attr("sln/attr"), respectively. Boolean attributes are stored with a value of 'TRUE'. That's the way boolean attributes are recognized when writing, so that they can be written in the shortened form.

```
$sln_attr->{backbone} = 1;
# would be ouput as "C[backbone=1]"

$sln_attr->{backbone} = 'TRUE';
# would be ouput as "C[backbone]"
```

Also note that attribute names are normalized to lowercase on reading.

OPTIONS

The following options are available when reading:

kekulize

Assign bond orders for unsatisfied valences or for aromatic bonds. For example, benzene read as C[1]H:CH:CH:CH:CH:CH@1 will be converted internally to something like C[1]H=CHCH=CHCH=CH@1. This is needed if another format or module expects a Kekule representation without an aromatic bond type.

The following options are available when writing:

mols

If this option points to an array of molecules, these molecules will be written, one per line, as in the example in the SYNOPSYS.

aromatic

Detect aromaticity before writing. This will ensure that aromatic bond types are used instead of alternate single and double bonds.

unique

Canonicalize before writing, and produce a unique strucure. NOTE: this option does not guarantee a unique representation for molecules with bracketed attributes.

name

Include the name of the molecule ($mol->name) in the output string.

coord3d, coords

Include the 3D coordinates of every atom in the molecule in the output string. `coord3d` and `coords` may be used interchangeably.

attr

Output the atom, bond, and molecule attributes found in $mol->attr("sln/attr"), etc.

CAVEATS

This version does not implement the full SLN specification. It supports simple structures and some attributes, but it does not support any of the following:

Macro atoms

Pattern matching options

Markush structures

2D Coordinates

The SLN specification is vague on several points, and I don't have a reference implementation available, so I had to make several arbitrary decisions. Also, this version of this module has not been tested exhaustively, so please report any bugs that you find.

If the parser doesn't understand a string, it only says "syntax error", which may not be very helpful.

VERSION

0.11

SEE ALSO

Chemistry::Mol, *Chemistry::File*, *Chemistry::File::SMILES*

The PerlMol website http://www.perlmol.org/

Ash, S.; Cline, M. A.; Homer, R. W.; Hurst, T.; Smith, G. B., SYBYL Line Notation (SLN): A Versatile Language for Chemical Structure Representation. J. Chem. Inf. Comput. Sci; 1997; 37(1); 71-79. DOI: 10.1021/ci960109j (http://dx.doi.org/10.1021/ci960109j)

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

27 Chemistry::File::SMARTS

SMARTS chemical substructure pattern linear notation parser

SYNOPSYS

```
#!/usr/bin/perl
use Chemistry::File::SMARTS;

# this string matches an oxygen next to an atom with three
# neighbors, one of which is a hydrogen, and a positive charge
my $smarts = 'O[D3H+]';

# parse a SMARTS string and compile it into a
# Chemistry::Pattern object
my $patt = Chemistry::Pattern->parse($smarts, format => 'smarts');

# find matches of the pattern in a Chemistry::Mol object $mol
my $mol = Chemistry::Mol->read("myfile.mol");
while ($patt->match($mol)) {
    print "pattern matches atoms: ", $patt->atom_map, "\n"
}

# NOTE: if the SMARTS pattern relies on aromaticity or ring
# properties, you have to make sure that the target
# molecule is "aromatized" first:
my $smarts = 'c:a';
my $patt = Chemistry::Pattern->parse($smarts, format => 'smarts');
use Chemistry::Ring 'aromatize_mol';
aromatize_mol($mol);  # <--- AROMATIZE!!!
while ($patt->match($mol)) {
    print "pattern matches atoms: ", $patt->atom_map, "\n"
}

# Note that "atom mapping numbers" end up as $atom->name
my $patt = Chemistry::Pattern->parse("[C:7][C:8]", format => 'smarts');
print $patt->atoms(1)->name;    # prints 7
```

DESCRIPTION

This module parse a SMARTS (SMiles ARbitrary Target Specification) string, generating a *Chemistry::Pattern* object. It is a file I/O driver for the PerlMol toolkit; it's not called directly but by means of the Chemistry::Pattern->parse class method.

For a detailed description of the SMARTS language, see http://www.daylight.com/dayhtml/doc/theory/theory.smarts.htm Note that this module doesn't implement the full language, as detailed under CAVEATS.

This module is part of the PerlMol project, http://www.perlmol.org/.

CAVEATS

The following features are not implemented yet:

chirality: @, @@

component-level gruouping

That is, the difference between these three cases:

```
(SMARTS)
(SMARTS).(SMARTS)
(SMARTS).SMARTS
```

The so-called parser is very lenient, so if you give it something that's not quite reasonable it will ignore it or interpret it in a strange way without warning.

As shown in the synopsis, you have to make sure that the molecule is "aromatized" if you want to apply to it a pattern that relies on aromaticity or ring properties.

VERSION

0.22

SEE ALSO

Chemistry::Pattern, *Chemistry::Mol*, *Chemistry::File*, *Chemistry::File::SMILES*.

For more information about SMARTS, see the SMARTS Theory Manual at http://www.daylight.com/dayhtml/doc/theor

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

28 Chemistry::File::SMILES

SMILES linear notation parser/writer

SYNOPSYS

```
#!/usr/bin/perl
use Chemistry::File::SMILES;

# parse a SMILES string
my $s = 'C1CC1(=O)[O-]';
my $mol = Chemistry::Mol->parse($s, format => 'smiles');

# print a SMILES string
print $mol->print(format => 'smiles');

# print a unique (canonical) SMILES string
print $mol->print(format => 'smiles', unique => 1);

# parse a SMILES file
my @mols = Chemistry::Mol->read("file.smi", format => 'smiles');

# write a multiline SMILES file
Chemistry::Mol->write("file.smi", mols => \@mols);
```

DESCRIPTION

This module parses a SMILES (Simplified Molecular Input Line Entry Specification) string. This is a File I/O driver for the PerlMol project. http://www.perlmol.org/. It registers the 'smiles' format with Chemistry::Mol.

This parser interprets anything after whitespace as the molecule's name; for example, when the following SMILES string is parsed, $mol->name will be set to "Methyl chloride":

```
CCl  Methyl chloride
```

The name is not included by default on output. However, if the `name` option is defined, the name will be included after the SMILES string, separated by a tab.

```
print $mol->print(format => 'smiles', name => 1);
```

Multiline SMILES and SMILES files

A file or string can contain multiple molecules, one per line.

```
CCl  Methyl chloride
CO   Methanol
```

Files with the extension '.smi' are assumed to have this format.

Atom Mapping Numbers

As an extension for reaction processing, SMILES strings may have atom mapping numbers, which are introduced after a colon in a bracketed atom. For example, [C:1]. The mapping number need not be unique. This module reads the mapping numbers and stores them as the name of the atom ($atom->name).

On output, atom names are not included by default. See the `number` and `auto_number` options below for ways of including them.

head1 OPTIONS

The following options are supported in addition to the options mentioned for *Chemistry::File*, such as `mol_class`, `format`, and `fatal`.

aromatic

On output, detect aromatic atoms and bonds by means of the Chemistry::Ring module, and represent the organic aromatic atoms with lowercase symbols.

unique

When used on output, canonicalize the structure if it hasn't been canonicalized already and generate a unique SMILES string. This option implies "aromatic".

number

For atoms that have a defined name, print the name as the "atom number". For example, if an ethanol molecule has the name "42" for the oxygen atom and the other atoms have undefined names, the output would be:

```
CC[OH:42]
```

auto_number

When used on output, number all the atoms explicitly and sequentially. The output for ethanol would look something like this:

```
[CH3:1][CH2:2][OH:3]
```

name

Include the molecule name on output, as described in the previous section.

kekulize

When used on input, assign single or double bond orders to "aromatic" or otherwise unspecified bonds (i.e., generate the Kekule structure). If false, the bond orders will remain single. This option is true by default. This uses `assign_bond_orders` from the *Chemistry::Bond::Find* module.

CAVEATS

Stereochemistry is not supported! Stereochemical descriptors such as @, @@, /, and \ will be silently ignored on input, and will certainly not be produced on output.

Reading branches that start before an atom, such as (OC)C, which should be equivalent to C(OC) and COC, according to some variants of the SMILES specification. Many other tools don't implement this rule either.

The kekulize option works by increasing the bond orders of atoms that don't have their usual valences satisfied. This may cause problems if you have atoms with explicitly low hydrogen counts.

VERSION

0.46

SEE ALSO

Chemistry::Mol, *Chemistry::File*

The SMILES Home Page at http://www.daylight.com/dayhtml/smiles/

The Daylight Theory Manual at http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

29 Chemistry::File::VRML

Generate VRML models for molecules

SYNOPSIS

```
use Chemistry::File::PDB;
use Chemistry::File::VRML;
use Chemistry::Bond::Find 'find_bonds';

my $mol = Chemistry::Mol->read('test.pdb');
find_bonds($mol, orders => 1);
$mol->write('test.wrl', format => 'vrml',
    center => 1,
    style  => 'ballAndWire',
    color  => 'byAtom',
);
```

DESCRIPTION

This module generates a VRML (Virtual Reality Modeling Language) representation of a molecule, which can then be visualized with any VRML viewer. This is a PerlMol file I/O plugin, and registers the 'vrml' format with *Chemistry::Mol*. Note however that this file plugin is write-only; there's no way of reading a VRML file back into a molecule.

This module is a modification of PDB2VRML by Horst Vollhardt, adapted to the *Chemistry::File* interface.

OPTIONS

The following options may be passed to $mol->write.

center

If true, shift the molecules center of geometry into the origin of the coordinate system. Note: this only affects the output; it does not affect the coordinates of the atoms in the original Chemistry::Mol object.

style

Sets the style for the VRML representation of the molecular structure. Default is 'Wireframe'. Currently supported styles are:

```
Wireframe, BallAndWire,
Stick, BallAndStick,
CPK
```

color

Set the overall color of the molecular structure. If the color is set to 'byAtom', the color the for atoms and bonds is defined by the atom type. Default is 'byAtom'. Currently supported colors are:

```
byAtom,
yellow, blue, red,
green, white, brown,
grey, purple
```

stick_radius

Defines the radius in Angstrom for the cylinders in the 'Stick' and 'BallAndStick' style. Default is 0.15 .

ball_radius

Defines the factor which is multiplied with the VDW radius for the spheres in the 'BallAndWire' and 'BallAndStick' style. Default is 0.2 .

compression

Turns on/off compression of the output. If turned on, all leading whitespaces are removed. This produces a less readable but approx. 20% smaller output, the speed is increased by 10% as well.

AUTHOR

PDB2VRML originally by Horst Vollhardt, horstv@yahoo.com, 1998. Modified and adapted as Chemistry::File::VRML by Ivan Tubert-Brohman, itub@cpan.org, 2005.

COPYRIGHT

SEE ALSO

PDB2VRML found at http://www.realitydiluted.com/mirrors/reality.sgi.com/horstv_basel/pdb2vrml/
PerlMol project at http://www.perlmol.org/

30 Chemistry::File::XYZ

XYZ molecule format reader/writer

SYNOPSIS

```
use Chemistry::File::XYZ;

# read an XYZ file
my $mol = Chemistry::Mol->read("myfile.xyz");

# write an XYZ file
$mol->write("out.xyz");
```

DESCRIPTION

This module reads XYZ files. It automatically registers the 'xyz' format with Chemistry::Mol, so that XYZ files may be identified and read by Chemistry::Mol->read().

The XYZ format is not strictly defined and there are various versions floating around; this module accepts the following:

First line: atom count (optional)

Second line: molecule name or comment (optional)

All other lines: (symbol or atomic number), x, y, and z coordinates separated by spaces, tabs, or commas.

If the first line doesn't look like a number, the atom count is deduced from the number of lines in the file. If the second line looks like it defines an atom, it is assumed that there was no name or comment.

OUTPUT OPTIONS

On writing, the default format is the following, giving H2 as an example.

```
2
Hydrogen molecule
H    0.0000   0.0000   0.0000
H    0.0000   0.7000   0.0000
```

That is: count line, name line, and atom lines (symbol, x, y, z). These format can be modified by means of certain options:

name

Control whether or not to include the name.

count

Control whether or not to include the count line.

symbol

If false, use the atomic numbers instead of the atomic symbols.

For example,

```
$mol->write("out.xyz", count => 0, name => 0, symbol => 0);
```

gives the following output:

```
1    0.0000   0.0000   0.0000
1    0.0000   0.7000   0.0000
```

VERSION

0.12

SEE ALSO

Chemistry::Mol, http://www.perlmol.org/.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

31 Chemistry::InternalCoords

Represent the position of an atom using internal coordinates and convert it to Cartesian coordinates.

SYNOPSIS

```
use Chemistry::InternalCoords;

# ... have a molecule in $mol
my $atom = $mol->new_atom;

# create an internal coordinate object for $atom
# with respect to atoms with indices 4, 3, and 2.
my $ic = Chemistry::InternalCoords->new(
    $atom, 4, 1.1, 3, 109.5, 2, 180.0
);

# can also use atom object references instead of indices
($atom4, $atom3, $atom2) = $mol->atoms(4,3,2);
my $ic = Chemistry::InternalCoords->new(
    $atom, $atom4, 1.1, $atom3, 109.5, $atom2, 180.0
);

# calculate the Cartesian coordinates for
# the atom from the internal coordinates
my $vector = $ic->cartesians;

# calculate and set permanently the Cartesian coordinates
# for the atom from the internal coordinates
my $vector = $ic->add_cartesians;
# same as $atom->coords($ic->cartesians);

# dump as string
print $ic;
# same as print $ic->stringify;
```

DESCRIPTION

This module implements an object class for representing internal coordinates and provides methods for converting them to Cartesian coordinates.

For generating an internal coordinate representation (aka a Z-matrix) of a molecule from its Cartesian coordinates, see the *Chemistry::InternalCoords::Builder* module.

This module is part of the PerlMol project, http://www.perlmol.org/.

METHODS

my $ic = Chemistry::InternalCoords->new($atom, $len_ref, $len_val, $ang_ref, $ang_val, $dih_ref, $dih_val)

Create a new internal coordinate object. $atom is the atom to which the coordinates apply. $len_ref, $ang_ref, and $dih_ref are either atom references or atom indices and are used to specify the distance, angle, and dihedral that are used to define the current position. $len_val, $ang_val, and $dih_val are the values of the distance, angle, and dihedral. The angle and the dihedral are expected to be in degrees.

For example,

```
my $ic = Chemistry::InternalCoords->new(
    $atom, 4, 1.1, 3, 109.5, 2, 180.0
);
```

means that $atom is 1.1 distance units from atom 4, the angle $atom-4-3 is 109.5 degrees, and the dihedral $atom-4-3-2 is 180.0 degrees.

The first three atoms in the molecule don't need all the internal coordinates: the first atom doesn't need anything (except for the atom reference $atom) because it will always be placed at the origin; the second atom only needs a distance, and it will be placed on the X axis; the third atom needs a distance and an angle, and it will be placed on the XY plane.

my ($atom, $distance) = $ic->distance

Return the atom reference and distance value contained in the Chemistry::InternalCoords object.

my ($atom, $angle) = $ic->angle

Return the atom reference and angle value contained in the Chemistry::InternalCoords object.

my ($atom, $dihedral) = $ic->dihedral

Return the atom reference and dihedral value contained in the Chemistry::InternalCoords object.

my $vector = $ic->cartesians

Calculate the Cartesian coordinates from an internal coordinate object. Returns a Math::VectorReal object. Note that the Cartesian coordinates of the atoms referenced by the $ic object should already be calculated.

my $vector = $ic->add_cartesians

Same as $ic->cartesians, but also adds the newly calculated Cartesian coordinates to the atom. It is just shorthand for the following:

```
$atom->coords($ic->cartesians);
```

The best way of calculating the Cartesian coordinates for an entire molecule, assuming that every atom is defined only in terms of previous atoms (as it should be), is the following:

```
# we have all the internal coords in @ics
for my $ic (@ics) {
    $ic->add_cartesians;
}
```

$ic->update

Update the values of the internal coordinates from the cartesian coordinates for the atom.

my $string = $ic->stringify

Dump the object as a string representation. May be useful for debugging. This method overloads the "" operator.

VERSION

0.20

SEE ALSO

Chemistry::InternalCoords::Builder, *Chemistry::Mol*, *Chemistry::Atom*, *Math::VectorReal*, http://www.perlmol.org/.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

32 Chemistry::Mok

Molecular awk interpreter

SYNOPSIS

```
use Chemistry::Mok;
$code = '/CS/g{ $n++; $l += $match->bond_map(0)->length }
    END { printf "Average C-S bond length: %.3f\n", $l/$n; }';

my $mok = Chemistry::Mok->new($code);
$mok->run({ format => mdlmol }, glob("*.mol"));
```

DESCRIPTION

This module is the engine behind the mok program. See mok(1) for a detailed description of the language. Mok is part of the PerlMol project, http://www.perlmol.org.

METHODS

Chemistry::Mok->new($code, %options)

Compile the code and return a Chemistry::Mok object. Available options:

`package`

If the `package` option is given, the code runs in the Chemistry::Mok::UserCode::$options{package} package instead of the Chemistry::Mok::UserCode::Default package. Specifying a package name is recommended if you have more than one mok object and you are using global varaibles, in order to avoid namespace clashes.

`pattern_format`

The name of the format which will be used for parsing slash-delimited patterns that don't define an explicit format. Mok versions until 0.16 only used the 'smiles' format, but newer versions can use other formats such as 'smarts', 'midas', 'formula_pattern', and 'sln', if available. The default is 'smarts'.

$mok->run($options, @args)

Run the code on the filenames contained in @args. $options is a hash reference with runtime options. Available options:

build_3d

Generate 3D coordinates using Chemistry::3DBuilder.

aromatize

"Aromatize" each molecule as it is read. This is needed for example for matching SMARTS patterns that use aromaticity or ring primitives.

delete_dummies

Delete dummy atoms after reading each molecule. A dummy atom is defined as an atom with an unknown symbol (i.e., it doesn't appear on the periodic table), or an atomic number of zero.

find_bonds

If set to a true value, find bonds. Use it when reading files with no bond information but 3D coordinates to detect the bonds if needed (for example, if you want to do match a pattern that includes bonds). If the file has explicit bonds, mok will not try to find the bonds, but it will reassign the bond orders from scratch.

format

The format used when calling $mol_class->read. If not given, $mol_class->read tries to identify the format automatically.

mol_class

The molecule class used for reading the files. Defaults to Chemistry::Mol.

VERSION

0.26

SEE ALSO

mok, http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

33 Chemistry::Bond::Find

Detect bonds in a molecule from atomic 3D coordinates and assign formal bond orders

SYNOPSIS

```
use Chemistry::Bond::Find ':all'; # export all available functions

# $mol is a Chemistry::Mol object
find_bonds($mol);
assign_bond_orders($mol);
```

DESCRIPTION

This module provides functions for detecting the bonds in a molecule from its 3D coordinates by using simple cutoffs, and for guessing the formal bond orders.

This module is part of the PerlMol project, http://www.perlmol.org/.

FUNCTIONS

These functions may be exported, although nothing is exported by default.

`find_bonds($mol, %options)`

Finds and adds the bonds in a molecule. Only use it in molecules that have no explicit bonds; for example, after reading a file with 3D coordinates but no bond orders.

Available options:

tolerance

Defaults to 1.1. Two atoms are considered to be bound if the distance between them is less than the sum of their covalent radii multiplied by the tolerance.

margin

NOTE: in general setting this option is not recommended, unless you know what you are doing. It is used by the space partitioning algorithm to determine the "bucket size". It defaults to 2 * Rmax * tolerance, where Rmax is the largest covalent radius among the elements found in the molecule. For example, if a molecule has C, H, N, O, and I, Rmax = R(I) = 1.33, so the margin defaults to 2 * 1.33 * 1.1 = 2.926. This margin ensures that no bonds are missed by the partitioning algorithm.

Using a smaller value gives faster results, but at the risk of missing some bonds. In this example, if you are certain that your molecule doesn't contain I-I bonds (but it has C-I bonds), you can set margin to (0.77 + 1.33) * 1.1 = 2.31 and you still won't miss any bonds (0.77 is the radius of carbon). This only has a significant impact for molecules with a thousand atoms or more, but it can reduce the execution time by 50% in some cases.

orders

If true, assign the bond orders after finding them, by calling `assign_bond_orders($mol, %opts)`.

bond_class

The class that will be used for creating the new bonds. The default is the bond class returned by `$mol->bond_class`.

`assign_bond_orders($mol, %opts)`

Assign the formal bond orders in a molecule. The bonds must already be defined, either by `find_bonds` or because the molecule was read from a file that included bonds but no bond orders. If the bond orders were already defined (maybe the molecule came from a file that did include bond orders after all), the original bond orders are erased and the process begins from scratch. Two different algorithms are available, and may be selected by using the "method" option:

```
assign_bond_orders($mol, method => 'itub');
assign_bond_orders($mol, method => 'baber');
```

itub

This is the default if no method is specified. Developed from scratch by the author of this module, this algorithm requires only the connection table information, and it requires that all hydrogen atoms be explicit. It looks for an atom with unsatisfied valence, increases a bond order, and then does the same recursively for the neighbors. If everybody's not happy at the end, it backtracks and tries another bond. The recursive part does not cover the whole molecule, but only the contiguous region of "unhappy" atoms next to the starting atom and their neighbors. This permits separating the molecule into independent regions, so that if one is solved and there's a problem in another, we don't have to backtrack to the first one.

The itub algorithm has the following additional options:

use_coords

Although the algorithm does not *require* 3D coordinates, it uses them by default to improve the initial guesses of which bond orders should be increased. To avoid using coordinates, add the `use_coords` option with a false value:

```
assign_bond_orders($mol, use_coords => 0);
```

The results are the same most of the time, but using good coordinates improves the results for complicated cases such as fused heteroaromatic systems.

scratch

If true, start the bond order assignment from scratch by assuming that all bond orders are 1. If false, start from the current bond orders and try to fix the unsatisfied valences. This option is true by default.

baber

A bond order assignment algorithm based on Baber, J. C.; Hodgkin, E. E. J. Chem. Inf. Comp. Sci. 1992, 32, 401-406 (with some interpretation).

This algorithm uses the 3D coordinates along with various cutoffs and confidence formulas to guess the bond orders. It then tries to resolve conflicts by looping through the atoms (but is not recursive or backtracking). It does not require explicit hydrogens (although it's better when they are available) because it was designed for use with real crystallographic data which often doesn't have hydrogen atoms.

This method doesn't always give a good answer, especially for conjugated and aromatic systems. The variation used in this module adds some random numbers to resolve some ambiguities and break loops, so the results are not even entirely deterministic (the 'itub' method is deterministic but the result may depend on the input order of the atoms).

VERSION

0.23

TO DO

Some future version should let the user specify the desired cutoffs, and not always create a bond but call a user-supplied function instead. This way these functions could be used for other purposes such as finding hydrogen bonds or neighbor lists.

Add some tests.

SEE ALSO

Chemistry::Mol, *Chemistry::Atom*, *Chemistry::Bond*, http://www.perlmol.org/.

AUTHOR

Ivan Tubert-Brohman<itub@cpan.org>

The new `find_bonds` algorithm was loosely based on a suggestion by BrowserUK on perlmonks.org (http://perlmonks.org/index.pl?node_id=352838).

COPYRIGHT

34 Chemistry::Canonicalize

Number the atoms in a molecule in a unique way

SYNOPSIS

```
use Chemistry::Canonicalize ':all';

# $mol is a Chemistry::Mol object
canonicalize($mol);
print "The canonical number for atom 1 is: ",
    $mol->atoms(1)->attr("canon/class");
print "The symmetry class for for atom 1 is: ",
    $mol->atoms(1)->attr("canon/symmetry_class");
```

DESCRIPTION

This module provides functions for "canonicalizing" a molecular structure; that is, to number the atoms in a unique way regardless of the input order.

The canonicalization algorithm is based on: Weininger, et. al., J. Chem. Inf. Comp. Sci. 29[2], 97-101 (1989)

This module is part of the PerlMol project, http://www.perlmol.org/.

ATOM ATTRIBUTES

During the canonicalization process, the following attributes are set on each atom:

canon/class

The unique canonical number; it is an integer going from 1 to the number of atoms.

canon/symmetry_class

The symmetry class number. Atoms that have the same symmetry class are considered to be topologicaly equivalent. For example, the two methyl carbons on 2-propanol would have the same symmetry class.

FUNCTIONS

These functions may be exported, although nothing is exported by default.

canonicalize($mol, %opts)

Canonicalizes the molecule. It adds the canon/class and canon/symmetry class to every atom, as discussed above. This function may take the following options:

sort

If true, sort the atoms in the molecule in ascending canonical number order.

invariants

This should be a subroutine reference that takes an atom and returns a number. These number should be based on the topological invariant properties of the atom, such as symbol, charge, number of bonds, etc.

VERSION

0.11

TO DO

Add some tests.

CAVEATS

Currently there is an atom limit of about 430 atoms.

These algorithm is known to fail to discriminate between non-equivalent atoms for some complicated cases. These are usually highly bridged structures explicitly designed to break canonicalization algorithms; I don't know of any "real-looking structure" (meaning something that someone would actually synthesize or find in nature) that fails, but don't say I didn't warn you!

SEE ALSO

Chemistry::Mol, *Chemistry::Atom*, *Chemistry::Obj*, http://www.perlmol.org/.

AUTHOR

Ivan Tubert <itub@cpan.org>

COPYRIGHT

35 Chemistry::InternalCoords::Builder

Build a Z-matrix from cartesian coordinates

SYNOPSIS

```
use Chemistry::InternalCoords::Builder 'build_zmat';

# $mol is a Chemistry::Mol object
build_zmat($mol);

# don't change the atom order!
build_zmat($mol, bfs => 0);
```

DESCRIPTION

This module builds a Z-matrix from the cartesian coordinates of a molecule, making sure that atoms are defined in a way that allows for efficient structure optimizations and Monte Carlo sampling.

By default, the algorithm tries to start at the center of the molecule and builds outward in a breadth-first fashion. Improper dihedrals are used to ensure clean rotation of groups without distortion. All distance and angle references use real bonds and bond angles where possible (the exception being disconnected structures).

This module is part of the PerlMol project, http://www.perlmol.org/.

FUNCTIONS

These functions may be exported, although nothing is exported by default. To export all functions, use the ":all" tag.

build_zmat($mol, %options)

Build a Z-matrix from the cartesian coordinates of the molecule. Side effect warning: by default, this function modifies the molecule heavily! First, it finds the bonds if there are no bonds defined already (for example, if the structure came from and XYZ file with no bond information). Second, it canonicalizes the molecule, as a means of finding the "topological center". Third, it builds the Z-matrix using a breadth-first search. Fourth, it sorts the atoms in the molecule in the order that they were defined in the Z-matrix.

Options:

bfs

Default: true. Follow the procedure described above. If bfs is false, then the atom order is not modified (that is, the atoms are added sequentially in the order in which they appear in the connection table, instead of using the breadth-first search).

sort

Default: true. Do the canonicalization step as described above. This option only applies when bfs => 1, otherwise it has no effect. If false and bfs => 1, the breadth-first search is done, but starting at the first atom in the connection table.

VERSION

0.20

CAVEATS

This version may not work properly for big molecules, because the canonicalization step has a size limit.

TO DO

Some improvements for handling disconnected structures, such as making sure that the intermolecular distance is short.

Allowing more control over how much the molecule will be modified: sort or not, canonicalize or not...

SEE ALSO

Chemistry::Mol, *Chemistry::Atom*, *Chemistry::InternalCoords*, *Chemistry::Bond::Find*, *Chemistry::Canonicalize*, http://www.perlmol.org/.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

36 Chemistry::3DBuilder

Generate 3D coordinates from a connection table

SYNOPSIS

```
# example: convert SMILES to MDL molfile
use Chemistry::3DBuilder qw(build_3d);
use Chemistry::File::SMILES;
use Chemistry::File::MDLMol;

my $s = '[O-]C(=O)C(N)C(C)CC';
my $mol = Chemistry::Mol->parse($s, format => 'smiles');

build_3d($mol);

print $mol->print(format => 'mdl');
```

DESCRIPTION

This module generates a three-dimensional molecular structure from a connection table, such as that obtained by a 2D representation of the molecule or from a SMILES string.

NOTE: this module is still at a very early stage of development so it has important limitations. 1) It doesn't handle rings or stereochemistry yet! 2) The bond lengths and atoms are very approximate as they don't really account for different elements. 3) Only the sp3, sp2, and sp hybridizations are supported.

SUBROUTINES

These subroutines may be exported; to export all, use the ':all' tag.

build_3d($mol)

Add internal and cartesian coordinates to the molecule `$mol`.

VERSION

0.10

SEE ALSO

Chemistry::Mol, *Chemistry::InternalCoords*.

The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

37 Chemistry::Ring::Find

Find the rings (cycles) in a molecule

SYNOPSIS

```
use Chemistry::Ring::Find ':all';

# find the smallest ring containing $atom
my $ring = find_ring($atom);

# find all the rings containing $bond
my @rings = find_ring($bond, all => 1);

# see below for more options

# find the six 4-atom rings in cubane
@rings = find_rings($cubane);

# find a cubane SSSR with five rings
@rings = find_rings($cubane, sssr => 1);
```

DESCRIPTION

The Chemistry::Ring::Find module implements a breadth-first ring finding algorithm, and it can find all rings that contain a given atom or bond, the Smallest Set of Smallest Rings (SSSR), or the "almost SSSR", which is an unambiguous set of rings for cases such as cubane.The algorithms are based on ideas from:

1) Leach, A. R.; Dolata, D. P.; Prout, P. Automated Conformational Analysis and Structure Generation: Algorithms for Molecular Perception J. Chem. Inf. Comput. Sci. 1990, 30, 316-324

2) Figueras, J. Ring perception using breadth-first search. J. Chem. Inf. Comput. Sci. 1996, 36, 986-991.

Ref. 2 is only used for find_ring, not for find_rings, because it has been shown that the overall SSSR method in ref 2 has bugs. Ref 1 inspired find_rings, which depends on find_ring.

This module is part of the PerlMol project, http://www.perlmol.org/.

FUNCTIONS

These functions may be exported explicitly, or all by using the :all tag, but nothing is exported by default.

find_ring($origin, %opts)

Find the smallest ring containg $origin, which may be either an atom or a bond. Returns a Chemistry::Ring object. Options:

all

If true, find all the rings containing $origin. If false, return the first ring found. Defaults to false. "All" is supposed to include only "simple" rings, that is, rings that are not a combination of smaller rings.

min

Only find rings with a the given minimum size. Defaults to zero.

max

Only find rings up to the given maximium size. Defaults to unlimited size.

size

Only find rings with this size. Same as setting min and max to the same size. Default: unspecified.

exclude

An array reference containing a list of atoms that must NOT be present in the ring. Defaults to the empty list.

mirror

If true, find each ring twice (forwards and backwards). Defaults to false.

@rings = find_rings($mol, %options)

Find "all" the rings in the molecule. In general it return the Smallest Set of Smallest Rings (SSSR). However, since it is well known that the SSSR is not unique for molecules such as cubane (where the SSSR consists of five unspecified four-member rings, even if the symmetry of the molecule would suggest that the six faces of the cube are equivalent), in such cases find_rings will return a non-ambiguous "non-smallest" set of smallest rings, unless the "sssr" option is given. For example,

```
@rings = find_rings($cubane);
# returns SIX four-member rings

@rings = find_rings($cubane, sssr => 1);
# returns FIVE four-member rings (an unspecified subset of
# the six rings above.)
```

BUGS

The "all" option in find_ring doesn't quite work as expected. It finds all simple rings and some bridged rings. It never finds fused rings (which is good).

VERSION

0.20

SEE ALSO

Chemistry::Ring, http://www.perlmol.org.

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

38 Chemistry::Isotope

Table of the isotopes exact mass data

SYNOPSIS

```
use Chemistry::Isotope ':all';

# get the exact atomic mass for an isotope
my $m  = isotope_mass(235, 92); # 235.043923094753

my $ab_table = isotope_abundance('C');
while (my ($A, $percent_ab) = each %$ab_table) {
    print "$A\t$percent_ab\n";
}
# this should print (the order may vary):
# 12    98.93
# 13    1.07
```

DESCRIPTION

This module contains the exact mass data from the table of the isotopes. It has an exportable function, isotope_mass, which returns the mass of an atom in mass units given its mass number (A) and atomic number (Z); and a function isotope_abundance which returns a table with the natural abundance of the isotopes given an element symbol.

The table of the masses includes 2931 nuclides and is taken from http://ie.lbl.gov/txt/awm95.txt (G. Audi and A.H. Wapstra, Nucl. Phys. A595, 409, 1995)

The table of natural abundances includes 288 nuclides and is taken from the Commission on Atomic Weights and Isotopic Abundances report for the International Union of Pure and Applied Chemistry in Isotopic Compositions of the Elements 1989, Pure and Applied Chemistry, 1998, 70, 217. http://www.iupac.org/publications/pac/1998/pdf/7001x0217.pdf

FUNCTIONS

isotope_mass($A, $Z)

Return the mass for the atom with the given mass number and atomic number, or undef if the nuclide is not in the data table.

isotope_abundance($symbol)

Returns a hash reference with the natural abundance information for the isotopes of a given element. The hash keys are the mass numbers, and the values are the abundance percentages. For example, isotope_abundance('C') returns the following structure:

```
{
    '13' => '1.07',
```

```
    '12' => '98.93'
};
```

VERSION

0.11

SEE ALSO

Chemistry::Atom
The PerlMol website http://www.perlmol.org/

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT

39 mok

An awk for molecules

SYNOPSIS

```
mok [OPTION]...  'CODE' FILE...
```

DESCRIPTION

The purpose of mok is to read all the molecules found in the files that are given in the command line, and for each molecule execute the CODE that is given. The CODE is given in Perl and it has at its disposal all of the methods of the PerlMol toolkit.

This mini-language is intended to provide a powerful environment for writing "molecular one-liners" for extracting and munging chemical information. It was inspired by the AWK programming language by Aho, Kernighan, and Weinberger, the SMARTS molecular pattern description language by Daylight, Inc., and the Perl programming language by Larry Wall.

Mok takes its name from Ookla the Mok, an unforgettable character from the animated TV series "Thundarr the Barbarian", and from shortening "molecular awk". For more details about the Mok mini-language, see LANGUAGE SPECIFICATION below.

Mok is part of the PerlMol project, http://www.perlmol.org.

OPTIONS

-3

Generate 3D coordinates using Chemistry::3DBuilder.

-a

"Aromatize" each molecule as it is read. This is needed for example for matching SMARTS patterns that use aromaticity or ring primitives.

-b

Find bonds. Use it when reading files with no bond information but 3D coordinates to detect the bonds if needed (for example, if you want to do match a pattern that includes bonds). If the file has explicit bonds, mok will not try to find the bonds, but it will reassign the bond orders from scratch.

-c CLASS

Use CLASS instead of Chemistry::Mol to read molecules

-d

Delete dummy atoms after reading each molecule. A dummy atom is defined as an atom with an unknown symbol (i.e., it doesn't appear on the periodic table), or an atomic number of zero.

-D

Print debugging information, such as the way the input program was tokenized and parsed into blocks and subs. This may be useful for diagnosing syntax errors when the default error mesage is not informative enough.

-f FILE

Run the code from FILE instead of the command line

-h

Print usage information and exit

-p TYPE

Parse patterns using the specified TYPE. Default: 'smarts'. Other options are 'smiles' and 'midas'.

-t TYPE

Assume that every file has the specified TYPE. Available types depend on which Chemistry::File modules are installed, but currently available types include mdl, sdf, smiles, formula, mopac, pdb.

LANGUAGE SPECIFICATION

A Mok script consists of a sequence of pattern-action statements and optional subroutine definitions, in a manner very similar to the AWK language.

```
pattern_type:/pattern/options { action statements }
{ action statements }
sub name { statements }
BEGIN { statements }
END { statements }
# comment
```

When the whole program consists of one unconditional action block, the braces may be omitted.

Program execution is as follows:

1) The BEGIN block is executed as soon as it's compiled, before any other actions are taken.

2) For each molecule in the files given in the command line, each pattern is applied in turn; if the pattern matches, the corresponding statement block is executed. The pattern is optional; statement blocks without a pattern are executed unconditionally. Subroutines are only executed when called explicitly.

3) Finally, the END block is executed.

The statements are evaluated as Perl statements in the Chemistry::Mok::UserCode::Default package. The following chemistry modules are conveniently loaded by default:

```
Chemistry::Mol;
Chemistry::Atom ':all';
Chemistry::Bond;
Chemistry::Pattern;
Chemistry::Pattern::Atom;
Chemistry::Pattern::Bond;
Chemistry::File;
Chemistry::File::*;
Math::VectorReal ':all';
```

Besides these, there is one more function available for convenience: `println`, which is defined by `sub println { print "\@_", "\n" }`.

Pattern Specification

The pattern must be a SMARTS string readable by the Chemistry::File::SMARTS module, unless a different type is specified by means of the -p option or a pattern_type is given explicitly before the pattern itself. The pattern is given within slashes, in a way reminiscent of AWK and Perl regular expressions. As in Perl, certain one-letter options may be included after the closing slash. An option is turned on by giving the corresponing lowercase letter and turned off by giving the corresponding uppercase letter.

g/G

Match globally (default: off). When not present, the Mok interpreter only matches a molecule once; when present, it tries matching again in other parts of the molecule. For example, /C/ matches butane only once (at an unspecified atom), while /C/g matches four times (once at each atom).

o/O

Overlap (default: on). When set and matching globally, matches may overlap. For example, /CC/go pattern could match twice on propane, but /CC/gO would match only once.

p/P

Permute (default: off). Sometimes there is more than one way of matching the same set of pattern atoms on the same set of molecule atoms. If true, return these "redundant" matches. For example, /CC/gp could match ethane with two different permutations (forwards and backwards).

Special Variables

When blocks with action statements are executed, some variables are defined automatically. The variables are local, so you can do whatever you want with them with no side effects. However, the objects themselves may be altered by using their methods.

NOTE: Mok 0.10 defined $file, $mol, $match, and $patt in lowercase. While they still work, the lowercase variables are deprecated and may be removed in the future.

$FILE

The current filename.

$MOL

A reference to the current molecule as a Chemistry::Mol object.

$MATCH

A reference to the current match as a Chemistry::Pattern object.

$PATT

The current pattern as a string.

$FH

The current input filehandle. This provides low-level access in case you want to rewind or seek into the file, tell the current position, etc. Playing with $FH may break things if you are not careful. Use at your own risk!

@A

The atoms that were matched. It is defined as @A = $MATCH->atom_map if a pattern was used, or @A = $MOL->atoms within an unconditional block. Remember that this is a Perl array, so it is zero-based, unlike the one-based numbering used by most file types and some PerlMol methods.

@B

The bonds that were matched. It is defined as @A = $MATCH->bond_map if a pattern was used, or @A = $MOL->bonds within an unconditional block. Remember Remember that this is a Perl array, so it is zero-based, unlike the one-based numbering used by most file types and some PerlMol methods.

Special Blocks

Within action blocks, the following block names can be used with Perl funcions such as `next` and `last`:

MATCH

BLOCK

MOL

FILE

EXAMPLES

Print the names of all the molecules found in all the .sdf files in the current directory:

```
mok 'println $MOL->name' *.sdf
```

Find esters among *.mol; print the filename, molecule name, and formula:

```
mok '/C(=O)OC/{ printf "$FILE: %s (%s)\n",
    $MOL->name, $MOL->formula }' *.mol
```

Find out the total number of atoms:

```
mok '{ $n += $MOL->atoms } END { print "Total: $n atoms\n" }' *.mol
```

Find out the average C-S bond length:

```
mok '/CS/g{ $n++; $len += $B[0]->length }
    END { printf "Average C-S bond length: %.3f\n", $len/$n; }' *.mol
```

Convert PDB files to MDL molfiles:

```
mok '{ $FILE =~ s/pdb/mol/; $MOL->write($FILE, format => "mdlmol") }' *.pdb
```

Find molecules with a given formula by overriding the formula pattern type globally (this example requires *Chemistry::FormulatPattern*):

```
mok -p formula_pattern '/C6H12O6/{ println $MOL->name }' *.sdf
```

Find molecules with a given formula by overriding the formula pattern type just for one specific pattern. This can be used when more than one pattern type is needed in one script.

```
mok 'formula_pattern:/C6H12O6/{ println $MOL->name }' *.sdf
```

SEE ALSO

awk(1), perl(1) *Chemistry::Mok*, *Chemistry::Mol*, *Chemistry::Pattern*, http://dmoz.org/Arts/Animation/Cartoons/Titles/T/T

Tubert-Brohman, I. Perl and Chemistry. The Perl Journal 2004-06 (http://www.tpj.com/documents/s=7618/tpj0406/).

The PerlMol project site at http://www.perlmol.org.

VERSION

0.26

AUTHOR

Ivan Tubert-Brohman <itub@cpan.org>

COPYRIGHT