CANDO documentation version 0.02

Christian Schafmeister, Temple University

December 20, 2009

Contents

Chapter 1

Cando Programs

Cando-Script consists of several executables and a library for incorporation into Python.

- cando - Executes *Cando-Script* code as a single process.

  Example:

  ```
  cat >hello.csc
  [println "Hello world" ]
  <control-d>

  cando hello.csc
  ```

  → Hello world

- candoMpi - Executes *Cando-Script* code using MPI - Message Passing Interface.

  Multiple copies of candoMpi are executed in parallel and communicate with each other using MPI.

  Example:

  ```
  cat >hello.csc
  [println ( "Hello world from process# %d" % [mpiRank] ) ]
  <control-d>
  ```

```
mpirun -np 4 candoMpi hello.csc
```

→ P1:Hello world from process # 1

→ P3:Hello world from process # 3

→ P2:Hello world from process # 2

→ P0:Hello world from process # 0

You culd also run this with:

```
mpirun -np 4 candoMpi -o hello.out hello.csc
```

Then every process will write its output into separate files named: hello.out0, hello.out1, hello.out2, hello.out3 (see below)

- candoView - View *Cando-Script* objects in a graphical environment.

  For an example you will need a molecule in Tripos "mol2" format. You can copy the $CANDO_HOME/examples/p53Mdm2/1ycr_p53Mdm2Complex.mol2 file to the current directory.

  Example:

  ```
  cp $CANDO_HOME/examples/p53Mdm2/1ycr_p53Mdm2Complex.mol2 .
  cat >render.csc
  ( mol = [ loadMol2 "1ycr_p53Mdm2Complex.mol2" ] )
  [save mol "mol.oml" ]
  <control-d>
  cando render.csc
  candoView -d full mol.oml
  ```

  All of these programs accept the following options:

  - –database (-d) *dbName*

    Tells candoView to load the standard database "dbName" before it does anything else. Many Cando objects require a database to be loaded before they can be loaded into candoView. Within *Cando-Script* scripts you can also load a database using the **standardDatabase** command.

- –output (-o) *filename*

  Tells “cando” to write all output that would go to stdout to the file *filename*. If this option is given to “candoMpi” then every process creates a filename by appending the process rank to *filename* and writes all output that would go to stdout to that file (example -o job.log will become job.log0 for process 0, job.log1 for process 1 etc.).

- –seed (-s) *integer*

  Tells “cando” and “candoMpi” to seed the random number generator with *integer*. By default, if this option is not given then each “candoMpi” process will seed its random number generator with its process rank. This is to avoid having every “candoMpi” process carry out exactly the same Monte Carlo searches.

To use these programs you need to do the following:

1. Put the path to the top of the CANDO development directory in the environment variable CANDO_HOME

2. Execute the “setup” script for your system.

   On a Mac that’s $CANDO_HOME/targets/setup.osx

Chapter 2

Cando Scripting Language

Cando-Script is a language tailored to constructing and searching virtual oligomer libraries. Cando-Script is modeled after the languages Lisp and Scheme with Smalltalk/Objective-C thrown in, not to be different but because these languages have a simple and compact syntax and they seemed to be a good match for the problem.

Cando-Script is designed to allow a chemists to easily define virtual oligomer libraries, build 3D models of members of of oligomer libraries, score members of these libraries and identify the best oligomer structures that present functional groups in a desired three-dimensional constellation.

Cando-Script commands are invoked using two forms:

- Prefix form - [**command** *arg1 arg2 arg3 ...*]

  This is the "prefix" form, where the **command** is given followed by its arguments. Square brackets are used to indicate that this is a prefix form command.

  When *Cando-Script* encounters a command in prefix form it does the following:

  1. It checks if the command is a macro name like "defClass" or "if" and if it is then the arguments are passed to the internal macro code for it to handle in its own way. *Cando-Script* then goes on to the next command.
  2. *Cando-Script* evaluates all of the arguments and constructs a list of evaluated arguments to pass to the function or method.

3. *Cando-Script* checks to see if the first evaluated argument object recognizes the method with the **command** name and if it does *Cando-Script* invokes the method with the evaluated argument list. *Cando-Script* then puts the result of the invocation into a growing argument list and goes to the next command.
4. If the **command** didn't match an object/method call then *Cando-Script* checks if **command** matches a function call. If it does then *Cando-Script* invokes the function with the evaluated arguments and puts the result into a growing result list and goes to the next command.
5. *Cando-Script* throws an error saying that the current command is not recognized.

Examples:

```
[save hitList "hits.oml" ]  # saves the hitList object
                            #    to the file: hits.oml.
[:= x 10.0]                 # assigns the value 10.0
                            #    to the global variable x.
[println "Hello world" ]    # prints "Hello world" to stdout
                            #    followed by a carriage return.
```

- Infix form - (object **command** *arg1 arg2 ...*)

  This is the "infix" form, where the **command** is sent to the *object* with the arguments *arg1 arg2 ....*

  Internally the "infix" form is automatically converted into "prefix" form by swapping the order of *object* and **command**.

  So the command: (*object* **command** *arg1 arg2 ...*)

  is converted to [**command** *object arg1 arg2 ...*]

  The purpose of the infix-form is to allow the programmer to use a more familiar notation for mathematical expressions and conditional expressions.

  Examples:

```
    ( x := 10.0 )                   # assigns the value 10.0
                                    #     to the global variable x.
    ( z = ( x * ( y + 10.0 ) ) )  # calculates y+10.0 and then
                                    # multiplies the result by x
                                    # and puts the results in the local variable z.
```

These forms can be mixed in any combination.
Examples:

```
[if ( y == 10 ) [println "y is ten" ] ]
( rect = [new Rectangle] )                        # Create a rectangle
( rect init [randomNumber01] [randomNumber01] ) # init with random width/height
```

A sample script is shown below:

```
[standardDatabase "full" ]
( builder = [ new Builder ] )
( acids = [ createMonomerPack "aaGlu"
    [parts
        [addPart "glu(S)" [aliasAtoms 'OE ] ]
        [addPart "glu(R)" [aliasAtoms 'OE ] ]
    ]
    [atomAliases 'carbO ]
] )
( builder addMonomerPack acids )

[setOligomer 'dipeptide
    [:
        [monomer 'a 'aaGlu [monomerAlias 'mon1 ] ]
        [link 'a 'dkp [monomer 'b 'aaGlu [monomerAlias 'mon2 ]]]
    ]
]

( builder addOligomer dipeptide )

[setGeometryScorer 'scorer
    [comparer
        [distance [alias 'mon1  'carbO] [alias 'mon2 'carbO] 2.0 ]
    ]
]


[setHitList 'hits 1000]

[set 'options [: [: 'UseRandomConformations false ] ] ]
[exhaustiveSearch builder scorer hits options ]
[save hits "_hits.xml"]
```

2.1 Control structures

CandoScript control structures like "if", "while" ...

2.1.1 ASSERT

[**ASSERT** *condition logString*]
If the condition is false then throw an exception with logString.

2.1.2 LOG

[**LOG** *logString*]
If debugging is on the *logString* is written to the log.

2.1.3 block

[**block** *command1 command2 ...*] → lastObject
Evaluates each command and returns the value *lastObject* from evaluating the last command. This is what you use to write blocks of code.

2.1.4 blockDEBUG

[**blockDEBUG** *command1 command2 ...*]
lastObject
Evaluate the block only if debugging is enabled. Return the last evaluated element or nil.

2.1.5 blockLOG

[**blockLOG** *"comment" command1 command2 ...*] → lastObject

Works just like "block" but if debugging is enabled then it prints a message to the log file when this block is entered and when it exists.

2.1.6 break

[**break**]

Break out of the current "foreach" or "while" loop.

2.1.7 cond

[**cond** *[[cond1 code1 ...] [cond2 code2 ...] ...]*]

Works just like lisp "cond" control structure. Evaluates each condition and for the first one that evaluates as true its associated block is evaluated.

2.1.8 continue

[**continue**]

Continue to the next iteration of the current "foreach" or "while" loop.

2.1.9 foreach

[**foreach** *localVariableName list code*]

For each element of the list put it in the localVariableName and evaluate the code.

2.1.10 if

[**if** *condition thenCode elseCode*]

[**if** *condition thenCode*]

If/then/else control statement.

2.1.11 ifTrue

[**ifFalse** *condition thenCode1 thenCode2 ...*]

If the condition is true then evaluate the thenCodes.

2.1.12 invoke

[**invoke** *Symbol::variable Cons::argumentList*]
Lookup the function in *variable* and call it with *argumentList*.

2.1.13 lambda

[**lambda** *arguments code*] → object
Creates an anonymous function that takes a list of *arguments* and evaluates *code* and returns the result. This is used for functional programming.

2.1.14 quote

[**quote** *object*] → unevaluatedObject
Returns the *object* without evaluating it.

2.1.15 return

[**return** *object*]
Returns from the current function/method and returns the object.

2.1.16 throw

[**throw** *messageString*]
Throw an exception. For now just throw string messages.

2.1.17 while

[**while** *condition code*]
While *condition* is True *code* is evaluated.
Example:

```
( x = 1 )
[while ( x < 10 ) [block
    [println ( "x = %d" % x ) ]
    ( x = ( x + 1 ) )
] ]
```

2.2 database

2.2.1 contextGrep

Text::contextKeySubstring
Search for contexts with keys that contain the substring.

2.2.2 setDatabase

[**standardDatabase** *directoryName:text*]
Set the database.

2.2.3 standardDatabase

[**standardDatabase** *directoryName:text*]
Load the database from $CANDO_RESOURCES/databases/*directoryName*.

2.3 Debugging

CandoScript commands used for debugging scripts.

2.4 general

2.4.1 add

[**add** *valueA:number valueB:number ...*] → number
[+ *valueA:number valueB:number ...*] → number
Return the sum of the arguments.

2.4.2 allGlobalNames

[**allGlobalNames**]
Return a string containing all of all variables in the namespace.

2.4.3 and

[**and** *boolA boolB*] → bool
Return boolA AND boolB.

2.4.4 caddr

[**caddr** *list*] → object
Return the third element of the list.

2.4.5 cadr

[**cadr** *list*] → object
Return the second element of the list.

2.4.6 car

[**car** *list*] → object
Return the first element of the list.

2.4.7 cdddr

[**cdddr** *list*] → object
Return the cdddr list after the first element is removed.

2.4.8 cddr

[**cddr** *list*] → object
Return the cddr list after the first element is removed.

2.4.9 cdr

[**cdr** *list*] → object
Return the rest of the list after the first element is removed.

2.4.10 className

[**className** *object*] → string
Return the name of the class the object belongs to.

2.4.11 cons

[**cons** *object list*] → cons
Create a "cons" with from object,list.

2.4.12 contentWithName

[**contentWithName** *object:matter name:text*]
Return the content of the Matter(Aggregate/Molecule/Residue) with the name *name*.

2.4.13 databaseDir

[**databaseDir**] → Text::
Return the path for the database directory.

2.4.14 debugDumpClassManager

[**debugDumpClassManager**]
Dump the class manager.

2.4.15 debugLogOff

[**debugLogOff** *true/false:bool*]
Turn on or off writing debug statements to the debug log. This is useful when running long scripts that crash, you can turn of debug logging up to the point where the crash happens and then examine the output.

2.4.16 debugLogOn

[**debugLogOn** *true/false:bool*]
Turn on or off writing debug statements to the debug log. This is useful when running long scripts that crash, you can turn of debug logging up to the point where the crash happens and then examine the output.

2.4.17 div

[**div** *valueA:number valueB:number*] → number
[/ *valueA:number valueB:number*] → number
Return the result of division of the arguments.

2.4.18 eq

[**eq** *valueA valueB*] → Bool::
(*valueA* == *valueB*) → Bool::
Return true if the objects are equal. For some objects (numbers,strings,bools) it compares the objects values. For more complex objects it returns true if they are identical.

2.4.19 format

[**format** *Text::format args ...*] → string
(*Text::format* % *args ...*) → string
Generates formatted output using the boost "format" library. It generates formatted output similar to the C-printf function. The result is returned as a string.

2.4.20 ge

[**ge** *valueA valueB*] → bool
(*valueA* >= *valueB*) → bool
Return true if valueA >= valueB.

2.4.21 gt

[**gt** *valueA valueB*] → bool
(*valueA* > *valueB*) → bool
Return true if valueA > valueB.

2.4.22 include

[**include** *Text::fileName*]
Open the *fileName*, compile and evaluate its contents. It looks through all of the directories in the global variable PATH and then the Scripts directory in the Cando application directory.

2.4.23 isOfClass

(*Object::object* **isOfClass** *Class::classObject*) → Bool::
Return true if *object* is a subclass of *classObject*.

2.4.24 isTopLevelScript

Return a true if this is a top level script or false if its an include file.

2.4.25 le

[**le** *valueA valueB*] → bool
(*valueA* <= *valueB*) → bool
Return true if valueA <= valueB.

2.4.26 length

[**length** *list*] → int
Return the length of the list.

2.4.27 let

[**let** *symbol object*]
(*symbol* = *object*)
Evaluate the arguments and put it into the local variable *symbol*.

2.4.28 list

[**list** *object1 object2 ...*] → list
[**:** *object1 object2 ...*] → list
Return a list formed by evaluating the arguments.

2.4.29 listref

[**listref** *list index*] → object
Return the element of the *list* at position *index*.

2.4.30 localVariableNames

[**localVariableNames**] → Text::
Return a list of all local variable names.

2.4.31 lt

[**lt** *valueA valueB*] → Bool::
(*valueA* < *valueB*) → Bool::
Return true if valueA < valueB.

2.4.32 map

2.4.33 max

[**max** *valueA:number valueB:number ...*] → number
Return the max of the arguments.

2.4.34 min

[**min** *valueA:number valueB:number ...*] → number
Return the min of the arguments.

2.4.35 mod

[**mod** *valueA:number valueB:number*] → number
Return the result of modulus of the arguments.

2.4.36 mul

[**mul** *Number::valueA Number::valueB ...*] → Number::
(*Number::valueA* ∗ *Number::valueB ...*) → Number::
Return the product of the arguments.

2.4.37 ne

[**ne** *valueA valueB*] → bool
(*valueA* **!=** *valueB*) → bool
Return true if the objects are not equal. For some objects (numbers,strings,bools) it compares the objects values. For more complex objects it returns true if they are not identical.

2.4.38 not

[**not** *boolA*] $\rightarrow$ bool
Return not boolA.

2.4.39 or

(*boolA* **or** *boolB*) $\rightarrow$ Bool::
Return boolA OR boolB.

2.4.40 print

[**print** *args ...*]
Print string representations of the arguments with no new line. See "println".

2.4.41 printPopPrefix

[**printPopPrefixln** *args ...*]
Pop a prefix to be printed everytime print is called the arguments followed by a new line.

2.4.42 printPushPrefix

[**printPushPrefixln** *args ...*]
Push a prefix to be printed everytime print is called the arguments followed by a new line.

2.4.43 repr

[**repr** *object*] $\rightarrow$ string
Return a string representation of the object.

2.4.44 set

[**set** *symbol object*]
(*symbol* **:=** *object*)
Evaluate the arguments and put it into the global variable *symbol*.

2.4.45 sub

[**sub** *valueA:number valueB:number*] → number
(*Number::valueA* − *Number::valueB*) → Number::
Return the difference of the arguments.

2.4.46 subClassOf

(*Object::object* **subClassOf** *Class::classObject*) → Bool::
Return true if *object* is a subclass of *classObject*.

2.4.47 scannerTest

[**testScanner** *Text::fileName*]
Open the *fileName*, run it through the scanner to test it.

2.5 Macro commands

These are special commands that manipulate the CandoScript environment. They don't evaluate their arguments in the same way that all other CandoScript commands do.

2.5.1 defClass

[**defClass** *Text::className instanceVariableNameList*]
[**defClass** *Text::className Class::baseCandoClass instanceVariableNameList*]
Define a class with the name *className*. The *baseClass* is optional and if provided then this new class will inherit all of the instance variables and methods of the base class. The *instanceVariableNameList* are the names of the instance variables (slots) for this class. Each instance variable ”x” will become part of the local namespace within methods for this class.

2.5.2 defFunction

[**defFunction** *functionName argumentNameList code...*]
Define a function with the name *functionName*. The *argumentNameList* defines the variables that are passed to the *code...*.

2.5.3 defMethod

[**defMethod** *Text::methodName Class::class argumentList code...*]

Define a method with the name *methodName* for the *class*. The first argument of the *argumentList* is the class instance (the "self" or "this" object) for which the method is being invoked.

2.6 monomerPack

2.6.1 createMonomerPack

[**createMonomerPack** *name:text monomersAndInterestingAtomNames:list atomAliases:list*]

[**createMonomerPack** *name:text monomersAndInterestingAtomNames:list*]

Create a MonomerPack and put it into the database with the name: *name*. A MonomerPack is a group of Stereoisomers each of which has zero or more atom names associated with it that will be built by CANDO during rapid searching through sequence and conformational space.

```
( aaGLu = [createMonomerPack "aaGlu"
    [parts
        [addPart 'glu(S) [aliasAtoms 'OE ] ]
        [addPart 'glu(R) [aliasAtoms 'OE ] ]
    ]
    [atomAliases 'carbO ]
] )
```

The command names "parts" and "atomAliases" are aliases for the "list" command.

2.6.2 setMonomerPack

[**setMonomerPack** *name:text monomersAndInterestingAtomNames:list atomAliases:list*]

[**setMonomerPack** *name:text monomersAndInterestingAtomNames:list*]

Create a MonomerPack and put it into the database with the name: *name*, also create a local variable with the name *name* containing this MonomerPack. A MonomerPack is a group of Stereoisomers each of which has zero or more atom names associated with it that will be built by CANDO during rapid searching through sequence and conformational space.

```
[setMonomerPack "aaGlu"
    [parts
        [addPart 'glu(S) [aliasAtoms 'OE ] ]
        [addPart 'glu(R) [aliasAtoms 'OE ] ]
    ]
    [atomAliases 'carbO ]
]
```

The commands "parts" and "atomAliases" are aliases for the "list" command.

2.7 Ring identification commands and objects

Commands to identify rings and to manage RingFinder objects.

2.7.1 identifyRings

[**identifyRings** *matter*]
Identify the Smallest Set of Smallest Rings (SSSR) for the Molecule or Aggregate *matter*. Set the ring membership flags of the atoms that are in rings.

Chapter 3

Cando Object Classes

This chapter describes the classes and methods available within Cando-Script.

3.1 Alias

3.1.1 Alias class methods

Alias

Text::monomerAlias Text::atomAlias
Create an Alias object that maintains a *monomerAlias* name and an *atomAlias* name.

3.2 AnchorOnOtherSideOfPlug

3.2.1 AnchorOnOtherSideOfPlug class methods

AnchorOnOtherSideOfPlug

plugName: Text::plugName required

3.3 AtomGrid

3.3.1 AtomGrid class methods

AtomGrid

Matter::matter gridResolution optional
optional
addRadius *withinSphere List::sphere* optional

3.4 Builder

Inherits from: Object

A Builder object builds three-dimensional structures of Oligomers. To achieve this, a Builder needs to be given at least one Oligomer object using "addOligomer" and any MonomerPacks that are used by the Oligomer using the "addMonomerPack" method.

A Builder object can be given any number of Oligomers and when its building an Oligomer it creates and manages an OligomerBuilder object that does the actual building of a single Oligomer.

A Builder object lets the user select between the oligomers that it has been given, select between the sequences of the current oligomer and select between the conformationsof the current sequence. It allows the user to build the entire three-dimensional structure of the current conformation or just the "interesting" atoms.

3.5 ChemDraw

3.5.1 ChemDraw class methods

ChemDraw

fileName: Text::name required
Define a ChemDraw object. Load a cdxml file from *name* and return the ChemDraw object define by it.

3.6 ExtractOthersFrame

3.6.1 ExtractOthersFrame class methods

ExtractOthersFrame

othersFrameName: Text::othersFrameName required
plugName: Text::myPlugName required
overlapsFrame: Frame::myFrame required
recognizer: FrameRecognizer::recognizer required
Create an object that will extract a frame of reference that has its origin atom in a preceeding monomer but overlaps this monomer. You must specify a frame of reference in this monomer *myFrame* that overlaps the others frame of reference and the FrameRecognizer that will recognize the others frame of reference.

3.7 Fragment

3.7.1 Fragment class methods

Fragment

name: Text::nameOfFragment required
atoms: Cons::listOfAtomNames required
Define a Fragment with *nameOfFragment* and containing the atom named in *listOfAtomNames*.

3.8 Frame

3.8.1 Frame class methods

Frame

name: Text::nameOfFrame required
origin: Text::nameOfOriginAtom required
recognizer: FrameRecognizer::recognizer required
Define a Frame with *nameOfFrame* and centered on the atom with name *nameOfOriginAtom* recognized by *recognizer*.

3.9 Hit

3.9.1 getBuiltMolecule

(*hit* **getBuiltMolecule**) → Molecule::structure

Looks up the "builderState" entry in the hit data and recreates a builder in the state that it was when the hit was identified. This method then returns the molecule with all atoms built in the hit conformation.

3.9.2 getData

(*hit* **getData**) → data::Dictionary

Return the Dictionary associated with the hit. The dictionary stores name/object pairs that describe the hit. The user can put any data they want into this dictionary.

3.9.3 getScore

→ *Real::score* Return the score value of the hit.

3.9.4 setScore

(*hit* **setScore** *Real::value*)

Set the score value of the hit. Only use this method on hits that haven't been added yet to a HitList - once the hit is in a HitList changing the score will mess up the ordering in the HitList.

3.10 HitList commands

Commands that operate on HitLists.

3.10.1 addAllHits

(*hitList* **addAllHits** *HitList::hits*)

Adds all of the hits in *hits* to *hitList*. If the *hitList* becomes overfull then the excess hits are discarded.

3.10.2 addHit

(*hitList* **addHit** *Hit::hit*)
Adds the *hit* to *hitList*. If the *hitList* becomes overfull then the excess hits are discarded.

3.10.3 describe

(*hitList* **describe**)
Print a description of the contents of the HitList.

3.10.4 getHit

(*hitList* **getHit** *Int::index*) → Hit::
Return a hit by its index value *index* (zero is the first entry). If the index is beyond the end of the *hitList* then the nil object [] is returned.

3.10.5 hitListGet

[**hitListGet** *hitList index*]
Return the hit at *index* from the *hitList*.

3.10.6 isAHit

(*hitList* **isAHit** *Real::score*) → Bool::
This method is used to evaluate if a new score represents a hit that is good enough to add to *hitList*. Compare the *score* to the scores of every hit in this list. If *score* is better than the worst score in the list then return true, if not return false.

3.10.7 numberOfHits

(*hitList* **numberOfHits**) → Int::
Return the number of hits in this HitList.

3.11 Mate

Inherits from: MonomerGrouper

A MonomerSet that keeps track of a capping monomer that is used to cap training oligomers when they are being defined. The capping monomer is supposed to be small and best represent the other members of the Mate.

3.11.1 Mate class methods

Mate

cap: Text::capName required

groupNames: List::groupNames required

Initialize a Mate object.

3.12 MultiScorer

3.12.1 MultiScorer class methods

MultiScorer

scorers Cons::scorers required

Create an MultiScorer object that maintains a list of Scorers.

3.13 Path

3.13.1 Path class methods

Path

*path:*Text::path

Create a Path object that maintains a system independant path to a file in the file system.

3.14 Plug

Defines one or two atoms of this monomer that can be plugged into, a plug name and a collection of Mate objects that can act as mates for this plug.

3.14.1 Plug class methods

Plug

name: Text::plugName required
bond0: Text::bond0AtomName required
bond1: Text::bond1AtomName optional
mates: Cons::listOfMates required
exportFrame: Frame::exportFrame optional
Initialize a Plug object. Plugs can have one bond (eg: amide) or two bonds (eg:diketopiperazine). Outgoing plugs export a frame of reference to the next monomer, use *exportFrame* to define this.

3.15 Scorer

3.15.1 Scorer class methods

Scorer

superposeAtoms Cons::superposeAtoms required
calculator ScoreOperation::calculator optional
Create an Scorer object.

3.16 StereoConfiguration

3.16.1 StereoConfiguration class methods

StereoConfiguration

atomName: Text::atom required
config: Text::configuration required
Provide the atom name and the stereo-configuration *configuration* of ”R” or ”S”.

3.17 StereoInformation

3.17.1 StereoInformation class methods

StereoInformation

stereoisomers: List::stereoisomers required
proChiralCenters: List:: optional
constrainedPiBonds: List:: optional

3.17.2 StereoInformation instance methods

addProChiralCenter

(*self* **addProChiralCenter** *ProChiralCenter::center*)
Add the *center* to the StereoInformation object.

3.18 Stereoisomer

3.18.1 Stereoisomer class methods

MultiStereoisomers

[**MultiStereoisomers** *nameTemplate:(Text::template) centers:(List::) configs:(List::)*] → Cons::stereoisomers

Stereoisomer

[**Stereoisomer** *name:(Text::name) pdb:(Text::pdb) configs:(List::)*] → StereoIsomer::

3.19 StringSet

3.19.1 StringSet class methods

StringSet

entries: Cons::listOfStrings required
Create a StringSet containing the strings in *listOfStrings*.

3.20 Superpose

3.20.1 Superpose class methods

Superpose

Alias::monomerAtomAlias OVector3::position
Create an Superpose object that maintains a *monomerAtomAlias* name and the point that it is supposed to superimpose on top of.

3.21 Class methods

In Cando-Script class names like "Hit" or "Real" return objects that are of the class "Class". These objects respond to the following methods.

3.21.1 describe

[**describe** *classObject*]
Dumps a description of the class to stdout.

3.22 HitList class

HitList objects store a sorted list of Hit objects as well as a Dictionary for name/object pairs.

3.22.1 setHitList

[**setHitList** *'symbol maxHits:int*]
Create a HitList that can store *maxHits* and put it into the variable named *'symbol*.

Chapter 4

MSMARTS chemical pattern matching

Based on SMARTS documentation at

`http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html`

MSMARTS is similar to SMARTS with the following differences.

- MSMARTS supports atom tags: numerical labels that can be attached to atoms as an MSMARTS substructure is matched to a molecule. The tagged atoms can then be referenced after the substructure is matched.

  For example: the MSMARTS string "[N&H1]1C2(=O3)" will recognize a secondary amide and the amide nitrogen, carbonyl carbon and carbonyl oxygen can be obtained using the tags "1", "2" and "3" after a successful match.

- The syntax for identifying rings is different. Rings are recognized with strings like: "C1CCC[C&?1]". The first "1" assigns a tag "1" to the first carbon, The "[C&?1]" atom tests if the atom is carbon and has the tag "1".

Substructure searching, the process of finding a particular pattern (subgraph) in a molecule (graph), is one of the most important tasks for computers in chemistry. It is used in virtually every application that employs a digital representation of a molecule, including depiction (to highlight a particular functional group), drug design (searching a database for similar structures and activity), analytical chemistry (looking for previously-characterized structures and comparing their data to that of an unknown), and a host of other problems.

MSMARTS expressions allow a chemist to specify substructures using rules that are straightforward extensions of SMILES. For example: to search a database for phenol-containing structures, one would use the SMARTS string [OH]c1cccc[c&?1], which is similar to SMILES (Note: the [c&?1] atom primative test is used to identify rings in MSMARTS). In fact, almost all SMILES specifications are valid SMARTS targets. Using SMARTS, flexible and efficient substructure-search specifications can be made in terms that are meaningful to chemists.

In the SMILES language, there are two fundamental types of symbols: atoms and bonds. Using these SMILES symbols, once can specify a molecule's graph (its "nodes" and "edges") and assign "labels" to the components of the graph (that is, say what type of atom each node represents, and what type of bond each edge represents).

The same is true in SMARTS: One uses atomic and bond symbols to specify a graph. However, in SMARTS the labels for the graph's nodes and edges (its "atoms" and "bonds") are extended to include "logical operators" and special atomic and bond symbols; these allow SMARTS atoms and bonds to be more general. For example, the SMARTS atomic symbol [C,N] is an atom that can be aliphatic C or aliphatic N; the SMARTS bond symbol (tilde) matches any bond.

Below is example code that uses SMARTS to find every amide bond in a molecule:

```
#
# Define a ChemInfo object that can carry out
# substructure searches
#
( amideSmarts = [ new ChemInfo] )
#
# Compile a substructure pattern using SMARTS code
#
( amideSmarts compileSmarts "N1~C2=O3" )
#
# Load a molecule
#
( p53 = [ loadMol2 "p53.mol2" ] )
```

```
#
# Iterate through every atom and if it matches
# the substructure search then extract the tagged
# atoms and print their names
#
[foreach a [ atoms p53 ] [block
    [ if ( amideSmarts matches a ) [block
        [println ( "-----Matching atom: %s" % ( a getName ) ) ]
        ( coAtom = ( amideSmarts getAtomWithTag "2" ) )
        ( oAtom = ( amideSmarts getAtomWithTag "3" ) )
        [ println ( "    Carbonyl carbon: %s" % ( coAtom getName ) ) ]
        [ println ( "    Carbonyl oxygen: %s" % ( oAtom getName ) ) ]
    ] ]
] ]
```

4.1 Atomic Primitives

SMARTS provides a number of primitive symbols describing atomic properties beyond those used in SMILES (atomic symbol, charge, and isotopic specifications). The following tables list the atomic primitives used in SMARTS (all SMILES atomic symbols are also legal). In these tables ¡n¿ stands for a digit, ¡c¿ for chiral class.

Note that atomic primitive H can have two meanings, implying a property or the element itself. [H] means hydrogen atom. [*H2] means any atom with exactly two hydrogens attached

Symbol	Symbol name	Atomic property requirements	Default
*	wildcard	any atom	(no default)
D*n*	APDegree	explicit connections	exactly one
H*n*	APTotalHCount	*n* attached hydrogens	exactly one
h*n*	APImplicitHCount	*n* implicit attached hydrogens	at least one
?*n*	APRingTest	Atom is tagged with *n*	(no default)
R*n*	APRingMemberCount	is in *n* SSSR rings (WORKS?)	any ring atom
r*n*	APRingSize	is in smallest SSSR size *n*	any ring atom
v*n*	APValence	total bond order *n*	exactly 1
X*n*	APConnectivity	*n* total connections	exactly 1
-*n*	APNegativeCharge	-*n* charge	exactly -1
–	APNegativeCharge 2x	-2 charge	exactly -2
—	APNegativeCharge 3x	-3 charge	exactly -3
+*n*	APPositiveCharge	+*n* charge	exactly +1
++	APPositiveCharge 2x	+2 charge	exactly +2
+++	APPositiveCharge 3x	+3 charge	exactly +3
#*n*	APAtomicNumber	atomic number *n*	(no default)
n	APAtomicMass	atomic mass *n*	(no default)
$(*MSMARTS*)	recursive MSMARTS	match recursive MSMARTS	(no default)

Some of these have not been debugged. Test before you trust them.

Examples:

[CH2]	aliphatic carbon with two hydrogens (methylene carbon)
[!C;R]	(NOT aliphatic carbon) AND in ring
[!C;!R0]	same as above (”!R0” means not in zero rings)
[n;H1]	H-pyrrole nitrogen
[n&H1]	same as above
[nH1]	same as above
[c,n&H1]	any arom carbon OR H-pyrrole nitrogen
[X3&H0]	atom with 3 total bonds and no H’s
[c,n;H1]	(arom carbon OR arom nitrogen) and exactly one H
[Cl]	any chlorine atom
[35*]	any atom of mass 35
[35Cl]	chlorine atom of mass 35
[F,Cl,Br,I]	the 1st four halogens.

4.2 Logical Operators

Atom and bond primitive specifications may be combined to form expressions by using logical operators. In the following table, e is an atom or bond SMARTS expression (which may be a primitive). The logical operators are listed in order of decreasing precedence (high precedence operators are evaluated first).

All atomic expressions which are not simple primitives must be enclosed in brackets. The default operation is & (high precedence "and"), i.e., two adjacent primitives without an intervening logical operator must both be true for the expression (or subexpression) to be true.

The ability to form expressions gives the SMARTS user a great deal of power to specify exactly what is desired. The two forms of the AND operator are used in SMARTS instead of grouping operators.

Symbol	**Expression**	**Meaning**
exclamation	!e1	not e1
ampersand	e1&e2	e1 and e2 (high precedence)
comma	e1,e2	e1 or e2
semicolon	e1;e2	e1 and e2 (low precedence)

4.3 Recursive MSMARTS

Any MSMARTS expression may be used to define an atomic environment by writing a SMARTS starting with the atom of interest in this form: $(*MSMARTS*) Such definitions may be considered atomic properties. These expressions can be used in same manner as other atomic primitives (also, they can be nested). Recursive SMARTS expressions are used in the following manner:

*C	atom connected to methyl or methylene carbon
*CC	atom connected to ethyl carbon
[$(*C);$(*CC)]	Atom in both above environments (matches CCC)

The additional power of such expressions is illustrated by the following example which derives an expression for methyl carbons which are ortho to oxygen and meta to a nitrogen on an aromatic ring.

CaaO	C ortho to O
CaaaN	C meta to N
Caa(O)aN	C ortho to O and meta to N (but 2O,3N only)
Ca(aO)aaN	C ortho to O and meta to N (but 2O,5N only)
C[$(aaO);$(aaaN)]	C ortho to O and meta to N (all cases)

Index