

---

# **matlab2cpp Documentation**

***Release 0.5***

**Jonathan Feinberg**

October 13, 2016



## CONTENTS

<b>1</b>	<b>User Manual</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	User interaction . . . . .	4
1.3	Configuring translation . . . . .	7
1.4	Translation rules . . . . .	13
1.5	Behind the frontends . . . . .	16
<b>2</b>	<b>Developer Manual</b>	<b>21</b>
2.1	Module overview . . . . .	21
2.2	Quick translation functions . . . . .	21
2.3	The tree constructor . . . . .	27
2.4	Node representation . . . . .	63
2.5	Datatypes . . . . .	70
2.6	Auto-configure datatype . . . . .	70
2.7	Collection . . . . .	70
2.8	Translation rules . . . . .	75
2.9	Datatype scope . . . . .	77
2.10	Testsuite . . . . .	77
	<b>Python Module Index</b>	<b>79</b>
	<b>Index</b>	<b>81</b>



## USER MANUAL

## 1.1 Introduction

matlab2cpp is a semi-automatic tool for converting code from Matlab to C++. At the moment, matlab2cpp is the name of the python module while m2cpp is the name of the python script. m2cpp is found in the root folder. When installing the matlab2cpp module, the python script is copied to a system folder so that the script is available in path. Then the m2cpp script can be executed by typing “m2cpp” in the command line interface (cmd in Windows, terminal in Linux).

Note that it is not meant as a complete tool for creating runnable C++ code. For example, the *eval*-function can not be supported because there is no general way to implement it in C++. Instead the program is aimed as a support tool, which aims at speed up the conversion process as much as possible for a user that needs to convert Matlab programs by hand anyway. The software does this by converting the basic structures of the Matlab-program (functions, branches, loops, etc.), adds variable declarations, and for some simple code, do a complete translation. And any problem the program encounters during conversion will be written in a log-file. From there manual conversions can be done by hand.

Currently, the code will not convert the large library collection of functions that Matlab currently possesses. However, there is no reason for the code not to support these features in time. The extension library is easy to extend.

### 1.1.1 Installation

Requirements:

- Python 2.7.3
- Armadillo (Not required for running, but generator creates armadillo code.)
- C++11 (Plotting and TBB require C++11)

Optional:

- TBB
- Sphinx (for compiling documentation)
- Argcomplete (for tab-completion support)

Linux/Mac:

As root, run the following command:

```
$ python setup.py install
```

In addition to installing the matlab2cpp module, the executable ‘m2cpp’ is copied to “/usr/local/bin/” The executable ‘m2cpp’ is now available from path.

Windows:

> Python setup.py install

A bat script is created so that m2cpp.py can be executed by typing m2cpp. The bat script and m2cpp.py is copied to “sys.executable”. “sys.executable” is the location where Python is installed. The executable ‘m2cpp’ is now available from path.

Linux, Mac, Windows:

If you want to put the executable m2cpp in another place, modify the setup.py file. From line 27 starts the code which makes the m2cpp.py file available from path. Alternatively, remove the code from line 27. The executable m2cpp.py can freely be copied or be added to path or environmental variables manually (with or without the .py extension).

Armadillo:

Armadillo is a linear algebra library for the C++ language. The Armadillo library can be found at <http://arma.sourceforge.net>. Some functionality in Armadillo rely on a math library like LAPACK, BLAS, OpenBLAS or MKL. When installing Armadillo, it will look for installed math libraries. If Armadillo is installed, the library can be linked with the link flag *-larmadillo*. Armadillo can also be linked directly, see the *FAQ* at the Armadillo webpage for more information.

I believe MKL is the fastest math library and it can be downloaded for free at <https://software.intel.com/en-us/articles/free-mkl>.

TBB:

By inserting pragmas in the Matlab code, for loops can be marked by the user. The program can then either insert OpenMP or TBB code to parallelize the for loop. To compile TBB code, the TBB library has to be installed. See *Parallel flags*, *-omp*, *-tbb* for more details.

Sphinx:

```
pip install sphinx
pip install sphinxcontrib-autoprogram
pip install sphinxcontrib-napoleon
pip install sphinx-argparse
```

Argcomplete:

```
pip install argcomplete
activate-global-python-argcomplete
```

This only works for Bash and would require a restart of terminal emulator.

## 1.1.2 An illustrating Example

Assuming Linux installation and *m2cpp* is available in path. Code works analogous in Mac and Windows.

Consider a file *example.m* with the following content:

```
function y=f(x)
    y = x+4
end
function g()
    x = [1,2,3]
    f(x)
end
```

Run conversion on the file:

```
$ m2cpp example.m
```

This will create two files: *example.m.hpp* and *example.m.py*.

In *example.m.hpp*, the translated C++ code is placed. It looks as follows:

```
#include <armadillo>
using namespace arma ;

TYPE f(TYPE x)
{
    TYPE y ;
    y = x+4 ;
    return y ;
}

void g()
{
    TYPE x ;
    x = [1, 2, 3] ;
    f(x) ;
}
```

Matlab doesn't declare variables explicitly, so m2cpp is unable to complete the translation. To create a full conversion, the variables must be declared. Declarations can be done in the file *example.m.py*. After the first run, it will look as follows:

```
# Supplement file
#
# Valid inputs:
#
# uint      int      float   double  cx_double
# uvec      ivec     fvec     vec      cx_vec
# urowvec   irowvec  frowvec  rowvec   cx_rowvec
# umat      imat     fmat     mat       cx_mat
# ucube     icube    fcube     cube     cx_cube
#
# char      string   struct   structs  func_lambda

functions = {
    "f" : {
        "y" : "",
        "x" : "",
    },
    "g" : {
        "x" : "",
    },
}
includes = [
    '#include <armadillo>',
    'using namespace arma ;',
]
```

In addition to defining includes at the bottom, it is possible to declare variables manually by inserting type names into the respective empty strings. However, some times it is possible to guess some of the variable types from context. To let the software try to guess variable types, run conversion with the *-s* flag:

```
$ m2cpp example.m -s
```

The file *example.m.py* will then automatically be populated with data types from context:

```
# ...

functions = {
    "f" : {
        "y" : "irowvec",
        "x" : "irowvec",
    },
    "g" : {
        "x" : "irowvec",
    },
}
includes = [
    '#include <armadillo>',
    'using namespace arma ;',
]
```

It will not always be successful and some of the types might in some cases be wrong. It is therefore also possible to adjust these values manually at any time.

Having run the conversion with the variables converted, creates a new output for *example.m.hpp*:

```
#include <armadillo>
using namespace arma ;

irowvec f(irowvec x)
{
    irowvec y ;
    y = x+4 ;
    return y ;
}

void g()
{
    irowvec x ;
    int _x [] = [1, 2, 3] ;
    x = irowvec(_x, 3, false) ;
    f(x) ;
}
```

This is valid and runnable C++ code. For such a small example, no manual adjustments were necessary.

## 1.2 User interaction

The simplest way to interact with the *Matlab2cpp*-toolbox is to use the *m2cpp* frontend. The script automatically creates files with various extensions containing translations and/or meta-information.

### 1.2.1 m2cpp

The toolbox frontend of the Matlab2cpp library. Use this to try to do automatic and semi-automatic translation. The program will create files with the same name as the input, but with various extra extensions. Scripts will receive the extension *.cpp*, headers and modules *.hpp*. A file containing data type and header information will be stored in a *.py* file. Any errors will be stored in *.log*.

usage: m2cpp [-h] [-o] [-c] [-s] [-S] [-r] [-t] [-T] [-d] [-p PATHS\_FILE] [-omp] [-tbb] [-l LINE] [-n] filename



**filename**

File containing valid Matlab code.

**-h, --help**

show this help message and exit

**-o, --original**

Include original Matlab code line as comment before the C++ translation of the code line

**-c, --comments**

Include Matlab comments in the generated C++ files.

**-s, --suggest**

Automatically populate the `<filename>.py` file with datatype with suggestions if possible.

**-S, --matlab-suggest**

Creates a folder `m2cpp_temp`. In the folder the matlab file(s) to be translated are also put. These matlab file(s) are slightly modified so that they output data-type information of the variables to file(s). This output can then be used to set the datatypes for the translation.

**-r, --reset**

Ignore the content of `<filename>.py` and make a fresh translation.

**-t, --tree**

Print the underlying node tree. Each line in the output represents a node and is formatted as follows:

```
<codeline> <position> <class> <backend> <datatype> <name> <translation>
```

The indentation represents the tree structure.

**-T, --tree-full**

Same as -t, but the full node tree, but include meta-nodes.

**-d, --disp**

Print out the progress of the translation process.

**-p <paths\_file>, --paths\_file <paths\_file>**

Flag and paths\_file (-p path\_to\_pathsfile). m2cpp will look for matlab files in the location specified in the paths\_file

**-omp, --enable-omp**

OpenMP code is inserted for Parfor and loops marked with the pragma `%%PARFOR` (in Matlab code) when this flag is set.

**-tbb, --enable-tbb**

TBB code is inserted for Parfor and loops marked with the pragma `%%PARFOR` (in Matlab code) when this flag is set.

**-l <line>, --line <line>**

Only display code related to code line number `<line>`.

**-n, --nargin**

Don't remove if and switch branches which use nargin variable.

For the user, the flags -o, -c, -s, -S, -r, -p -omp, -tbb are the useful flags. The flags -t, -T are good for debugging because they print the structure of the Abstract Syntax Tree (AST). The -d flag gives useful information on the parsing of the Matlab code and insight in how the AST is built.

## 1.2.2 Suggest flags, -s, -S

Read the section *Suggestion engine* first. When using m2cpp the corresponding suggest is set with the flag -s. The suggest engine works well for simple cases. For more complex cases, not all the variables get a type suggestion and

the suggested type could be wrong.

The other suggest flag -S get the datatypes by running the (Matlab) code with Matlab. Information of the datatypes are written to files which can be extracted by the code translator. For this flag to work, in addition to having Matlab installed, the Matlab Engine API for Python has to be installed (see: [Install MATLAB Engine API for Python](#)). Matlab has to be able to run the code to extract the datatypes. So if the code require datafiles or special Matlab modules (e.g. numerical modules), these have to be available for this option to work. The Matlab suggest option is not 100%, but still quite good at suggesting datatypes. A downside with the using Matlab to suggest datatypes, is that Matlab takes some time to start up and then run the (Matlab) code.

### 1.2.3 Multiple directories, -p paths\_file

In Matlab the script and function files have to be in the same folder for the function files to be found. To call a function script located in a different folder, the folder has to be added to path. This can be done with *addpath* or *path*. In a separate file from the Matlab main and function scripts, a separate script can be written to set the path to different folders:

```
Dir='/path_to_folder/SeismicLab/codes/';
path(path, strcat(Dir, 'bp_filter/'));
path(path, strcat(Dir, 'decon'));
path(path, strcat(Dir, 'dephasing'));
path(path, strcat(Dir, 'fx'));
...
```

The flag option -p *paths\_file* can be set to parse such a file. Then Matlab as well as m2cpp can find function scripts that are located in other directories.

### 1.2.4 Parallel flags, -omp, -tbb

The program m2cpp can do parallelization of simple for loops (so called embarrassingly parallel). To let the program know which loops the user wants to parallelize, use the pragma *%#PARFOR* before the loop (similar to the way its done in OpenMP). The flags -omp and -tbb can then be used to chose if OpenMP code or TBB code will be inserted to parallelize the code. Matlab's *parfor* doesn't require the pragma *%#PARFOR* to parallelize. If neither -omp nor -tbb flag is used, no OpenMP or TBB code is inserted and we will get a sequential for loop. When compiling, try link flags -fopenmp for OpenMP and -ltbb for TBB. OpenMP is usually available for the compiler out of the box. TBB needs to be installed (see: <https://www.threadingbuildingblocks.org/>). The TBB code makes use of lambda functions which is a C++ feature. C++11 is probably not set as standard for the compiler, i.e., in the GNU compiler g++, the flag -std=c++11 is required to make use of C++11 features.

### 1.2.5 Quick translation functions

Even though *m2cpp* is sufficient for performing all code translation, many of the examples in this manual are done through a python interface, since some of the python functionality also will be discussed. Given that *Matlab2cpp* is properly installed on your system, the python library is available in Python's path. The module is assumed imported as:

```
>>> import matlab2cpp as mc
```

Quick functions collection of frontend tools for performing code translation. Each of the function *qcpp()*, *qhpp()*, *qpy()* and *qlog()* are directly related to the functionality of the **m2cpp** script. The name indicate the file extension that the script will create. In addition there are the three functions *qtree()* and *qscript()*. The former represents a summary of the created node tree. The latter is a simple translation tool that is more of a one-to-one translation.

For an overview of the various quick-functions, see [Quick translation functions](#).

## 1.2.6 Plotting functionality

Plotting functionality is available through a wrapper, which calls Python's matplotlib. If a Matlab code with plotting calls is translated, the file *SPlot.h* is generated. The C++ file that is generated also *#include* this file. To compile the generated code, the Python have to be included. The code in *SPlot.h* makes of C++11 features, so compiler options for C++11 may be needed as well. With the GNU compiler g++, I can compile the generated code with: `g++ my_cpp_file.cpp -o runfile -I /usr/include/python2.7/ -lpython2.7 -larmadillo -std=c++11`

Additional flags could be `-O3` (optimization) `-ltbb` (in case of TBB parallelization)

## 1.3 Configuring translation

One of the translation challenges is how each variable type is determined. In C++ all variables have to be explicitly declared, while in Matlab they are declared implicitly at creation. When translating between the two languages, there are many variables where the data types are unknown and impossible for the Matlab2cpp software to translate. How to translate the behavior of an integer is vastly different from an float matrix.

To differentiate between types, each node have an attribute `type` which represents the node datatype. Datatypes can be roughly split into two groups: **numerical** and **non-numerical** types. The numerical types are as follows:

	unsigned int	integer	float	double	complex
<i>scalar</i>	uword	int	float	double	cx_double
<i>vector</i>	uvec	ivec	fvec	vec	cx_vec
<i>row-vector</i>	urowvec	irowvec	frowvec	rowvec	cx_rowvec
<i>matrix</i>	umat	imat	fmat	mat	cx_mat
<i>cube</i>	ucube	icube	fcube	cube	cx_cube

Values along the horizontal axis represents the amount of memory reserved per element, and the along the vertical axis represents the various number of dimensions. The names are equivalent to the ones in the Armadillo package.

The non-numerical types are as follows:

Name	Description
<i>char</i>	Single text character
<i>string</i>	Text string
<i>struct</i>	Struct container
<i>structs</i>	Struct array container
<i>func_lambda</i>	Anonymous function

### 1.3.1 Function scope

If not specified otherwise, the program will not assign datatype types to any of variables. The user could in theory navigate the node tree and assign the variables one by one using the node attributes to navigate. (See section [Behind the frontends](#) for details.) However that would be very cumbersome. Instead the datatypes are define collectively inside their scope. In the case of variables in functions, the scope variables are the variables declaration `Declares` and function parameters `Params`. To reach the variable that serves as a scope-wide type, the node attribute `declare` can be used.

Manually interacting with the variable scope is simpler then iterating through the full tree, but can in many cases still be cumbersome. To simplify interaction with datatype scopes, each program has an suppliment attribute `ftypes`. The attribute is a nested dictionary where the outer shell represents the function name the variables are defined. The inner shell is the variables where keys are variable names and values are types. It can be used to quickly retrieving and inserting datatypes. For example:

```
>>> tree = mc.build("function f(a)")
>>> print tree.ftypes
{'f': {'a': ''}}
>>> tree.ftypes = {"f": {"a": "int"}}
>>> print mc.qscript(tree)
void f(int a)
{
    // Empty block
}
```

## 1.3.2 Anonymous functions

In addition to normal function, Matlab have support for anonymous function through the name prefix @. For example:

```
>>> print mc.qscript("function f(); g = @(x) x^2; g(4)")
void f()
{
    std::function<double(int)> g ;
    g = [] (int x) {pow(x, 2) ; } ;
    g(4) ;
}
```

The translator creates an C++11 lambda function with equivalent functionality. To achieve this, the translator creates an extra function in the node-tree. The name of the function is the same as assigned variable with a `_`-prefix (and a number postfix, if name is taken). The information about this function dictate the behaviour of the output. The supplement file have the following form:

```
>>> print mc.qpy("function f(); g = @(x) x^2; g(4)")
functions = {
    "_g" : {
        "x" : "int",
    },
    "f" : {
        "g" : "func_lambda",
    },
}
includes = [
    '#include <armadillo>',
    'using namespace arma ;',
]
```

The function `g` is a variable inside `f`'s function scope. It has the datatype `func_lambda` to indicate that it should be handled as a function. The associated function scope `_g` contains the variables inside the definition of the anonymous function.

## 1.3.3 Data structure

Data structures in Matlab can be constructed explicitly through the `struct`-function. However, they can also be constructed implicitly by direct assignment. For example will `a.b=4` create a `struct` with name `a` that has one field `b`. When translating such a snippet, it creates a C++-struct, such that:

```
>>> print mc.qhpp("function f(); a.b = 4.", suggest=True)
#include <armadillo>
using namespace arma ;

struct _A
```

```
{
    double b ;
} ;

void f()
{
    _A a ;
    a.b = 4. ;
}
```

In the supplement file, the local variable *a* will be assigned as a *struct*. In addition, since the struct has content, the supplement file creates a new section for structs. It will have the following form:

```
>>> print mc.qpy("function f(); a.b = 4.", suggest=True)
functions = {
    "f" : {
        "a" : "struct",
    },
}
structs = {
    "a" : {
        "b" : "double",
    },
}
includes = [
    '#include <armadillo>',
    'using namespace arma ;',
]
```

Quick retrieving and inserting struct variables can be done through the `stypes` attribute:

```
>>> tree = mc.build("a.b = 4")
>>> tree.ftypes = {"f": {"a": "struct"}}
>>> tree.stypes = {"a": {"b": "double"}}
>>> print mc.qcpp(tree)
#include <armadillo>
using namespace arma ;

struct _A
{
    double b ;
} ;

int main(int argc, char** argv)
{
    _A a ;
    a.b = 4 ;
    return 0 ;
}
```

### 1.3.4 Struct tables

Given that the data structure is indexed, e.g. `a(1).b`, it forms a struct table. Very similar to regular *structs*, which only has one value per element. There are a couple of differences in the translation. First, the struct is declared as an array:

```
>>> print mc.qhpp("function f(); a(1).b = 4.", suggest=True)
#include <armadillo>
using namespace arma ;

struct _A
{
    double b ;
} ;

void f()
{
    _A a[100] ;
    a[0].b = 4. ;
}
```

The translation assigned reserves 100 pointers for the content of `a`. Obviously, there are situations where this isn't enough (or too much), and the number should be increased. So second, to adjust this number, the supplement file specifies the number of elements in the integer `_size`:

```
>>> print mc.qpy("function f(); a(1).b = 4.", suggest=True)
functions = {
    "f" : {
        "a" : "structs",
    },
}
structs = {
    "a" : {
        "_size" : 100,
        "b" : "double",
    },
}
includes = [
    '#include <armadillo>',
    'using namespace arma ;',
]
```

### 1.3.5 Suggestion engine

The examples so far, when the functions `qcpp()`, `qhpp()` and `qpy()` are used, the argument `suggest=True` have been used, and all variable types have been filled in. Consider the following program where this is not the case:

```
>>> print mc.qhpp("function c=f(); a = 4; b = 4.; c = a+b", suggest=False)
#include <armadillo>
using namespace arma ;

TYPE f()
{
    TYPE a, b, c ;
    a = 4 ;
    b = 4. ;
    c = a+b ;
    return c ;
}
```

Since all variables are unknown, the program decides to fill in the dummy variable `TYPE` for each unknown variable. Any time variables are unknown, `TYPE` is used. The supplement file created by `m2cpp` or `qpy()` reflects all these unknown variables as follows:

```
>>> print mc.qpy("function c=f(); a = 4; b = 4.; c = a+b", suggest=False)
functions = {
  "f" : {
    "a" : "", # int
    "b" : "", # double
    "c" : "",
  },
}
includes = [
  '#include <armadillo>',
  'using namespace arma ;',
]
```

By flipping the boolean to True, all the variables get assigned datatypes:

```
>>> print mc.qpy("function c=f(); a = 4; b = 4.; c = a+b", suggest=True)
functions = {
  "f" : {
    "a" : "int",
    "b" : "double",
    "c" : "double",
  },
}
includes = [
  '#include <armadillo>',
  'using namespace arma ;',
]
```

The resulting program will have the following complete form:

```
>>> print mc.qhpp(
...     "function c=f(); a = 4; b = 4.; c = a+b", suggest=True)
#include <armadillo>
using namespace arma ;

double f()
{
  double b, c ;
  int a ;
  a = 4 ;
  b = 4. ;
  c = a+b ;
  return c ;
}
```

Note here though that the variable `c` didn't have a suggestion. The suggestion is an interactive process such that `a` and `b` both must be known beforehand. The variable `a` and `b` get assigned the datatypes `int` and `double` because of the direct assignment of variable. After this, the process starts over and tries to find other variables that suggestion could fill out for. In the case of the `c` variable, the assignment on the right were an addition between `int` and `double`. To not lose precision, it then chooses to keep `double`, which is passed on to the `c` variable. In practice the suggestions can potentially fill in all datatypes automatically in large programs, and often quite intelligently. For example, variables get suggested across function call scope:

```
>>> print mc.qscript('function y=f(x); y=x; function g(); z=f(4)')
int f(int x)
{
  int y ;
  y = x ;
  return y ;
}
```

```
}

void g()
{
    int z ;
    z = f(4) ;
}
```

And accross multiple files:

```
>>> builder = mc.Builder()
>>> builder.load("f.m", "function y=f(x); y=x")
>>> builder.load("g.m", "function g(); z=f(4) ")
>>> builder.configure(suggest=True)
>>> tree_f, tree_g = builder[:]
>>> print mc.qscript(tree_f)
int f(int x)
{
    int y ;
    y = x ;
    return y ;
}
>>> print mc.qscript(tree_g)
void g()
{
    int z ;
    z = f(4) ;
}
```

### 1.3.6 Verbatim translations

In some cases, the translation can not be performed. For example, the Matlab function `eval` can not be properly translated. Matlab is interpreted, and can easily take a string from local name space, and feed it to the interpreter. In C++ however, the code must be pre-compiled. Not knowing what the string input is before runtime, makes this difficult. So instead it makes more sense to make some custom translation by hand.

Since `matlab2cpp` produces C++ files, it is possible to edit them after creation. However, if changes are made to the Matlab-file at a later point, the custom edits have to be added manually again. To resolve this, `matlab2cpp` supports verbatim translations through the supplement file `.py` and through the node attribute `vtypes`. `vtype` is a dictionary where the keys are string found in the original code, and the values are string of the replacement.

Performing a verbatim replacement has to be done before the node tree is constructed. Assigning `vtypes` doesn't work very well. Instead the replacement dictionary can be passed as argument to `build()`:

```
>>> tree = mc.build(''a=1
... b=2
... c=3'', vtypes = {"b": "_replaced_text_"})
>>> print mc.qscript(tree)
a = 1 ;
// b=2
_replaced_text_
c = 3 ;
```

Note that when a match is found, the whole line is replaced. Note also how the source code is retained a comment above the verbatim translation. The verbatim key can only match a single line, however the replacement might span multiple lines. For example:



```
>>> replace_code = '''one line
... two line
... three line'''
>>> tree = mc.build(''a=1
... b=2
... c=3'', vtypes={"b": replace_code})
>>> print mc.qscript(tree)
a = 1 ;
// b=2
one line
two line
three line
c = 3 ;
```

Verbatims can also be utilized by modifying the .py file. Consider the Matlab script:

```
a = 1 ;
b = 2 ;
c = 3 ;
```

Using the m2cpp script to translate the Matlab script produces a C++ file and a .py file. By adding code to the .py file, verbatim translation can be added. This is done by using the keyword verbatims and setting it to a python dictionary. Similar to vtype, keys are strings found in the original code, and the values are string of the replacement:

```
functions = {
"main" : {
"a" : "int",
"b" : "int",
"c" : "int",
},
}
includes = [
'#include <armadillo>',
'using namespace arma ;',
]
verbatims = {"b = 2 ;" : '''one line
two line
tree line'''
}
```

In the generated C++ file the second assignment is replaced with the verbatim translation:

```
int main(int argc, char** argv)
{
    int a, c ;
    a = 1 ;
    // b = 2 ;
    one line
    two line
    tree line
    c = 3 ;
    return 0 ;
}
```

## 1.4 Translation rules

In Matlab2cpp, the simplest form for translation is a simple string saved to a variable. For example:

```
>>> Int = "6"
```

The name *Int* (with capital letter) represents the node the rule is applicable for integers. The right hand side when it is a string, will be used as the translation every time *Int* occurs. To illustrate this, consider the following simple example, where we pass a snippet to the *qscript()* function:

```
>>> print mc.qscript("5")
5 ;
```

To implement the new rule we (globally) insert the rule for all instances of *Int* as follows:

```
>>> print mc.qscript("5", Int=Int)
6 ;
```

Obviously, this type of translation is not very useful except for a very few exceptions. First of all, each *Int* (and obviously many other nodes) contain a value. To represent this value, the translation rule uses string interpolation. This can be implemented as follows:

```
>>> Int = "|%(value)s|"
>>> print mc.qscript("5", Int=Int)
|5| ;
```

There are also other attributes than *value*. For example variables, represented through the node *Var* have a name, which refer to it's scope defined name. For example:

```
>>> Var = "__%(name)s__"
>>> print mc.qscript("a = 4", Var=Var)
__a__ = 4 ;
```

Since all the code is structured as a node tree, many of the nodes have node children. The translation is performed leaf-to-root, implying that at the time of translation of any node, all of it's children are already translated and available in interpolation. The children are indexed by number, counting from 0. Consider the simple example of a simple addition:

```
>>> print mc.qtree("2+3", core=True)
1 1Block      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Plus    expression  int
1 1| | | Int   int         int
1 3| | | Int   int         int
>>> print mc.qscript("2+3")
2+3 ;
```

The node tree (at it's core) consists of a *Block*. That code *Block* contains one *Statement*. The *Statement* contains the *Plus* operator, which contains the two *Int*. Each *Node* in the tree represents one token to be translated.

From the perspective of the addition *Plus*, the two node children of class *Int* are available in translation respectively as index 0 and 1. In interpolation we can do as follows:

```
>>> Plus = "%(1)s+%(0)s"
>>> print mc.qscript("2+3", Plus=Plus)
3+2 ;
```

One obvious problem with this approach is that the number of children of a node might not be fixed. For example the *Plus* in "2+3" has two children while "1+2+3" has three. To address nodes with variable number of node children, alternative representation can be used. Instead of defining a string, a tuple of three string can be used. They represents prefix, infix and postfix between each node child. For example:

```
>>> Plus = "", "+", ""
```

It implies that there should be a “+” between each children listed, and nothing at the ends. In practice we get:

```
>>> print mc.qscript("2+3", Plus=Plus)
2+3 ;
>>> print mc.qscript("1+2+3", Plus=Plus)
1+2+3 ;
```

And this is the extent of how the system uses string values. However, in practice, they are not used much. Instead of strings and tuples functions are used. They are defined with the same name the string/tuple. This function should always take a single argument of type *Node* which represents the current node in the node tree. The function should return either a *str* or *tuple* as described above. For example, without addressing how one can use *node*, the following is equivalent:

```
>>> Plus = "", "+", ""
>>> print mc.qscript("2+3", Plus=Plus)
2+ 3 ;
>>> def Plus(node):
...     return "", " +", ""
...
>>> print mc.qscript("2+3", Plus=Plus)
2 +3 ;
```

One application of the *node* argument is to use it to configure datatypes. As discussed in the previous section *Configuring translation*, the *node* attribute *type* contains information about datatype. For example:

```
>>> def Var(node):
...     if node.name == "x": node.type = "vec"
...     if node.name == "y": node.type = "rowvec"
...     return node.name
>>> print mc.qscript("function f(x,y)", Var=Var)
void f(vec x, rowvec y)
{
    // Empty block
}
```

For more details on the behavior of the *node* argument, see section on node *Behind the frontends*. For an extensive list of the various nodes available, see developer manual *Collection*.

### 1.4.1 Rule context

In the basic translation rule described above, each node type have one universal rule. However, depending on context, various nodes should be addressed differently. For example the snippet *sum(x)* lend itself naturally to have a rule that targets the name *sum* which is part of the Matlab standard library. Its translation should is as follows:

```
>>> print mc.qscript("sum(x)")
arma::sum(x) ;
```

However, if there is a line *sum = [1,2,3]* earlier in the code, then the translation for *sum(x)* become very different. *sum* is now an array, and the translation adapts:

```
>>> print mc.qscript("sum=[1,2,3]; sum(x)")
sword _sum [] = {1, 2, 3} ;
sum = irowvec(_sum, 3, false) ;
sum(x-1) ;
```

To address this in the same node will quickly become very convoluted. So instead, the rules are split into various backends. This simplifies things for each rule that have multiple interpretations, but also ensures that code isn't to bloated. For an overview of the various backend, see the developer manual [Translation rules](#).

## 1.4.2 Reserved rules

The example above with `sum(x)` is handled by two rules. In the second iteration, it is a *datatype* of type *irowvec* and is therefore processed in the corresponding rule for *irowvec*. However, in the former case, `sum` is a function from the Matlab standard library. In principle there is only one rule for all function calls like this. However, since the standard library is large, the rules are segmented into rules for each name.

In the rule `rules._reserved`, each function in the standard library (which matlab2cpp supports) is listed in the variable `rules.reserved`. The context for reserved function manifest itself into the rules for function calls *Get*, variables *Var* and in some cases, multivariate assignment *Assigns*. As described above, the rules should then have these names respectively. However to indicate the name, the rules also includes node names as suffix. For example, the function call for `sum` is handled in the rule `Get_sum()`.

In practice this allows us to create specific rules for any node with names, which includes variables, function calls, functions, to name the obvious. For example, if we want to change the summation function from *armadillo* to the *accumulation* from *numeric* module, it would be implemented as follows:

```
>>> Get_sum = "std::accumulate(", " ", " ", ")"
>>> print mc.qscript("sum(x)", Get_sum=Get_sum)
std::accumulate(x) ;
```

This rules is specific for all function calls with name `sum` and wont be applied for other functions:

```
>>> Get_sum = "std::accumulate(", " ", " ", ")"
>>> print mc.qscript("min(x)", Get_sum=Get_sum)
min(x) ;
```

There are many rules to translation rule backends in matlab2cpp. This is mainly because each datatype have a corresponding backend.

## 1.5 Behind the frontends

Common for all the various frontends in *qfunctions* are two classes: *Builder* and *Node*. The former scans Matlab code and constructs a node tree consisting of instances of the latter.

### 1.5.1 The Builder class

Iterating through Matlab code always starts with constructing a *Builder*:

```
>>> builder = mc.Builder()
```

This is an empty shell without any content. To give it content, we supply it with code:

```
>>> builder.load("file1.m", "a = 4")
```

The function saves the code locally as `builder.code` and initiate the `create_program` method with index 0. The various `create_*` methods are then called and used to populate the node tree. The code is considered static, instead the index, which refer to the position in the code is increased to move forward in the code. The various constructors uses the support modules in the `qtree` to build a full token tree. The result is as follows:

```

>>> print builder
  Project      unknown      TYPE
    | Program      unknown      TYPE      file1.m
    | | Includes      unknown      TYPE
1 1| | Funcs      unknown      TYPE      file1.m
1 1| | | Main      unknown      TYPE      main
1 1| | | | Declares      unknown      TYPE
1 1| | | | | Var      unknown      TYPE      a
1 1| | | | Returns      unknown      TYPE
1 1| | | | Params      unknown      TYPE
1 1| | | | Block      unknown      TYPE
1 1| | | | | Assign      unknown      TYPE
1 1| | | | | Var      unknown      TYPE      a
1 5| | | | | Int      unknown      TYPE
    | | Inlines      unknown      TYPE      file1.m
    | | Structs      unknown      TYPE      file1.m
    | | Headers      unknown      TYPE      file1.m
    | | Log          unknown      TYPE      file1.m

```

It is possible to get a detailed output of how this process is done, by turning the `disp` flag on:

```

>>> builder = mc.Builder(disp=True)
>>> builder.load("file1.m", "a = 4")
loading file1.m
  Program      functions.program
0 Main        functions.main
0 Codeblock   codeblock.codeblock
0 Assign      assign.single      'a = 4'
0 Var         variables.assign    'a'
4 Expression  expression.create   '4'
4 Int         misc.number         '4'

```

This printout lists the core Matlab translation. In the four columns the first is the index to the position in the Matlab code, the second is the node created, the third is the file and function where the node was created, and lastly the fourth column is a code snippet from the Matlab code. This allows for quick diagnostics about where an error in interpretation might have occurred.

Note that the tree above for the most part doesn't have any backends or datatypes configured. They are all set to unknown and TYPE respectively. To configure backends and datatypes, use the `configure()` method:

```

>>> builder.configure(suggest=True)
>>> print builder
  Project      program      TYPE
    | Program      program      TYPE      file1.m
    | | Includes      program      TYPE
1 1| | Funcs      program      TYPE      file1.m
1 1| | | Main      func_return TYPE      main
1 1| | | | Declares      func_return TYPE
1 1| | | | | Var      int      int      a
1 1| | | | Returns      func_return TYPE
1 1| | | | Params      func_return TYPE
1 1| | | | Block      code_block TYPE
1 1| | | | | Assign      int      int
1 1| | | | | Var      int      int      a
1 5| | | | | Int      int      int
    | | Inlines      program      TYPE      file1.m
    | | Structs      program      TYPE      file1.m
    | | Headers      program      TYPE      file1.m
    | | Log          program      TYPE      file1.m

```

Multiple program can be loaded into the same builder. This allows for building of projects that involves multiple files. For example:

```
>>> builder = mc.Builder()
>>> builder.load("a.m", "function y=a(x); y = x+1")
>>> builder.load("b.m", "b = a(2)")
```

The two programs refer to each other through their names. This can the suggestion engine use:

```
>>> print mc.qscript(builder[0])
int a(int x)
{
    int y ;
    y = x+1 ;
    return y ;
}
>>> print mc.qscript(builder[1])
b = a(2) ;
```

Note that the frontend functions (like `qscript()`) configure the tree if needed.

## 1.5.2 The Node class

So far the translation has been for the most part fairly simple, where the translation is reduced to either a string or a tuple of strings for weaving sub-nodes together. Consider the following example:

```
>>> def Plus(node):
...     return "", " +", ""
...
>>> print mc.qscript("2+3", Plus=Plus)
2 +3 ;
```

Though not used, the argument *node* represents the current node in the tree as the tree is translated. We saw this being used in the last section *Translation rules* to get and set node datatype. However, there are much more you can get out of the node. First, to help understand the current situation from a coding perspective, one can use the help function `summary()`, which gives a quick summary of the node and its node children. It works the same way as the function `qtree()`, but can be used mid translation. For example:

```
>>> def Plus(node):
...     print node.summary()
...     return "", " +", ""
...
>>> print mc.qscript("2+3", Plus=Plus)
1 1Plus      expression  int
1 1| Int      int        int
1 3| Int      int        int
2 +3 ;
```

The first line represent the current node *Plus*.

introduce the node tree structure and how the nodes relate to each other. This will vary from program to program, so a printout of the current state is a good startingpoint. This can either be done through the function `qtree`, or manually as follows:

```
>>> builder = mc.Builder()
>>> builder.load("unnamed", "function y=f(x); y=x+4")
>>> builder[0].ftypes = {"f" : {"x": "int", "y": "double"}}
>>> builder.translate()
```

```

>>> print builder
    Project      program      TYPE
    | Program    program      TYPE    unnamed
    | | Includes  program      TYPE
    | | | Include  program      TYPE    #include <armadillo>
    | | | Include  program      TYPE    using namespace arma ;
1 1| | Funcs      program      TYPE    unnamed
1 1| | | Func      func_return  TYPE    f
1 1| | | | Declares  func_return  TYPE
1 1| | | | | Var      double      double  y
1 1| | | | Returns  func_return  TYPE
1 10| | | | | Var      double      double  y
1 13| | | | Params  func_return  TYPE
1 14| | | | | Var      int          int      x
1 16| | | | Block    code_block   TYPE
1 18| | | | | Assign  expression  int
1 18| | | | | | Var      double      double  y
1 20| | | | | Plus    expression  int
1 20| | | | | | Var      int          int      x
1 22| | | | | | Int      int          int
    | | Inlines    program      TYPE    unnamed
    | | Structs    program      TYPE    unnamed
    | | Headers    program      TYPE    unnamed
    | | | Header    program      TYPE    f
    | | | Log       program      TYPE    unnamed

```

The Project is the root of the tree. To traverse the tree in direction of the leafs can be done using indexing:

```

>>> funcs = builder[0][1]
>>> func = funcs[0]
>>> assign = func[3][0]
>>> var_y, plus = assign
>>> var_x, int_4 = plus

```

Moving upwards towards the root of the tree is done using the parent reference:

```

>>> block = assign.parent
>>> print assign is var_y.parent
True

```

The node provided in the node function is the current node for which the parser is trying to translate. This gives each node full control over context of which is is placed. For example, consider the following:

```

>>> print mc.qtree("x(end, end)", core=True)
1 1Block      code_block   TYPE
1 1| Statement  code_block   TYPE
1 1| | Get      unknown     TYPE    x
1 3| | | End      expression  int
1 8| | | End      expression  int
>>> def End(node):
...     if node is node.parent[0]:
...         return node.parent.name + ".n_rows"
...     if node is node.parent[1]:
...         return node.parent.name + ".n_cols"
...
>>> print mc.qscript("x(end, end)", End=End)
x(x.n_rows, x.n_cols) ;

```

Here the rule *End* was called twice, where the if-test produces two different results. Also, information about the parent

is used in the value returned.

**See also:**

*Builder*



## DEVELOPER MANUAL

### 2.1 Module overview

The toolbox is sorted into the following modules:

Module	Description
<i>qfunctions</i>	Functions for performing simple translations
<i>Builder</i>	Constructing a tree from Matlab code
<i>Node</i>	Components in the tree representation of the code
<i>collection</i>	The collection of various node
<i>configure</i>	Routine for setting datatypes and backends of the various nodes
<i>rules</i>	Translation rules
<i>supplement</i>	Functions for inserting and extraction datatypes
<i>testsuite</i>	Suite for testing software

The simplest way to use the library is to use the quick translation functions. They are available through the *mc.qfunctions* module and mirrors the functionality offered by the *m2cpp* function.

### 2.2 Quick translation functions

For simplest use of the module, these function works as an alternative frontend to the *mconvert* script.

Function	Description
<i>build()</i>	Build token tree representation of code
<i>qcpp()</i>	Create content of <i>.cpp</i> file
<i>qhpp()</i>	Create content of <i>.hpp</i> file
<i>qpy()</i>	Create content of supplement <i>.py</i> file
<i>qlog()</i>	Create content of <i>.log</i> file
<i>qscript()</i>	Create script translation
<i>qtree()</i>	Create summary of node tree

`matlab2cpp.build(code, disp=False, retall=False, suggest=True, comments=True, vtypes=None, **kws)`

Build a token tree out of Matlab code. This function is used by the other quick-functions as the first step in code translation.

The function also handles syntax errors in the Matlab code. It will highlight the line it crashed on and explain as far as it can why it crashed.

#### Parameters

- **code** (*str*) – Code to be interpreted

- **disp** (*bool*) – If true, print out diagnostic information while interpreting.
- **retall** (*bool*) – If true, return full token tree instead of only code related.
- **suggest** (*bool*) – If true, suggestion engine will be used to fill in datatypes.
- **comments** (*bool*) – If true, comments will be striped away from the solution.
- **vtypes** (*dict*) – Verbatim translations added to tree before process.
- **\*\*kws** – Additional arguments passed to *Builder*.

**Returns** The tree constructor if *retall* is true, else the root node for code.

**Return type** Builder,Node

**Example use::**

```
>>> builder = mc.build("a=4", retall=True)
>>> print isinstance(builder, mc.Builder)
True
>>> node = mc.build("a=4", retall=False)
>>> print isinstance(node, mc.Node)
True
>>> print mc.build("a**b")
Traceback (most recent call last):
...
SyntaxError: line 1 in Matlab code:
a**b
  ^
Expected: expression start
```

**See also:**

*mtree()*, *Builder*, *Node*

matlab2cpp.**qcpp**(*code*, *suggest=True*, **\*\*kws**)

Quick code translation of matlab script to C++ executable. For Matlab modules, code that only consists of functions, will be placed in the *qhpp()*. In most cases, the two functions must be used together to create valid runnable code.

**Parameters**

- **code** (*str*, *Node*, *Builder*) – A string or tree representation of Matlab code.
- **suggest** (*bool*) – If true, use the suggest engine to guess data types.
- **\*\*kws** – Additional arguments passed to *Builder*.

**Returns** Best estimate of script. If code is a module, return an empty string.

**Return type** *str*

**Example**

```
>>> code = "a = 4; b = 5.; c = 'abc'"
>>> print mc.qcpp(code, suggest=False)
#include <armadillo>
using namespace arma ;

int main(int argc, char** argv)
{
```

```

TYPE a, b, c ;
a = 4 ;
b = 5. ;
c = "abc" ;
return 0 ;
}
>>> print mc.qcpp(code, suggest=True)
#include <armadillo>
using namespace arma ;

int main(int argc, char** argv)
{
    double b ;
    int a ;
    std::string c ;
    a = 4 ;
    b = 5. ;
    c = "abc" ;
    return 0 ;
}
>>> build = mc.build(code, retall=True)
>>> print mc.qcpp(build) == mc.qcpp(code)
True

```

**See also:**

*qscript()*, *qhpp()*, *Builder*

matlab2cpp.**qhpp**(code, suggest=False)

Quick module translation of Matlab module to C++ library. If the code is a script, executable part of the code will be placed in *qcpp()*.

#### Parameters

- **code** (*str*, *Node*, *Builder*) – A string or tree representation of Matlab code.
- **suggest** (*bool*) – If true, use the suggest engine to guess data types.
- **\*\*kws** – Additional arguments passed to *Builder*.

**Returns** C++ code of module.

**Return type** *str*

#### Example

```

>>> code = "function y=f(x); y=x+1; end; function g(); f(4)"
>>> print mc.qhpp(code)
#include <armadillo>
using namespace arma ;

TYPE f(TYPE x) ;
void g() ;

TYPE f(TYPE x)
{
    TYPE y ;
    y = x+1 ;
    return y ;
}

```

```
void g()
{
    f(4) ;
}

>>> print mc.qhpp(code, suggest=True)
#include <armadillo>
using namespace arma ;

int f(int x) ;
void g() ;

int f(int x)
{
    int y ;
    y = x+1 ;
    return y ;
}

void g()
{
    f(4) ;
}
```

**See also:**

[`qcpp\(\)`](#), [`Builder`](#)

matlab2cpp.[`qpy`](#)(*code*, *suggest=True*, *prefix=False*)

Create annotation string for the supplement file containing datatypes for the various variables in various scopes.

**Parameters**

- **code** (*str*, *Builder*, *Node*) – Representation of the node tree.
- **suggest** (*bool*) – Use the suggestion engine if appropriate.
- **prefix** (*bool*) – include a helpful comment in the beginning of the string.

**Returns** Supplement string

**Return type** *str*

**Example**

```
>>> code = "a = 4; b = 5.; c = 'abc'"
>>> print mc.qpy(code, suggest=False)
functions = {
    "main" : {
        "a" : "", # int
        "b" : "", # double
        "c" : "", # string
    },
}
includes = [
    '#include <armadillo>',
    'using namespace arma ;',
]
>>> print mc.qpy(code, suggest=True)
functions = {
```

```

"main" : {
  "a" : "int",
  "b" : "double",
  "c" : "string",
},
}
includes = [
  '#include <armadillo>',
  'using namespace arma ;',
]

```

**See also:**

[supplement](#), [datatype](#)

`matlab2cpp.qlog` (*code*, *suggest=False*, *\*\*kws*)

Retrieve all errors and warnings generated through the code translation and summarize them into a string. Each entry uses four lines. For example:

```

Error in class Var on line 1:
function f(x)
      ^
unknown data type

```

First line indicate at what node and line-number the error occurred. The second and third prints the Matlab-code line in question with an indicator to where the code failed. The last line is the error or warning message generated.

#### Parameters

- **code** (*str*, *Builder*, *Node*) – Representation of the node tree.
- **suggest** (*bool*) – Use suggestion engine where appropriate.
- **\*\*kws** – Additional arguments passed to [Builder](#).

**Returns** A string representation of the log

**Return type** [str](#)

#### Example

```

>>> print mc.qlog("function f(x); x=4")
Error in class Var on line 1:
function f(x); x=4
      ^
unknown data type

Error in class Var on line 1:
function f(x); x=4
      ^
unknown data type

```

**See also:** [error\(\)](#), [warning\(\)](#)

`matlab2cpp.qscript` (*code*, *suggest=True*, *ftypes={}*, *\*\*kws*)

Perform a full translation (like [qcpp\(\)](#) and [qhpp\(\)](#)), but only focus on the object of interest. If for example code is provided, then only the code part of the translation will be include, without any wrappers. It will be as

close to a one-to-one translation as you can get. If a node tree is provided, current node position will be source of translation.

**Parameters**

- **code** (*str*, *Builder*, *Node*) – Representation of the node tree.
- **suggest** (*bool*) – Use suggestion engine where appropriate.
- **\*\*kws** – Additional arguments passed to *Builder*.

**Returns** A code translation in C++.

**Return type** *str*

**Example**

```
>>> print mc.qscript("a = 4")
a = 4 ;
```

matlab2cpp.**qtree** (*code*, *suggest=False*, *core=False*)

Summarize the node tree with relevant information, where each line represents a node. Each line will typically look as follows:

1	10				Var	unknown	TYPE	y
---	----	--	--	--	-----	---------	------	---

The line can be interpreted as follows:

Column	Description	Object
1	Matlab code line number	line
2	Matlab code cursor number	cur
3	The node categorization type	cls
4	The rule used for translation	backend
5	The data type of the node	type
6	Name of the node (if any)	name

The vertical bars represents branches. The right most bar on each line points upwards towards its node parent.

**Parameters**

- **code** (*str*, *Builder*, *Node*) – Representation of the node tree.
- **suggest** (*bool*) – Use suggestion engine where appropriate.
- **core** (*bool*) – Unly display nodes generated from Matlab code directly.
- **\*\*kws** – Additional arguments passed to *Builder*.

**Returns** A summary of the node tree.

**Return type** *str*

**Example**

```
>>> print mc.qtree("function y=f(x); y=x+4")
Program      program      TYPE      unnamed
| Includes   program      TYPE
| | Include  program      TYPE      #include <armadillo>
| | Include  program      TYPE      using namespace arma ;
1  1| Funcs   program      TYPE      unnamed
```

```

1  1| | Func      func_return TYPE  f
1  1| | | Declares func_return TYPE
1  1| | | | Var    unknown    TYPE  y
1  1| | | | Returns func_return TYPE
1 10| | | | Var    unknown    TYPE  y
1 13| | | | Params func_return TYPE
1 14| | | | Var    unknown    TYPE  x
1 16| | | | Block  code_block  TYPE
1 18| | | | Assign  expression  TYPE
1 18| | | | | Var    unknown    TYPE  y
1 20| | | | | Plus   expression  TYPE
1 20| | | | | Var    unknown    TYPE  x
1 22| | | | | Int    int         int
    | Inlines      program      TYPE  unnamed
    | Structs      program      TYPE  unnamed
    | Headers      program      TYPE  unnamed
    | | Header     program      TYPE  f
    | Log          program      TYPE  unnamed
    | | Error      program      TYPE  Var:0
    | | Error      program      TYPE  Var:9
    | | Error      program      TYPE  Var:13
    | | Error      program      TYPE  Var:17
    | | Error      program      TYPE  Var:19
    | | Error      program      TYPE  Plus:19

```

See also:

[\*matlab2cpp.tree\*](#), [\*matlab2cpp.node\*](#)

## 2.3 The tree constructor

Parsing of Matlab code is solely done through the [\*Builder\*](#) class. It contains three main use methods: [\*load\(\)\*](#), [\*configure\(\)\*](#) and [\*translate\(\)\*](#). In addition there are a collection of method with names starting with [\*create\\_\*](#) that creates various structures of the node tree.

In addition to [\*Builder\*](#) there are submodules with support function for modules. Constructor help functions are as follows:

Module	Description
<a href="#"><i>matlab2cpp.tree.assign</i></a>	Support functions for variable assignments
<a href="#"><i>matlab2cpp.tree.branches</i></a>	Support functions for if-tests, loops, try-blocks
<a href="#"><i>matlab2cpp.tree.codeblocks</i></a>	Support functions for filling in codeblock content
<a href="#"><i>matlab2cpp.tree.expressions</i></a>	Support functions for filling in expression content
<a href="#"><i>matlab2cpp.tree.functions</i></a>	Support functions for constructing Functions, both explicit and lambda, and program content
<a href="#"><i>matlab2cpp.tree.misc</i></a>	Miscelenious support functions
<a href="#"><i>matlab2cpp.tree.variables</i></a>	Support functions for constructing various variables

In addition a collectio of genereal purpose modules are available:

Module	Description
<a href="#"><i>matlab2cpp.tree.constants</i></a>	A collection of usefull constants used by various interpretation rules
<a href="#"><i>matlab2cpp.tree.findend</i></a>	Look-ahead functions for finding the end of various code structures
<a href="#"><i>matlab2cpp.tree.identify</i></a>	Look-ahead functions for identifying ambiguous contexts
<a href="#"><i>matlab2cpp.tree.iterate</i></a>	Support functions for segmentation of lists

### 2.3.1 The Builder class

**class** matlab2cpp.**Builder** (*disp=False, comments=True, original=False, enable\_omp=False, enable\_tbb=False, \*\*kws*)

Convert Matlab-code to a tree of nodes.

Given that one or more Matlab programs are loaded, each one can be accessed through indexing the Builder instance. For example:

```
>>> builder = mc.Builder()
>>> builder.load("prg1.m", "function y=prg1(x); y=x")
>>> builder.load("prg2.m", "prg1(4) ")
>>> builder.configure(suggest=True)
>>> builder.translate()
>>> prg1, prg2 = builder
>>> print prg1.cls, prg1.name
Program prg1.m
>>> print prg2.cls, prg2.name
Program prg2.m
```

Programs that are loaded, configured and translated, can be converted into C++ code through the front end functions in *matlab2cpp.qfunctions*:

```
>>> print mc.qhpp(prg1)
#include <armadillo>
using namespace arma ;

int prg1(int x)
{
    int y ;
    y = x ;
    return y ;
}
>>> print mc.qcpp(prg2)
#include <armadillo>
using namespace arma ;

int main(int argc, char** argv)
{
    prg1(4) ;
    return 0 ;
}
```

**\_\_getitem\_\_** (*index*)

Get root node for a program through indexing

`builder[index] <=> Builder.__getitem__(builder, index)`

**Parameters** *index* (*int*) – Loaded order

#### Example

```
>>> builder = mc.Builder()
>>> builder.load("prg1.m", "function y=prg1(x); y=x")
>>> builder.load("prg2.m", "prg1(4) ")
>>> prg1 = builder[0]
>>> prg2 = builder[1]
```



`__init__` (*disp=False*, *comments=True*, *original=False*, *enable\_omp=False*, *enable\_tbb=False*, *\*\*kws*)

#### Parameters

- **disp** (*bool*) – Verbose output while loading code
- **comments** (*bool*) – Include comments in the code interpretation
- **\*\*kws** – Optional arguments are passed to `matlab2cpp.rules`

`__str__` ()

Summary of all node trees

Same as `matlab2cpp.Node.summary()`, but for the whole project.

`str(builder) <=> Builder.__str__(builder)`

#### Example

```
>>> builder = mc.Builder()
>>> print builder
Project      unknown      TYPE
```

#### See also:

`matlab2cpp.Node.summary()`

`__weakref__`

list of weak references to the object (if defined)

**configure** (*suggest=True*, *\*\*kws*)

Configure node tree with datatypes.

**Parameters** **suggest** (*bool*) – Uses suggestion engine to fill in types

#### Example

```
>>> builder = mc.Builder()
>>> builder.load("unnamed.m", "a=1; b=2.; c='c'")
>>> print builder
Project      unknown      TYPE
| Program    unknown      TYPE      unnamed.m
| | Includes  unknown      TYPE
1 1| | Funcs   unknown      TYPE      unnamed.m
1 1| | | Main   unknown      TYPE      main
1 1| | | | Declares  unknown      TYPE
1 1| | | | | Var      unknown      TYPE      a
1 1| | | | | Var      unknown      TYPE      b
1 1| | | | | Var      unknown      TYPE      c
1 1| | | | Returns  unknown      TYPE
1 1| | | | Params   unknown      TYPE
1 1| | | | Block    unknown      TYPE
1 1| | | | | Assign   unknown      TYPE
1 1| | | | | | Var      unknown      TYPE      a
1 3| | | | | | Int      unknown      TYPE
1 6| | | | | Assign   unknown      TYPE
1 6| | | | | | Var      unknown      TYPE      b
1 8| | | | | | Float    unknown      TYPE
1 12| | | | | Assign   unknown      TYPE
```

```

1 12| | | | | Var      unknown    TYPE      c
1 14| | | | | String   unknown    TYPE
      | | Inlines      unknown    TYPE      unnamed.m
      | | Structs      unknown    TYPE      unnamed.m
      | | Headers      unknown    TYPE      unnamed.m
      | | Log          unknown    TYPE      unnamed.m
>>> builder.configure(suggest=True)
>>> print builder
      Project      program      TYPE
      | Program    program      TYPE      unnamed.m
      | | Includes  program      TYPE
1 1| | | Funcs      program      TYPE      unnamed.m
1 1| | | Main       func_return  TYPE      main
1 1| | | | Declares func_return  TYPE
1 1| | | | Var      int          int      a
1 1| | | | Var      double       double   b
1 1| | | | Var      string       string   c
1 1| | | | Returns  func_return  TYPE
1 1| | | | Params   func_return  TYPE
1 1| | | | Block    code_block   TYPE
1 1| | | | | Assign  int          int
1 1| | | | | Var     int          int      a
1 3| | | | | Int     int          int
1 6| | | | | Assign  double       double
1 6| | | | | Var     double       double   b
1 8| | | | | Float   double       double
1 12| | | | | Assign  string       string
1 12| | | | | Var     string       string   c
1 14| | | | | String  string       string
      | | Inlines      program      TYPE      unnamed.m
      | | Structs      program      TYPE      unnamed.m
      | | Headers      program      TYPE      unnamed.m
      | | Log          program      TYPE      unnamed.m

```

**Raises** RuntimeError – Method can only be run once.

**create\_assign** (*parent, cur, eq\_loc*)

Create assignment with single return

Structure:

Assign

| <return var>

| Get/Var

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where assignment is identified
- **eq\_loc** (*int*) – position of assignment equal sign

**Returns** position where assignment ends

**Return type** int

**See also:**

`matlab2cpp.tree.assign.single()`

**create\_assign\_variable** (*parent, cur, end=None*)

Create right-hand-side variable (Expression)

**Structure:**

Cset|Cvar|Fset|Fvar|Nset|Var|Set|Sset|Svar  
| <list of expression>?

**Parameters**

- **parent** (*Node*) – Reference to parent node
- **cur** (*int*) – position where variable is identified
- **end** (*int, optional*) – position where variable ends

**Returns** position where variable ends

**Return type** int

**See also:**

`matlab2cpp.tree.variables.assign()`

**create\_assigns** (*parent, cur, eq\_loc*)

Create assignment with multiple returns

**Structure:**

Assigns  
| <list of return vars>  
| Get|Var

**Parameters**

- **parent** (*Block*) – Reference to parent node
- **cur** (*int*) – position where assignments is identified
- **eq\_loc** (*int*) – position of assignment equal sign

**Returns** position where assignments ends

**Return type** int

**See also:**

`matlab2cpp.tree.assign.multi()`

**create\_cell** (*parent, cur*)

Create cell-structure (expression)

**Structure:**

Cell  
| <expression>+

**Parameters**

- **parent** ([Node](#)) – Reference to parent node
- **cur** (*int*) – position where cell is identified

**Returns** position where cell ends

**Return type** int

**See also:**

`matlab2cpp.tree.misc.cell()`

**create\_codeblock** (*parent, cur*)

Create codeblock Block

Structure:

Assign|Assigns|Bcomment|Ecomment|Lcomment|Statement

*Statements* are handled locally and evoces <expression>

Legal parents: Case, Catch, Elif, Else, For, Func, If, Main, Otherwise, Switch, Try, While

**Parameters**

- **parent** ([Node](#)) – Reference to parent node
- **cur** (*int*) – position where codeblock is identified

**Returns** position where codeblock ends

**Return type** int

**See also:**

`matlab2cpp.tree.codeblock.codeblock()`

**create\_comment** (*parent, cur*)

Create comment

Structure:

Bcomment|Ecomment|Lcomment

**Parameters**

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where comment is identified

**Returns** position where comment ends

**Return type** int

**See also:**

`matlab2cpp.tree.misc.comment()`

**create\_expression** (*parent, cur, end=None*)

Create expression

Main engine for creating expression.

#### Parameters

- **parent** ([Node](#)) – Reference to parent node
- **cur** (*int*) – position where expression is identified
- **end** (*int, optional*) – position where expression ends

**Returns** position where expression ends

**Return type** int

**See also:**

`matlab2cpp.tree.expression.create()`

**create\_for** (*parent, cur*)

Create For-loop

Structure:

```
For
| <loop variable>
| <loop expression>
| <code block>
```

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where for-loop is identified

**Returns** position where for-loop ends

**Return type** int

**See also:**

`matlab2cpp.tree.branches.forloop()`

**create\_function** (*parent, cur*)

Create function (not main)

Structure:

```
Func
| Declares
| Returns
| Params
| <code block>
```

#### Parameters

- **parent** ([Funcs](#)) – Reference to parent node
- **cur** (*int*) – position where function is identified

**Returns** position where function ends

**Return type** int

**See also:**

`matlab2cpp.tree.functions.function()`

**create\_if** (*parent*, *cur*)

Create if-branch

Structure (main):

```
Branch
| If
| | <cond expression>
| | <code block>
| <else if> *
| <else> ?
```

Structure (else if):

```
Elif
| <cond expression>
| <code block>
```

Structure (else):

```
Else
| <code block>
```

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where if-branch is identified

**Returns** position where if-branch ends

**Return type** int

**See also:**

`matlab2cpp.tree.branches.ifbranch()`

**create\_lambda\_assign** (*parent*, *cur*, *eq\_loc*)

Create assignments involving lambda functions

Structure:

```
Assign
| <assign variable>
| <lambda function>
```

#### Parameters

- **parent** ([Block](#)) – Reference to parent node

- **cur** (*int*) – position where Lambda assignment is identified
- **eq\_loc** (*int*) – position of assignment equal sign

**Returns** position where Lambda assignment ends

**Return type** int

**See also:**

`matlab2cpp.tree.functions.lambda_assign()`

**create\_lambda\_func** (*parent, cur*)

Create lambda function

Structure (function part):

```
Func
| Declares
| Returns
|| Var (_retval)
| Params
| Block
|| Assign
||| Var (_retval)
||| <expression>
```

Structure (lambda part):

```
Lambda
```

#### Parameters

- **parent** (*Assign*) – Reference to parent node
- **cur** (*int*) – position where Lambda function is identified

**Returns** position where Lambda function ends

**Return type** int

**See also:**

`matlab2cpp.tree.functions.lambda_func()`

**create\_list** (*parent, cur*)

Create list of expressions

Structure:

```
<expression>*
```

#### Parameters

- **parent** (*Node*) – Reference to parent node
- **cur** (*int*) – position where list is identified

**Returns** position where list ends

**Return type** int

**See also:**

`matlab2cpp.tree.misc.list()`

**create\_main** (*parent*, *cur*)

Create main function

Structure:

Main  
| Declares  
| Returns  
| Params  
| <code block>

**Parameters**

- **parent** ([Funcs](#)) – Reference to parent node
- **cur** (*int*) – position where main function is identified

**Returns** position where main function ends

**Return type** int

**See also:**

`matlab2cpp.tree.functions.main()`

**create\_matrix** (*parent*, *cur*)

Create matrix (Expression)

Structure (main):

Matrix  
| <vector>\*

Structure (vector):

Vector  
| <expression>\*

**Parameters**

- **parent** ([Node](#)) – Reference to parent node
- **cur** (*int*) – position where matrix is identified

**Returns** position where matrix ends

**Return type** int

**See also:**

`matlab2cpp.tree.misc.matrix()`



**create\_number** (*parent, cur*)  
Create number (Expression)

Structure:

Int|Float|Imag

#### Parameters

- **parent** ([Node](#)) – Reference to parent node
- **cur** (*int*) – position where number is identified

**Returns** position where number ends

**Return type** int

**See also:**

`matlab2cpp.tree.misc.number()`

**create\_parfor** (*parent, cur*)  
Create parfor-loop

Structure:

Parfor  
| <loop variable>  
| <loop expression>  
| <code block>

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where for-loop is identified

**Returns** position where for-loop ends

**Return type** int

**create\_program** (*name*)  
Create program meta variables and initiates to fill them

Structure:

Program  
| Includes  
| Funcs  
| Inlines  
| Structs  
| Headers  
| Log

**Parameters** **name** (*str*) – filename of program

**Returns** position in program when scanning is complete.

**Return type** int

**See also:**

`matlab2cpp.tree.functions.program()`

**create\_reserved** (*parent, cur*)

Create Matlab reserved keywords.

Some words like “hold”, “grid” and “clear”, behaves differently than regular Matlab. They take arguments after space, not in parenthesis.

Structure (main):

Get

| <string>\*

Structure (string):

String

**Parameters**

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where reserved statement is identified

**Returns** position where reserved statement ends

**Return type** int

**See also:**

`matlab2cpp.tree.misc.reserved()`

**create\_string** (*parent, cur*)

Create string (Expression)

Structure:

String

**Parameters**

- **parent** ([Node](#)) – Reference to parent node
- **cur** (*int*) – position where string is identified

**Returns** position where string ends

**Return type** int

**See also:**

`matlab2cpp.tree.misc.string()`

**create\_switch** (*parent, cur*)

Create switch-branch

Structure (main):

```
Switch
| <cond expression>
| <case>+
| <otherwise>?
```

Structure (case):

```
Case
| <cond expression>
| <code block>
```

Structure (otherwise):

```
Otherwise
| <code block>
```

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where switch is identified

**Returns** position where switch ends

**Return type** int

**See also:**

*matlab2cpp.tree.branches.switch()*

**create\_try** (*parent, cur*)

Create try-block

Structure:

```
Tryblock
| Try
| | <code block>
| Catch
| | <code block>
```

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where try-block is identified

**Returns** position where try-block ends

**Return type** int

**See also:**

*matlab2cpp.tree.branches.trybranch()*

**create\_variable** (*parent, cur*)

Create left-hand-side variable (Expression)

Structure:

Cget|Cvar|Fget|Fvar|Get|Nget|Var|Sget|Svar  
| <list of expression>?

#### Parameters

- **parent** ([Node](#)) – Reference to parent node
- **cur** (*int*) – position where variable is identified

**Returns** position where variable ends

**Return type** int

**See also:**

*matlab2cpp.tree.variables.variable()*

**create\_verbatim** (*parent, cur*)

Create verbatim translation

A manual overrides switch provided by the user to perform translations.

Structure:

Verbatim

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where verbatim is identified

**Returns** position where verbatim ends

**Return type** int

**See also:**

*matlab2cpp.tree.misc.verbatim()*

**create\_while** (*parent, cur*)

Create while-loop

Structure:

While  
| <cond expression>  
| <code block>

#### Parameters

- **parent** ([Block](#)) – Reference to parent node
- **cur** (*int*) – position where while-loop is identified

**Returns** position where while-loop ends

**Return type** int

See also:

`matlab2cpp.tree.branches.whileloop()`

**get\_unknowns** (*index=-1*)

Get unknown variables and function calls names in a program.

**Parameters** *index* (*int, str*) – Either loading index or the name of the program.

**Returns** strings of the names of the unknown variables and calls.

**Return type** *list*

### Example

```
>>> builder = Builder(); builder.load("prg.m", "a;b;c")
>>> print builder.get_unknowns()
['a', 'c', 'b']
```

**load** (*name, code*)

Load a Matlab code into the node tree.

The code is inserted into the attribute *self.code* and initiate the `matlab2cpp.Builder.create_program()`, which evokes various other `create_*` methods. Each method creates nodes and/or pushes the job over to other create methods.

**Parameters**

- **name** (*str*) – Name of program (usually valid filename).
- **code** (*str*) – Matlab code to be loaded

**Raises** `SyntaxError` – Error in the Matlab code.

### Example

```
>>> builder = mc.Builder()
>>> builder.load("unnamed.m", "")
>>> print builder
Project      unknown      TYPE
| Program    unknown      TYPE    unnamed.m
| | Includes  unknown      TYPE
1 1| | Funcs   unknown      TYPE    unnamed.m
| | Inlines   unknown      TYPE    unnamed.m
| | Structs   unknown      TYPE    unnamed.m
| | Headers   unknown      TYPE    unnamed.m
| | Log        unknown      TYPE    unnamed.m
```

**syntaxerror** (*cur, text*)

Raise an `SyntaxError` related to the Matlab code. Called from various `create_*` methods when code is invalid.

**Parameters**

- **cur** (*int*) – Current location in the Matlab code
- **text** (*str*) – The related rational presented to the user

**Raises** `SyntaxError` – Error in the Matlab code.

### Example

```
>>> builder = mc.Builder()
>>> prg = builder.load("unnamed.m", "0123456789")
>>> builder.syntaxerror(7, "example of error")
Traceback (most recent call last):
...
SyntaxError: line 1 in Matlab code:
0123456789
    ^
Expected: example of error
```

#### **translate()**

Perform translation on all nodes in all programs in builder. Also runs configure if not done already.

#### **See also:**

*matlab2cpp.rules*

## 2.3.2 Assignment constructors

Support functions for identifying assignments.

Function	Description
<i>single()</i>	Assignment with single return
<i>multi()</i>	Assignment with multiple returns

`matlab2cpp.tree.assign.multi(self, parent, cur, eq_loc)`

Assignment with multiple return

#### **Parameters**

- **self** (**Builder**) – Code constructor.
- **parent** (**Node**) – Parent node
- **cur** (*int*) – Current position in code
- **eq\_loc** (*int*) – position of the assignment marker ('='-sign)

**Returns** Index to end of assignment

**Return type** `int`

### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "[a,b] = c")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock   codeblock.codeblock
0 Assigns     assign.multi      '[a,b] = c'
1 Var         variables.assign  'a'
3 Var         variables.assign  'b'
8 Expression  expression.create 'c'
8 Var         variables.variable 'c'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
```

```

1 1Block      code_block  TYPE
1 1| Assigns  unknown    TYPE  c
1 2| | Var    unknown    TYPE  a
1 4| | Var    unknown    TYPE  b
1 9| | Var    unknown    TYPE  c

```

`matlab2cpp.tree.assign.single(self, parent, cur, eq_loc)`

Assignment with single return.

#### Parameters

- **self** ([Builder](#)) – Code constructor
- **parent** ([Node](#)) – Parent node
- **cur** (*int*) – Current position in code
- **eq\_loc** (*int*) – position of the assignment marker ('='-sign)

**Returns** Index to end of assignment

**Return type** `int`

#### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a=b")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock   codeblock.codeblock
0 Assign      assign.single      'a=b'
0 Var         variables.assign   'a'
2 Expression  expression.create  'b'
2 Var         variables.variable 'b'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 1Block      code_block  TYPE
1 1| Assigns  unknown    TYPE  b
1 1| | Var    unknown    TYPE  a
1 3| | Var    unknown    TYPE  b

```

## 2.3.3 Loop and branch constructors

Interpretors related to branches, loops and try.

Function	Description
<a href="#"><code>trybranch()</code></a>	Try-catch block
<a href="#"><code>switch()</code></a>	Switch-case branch
<a href="#"><code>whileloop()</code></a>	While loop
<a href="#"><code>forloop()</code></a>	For loop
<a href="#"><code>ifbranch()</code></a>	If-iffelse-else branch

`matlab2cpp.tree.branches.forloop(self, parent, cur)`

For loop

#### Parameters

- **self** (*Builder*) – Code constructor.
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of code block

**Return type** int

### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed",
...   """for a = b
...   c
...   end""")
loading unnamed
  Program      functions.program
  0 Main        functions.main
  0 Codeblock   codeblock.codeblock
  0   For       'for a = b' branches.forloop
  4   Var       variables.variable 'a'
  8   Expression expression.create 'b'
  8   Var       variables.variable 'b'
 12 Codeblock   codeblock.codeblock
 12  Statement  codeblock.codeblock 'c'
 12  Expression expression.create 'c'
 12  Var        variables.variable 'c'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 1Block      code_block  TYPE
1 1| For      code_block  TYPE
1 5| | Var    unknown     (int)  a
1 9| | Var    unknown     TYPE    b
2 13| | Block  code_block  TYPE
2 13| | | Statement code_block TYPE
2 13| | | | Var    unknown     TYPE    c
```

matlab2cpp.tree.branches.**ifbranch** (*self*, *parent*, *start*)

If-iffelse-else branch

### Parameters

- **self** (*Builder*) – Code constructor.
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of code block

**Return type** int

### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed",
...   """if a
```



```

...     b
... elseif c
...     d
... end"")
loading unnamed
  Program      functions.program
  0 Main       functions.main
  0 Codeblock  codeblock.codeblock
  0   If       branches.ifbranch  'if a'
  3   Expression expression.create  'a'
  3   Var      variables.variable  'a'
  4 Codeblock  codeblock.codeblock
  7   Statement codeblock.codeblock  'b'
  7   Expression expression.create  'b'
  7   Var      variables.variable  'b'
  9   Else if  branches.ifbranch  'elseif c'
  16 Expression expression.create  'c'
  16 Var      variables.variable  'c'
  17 Codeblock  codeblock.codeblock
  20 Statement codeblock.codeblock  'd'
  20 Expression expression.create  'd'
  20 Var      variables.variable  'd'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Branch   code_block  TYPE
1 4| | If     code_block  TYPE
1 4| | | Var  unknown     TYPE   a
1 5| | | Block code_block  TYPE
2 8| | | | Statement code_block  TYPE
2 8| | | | | Var  unknown     TYPE   b
3 10| | Elif   code_block  TYPE
3 17| | | Var  unknown     TYPE   c
3 18| | | Block code_block  TYPE
4 21| | | | Statement code_block  TYPE
4 21| | | | | Var  unknown     TYPE   d

```

matlab2cpp.tree.branches.**switch** (*self*, *parent*, *cur*)

Switch-case branch

#### Parameters

- **self** (*Builder*) – Code constructor.
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of codeblock

**Return type** int

#### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed",
...  """switch a
...  case b
...  c

```

```

... case d
... d
... end"")
loading unnamed
  Program      functions.program
  0 Main        functions.main
  0 Codeblock   codeblock.codeblock
  0 Switch      branches.switch      'switch a'
  7 Expression  expression.create    'a'
  7 Var         variables.variable   'a'
  9 Case        branches.switch      'case b'
  14 Expression expression.create    'b'
  14 Var        variables.variable   'b'
  18 Codeblock   codeblock.codeblock
  18 Statement   codeblock.codeblock 'c'
  18 Expression  expression.create    'c'
  18 Var         variables.variable   'c'
  20 Case        branches.switch      'case d'
  25 Expression  expression.create    'd'
  25 Var         variables.variable   'd'
  29 Codeblock   codeblock.codeblock
  29 Statement   codeblock.codeblock 'd'
  29 Expression  expression.create    'd'
  29 Var         variables.variable   'd'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 1Block      code_block  TYPE
1 1| Switch    code_block  TYPE
1 8| | Var     unknown    TYPE    a
2 10| | Case   code_block  TYPE
2 15| | | Var  unknown    TYPE    b
3 19| | | Block code_block  TYPE
3 19| | | | Statement code_block TYPE
3 19| | | | | Var  unknown    TYPE    c
4 21| | Case   code_block  TYPE
4 26| | | Var  unknown    TYPE    d
5 30| | | Block code_block  TYPE
5 30| | | | Statement code_block TYPE
5 30| | | | | Var  unknown    TYPE    d

```

matlab2cpp.tree.branches.**trybranch**(*self*, *parent*, *cur*)

Try-catch block

#### Parameters

- **self** (**Builder**) – Code constructor.
- **parent** (**Node**) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of block

**Return type** int

### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed",
...   """try
...   a
... catch
...   b""")
loading unnamed
  Program      functions.program
  0 Main        functions.main
  0 Codeblock   codeblock.codeblock
  0 Try         branches.trybranch 'try'
  6 Codeblock   codeblock.codeblock
  6 Statement   codeblock.codeblock 'a'
  6 Expression  expression.create 'a'
  6 Var         variables.variable 'a'
 16 Codeblock   codeblock.codeblock
 16 Statement   codeblock.codeblock 'b'
 16 Expression  expression.create 'b'
 16 Var         variables.variable 'b'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 1Block      code_block  TYPE
1 1| Tryblock  code_block  TYPE
1 1| | Try     code_block  TYPE
2 7| | | Block code_block  TYPE
2 7| | | | Statement code_block TYPE
2 7| | | | | Var      unknown  TYPE    a
3 9| | Catch   code_block  TYPE
4 17| | | Block code_block  TYPE
4 17| | | | Statement code_block TYPE
4 17| | | | | Var      unknown  TYPE    b

```

matlab2cpp.tree.branches.**whileloop**(*self*, *parent*, *cur*)

While loop

#### Parameters

- **self** (**Builder**) – Code constructor.
- **parent** (**Node**) – Parent node
- **cur** (**int**) – Current position in code

**Returns** Index to end of code block

**Return type** int

### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed",
...   """while a
...   b
... end""")
loading unnamed
  Program      functions.program
  0 Main        functions.main

```

```

0 Codeblock    codeblock.codeblock
0   While      branches.whileloop   'while a'
6   Expression expression.create     'a'
6   Var        variables.variable    'a'
10 Codeblock   codeblock.codeblock
10  Statement  codeblock.codeblock   'b'
10  Expression expression.create     'b'
10  Var        variables.variable    'b'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 1Block      code_block    TYPE
1 1| While    code_block    TYPE
1 7| | Var    unknown       TYPE    a
2 11| | Block code_block    TYPE
2 11| | | Statement code_block TYPE
2 11| | | | Var    unknown   TYPE    b

```

### 2.3.4 Code block constructor

The main codeblock loop

matlab2cpp.tree.codeblock.**codeblock** (*self, parent, start*)  
If-elseif-else branch

#### Parameters

- **self** (*Builder*) – Code constructor
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of codeblock

**Return type** int

#### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a; 'b'; 3")
loading unnamed
  Program      functions.program
0 Main         functions.main
0 Codeblock    codeblock.codeblock
0  Statement   codeblock.codeblock 'a'
0  Expression  expression.create    'a'
0  Var         variables.variable    'a'
3  Statement   codeblock.codeblock "'b'"
3  String      misc.string           "'b'"
8  Statement   codeblock.codeblock '3'
8  Expression  expression.create     '3'
8  Int         misc.number            '3'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 1Block      code_block    TYPE
1 1| Statement code_block    TYPE
1 1| | Var    unknown       TYPE    a

```

1	4	Statement	code_block	TYPE
1	4	String	string	string
1	9	Statement	code_block	TYPE
1	9	Int	int	int

### 2.3.5 Expression constructor

Expression interpreter

matlab2cpp.tree.expression.**create** (*self, node, start, end=None, start\_opr=None*)

Create expression in three steps:

1. In order, split into sub-expressions for each dividing operator
2. Address prefixes, postfixes, parentheses, etc.
3. Identify the remaining singleton

#### Parameters

- **self** ([Builder](#)) – Code constructor.
- **node** ([Node](#)) – Reference to the parent node
- **start** (*int*) – current position in code
- **end** (*int, optional*) – end of expression. Required for space-delimited expression.
- **start\_opr** (*str, optional*) – At which operator the recursive process is. (For internal use)

**Returns** index to end of the expression

**Return type** int

#### Examples

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a*b+c/d")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock    codeblock.codeblock
0 Statement    codeblock.codeblock 'a*b+c/d'
0 Expression   expression.create    'a*b+c/d'
0 Expression   expression.create    'a*b'
0 Expression   expression.create    'a'
0 Var          variables.variable   'a'
2 Expression   expression.create    'b'
2 Var          variables.variable   'b'
4 Expression   expression.create    'c/d'
4 Expression   expression.create    'c'
4 Var          variables.variable   'c'
6 Expression   expression.create    'd'
6 Var          variables.variable   'd'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 1Block      code_block    TYPE
1 1| Statement code_block    TYPE
1 1| | Plus    expression    TYPE
```

1	1				Mul	expression	TYPE	
1	1				Var	unknown	TYPE	a
1	3				Var	unknown	TYPE	b
1	5				Matrixdivision	expression	TYPE	
1	5				Var	unknown	TYPE	c
1	7				Var	unknown	TYPE	d

matlab2cpp.tree.expression.**retrieve\_operator**(self, opr)

Retrieve operator class by string

**Parameters** **opr** (str) – operator string

**Returns** class of corresponding operator

**Return type** *Node*

## 2.3.6 Function constructors

Functions, programs and meta-nodes

Functions	Description
<code>program()</code>	Program outer shell
<code>function()</code>	Explicit functions
<code>main()</code>	Main script
<code>lambda_assign()</code>	Anonymous function assignment
<code>lambda_func()</code>	Anonymous function content

matlab2cpp.tree.functions.**function**(self, parent, cur)

Explicit functions

**Parameters**

- **self** (*Builder*) – Code constructor
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of function

**Return type** int

### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "function f(); end")
loading unnamed
  Program      functions.program
  0 Function    functions.function  'function f()'
  12 Codeblock  codeblock.codeblock
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 1Funcs      program      TYPE      unnamed
1 1| Func      func_returns TYPE      f
1 1| | Declares func_returns TYPE
1 1| | Returns  func_returns TYPE
1 11| | Params  func_returns TYPE
1 13| | Block   code_block  TYPE
```

matlab2cpp.tree.functions.lambda\_assign(self, node, cur, eq\_loc)

Anonymous function constructor

#### Parameters

- **self** (**Builder**) – Code constructor
- **parent** (**Node**) – Parent node
- **cur** (**int**) – Current position in code
- **eq\_loc** (**int**) – location of assignment sign ('=')

**Returns** Index to end of function line

**Return type** int

#### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "f = @(x) 2*x")
loading unnamed
  Program      functions.program
  0 Main        functions.main
  0 Codeblock   codeblock.codeblock
  0 Assign      'f = @(x) 2*x' functions.lambda_assign
  0 Var         variables.assign      'f'
  4 Lambda      functions.lambda_func '@(x) 2*x'
  6 Expression  expression.create     'x'
  6 Var         variables.variable    'x'
  9 Expression  expression.create     '2*x'
  9 Expression  expression.create     '2'
  9 Int         misc.number           '2'
 11 Expression  expression.create     'x'
 11 Var         variables.variable    'x'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder)
  Program      program      TYPE      unnamed
  | Includes   program      TYPE
  1 1| Funcs   program      TYPE      unnamed
  1 1| | Main   func_return  TYPE      main
  1 1| | | Declares func_return TYPE
  1 1| | | | Var   func_lambda TYPE      f
  1 1| | | Returns func_return TYPE
  1 1| | | Params  func_return TYPE
  1 1| | | Block   code_block  TYPE
  1 1| | | | Assign func_lambda func_lambda
  1 1| | | | | Var   func_lambda TYPE      f
  1 1| | | | | Lambda func_lambda func_lambda_f
  1 5| | Func    func_lambda TYPE      _f
  1 5| | | Declares func_lambda TYPE
  1 5| | | | Var    unknown    TYPE      _retval
  1 5| | | Returns func_lambda TYPE
  1 5| | | | Var    unknown    TYPE      _retval
  1 5| | | Params  func_lambda TYPE
  1 7| | | | Var    unknown    TYPE      x
  1 5| | | Block   code_block  TYPE
  1 5| | | | Assign expression  TYPE
  1 5| | | | | Var    unknown    TYPE      _retval
  1 10| | | | | Mul   expression  TYPE
```

1	10						Int	int	int	
1	12						Var	unknown	TYPE	x
			Inlines				program	TYPE	unnamed	
			Structs				program	TYPE	unnamed	
			Headers				program	TYPE	unnamed	
			Log				program	TYPE	unnamed	

matlab2cpp.tree.functions.**lambda\_func**(*self, node, cur*)  
 Anonymous function content. Support function of *lambda\_assign*.

#### Parameters

- **self** (*Builder*) – Code constructor
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of function line

**Return type** int

matlab2cpp.tree.functions.**main**(*self, parent, cur*)

Main script

#### Parameters

- **self** (*Builder*) – Code constructor
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** Index to end of script

**Return type** int

#### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock   codeblock.codeblock
0 Statement   codeblock.codeblock 'a'
0 Expression  expression.create   'a'
0 Var         variables.variable  'a'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder)
  Program      program      TYPE      unnamed
  | Includes   program      TYPE
1 1| Funcs     program      TYPE      unnamed
1 1| | Main    func_return  TYPE      main
1 1| | | Declares func_return TYPE
1 1| | | Returns func_return TYPE
1 1| | | Params  func_return TYPE
1 1| | | Block   code_block  TYPE
1 1| | | | Statement code_block TYPE
1 1| | | | | Var   unknown    TYPE      a
  | Inlines    program      TYPE      unnamed
```



Structs	program	TYPE	unnamed
Headers	program	TYPE	unnamed
Log	program	TYPE	unnamed

matlab2cpp.tree.functions.**program**(*self*, *name*)

The outer shell of the program

#### Parameters

- **self** (*Builder*) – Code constructor
- **name** (*str*) – Name of the program

**Returns** The root node of the constructed node tree

**Return type** *Node*

#### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a")
loading unnamed
  Program      functions.program
  0 Main       functions.main
  0 Codeblock  codeblock.codeblock
  0 Statement  codeblock.codeblock 'a'
  0 Expression expression.create   'a'
  0 Var        variables.variable  'a'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder)
  Program      program      TYPE      unnamed
  | Includes   program      TYPE
  1 1| Funcs   program      TYPE      unnamed
  1 1| | Main   func_return  TYPE      main
  1 1| | | Declares func_return TYPE
  1 1| | | Returns func_return TYPE
  1 1| | | Params func_return TYPE
  1 1| | | Block  code_block  TYPE
  1 1| | | | Statement code_block TYPE
  1 1| | | | | Var      unknown  TYPE      a
  | Inlines    program      TYPE      unnamed
  | Structs    program      TYPE      unnamed
  | Headers    program      TYPE      unnamed
  | Log        program      TYPE      unnamed
```

## 2.3.7 Miscelenious constructors

Interpretors that didn't fit other places

matlab2cpp.tree.misc.**cell**(*self*, *node*, *cur*)

Verbatim cells

#### Parameters

- **self** (*Builder*) – Code constructor
- **node** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** End of cell

**Return type** int

### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "{1, 2}")
loading unnamed
    Program      functions.program
0 Main          functions.main
0 Codeblock     codeblock.codeblock
0 Statement     codeblock.codeblock '{1, 2}'
0 Expression    expression.create   '{1, 2}'
0 Cell          misc.cell           '{1, 2}'
1 Expression    expression.create   '1'
1 Int           misc.number         '1'
4 Expression    expression.create   '2'
4 Int           misc.number         '2'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Cell    cell       TYPE
1 2| | | Int   int        int
1 5| | | Int   int        int
```

matlab2cpp.tree.misc.**comment** (*self*, *parent*, *cur*)

Comments on any form

### Parameters

- **self** (*Builder*) – Code constructor
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** End of comment

**Return type** int

### Example

```
>>> builder = mc.Builder(True, comments=True)
>>> builder.load("unnamed", "4 % comment")
loading unnamed
    Program      functions.program
0 Main          functions.main
0 Codeblock     codeblock.codeblock
0 Statement     codeblock.codeblock '4'
0 Expression    expression.create   '4'
0 Int           misc.number         '4'
2 Comment       misc.comment        '% comment'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Statement code_block  TYPE
```

```

1 1| | Int      int      int
1 3| Ecomment  code_block TYPE

```

matlab2cpp.tree.misc.**list** (*self*, *parent*, *cur*)

A list (both comma or space delimited)

#### Parameters

- **self** (*Builder*) – Code constructor
- **parent** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** End of list

**Return type** int

#### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "[2 -3]")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock    codeblock.codeblock
0 Statement    codeblock.codeblock '[2 -3]'
0 Expression   expression.create    '[2 -3]'
0 Matrix       misc.matrix          '[2 -3]'
1 Vector       misc.matrix          '2 -3'
1 Expression   expression.create    '2'
1 Int          misc.number          '2'
3 Expression   expression.create    '-3'
4 Int          misc.number          '3'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Matrix  matrix      irowvec
1 2| | | Vector matrix      irowvec
1 2| | | | Int  int         int
1 4| | | | Neg  expression  int
1 5| | | | | Int  int         int

```

matlab2cpp.tree.misc.**matrix** (*self*, *node*, *cur*)

Verbatim matrices

#### Parameters

- **self** (*Builder*) – Code constructor
- **node** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** End of matrix

**Return type** int

**Example**

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "[[1 2] [3 4]]")
loading unnamed
    Program      functions.program
0 Main          functions.main
0 Codeblock     codeblock.codeblock
0 Statement     codeblock.codeblock  '[[1 2] [3 4]]'
0 Expression    expression.create     '[[1 2] [3 4]]'
0 Matrix        misc.matrix           '[[1 2] [3 4]]'
1 Vector        misc.matrix           '[1 2] [3 4]'
1 Expression    expression.create     '[1 2]'
1 Matrix        misc.matrix           '[1 2]'
2 Vector        misc.matrix           '1 2'
2 Expression    expression.create     '1'
2 Int           misc.number           '1'
4 Expression    expression.create     '2'
4 Int           misc.number           '2'
7 Expression    expression.create     '[3 4]'
7 Matrix        misc.matrix           '[3 4]'
8 Vector        misc.matrix           '3 4'
8 Expression    expression.create     '3'
8 Int           misc.number           '3'
10 Expression   expression.create     '4'
10 Int          misc.number           '4'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 1Block      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Matrix  matrix      irowvec
1 2| | | Vector matrix      irowvec
1 2| | | | Matrix matrix      irowvec
1 3| | | | | Vector matrix      irowvec
1 3| | | | | Int    int        int
1 5| | | | | Int    int        int
1 8| | | | Matrix  matrix      irowvec
1 9| | | | | Vector matrix      irowvec
1 9| | | | | Int    int        int
1 11| | | | | Int    int        int

```

matlab2cpp.tree.misc.**number** (*self*, *node*, *start*)

Verbatim number

**Parameters**

- **self** (*Builder*) – Code constructor
- **node** (*Node*) – Parent node
- **start** (*int*) – Current position in code

**Returns** End of number

**Return type** int

### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "42.")
loading unnamed
    Program      functions.program
  0 Main        functions.main
  0 Codeblock   codeblock.codeblock
  0 Statement   codeblock.codeblock  '42.'
  0 Expression  expression.create    '42.'
  0 Float       misc.number          '42.'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Float   double     double

```

matlab2cpp.tree.misc.**reserved**(*self*, *node*, *start*)

Reserved keywords

matlab2cpp.tree.misc.**string**(*self*, *parent*, *cur*)

Verbatim string

#### Parameters

- **self** (**Builder**) – Code constructor
- **parent** (**Node**) – Parent node
- **start** (*int*) – Current position in code

**Returns** End of string

**Return type** int

### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "'abc'")
loading unnamed
    Program      functions.program
  0 Main        functions.main
  0 Codeblock   codeblock.codeblock
  0 Statement   codeblock.codeblock  "'abc'"
  0 String      misc.string          "'abc'"
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | String  string      string

```

matlab2cpp.tree.misc.**verbatim**(*self*, *parent*, *cur*)

Verbatim, indicated by \_

#### Parameters

- **self** (**Builder**) – Code constructor
- **parent** (**Node**) – Parent node

- **cur** (*int*) – Current position in code

**Returns** End of verbatim

**Return type** int

## 2.3.8 Variable constructors

Variable interpreter

`matlab2cpp.tree.variables.assign(self, node, cur, end=None)`

Variable left side of an assignment

### Parameters

- **self** (*Builder*) – Code constructor
- **node** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Kwargs:** `end` (*int*, optional): End of variable

**Returns** End of variable

**Return type** int

### Example

```
>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a = 4")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock   codeblock.codeblock
0 Assign      assign.single      'a = 4'
0 Var         variables.assign   'a'
4 Expression  expression.create  '4'
4 Int         misc.number        '4'
>>> builder.configure(suggest=False)
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block      TYPE
1 | Assign    int             int
1 | | Var     unknown         (int)  a
1 | 5 | Int   int             int
```

`matlab2cpp.tree.variables.cell_arg(self, cset, cur)`

Argument of a cell call. Support function to *assign* and *variable*.

### Parameters

- **self** (*Builder*) – Code constructor
- **cset** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Returns** End of argument

**Return type** int

### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a{b}")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock   codeblock.codeblock
0 Statement   codeblock.codeblock 'a{b}'
0 Expression  expression.create    'a{b}'
0 Cvar        variables.variable   'a{b}'
2 Expression  expression.create    'b'
2 Var         variables.variable   'b'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Cvar    cell       TYPE    a
1 3| | | Var   unknown    TYPE    b

```

matlab2cpp.tree.variables.**variable** (*self, parent, cur*)

Variable not on the left side of an assignment

#### Parameters

- **self** (*Builder*) – Code constructor
- **node** (*Node*) – Parent node
- **cur** (*int*) – Current position in code

**Kwargs:** end (int, optional): End of variable

**Returns** End of variable

**Return type** int

### Example

```

>>> builder = mc.Builder(True)
>>> builder.load("unnamed", "a")
loading unnamed
  Program      functions.program
0 Main        functions.main
0 Codeblock   codeblock.codeblock
0 Statement   codeblock.codeblock 'a'
0 Expression  expression.create    'a'
0 Var         variables.variable   'a'
>>> builder.configure()
>>> print mc.qtree(builder, core=True)
1 lBlock      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Var     unknown    TYPE    a

```

## 2.3.9 Find-end functions

Look-ahead routines to find end character.

Function	Description
<code>expression()</code>	Find end of expression (non-space delimited)
<code>expression_space()</code>	Find end of expression (space delimited)
<code>matrix()</code>	Find end of matrix construction
<code>string()</code>	Find end of string
<code>comment()</code>	Find end of comment
<code>dots()</code>	Find continuation after ellipse
<code>paren()</code>	Find matching parenthesis
<code>cell()</code>	Find matching cell-parenthesis

`matlab2cpp.tree.findend.cell(self, start)`

Find matching cell-parenthesis

**Parameters**

- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of matching cell-parenthesis

**Return type** *int*

`matlab2cpp.tree.findend.comment(self, start)`

Find end of comment

**Parameters**

- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of end of comment

**Return type** *int*

`matlab2cpp.tree.findend.dots(self, start)`

Find continuation of expression after ellipse

**Parameters**

- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of end of ellipse

**Return type** *int*

`matlab2cpp.tree.findend.expression(self, start)`

Find end of expression (non-space delimited)

**Parameters**

- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of end of expression

**Return type** *int*

`matlab2cpp.tree.findend.expression_space(self, start)`

Find end of expression (space delimited)

**Parameters**



- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of end of expression

**Return type** *int*

`matlab2cpp.tree.findend.matrix(self, start)`  
Find end of matrix construction

**Parameters**

- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of end of matrix

**Return type** *int*

`matlab2cpp.tree.findend.paren(self, start)`  
Find matching parenthesis

**Parameters**

- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of matching parenthesis

**Return type** *int*

`matlab2cpp.tree.findend.string(self, start)`  
Find end of string

**Parameters**

- **self** (*Builder*) – Code constructor
- **start** (*int*) – current position in code

**Returns** index location of end of string

**Return type** *int*

`matlab2cpp.tree.findend.verbatim(self, start)`  
Find end of verbatim

**Arg:** *self*(*Builder*): Code constructor *start* (*int*): current position in code

**Returns** index location of end of verbatim

**Return type** *int*

## 2.3.10 Iterators

Routines for iterating lists

Functions	Description
<code>comma_list()</code>	Iterate over a comma separated list
<code>space_list()</code>	Iterate over a space delimited list

`matlab2cpp.tree.iterate.comma_list(self, start)`  
Iterate over a comma separated list

**Parameters**

- **self** ([Builder](#)) – Code constructor
- **start** (*int*) – Current position in code

**Returns** A list of 2-tuples that represents index start and end for each expression in list

**Return type** *list*

`matlab2cpp.tree.iterate.space_list(self, start)`

Iterate over a space delimited list

**Parameters**

- **self** ([Builder](#)) – Code constructor
- **start** (*int*) – Current position in code

**Returns** A list of 2-tuples that represents index start and end for each expression in list

**Return type** *list*

## 2.3.11 Identify structures

Routines for identifying code structure.

Function	Description
<a href="#"><i>space_delimiter()</i></a>	Check if at expression space-delimiter
<a href="#"><i>string()</i></a>	Check if at string start
<a href="#"><i>space_delimited()</i></a>	Check if list is space-delimited

`matlab2cpp.tree.identify.space_delimited(self, start)`

Check if list is space-delimited

**Parameters**

- **self** ([Builder](#)) – Code constructor
- **start** (*int*) – Current position in code

**Returns** True if list consists of whitespace delimiters

**Return type** `bool`

`matlab2cpp.tree.identify.space_delimiter(self, start)`

Check if mid-expression space-delimiter. This already assumes that position is in the middle of a space delimited list. Use *space\_delimited* to check if a list is space or comma delimited.

**Parameters**

- **self** ([Builder](#)) – Code constructor
- **start** (*int*) – Current position in code

**Returns** True if whitespace character classifies as a delimiter

**Return type** `bool`

`matlab2cpp.tree.identify.string(self, k)`

Check if at string start

**Parameters**

- **self** ([Builder](#)) – Code constructor

- **k** (*int*) – Current position in code

**Returns** True if character classifies as start of string.

**Return type** bool

### 2.3.12 Matlab constants

Matlab consists of various legal start and end characters depending on context. This module is a small collection of constants available to ensure that context is defined correctly.

`matlab2cpp.tree.constants.e_start`

*str*

characters allowed in expression start

`matlab2cpp.tree.constants.e_end`

*str*

characters allowed to terminate expression

`matlab2cpp.tree.constants.l_start`

*str*

characters allowed in list start

`matlab2cpp.tree.constants.l_end`

*str*

characters allowed to terminate list

`matlab2cpp.tree.constants.prefixes`

*str*

characters allowed as prefix univary operators

`matlab2cpp.tree.constants.postfix1`

*str*

characters allowed as postfix univary operators

`matlab2cpp.tree.constants.postfix2`

*tuple*

same as postfix1, but tuple of multi-char operators

`matlab2cpp.tree.constants.op1`

*str*

characters allowed as infix operators

`matlab2cpp.tree.constants.op2`

*tuple*

same as op1, but tuple of multi-char operators

## 2.4 Node representation

The module contains the following submodules.

### 2.4.1 The Node class

**class** `matlab2cpp.Node` (*parent=None, name='', value='', pointer=0, line=None, cur=None, code=None*)

A representation of a node in a node tree.

**backend**

*str*

The currently set translation backend. Available in the string format as *%(backend)s*.

**children**

*list*

A list of node children ordered from first to last child. Accessible using indexing (*node[0]*, *node[1]*, ...). Also available in the string format as *%(0)s*, *%(1)s*, ...

**cls**

*str*

A string representation of the class name. Available in the string format as *%(class)s*

**code**

*str*

The code that conceived this node.

**cur**

*int*

The index to the position in the code where this node was conceived. It takes the value 0 for nodes not created from code.

**declare**

*Node*

A reference to the node of same name where it is defined. This would be under *Declares*, *Params* or *Struct*. Useful for setting scope defined common datatypes. Returns itself if no declared variable has the same name as current node.

**dim**

*int*

The number of dimensions in a numerical datatype. The values 0 through 4 represents scalar, column vector, row vector, matrix and cube respectively. The value is None if datatype is not numerical. Interconnected with *type*.

**file**

*str*

Name of the program. In projects, it should be the absolute path to the Matlab source file. Available in the string format as *%(file)s*.

**ftypes**

*dict*

Input/output function scoped datatypes.

**func**

*Node*

A reference to Func (function) ancestor. Uses root if not found.

**group***Node*

A reference to the first ancestor where the datatype does not automatically affect nodes upwards. A list of these nodes are listed in *mc.reference.groups*.

**itype***list*

Input/output include scope statements

**line***int*

The codeline number in original code where this node was conceived. It takes the value 0 for nodes not created from code.

**mem***int*

The amount of type-space reserved per element in a numerical datatype. The value 0 through 4 represents unsigned int, int, float, double and complex. The value is None if datatype is not numerical. Interconnected with *type*.

**name***str*

The name of the node. Available in the string format as *%(name)s*.

**names***list*

A list of the names (if any) of the nodes children.

**num***bool*

A bool value that is true if and only if the datatype is numerical. Interconnected with *type*.

**parent***Node*

A reference to the direct node parent above the current one.

**pointer***int*

A numerical value of the reference count. The value 0 imply that the node refer to the actual variable, 1 is a reference to the variable, 2 is a reference of references, and so on.

**program***Node*

A reference to program ancestor. Uses root if not found.

**project***Node*

A reference to root node.

**reference***Node*

If node is a lambda function (backend *func\_lambda*), the variable is declared locally, but it's content might be available in it's own function. If so, the node will have a *reference* attribute to that function. Use *hasattr* to ensure it is the case.

**ret**

*tuple*

The raw translation of the node. Same as (str): *node.str*, but on the exact form the translation rule returned it.

**str**

*str*

The translation of the node. Note that the code is translated leaf to root, and parents will not be translated before after current node is translated. Current and all ancestors will have an empty string.

**stypes**

*dict*

Input/Output struct scoped datatypes.

**suggest**

*str*

A short string representation of the suggested datatype. It is used for suggesting datatype in general, and can only be assigned, not read. Typically only the declared variables will be read, so adding a suggestion is typically done *node.declare.type = "..."*.

**type**

*str*

A short string representation of the nodes datatype. Interconnected with *dim*, *mem* and *num*. Available in string format as *%(type)s*

**value**

*str*

A free variable reserved for content. The use varies from node to node. Available in the string format as *%(value)s*.

**vtypes**

*dict*

Verbatim translation in tree (read-only)

**auxiliary** (*type=None, convert=False*)

Create a auxiliary variable and rearrange nodes to put current node on its own line before.

**Parameters**

- **type** (*str, None*) – If provided, auxiliary variable type will be converted
- **convert** (*bool*) – If true, add an extra function call `conv_to` to convert datatype in Armadillo.

**Example**

Many statements that works inline in Matlab, must be done on multiple lines in C++. Take for example the statement `[1, 2] + 3`. In C++, the rowvec `[1, 2]` must first be initialized and converted into `rowvec` before arithmetics can be used:

```
>>> print mc.qscript("[1,2]+3")
```

```
sword __aux_irowvec_1 [] = {1, 2} ; __aux_irowvec_1 = irowvec(__aux_irowvec_1, 2, false) ;
__aux_irowvec_1+3;
```

The difference in tree structure is as follows:

```
>>> print mc.qtree("[1,2]", core=True)
1 1Block      code_block  TYPE
1 1| Statement code_block  TYPE
1 1| | Matrix   matrix    irowvec
1 2| | | Vector   matrix    irowvec
1 2| | | | Int     int      int
1 4| | | | Int     int      int
>>> print mc.qtree("[1,2]+3", core=True)
1 1Block      code_block  TYPE
1 1| Assign    matrix     int
1 1| | Var      irowvec    irowvec __aux_irowvec_1
1 1| | Matrix   matrix     irowvec
1 2| | | Vector   matrix    irowvec
1 2| | | | Int     int      int
1 4| | | | Int     int      int
1 1| Statement code_block  TYPE
1 1| | Plus      expression irowvec
1 1| | | Var      unknown   irowvec __aux_irowvec_1
1 7| | | Int      int       int
```

#### **create\_declare()**

Investigate if the current node is declared (either in Params, Declares or in Structs), and create such a node if non exists in Declares.

The declared variable's datatype will be the same as current node.

**Returns** the (newly) declared node

**Return type** *Node*

#### **error(msg)**

Add an error to the log file.

**Parameters** **msg** (*str*) – Content of the error

#### **Example**

```
>>> print mc.qlog(" a")
Error in class Var on line 1:
  a
  ^
unknown data type
```

#### **flatten(ordered=False, reverse=False, inverse=False)**

Return a list of all nodes

Structure:

```
A
| B
|| D
```

```
|| E
| C
|| F
|| G
```

Sorted [o]rdered, [r]everse and [i]nverse:

```
ori
___: A B D E C F G
o__ : A B C D E F G
_r_ : A C G F B E D
__i : D E B F G C A
or_ : A C B G F E D
o_i : D E F G B C A
_r_i : E D B G F C A
ori : G F E D C B A
```

### Parameters

- **node** (*Node*) – Root node to start from
- **ordered** (*bool*) – If True, make sure the nodes are hierarcically ordered.
- **reverse** (*bool*) – If True, children are itterated in reverse order.
- **inverse** (*bool*) – If True, tree is itterated in reverse order.

**Returns** All nodes in a flatten list.

**Return type** *list*

**include** (*name*, *\*\*kws*)

Include library in the header of the file.

These include:

```
+-----+-----+
```

Name | Description |

```
+=====+=====+ | SPlot | Local SPlot library | +-----+
+-----+ | m2cpp | Local M2cpp library | +-----+ | arma |
Global Armadillo library | +-----+ | iostream | Global iostream library |
+-----+ +-----+
```

### Parameters

- **name** (*str*) – Name of header to include
- **\*\*kws** (*str*; *optional*) – Optional args for header. Mostly not in use.

**plotting** ()

Prepare the code for plotting functionality.





**Parameters**

- **opt** (*argparse.Namespace, optional*) – Extra arguments provided by argparse
- **only** (*bool*) – If true, translate current node only.

**wall\_clock()**

Prepare for the use of `tic` and `toc` functionality in code.

Does nothing if called before.

**warning** (*msg*)

Add a warning to the log file.

**Parameters** **msg** (*str*) – Content of the warning

**See also:**

`error()`

## 2.4.2 Node backend

## 2.4.3 Quick references

Each node has a set of attributes that allows for quick access to properties and other node of interest. For example, if *node* has a name, it can be referred to by *node.name*. Another example is to access the parent node by *node.parent*.

Note that, if a reference does not exist, the node itself will be returned.

## 2.5 Datatypes

The following constructor classes exists here:

Class	Description
Type	Frontend for the datatype string
Dim	Reference to the number of dimensions
Mem	Reference to the memory type
Num	Numerical value indicator
Suggest	Frontend for suggested datatype

## 2.6 Auto-configure datatype

## 2.7 Collection

A full summary of all nodes.

Name	Children	Example	Description
All		:	Colon operator w/o range
Assign	<i>Expr Expr</i>	<i>a=b</i>	Assignment one var
Assigns	<i>Expr Expr+</i>	<i>[a,b]=c</i>	Assignment multi vars
Band	<i>Expr Expr+</i>	<i>a&amp;b</i>	Binary AND operator
Bcomment		<i>%{ . %}</i>	Block comment

Continued on next page

Table 2.1 – continued from previous page

Name	Children	Example	Description
Block	<i>Line*</i>	<i>a</i>	Code block
Bor	<i>Expr Expr+</i>	<i>a b</i>	Binary OR operator
Branch	<i>If Ifse* Else?</i>	<i>if a; end</i>	If chain container
Break		<i>break</i>	Break statement
Case	<i>Var Block</i>	<i>case a</i>	Case part of Switch
Catch	<i>Block</i>	<i>catch a</i>	Catch part of Tryblock
Cell	<i>Expr*</i>	<i>{a}</i>	Cell array
Cget	<i>Expr+</i>	<i>a{b}(c)</i>	Cell retrieval
Colon	<i>Expr Expr Expr?</i>	<i>a:b</i>	Colon operator w range
Counter			Struct array size
Cset	<i>Expr+</i>	<i>a{b}(c)=d</i>	Cell array assignment
Ctranspose	<i>Expr</i>	<i>a'</i>	Complex transform
Cvar	<i>Expr+</i>	<i>a{b}</i>	Cell variable
Declares	<i>Var*</i>		Declared variable list
Ecomment		<i>a%b</i>	End-of-line comment
Elementdivision	<i>Expr Expr+</i>	<i>a./b</i>	Scalars division
Elexp	<i>Expr Expr+</i>	<i>a.^b</i>	Element-wise exponent
Elif	<i>Expr Block</i>	<i>elseif a</i>	Else-if part of Branch
Elmul	<i>Expr Expr+</i>	<i>a.*b</i>	Element-wise multiplication
Else	<i>Block</i>	<i>else</i>	Else part of Branch
End		<i>end</i>	End-expression
Eq	<i>Expr Expr</i>	<i>a==b</i>	Equality sign
Error			Error node
Exp	<i>Expr Expr+</i>	<i>a^b</i>	Exponential operator
Fget	<i>Expr*</i>	<i>a.b(c)</i>	Fieldarray retrieval
Float		<i>4.</i>	Float-point number
For	<i>Var Expr Block</i>	<i>for a=b;end</i>	For-loop container
Fset	<i>Expr Expr+</i>	<i>a.b(c)=d</i>	Fieldname assignment
Func	<i>Declares Returns Params Block</i>	<i>function f() end</i>	Function container
Funcs	<i>[Main Func+]</i>		Root of all functions
Fvar		<i>a.b</i>	Fieldname variable
Ge	<i>Expr Expr</i>	<i>a&gt;=b</i>	Greater-or-equal operator
Get	<i>Expr*</i>	<i>a(b)</i>	Function or retrieval
Gt	<i>Expr Expr</i>	<i>a&gt;b</i>	Greater operator
Header			File header element
Headers			Collection header lines
If	<i>Expr Block</i>	<i>if a</i>	If part of Branch
Imag		<i>i</i>	Imaginary unit
Include			Include statement
Includes			Collection of includes
Int		<i>1</i>	Integer value
Lambda		<i>f=@()1</i>	Lambda function expression
Land	<i>Expr Expr+</i>	<i>a&amp;&amp; b</i>	Logical AND operator
Lcomment		<i>%a</i>	Line-comment
Le	<i>Expr Expr</i>	<i>a&lt;=b</i>	Less-or-equal operator
Leftelementdivision	<i>Expr Expr+</i>	<i>a./b</i>	Left scalar division
Leftmatrixdivision	<i>Expr Expr+</i>	<i>a/b</i>	Left matrix division
Log	<i>[Error Warning]+</i>		Collection of Errors
Lor	<i>Expr Expr</i>	<i>a  b</i>	Logical OR operator

Continued on next page

Table 2.1 – continued from previous page

Name	Children	Example	Description
Lt	<i>Expr Expr</i>	<i>a&lt;b</i>	Less-than operator
Main	<i>Declares Returns Params Block</i>	<i>function f() end</i>	Container for main function
Matrix	<i>Vector*</i>	<i>[a]</i>	Matrix container
Matrixdivision	<i>Expr Expr+</i>	<i>a/b</i>	Matrix division
Minus	<i>Expr Expr+</i>	<i>a-b</i>	Minus operator
Mul	<i>Expr Expr+</i>	<i>a*b</i>	Multiplication operator
Ne	<i>Expr Expr</i>	<i>a~=b</i>	Not-equal operator
Neg	<i>Expr</i>	<i>-a</i>	Unary negative sign
Nget	<i>Expr</i>	<i>a.(b)</i>	Namefield retrieval
Not	<i>Expr</i>	<i>~a</i>	Not operator
Nset	<i>Expr</i>	<i>a.(b)=c</i>	Namefield assignment
Otherwise	<i>Block</i>	<i>otherwise</i>	Otherwise part of Switch
Params	<i>Var*</i>		Function parameter container
Parfor	<i>Var Expr Block</i>	<i>‘parfor a=b;end’</i>	Parallel for-loop container
Plus	<i>Expr Expr+</i>	<i>a+b</i>	Addition operator
Pragma_for		<i>%%PARFOR str</i>	For-loop pragma
Program	<i>Includes Funcs Inlines Structs Headers Log</i>		Program root
Project	<i>Program+</i>		Root of all programs
Return		<i>return</i>	Return statement
Returns	<i>Var*</i>		Return value collection
Set	<i>Expr*</i>	<i>a(b)=c</i>	Array value assignment
Sget	<i>Expr+</i>	<i>a.b(c)</i>	Submodule function/retrieval
Sset	<i>Expr+</i>	<i>a.b(c)=d</i>	Submodule assignment
Statement	<i>Expr</i>	<i>a</i>	Stand alone statement
String		<i>‘a’</i>	String representation
Struct			Struct container
Structs			Container for structs
Switch	<i>Var Case+ Other</i>	<i>case a; end</i>	Container for Switch branch
Transpose	<i>Expr</i>	<i>a’</i>	Transpose operator
Try	<i>Block</i>	<i>try</i>	Try part of Tryblock
Tryblock	<i>Try Catch</i>	<i>try; end</i>	Container for try-blocks
Var		<i>a</i>	Variable
Vector	<i>Expr*</i>	<i>[a]</i>	Row-vector part of Matrix
Warning			Element in Log
While	<i>Expr Block</i>	<i>while a;end</i>	While-loop container

```
class matlab2cpp.collection.All (parent, **kws)
```

```
class matlab2cpp.collection.Assign (parent=None, name='', value='', pointer=0, line=None,
                                     cur=None, code=None)
```

```
class matlab2cpp.collection.Assigns (parent, **kws)
```

```
class matlab2cpp.collection.Band (parent, **kws)
```

```
class matlab2cpp.collection.Bcomment (parent, value, **kws)
```

```
class matlab2cpp.collection.Block (parent, **kws)
```

```
class matlab2cpp.collection.Bor (parent, **kws)
```

```
class matlab2cpp.collection.Branch (parent, **kws)
```

```
class matlab2cpp.collection.Break (parent, **kws)
```

```

class matlab2cpp.collection.Case (parent, **kws)
class matlab2cpp.collection.Catch (parent, **kws)
class matlab2cpp.collection.Cell (parent, **kws)
class matlab2cpp.collection.Cget (parent, name, **kws)
class matlab2cpp.collection.Colon (parent, **kws)
class matlab2cpp.collection.Counter (parent, name, value, **kws)
class matlab2cpp.collection.Cset (parent, name, **kws)
class matlab2cpp.collection.Ctranspose (parent, **kws)
class matlab2cpp.collection.Cvar (parent, name, **kws)
class matlab2cpp.collection.Declares (parent=None, name='', value='', pointer=0, line=None,
                                     cur=None, code=None)
class matlab2cpp.collection.Ecomment (parent, value, **kws)
class matlab2cpp.collection.Elementdivision (parent, **kws)
class matlab2cpp.collection.Elexp (parent, **kws)
class matlab2cpp.collection.Elif (parent, **kws)
class matlab2cpp.collection.Elmul (parent, **kws)
class matlab2cpp.collection.Else (parent, **kws)
class matlab2cpp.collection.End (parent, **kws)
class matlab2cpp.collection.Eq (parent, **kws)
class matlab2cpp.collection.Error (parent, name, value, **kws)
class matlab2cpp.collection.Exp (parent, **kws)
class matlab2cpp.collection.Expr (parent, **kws)
class matlab2cpp.collection.Fget (parent, name, value, **kws)
class matlab2cpp.collection.Float (parent, value, **kws)
class matlab2cpp.collection.For (parent, **kws)
class matlab2cpp.collection.Fset (parent, name, value, **kws)
class matlab2cpp.collection.Func (parent=None, name='', value='', pointer=0, line=None,
                                  cur=None, code=None)
class matlab2cpp.collection.Funcs (parent, line=1, **kws)
class matlab2cpp.collection.Fvar (parent, name, value, **kws)
class matlab2cpp.collection.Ge (parent, **kws)
class matlab2cpp.collection.Get (parent, name, **kws)
class matlab2cpp.collection.Gt (parent, **kws)
class matlab2cpp.collection.Header (parent, name, **kws)
class matlab2cpp.collection.Headers (parent, **kws)
class matlab2cpp.collection.If (parent, **kws)
class matlab2cpp.collection.Imag (parent, value, **kws)

```

```
class matlab2cpp.collection.Include (parent, name, **kws)
class matlab2cpp.collection.Includes (parent, **kws)
class matlab2cpp.collection.Inline (parent, name, **kws)
class matlab2cpp.collection.Inlines (parent, **kws)
class matlab2cpp.collection.Int (parent, value, **kws)
class matlab2cpp.collection.Lambda (parent, name='', **kws)
class matlab2cpp.collection.Land (parent, **kws)
class matlab2cpp.collection.Lcomment (parent, value, **kws)
class matlab2cpp.collection.Le (parent, **kws)
class matlab2cpp.collection.Leftelementdivision (parent, **kws)
class matlab2cpp.collection.Leftmatrixdivision (parent, **kws)
class matlab2cpp.collection.Log (parent, **kws)
class matlab2cpp.collection.Lor (parent, **kws)
class matlab2cpp.collection.It (parent, **kws)
class matlab2cpp.collection.Main (parent, name='main', **kws)
class matlab2cpp.collection.Matrix (parent, **kws)
class matlab2cpp.collection.Matrixdivision (parent, **kws)
class matlab2cpp.collection.Minus (parent, **kws)
class matlab2cpp.collection.Mul (parent, **kws)
class matlab2cpp.collection.Ne (parent, **kws)
class matlab2cpp.collection.Neg (parent, **kws)
class matlab2cpp.collection.Nget (parent, name, **kws)
class matlab2cpp.collection.Not (parent, **kws)
class matlab2cpp.collection.Nset (parent, name, **kws)
class matlab2cpp.collection.Opr (parent, **kws)
class matlab2cpp.collection.Otherwise (parent, **kws)
class matlab2cpp.collection.Params (parent=None, name='', value='', pointer=0, line=None,
                                   cur=None, code=None)
class matlab2cpp.collection.Paren (parent, **kws)
class matlab2cpp.collection.Plus (parent, **kws)
class matlab2cpp.collection.Program (parent, name, **kws)
class matlab2cpp.collection.Project (name='', cur=0, line=0, code='', **kws)
class matlab2cpp.collection.Resize (parent, **kws)
class matlab2cpp.collection.Return (parent, **kws)
class matlab2cpp.collection>Returns (parent=None, name='', value='', pointer=0, line=None,
                                   cur=None, code=None)
class matlab2cpp.collection.Set (parent, name, **kws)
```

```

class matlab2cpp.collection.Sget (parent, name, value, **kws)
class matlab2cpp.collection.Sset (parent, name, value, **kws)
class matlab2cpp.collection. (parent, **kws)
class matlab2cpp.collection.String (parent, value, **kws)
class matlab2cpp.collection.Struct (parent, **kws)
class matlab2cpp.collection.Structs (parent, **kws)
class matlab2cpp.collection.Switch (parent, **kws)
class matlab2cpp.collection.Transpose (parent, **kws)
class matlab2cpp.collection.Try (parent, **kws)
class matlab2cpp.collection.Tryblock (parent, **kws)
class matlab2cpp.collection.Var (parent, name, **kws)
class matlab2cpp.collection.Vector (parent, **kws)
class matlab2cpp.collection.Warning (parent, name, value, **kws)
class matlab2cpp.collection.While (parent, **kws)

```

## 2.8 Translation rules

Datatype driven rules have the same name as datatypes reference in *datatype*. They are as follows:

Datatype	Rule	Description
cell	<code>__cell</code>	Cell structure
char	<code>__char</code>	Word character
cube	<code>__cube</code>	Armadillo cube
cx_cube	<code>__cx_cube</code>	Armadillo cube
cx_double	<code>__cx_double</code>	Scalar complex
cx_mat	<code>__cx_mat</code>	Armadillo matrix
cx_rowvec	<code>__cx_rowvec</code>	Armadillo rowvec
cx_vec	<code>__cx_vec</code>	Armadillo colvec
double	<code>__double</code>	Scalar double
fcube	<code>__fcube</code>	Armadillo cube
float	<code>__float</code>	Scalar float
fmat	<code>__fmat</code>	Armadillo matrix
frowvec	<code>__frowvec</code>	Armadillo rowvec
fvec	<code>__fvec</code>	Armadillo colvec
icube	<code>__icube</code>	Armadillo cube
imat	<code>__imat</code>	Armadillo matrix
int	<code>__int</code>	Scalar integer
irowvec	<code>__irowvec</code>	Armadillo rowvec
ivec	<code>__ivec</code>	Armadillo colvec
mat	<code>__mat</code>	Armadillo matrix
rowvec	<code>__rowvec</code>	Armadillo rowvec
string	<code>__string</code>	Character string
struct	<code>__struct</code>	Struct
structs	<code>__structs</code>	Array of structs

Continued on next page

Table 2.2 – continued from previous page

Datatype	Rule	Description
ucube	<code>__ucube</code>	Armadillo cube
umat	<code>__umat</code>	Armadillo matrix
urowvec	<code>__urowvec</code>	Armadillo rowvec
uvec	<code>__uvec</code>	Armadillo colvec
uword	<code>__uword</code>	Scalar uword
vec	<code>__vec</code>	Armadillo colvec

These basic types are then glued together through the following:

Rule	Description
<code>__code_block</code>	Branches, loops etc.
<code>__expression</code>	Operators and special characters
<code>__func_lambda</code>	Anonymous functions
<code>__func_return</code>	Functions with one return value
<code>__func_returns</code>	Functions with multiple return values
<code>__matrix</code>	Matrix constructor
<code>__program</code>	Program postprocessing
<code>__reserved</code>	Reserved names from Matlab library
<code>__unknown</code>	Structures with unknown origin
<code>__verbatim</code>	Special verbatim translations

## 2.8.1 Datatype driven rules

## 2.8.2 Other rules

This module contains all the codeblock related nodes. Each node can then here be nested on top of each other. They are static in the sense that there only exists one copy, unaffected by type and have the backend fixed to `code_block`.  
Anonymous/Lambda Functions Functions with single return

### Nodes

**Func** [Function definition] Contains: Declares, Returns, Params, Block Property: name

**Returns** [Function return variables] Contains: Var, ...

Params : Function parameter variables

**Get** [Function call] Example: “y(4)” Contains: Gets Property: name

**Var** [Function call hidden as variable] Example “y” Contains: nothing

Functions with multiple returns Matrix declaration rules

### Nodes

**Matrix** [Matrix container] Example: “[x;y]” Contains: Vector, ...

**Vector** [(Column)-Vector container] Contains: Expr, ...

Reserved translation rules

See `rules.reserved` for a collection of set of the various reserved words implemented into matlab2cpp.



## 2.9 Datatype scope

### 2.9.1 Input/Output classes

**class** matlab2cpp.supplement.**Ftypes**  
Access to function types from program node

**class** matlab2cpp.supplement.**Stypes**

**class** matlab2cpp.supplement.**Itypes**

**class** matlab2cpp.supplement.**Vtypes**

### 2.9.2 Stringify supplement file

matlab2cpp.supplement.**str\_variables** (*types\_f*={}, *types\_s*={}, *types\_i*=[], *suggest*={}, *pre-fix*=True, *types\_v*={})

Convert a nested dictionary for types, suggestions and structs and use them to create a suppliment text ready to be saved.

**Kwargs:** *types\_f* (dict): Function variables datatypes *types\_s* (dict): Struct variables datatypes *types\_i* (list): Includes in header *types\_v* (dict): Verbatim translations *suggest* (dict): Suggested datatypes for *types\_f* and *types\_s* *prefix* (bool): True if the type explanation should be included

**Returns:** **str** String representation of suppliment file

#### Example

```
>>> types_f = {"f" : {"a":"int"}, "g" : {"b":""}}
>>> types_s = {"c" : {"d":""}}
>>> types_i = ["#include <armadillo>"]
>>> suggest = {"g" : {"b":"float"}, "c" : {"d":"vec"}}
>>> print str_variables(types_f, types_s, types_i, suggest, prefix=False)
functions = {
  "f" : {
    "a" : "int",
  },
  "g" : {
    "b" : "", # float
  },
}
structs = {
  "c" : {
    "d" : "", # vec
  },
}
includes = [
  '#include <armadillo>',
]
```

## 2.10 Testsuite



**m**

- matlab2cpp, 21
- matlab2cpp.collection, 70
- matlab2cpp.configure, 70
- matlab2cpp.datatype, 70
- matlab2cpp.manual.usr01\_interaction, 4
- matlab2cpp.manual.usr02\_datatype, 7
- matlab2cpp.manual.usr03\_rules, 13
- matlab2cpp.manual.usr04\_node, 16
- matlab2cpp.node, 63
- matlab2cpp.node.backend, 70
- matlab2cpp.node.reference, 70
- matlab2cpp.qfunctions, 21
- matlab2cpp.rules, 75
- matlab2cpp.rules.\_cell, 76
- matlab2cpp.rules.\_char, 76
- matlab2cpp.rules.\_code\_block, 76
- matlab2cpp.rules.\_cube, 76
- matlab2cpp.rules.\_cx\_cube, 76
- matlab2cpp.rules.\_cx\_double, 76
- matlab2cpp.rules.\_cx\_mat, 76
- matlab2cpp.rules.\_cx\_rowvec, 76
- matlab2cpp.rules.\_cx\_vec, 76
- matlab2cpp.rules.\_double, 76
- matlab2cpp.rules.\_expression, 76
- matlab2cpp.rules.\_fcube, 76
- matlab2cpp.rules.\_float, 76
- matlab2cpp.rules.\_fmat, 76
- matlab2cpp.rules.\_frowvec, 76
- matlab2cpp.rules.\_func\_lambda, 76
- matlab2cpp.rules.\_func\_return, 76
- matlab2cpp.rules.\_func\_returns, 76
- matlab2cpp.rules.\_fvec, 76
- matlab2cpp.rules.\_icube, 76
- matlab2cpp.rules.\_imat, 76
- matlab2cpp.rules.\_int, 76
- matlab2cpp.rules.\_irowvec, 76
- matlab2cpp.rules.\_ivec, 76
- matlab2cpp.rules.\_mat, 76
- matlab2cpp.rules.\_matrix, 76
- matlab2cpp.rules.\_program, 76
- matlab2cpp.rules.\_reserved, 76
- matlab2cpp.rules.\_rowvec, 76
- matlab2cpp.rules.\_string, 76
- matlab2cpp.rules.\_struct, 76
- matlab2cpp.rules.\_structs, 76
- matlab2cpp.rules.\_ucube, 76
- matlab2cpp.rules.\_umat, 76
- matlab2cpp.rules.\_unknown, 76
- matlab2cpp.rules.\_urowvec, 76
- matlab2cpp.rules.\_uvec, 76
- matlab2cpp.rules.\_uword, 76
- matlab2cpp.rules.\_vec, 76
- matlab2cpp.rules.\_verbatim, 76
- matlab2cpp.supplement, 77
- matlab2cpp.testsuite, 77
- matlab2cpp.tree, 27
- matlab2cpp.tree.assign, 42
- matlab2cpp.tree.branches, 43
- matlab2cpp.tree.codeblock, 48
- matlab2cpp.tree.constants, 63
- matlab2cpp.tree.expression, 49
- matlab2cpp.tree.findend, 59
- matlab2cpp.tree.functions, 50
- matlab2cpp.tree.identify, 62
- matlab2cpp.tree.iterate, 61
- matlab2cpp.tree.misc, 53
- matlab2cpp.tree.variables, 58



## Symbols

-S, `-matlab-suggest`  
     m2cpp command line option, 5

-T, `-tree-full`  
     m2cpp command line option, 5

-c, `-comments`  
     m2cpp command line option, 5

-d, `-disp`  
     m2cpp command line option, 5

-h, `-help`  
     m2cpp command line option, 5

-l `<line>`, `-line <line>`  
     m2cpp command line option, 5

-n, `-nargin`  
     m2cpp command line option, 5

-o, `-original`  
     m2cpp command line option, 5

-omp, `-enable-omp`  
     m2cpp command line option, 5

-p `<paths_file>`, `-paths_file <paths_file>`  
     m2cpp command line option, 5

-r, `-reset`  
     m2cpp command line option, 5

-s, `-suggest`  
     m2cpp command line option, 5

-t, `-tree`  
     m2cpp command line option, 5

-tbb, `-enable-tbb`  
     m2cpp command line option, 5

`__getitem__()` (matlab2cpp.Builder method), 28

`__init__()` (matlab2cpp.Builder method), 28

`__str__()` (matlab2cpp.Builder method), 29

`__weakref__` (matlab2cpp.Builder attribute), 29

## A

All (class in matlab2cpp.collection), 72

Assign (class in matlab2cpp.collection), 72

`assign()` (in module matlab2cpp.tree.variables), 58

Assigns (class in matlab2cpp.collection), 72

`auxiliary()` (matlab2cpp.Node method), 66

## B

`backend` (matlab2cpp.node.Node attribute), 64

Band (class in matlab2cpp.collection), 72

Bcomment (class in matlab2cpp.collection), 72

Block (class in matlab2cpp.collection), 72

Bor (class in matlab2cpp.collection), 72

Branch (class in matlab2cpp.collection), 72

Break (class in matlab2cpp.collection), 72

`build()` (in module matlab2cpp), 21

Builder (class in matlab2cpp), 28

## C

Case (class in matlab2cpp.collection), 72

Catch (class in matlab2cpp.collection), 73

Cell (class in matlab2cpp.collection), 73

`cell()` (in module matlab2cpp.tree.findend), 60

`cell()` (in module matlab2cpp.tree.misc), 53

`cell_arg()` (in module matlab2cpp.tree.variables), 58

Cget (class in matlab2cpp.collection), 73

`children` (matlab2cpp.node.Node attribute), 64

`cls` (matlab2cpp.node.Node attribute), 64

`code` (matlab2cpp.node.Node attribute), 64

`codeblock()` (in module matlab2cpp.tree.codeblock), 48

Colon (class in matlab2cpp.collection), 73

`comma_list()` (in module matlab2cpp.tree.iterate), 61

`comment()` (in module matlab2cpp.tree.findend), 60

`comment()` (in module matlab2cpp.tree.misc), 54

`configure()` (matlab2cpp.Builder method), 29

Counter (class in matlab2cpp.collection), 73

`create()` (in module matlab2cpp.tree.expression), 49

`create_assign()` (matlab2cpp.Builder method), 30

`create_assign_variable()` (matlab2cpp.Builder method), 31

`create_assigns()` (matlab2cpp.Builder method), 31

`create_cell()` (matlab2cpp.Builder method), 31

`create_codeblock()` (matlab2cpp.Builder method), 32

`create_comment()` (matlab2cpp.Builder method), 32

`create_declare()` (matlab2cpp.Node method), 67

`create_expression()` (matlab2cpp.Builder method), 32

`create_for()` (matlab2cpp.Builder method), 33

`create_function()` (matlab2cpp.Builder method), 33

`create_if()` (matlab2cpp.Builder method), 34

`create_lambda_assign()` (matlab2cpp.Builder method), 34  
`create_lambda_func()` (matlab2cpp.Builder method), 35  
`create_list()` (matlab2cpp.Builder method), 35  
`create_main()` (matlab2cpp.Builder method), 36  
`create_matrix()` (matlab2cpp.Builder method), 36  
`create_number()` (matlab2cpp.Builder method), 36  
`create_parfor()` (matlab2cpp.Builder method), 37  
`create_program()` (matlab2cpp.Builder method), 37  
`create_reserved()` (matlab2cpp.Builder method), 38  
`create_string()` (matlab2cpp.Builder method), 38  
`create_switch()` (matlab2cpp.Builder method), 38  
`create_try()` (matlab2cpp.Builder method), 39  
`create_variable()` (matlab2cpp.Builder method), 39  
`create_verbatim()` (matlab2cpp.Builder method), 40  
`create_while()` (matlab2cpp.Builder method), 40  
`Cset` (class in matlab2cpp.collection), 73  
`Ctranspose` (class in matlab2cpp.collection), 73  
`cur` (matlab2cpp.node.Node attribute), 64  
`Cvar` (class in matlab2cpp.collection), 73

## D

`declare` (matlab2cpp.node.Node attribute), 64  
`Declares` (class in matlab2cpp.collection), 73  
`dim` (matlab2cpp.node.Node attribute), 64  
`dots()` (in module matlab2cpp.tree.findend), 60

## E

`e_end` (in module matlab2cpp.tree.constants), 63  
`e_start` (in module matlab2cpp.tree.constants), 63  
`Ecomment` (class in matlab2cpp.collection), 73  
`Elementdivision` (class in matlab2cpp.collection), 73  
`Elexp` (class in matlab2cpp.collection), 73  
`Elif` (class in matlab2cpp.collection), 73  
`Elmul` (class in matlab2cpp.collection), 73  
`Else` (class in matlab2cpp.collection), 73  
`End` (class in matlab2cpp.collection), 73  
`Eq` (class in matlab2cpp.collection), 73  
`Error` (class in matlab2cpp.collection), 73  
`error()` (matlab2cpp.Node method), 67  
`Exp` (class in matlab2cpp.collection), 73  
`Expr` (class in matlab2cpp.collection), 73  
`expression()` (in module matlab2cpp.tree.findend), 60  
`expression_space()` (in module matlab2cpp.tree.findend), 60

## F

`Fget` (class in matlab2cpp.collection), 73  
`file` (matlab2cpp.node.Node attribute), 64  
`filename`  
    matlab2cpp command line option, 4  
`flatten()` (matlab2cpp.Node method), 67  
`Float` (class in matlab2cpp.collection), 73  
`For` (class in matlab2cpp.collection), 73  
`forloop()` (in module matlab2cpp.tree.branches), 43

`Fset` (class in matlab2cpp.collection), 73  
`Ftypes` (class in matlab2cpp.supplement), 77  
`ftypes` (matlab2cpp.node.Node attribute), 64  
`Func` (class in matlab2cpp.collection), 73  
`func` (matlab2cpp.node.Node attribute), 64  
`Funcs` (class in matlab2cpp.collection), 73  
`function()` (in module matlab2cpp.tree.functions), 50  
`Fvar` (class in matlab2cpp.collection), 73

## G

`Ge` (class in matlab2cpp.collection), 73  
`Get` (class in matlab2cpp.collection), 73  
`get_unknowns()` (matlab2cpp.Builder method), 41  
`group` (matlab2cpp.node.Node attribute), 64  
`Gt` (class in matlab2cpp.collection), 73

## H

`Header` (class in matlab2cpp.collection), 73  
`Headers` (class in matlab2cpp.collection), 73

## I

`If` (class in matlab2cpp.collection), 73  
`ifbranch()` (in module matlab2cpp.tree.branches), 44  
`Imag` (class in matlab2cpp.collection), 73  
`Include` (class in matlab2cpp.collection), 73  
`include()` (matlab2cpp.Node method), 68  
`Includes` (class in matlab2cpp.collection), 74  
`Inline` (class in matlab2cpp.collection), 74  
`Inlines` (class in matlab2cpp.collection), 74  
`Int` (class in matlab2cpp.collection), 74  
`itype` (matlab2cpp.node.Node attribute), 65  
`Itypes` (class in matlab2cpp.supplement), 77

## L

`l_end` (in module matlab2cpp.tree.constants), 63  
`l_start` (in module matlab2cpp.tree.constants), 63  
`Lambda` (class in matlab2cpp.collection), 74  
`lambda_assign()` (in module matlab2cpp.tree.functions), 50  
`lambda_func()` (in module matlab2cpp.tree.functions), 52  
`Land` (class in matlab2cpp.collection), 74  
`Lcomment` (class in matlab2cpp.collection), 74  
`Le` (class in matlab2cpp.collection), 74  
`Leftelementdivision` (class in matlab2cpp.collection), 74  
`Leftmatrixdivision` (class in matlab2cpp.collection), 74  
`line` (matlab2cpp.node.Node attribute), 65  
`list()` (in module matlab2cpp.tree.misc), 55  
`load()` (matlab2cpp.Builder method), 41  
`Log` (class in matlab2cpp.collection), 74  
`Lor` (class in matlab2cpp.collection), 74  
`Lt` (class in matlab2cpp.collection), 74

## M

matlab2cpp command line option

- S, `--matlab-suggest`, 5
  - T, `--tree-full`, 5
  - c, `--comments`, 5
  - d, `--disp`, 5
  - h, `--help`, 5
  - l `<line>`, `--line <line>`, 5
  - n, `--nargin`, 5
  - o, `--original`, 5
  - omp, `--enable-omp`, 5
  - p `<paths_file>`, `--paths_file <paths_file>`, 5
  - r, `--reset`, 5
  - s, `--suggest`, 5
  - t, `--tree`, 5
  - tbb, `--enable-tbb`, 5
  - filename, 4
  - Main (class in `matlab2cpp.collection`), 74
  - main() (in module `matlab2cpp.tree.functions`), 52
  - `matlab2cpp` (module), 21
  - `matlab2cpp.collection` (module), 70
  - `matlab2cpp.configure` (module), 70
  - `matlab2cpp.datatype` (module), 70
  - `matlab2cpp.manual.usr01_interaction` (module), 4
  - `matlab2cpp.manual.usr02_datatype` (module), 7
  - `matlab2cpp.manual.usr03_rules` (module), 13
  - `matlab2cpp.manual.usr04_node` (module), 16
  - `matlab2cpp.node` (module), 63
  - `matlab2cpp.node.backend` (module), 70
  - `matlab2cpp.node.reference` (module), 70
  - `matlab2cpp.qfunctions` (module), 21
  - `matlab2cpp.rules` (module), 75
  - `matlab2cpp.rules._cell` (module), 76
  - `matlab2cpp.rules._char` (module), 76
  - `matlab2cpp.rules._code_block` (module), 76
  - `matlab2cpp.rules._cube` (module), 76
  - `matlab2cpp.rules._cx_cube` (module), 76
  - `matlab2cpp.rules._cx_double` (module), 76
  - `matlab2cpp.rules._cx_mat` (module), 76
  - `matlab2cpp.rules._cx_rowvec` (module), 76
  - `matlab2cpp.rules._cx_vec` (module), 76
  - `matlab2cpp.rules._double` (module), 76
  - `matlab2cpp.rules._expression` (module), 76
  - `matlab2cpp.rules._fcube` (module), 76
  - `matlab2cpp.rules._float` (module), 76
  - `matlab2cpp.rules._fmat` (module), 76
  - `matlab2cpp.rules._frowvec` (module), 76
  - `matlab2cpp.rules._func_lambda` (module), 76
  - `matlab2cpp.rules._func_return` (module), 76
  - `matlab2cpp.rules._func_returns` (module), 76
  - `matlab2cpp.rules._fvec` (module), 76
  - `matlab2cpp.rules._icube` (module), 76
  - `matlab2cpp.rules._imat` (module), 76
  - `matlab2cpp.rules._int` (module), 76
  - `matlab2cpp.rules._irowvec` (module), 76
  - `matlab2cpp.rules._ivec` (module), 76
  - `matlab2cpp.rules._mat` (module), 76
  - `matlab2cpp.rules._matrix` (module), 76
  - `matlab2cpp.rules._program` (module), 76
  - `matlab2cpp.rules._reserved` (module), 76
  - `matlab2cpp.rules._rowvec` (module), 76
  - `matlab2cpp.rules._string` (module), 76
  - `matlab2cpp.rules._struct` (module), 76
  - `matlab2cpp.rules._structs` (module), 76
  - `matlab2cpp.rules._ucube` (module), 76
  - `matlab2cpp.rules._umat` (module), 76
  - `matlab2cpp.rules._unknown` (module), 76
  - `matlab2cpp.rules._urowvec` (module), 76
  - `matlab2cpp.rules._uvec` (module), 76
  - `matlab2cpp.rules._uword` (module), 76
  - `matlab2cpp.rules._vec` (module), 76
  - `matlab2cpp.rules._verbatim` (module), 76
  - `matlab2cpp.supplement` (module), 77
  - `matlab2cpp.testsuite` (module), 77
  - `matlab2cpp.tree` (module), 27
  - `matlab2cpp.tree.assign` (module), 42
  - `matlab2cpp.tree.branches` (module), 43
  - `matlab2cpp.tree.codeblock` (module), 48
  - `matlab2cpp.tree.constants` (module), 63
  - `matlab2cpp.tree.expression` (module), 49
  - `matlab2cpp.tree.findend` (module), 59
  - `matlab2cpp.tree.functions` (module), 50
  - `matlab2cpp.tree.identify` (module), 62
  - `matlab2cpp.tree.iterate` (module), 61
  - `matlab2cpp.tree.misc` (module), 53
  - `matlab2cpp.tree.variables` (module), 58
  - Matrix (class in `matlab2cpp.collection`), 74
  - matrix() (in module `matlab2cpp.tree.findend`), 61
  - matrix() (in module `matlab2cpp.tree.misc`), 55
  - Matrixdivision (class in `matlab2cpp.collection`), 74
  - mem (`matlab2cpp.node.Node` attribute), 65
  - Minus (class in `matlab2cpp.collection`), 74
  - Mul (class in `matlab2cpp.collection`), 74
  - multi() (in module `matlab2cpp.tree.assign`), 42
- ## N
- name (`matlab2cpp.node.Node` attribute), 65
  - names (`matlab2cpp.node.Node` attribute), 65
  - Ne (class in `matlab2cpp.collection`), 74
  - Neg (class in `matlab2cpp.collection`), 74
  - Nget (class in `matlab2cpp.collection`), 74
  - Node (class in `matlab2cpp`), 64
  - Not (class in `matlab2cpp.collection`), 74
  - Nset (class in `matlab2cpp.collection`), 74
  - num (`matlab2cpp.node.Node` attribute), 65
  - number() (in module `matlab2cpp.tree.misc`), 56
- ## O
- op1 (in module `matlab2cpp.tree.constants`), 63
  - op2 (in module `matlab2cpp.tree.constants`), 63

Opr (class in matlab2cpp.collection), 74  
Otherwise (class in matlab2cpp.collection), 74

## P

Params (class in matlab2cpp.collection), 74  
Paren (class in matlab2cpp.collection), 74  
paren() (in module matlab2cpp.tree.findend), 61  
parent (matlab2cpp.node.Node attribute), 65  
plotting() (matlab2cpp.Node method), 68  
Plus (class in matlab2cpp.collection), 74  
pointer (matlab2cpp.node.Node attribute), 65  
postfix1 (in module matlab2cpp.tree.constants), 63  
postfix2 (in module matlab2cpp.tree.constants), 63  
prefixes (in module matlab2cpp.tree.constants), 63  
Program (class in matlab2cpp.collection), 74  
program (matlab2cpp.node.Node attribute), 65  
program() (in module matlab2cpp.tree.functions), 53  
Project (class in matlab2cpp.collection), 74  
project (matlab2cpp.node.Node attribute), 65  
properties() (matlab2cpp.Node method), 68

## Q

qcpp() (in module matlab2cpp), 22  
qhpp() (in module matlab2cpp), 23  
qlog() (in module matlab2cpp), 25  
qpy() (in module matlab2cpp), 24  
qscript() (in module matlab2cpp), 25  
qtree() (in module matlab2cpp), 26

## R

reference (matlab2cpp.node.Node attribute), 65  
reserved() (in module matlab2cpp.tree.misc), 57  
Resize (class in matlab2cpp.collection), 74  
resize() (matlab2cpp.Node method), 69  
ret (matlab2cpp.node.Node attribute), 66  
retrieve\_operator() (in module  
lab2cpp.tree.expression), 50  
Return (class in matlab2cpp.collection), 74  
Returns (class in matlab2cpp.collection), 74

## S

Set (class in matlab2cpp.collection), 74  
Sget (class in matlab2cpp.collection), 74  
single() (in module matlab2cpp.tree.assign), 43  
space\_delimited() (in module matlab2cpp.tree.identify),  
62  
space\_delimiter() (in module matlab2cpp.tree.identify),  
62  
space\_list() (in module matlab2cpp.tree.iterate), 62  
Sset (class in matlab2cpp.collection), 75  
Statement (class in matlab2cpp.collection), 75  
str (matlab2cpp.node.Node attribute), 66  
str\_variables() (in module matlab2cpp.supplement), 77

String (class in matlab2cpp.collection), 75  
string() (in module matlab2cpp.tree.findend), 61  
string() (in module matlab2cpp.tree.identify), 62  
string() (in module matlab2cpp.tree.misc), 57  
Struct (class in matlab2cpp.collection), 75  
Structs (class in matlab2cpp.collection), 75  
Stypes (class in matlab2cpp.supplement), 77  
stypes (matlab2cpp.node.Node attribute), 66  
suggest (matlab2cpp.node.Node attribute), 66  
suggest\_datatype() (matlab2cpp.Node method), 69  
summary() (matlab2cpp.Node method), 69  
Switch (class in matlab2cpp.collection), 75  
switch() (in module matlab2cpp.tree.branches), 45  
syntaxerror() (matlab2cpp.Builder method), 41

## T

translate() (matlab2cpp.Builder method), 42  
translate() (matlab2cpp.Node method), 69  
Transpose (class in matlab2cpp.collection), 75  
Try (class in matlab2cpp.collection), 75  
Tryblock (class in matlab2cpp.collection), 75  
trybranch() (in module matlab2cpp.tree.branches), 46  
type (matlab2cpp.node.Node attribute), 66

## V

value (matlab2cpp.node.Node attribute), 66  
Var (class in matlab2cpp.collection), 75  
variable() (in module matlab2cpp.tree.variables), 59  
Vector (class in matlab2cpp.collection), 75  
verbatim() (in module matlab2cpp.tree.findend), 61  
verbatim() (in module matlab2cpp.tree.misc), 57  
Vtypes (class in matlab2cpp.supplement), 77  
vtypes (matlab2cpp.node.Node attribute), 66

## W

mat- wall\_clock() (matlab2cpp.Node method), 70  
Warning (class in matlab2cpp.collection), 75  
warning() (matlab2cpp.Node method), 70  
While (class in matlab2cpp.collection), 75  
whileloop() (in module matlab2cpp.tree.branches), 47