matlab2cpp Documentation

Release 1.0

Jonathan Feinberg

October 16, 2015

CONTENTS

CHAPTER

ONE

INTRODUCTION

Matlab2cpp is a semi-automatic tool for converting code from Matlab to C++.

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 Installation

Requirements:

- Python 2.7.3
- Armadillo (Not required for running, but generator creates armadillo code.)

Linux/Mac:

As root, run the following command:

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

The executable ´mconvert´ is now available from path.

Windows:

```
> Python setup.py install
```

The executable mconvert.py can freely be copied or be added to environmental variables manually (with or without the *.py* extension).

1.2 An illustrating Example

Assuming Linux installation and *mconvert* 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:

```
$ mconvert 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 Matlab2cpp 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:

```
$ mconvert 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 necesarry.

CHAPTER

TWO

USER INTERACTION

The simplest way to interact with the *Matlab2cpp*-toolbox is to use the *mconvert* frontend. The script automatically creates files with various extensions containing translations and/or meta-information. Even though *mconvert* 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. For the examples, the module is assumed imported as

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

The toolbox is sorted into the following modules:

Module	Description
`qfunctions`	Functions for performing simple translations
`tree`	Constructing a tree from Matlab code
`datatype`	The various node data types
`node`	Node behavior
`collection`	The collcetion of various node
`rule`	Translation rules
`inlines`	Code insertion

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 *mconvert* function.

2.1 mconvert

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: mconvert [-h] [-t] [-T] [-s] [-r] [-d] [-c] [-l LINE] filename

`filename`
File containing valid Matlab code.

`-h, --help`
show this help message and exit

`-t, --tree`
Print the underlying node tree. Each line in the output represents a node and is formated 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, not only code related.

`-s, --suggest`
: Automatically populate the <*filename*>*.py* file with datatype with suggestions if possible.

`-r, --reset`
: Ignore the content of <*filename*>*.py* and make a fresh translation.

`-d, --disp`
: Print out the progress of the translation process.

`-c, --comments`
: Strip away all the comments in the output of the translation.

`-l` `<line>`, **`--line`** `<line>`
: Only display code related to code line number <*line*>.

2.2 Quick translation functions

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 **`mconvert`** 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.

Function	Description
`build()`	Build token tree
`qcpp()`	Create content of *.cpp* file
`qhpp()`	Create content of *.hpp* file
`qpy()`	Create content of supplement *.py* file
`qlog()`	Create content of *.log* file
`qscript()`	Create quick code translation
`qtree()`	Create summary of node tree

`matlab2cpp.`**`build`**(*code*, *disp=False*, *retall=False*, *suggest=False*, *comments=False*, ***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.
- **`**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:

`qtree()`, `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)
{
  int a ;
```

```
    double b ;
    string c ;
    a = 4 ;
    b = 5. ;
    c = "abc" ;
    return 0 ;
  }
  >>> build = mc.build(code, retall=True)
  >>> build.configure()
  >>> 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 occured. 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 alse: `error()`, `warning()`

`matlab2cpp.`**`qscript`**`(`*code*, *suggest=False*, ***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", suggest=True)
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 content is described in details in `tree`.

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    unamed
        Includes    program       TYPE
        | Include     program       TYPE    #include <armadillo>
        | Include     program       TYPE    using namespace arma ;
  1   1 Funcs       program       TYPE    unamed
  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      unknown       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    unamed
        Structs     program       TYPE    unamed
        Headers     program       TYPE    unamed
        | Header      program       TYPE    f
        Log         program       TYPE    unamed
        | 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`*

CHAPTER

THREE

CREATING NODE TREE

Translating Matlab code is done in three steps: interpret Matlab code and construct the node tree, configuring the data types in the tree and translating the tree into C++ code. Here we are going to look at the first step: interpreting the Matlab code using the `Builder` class. See `datatype` and `rules` for the two other steps respecticly.

The class `Builder` creates a tree representation of the code where each segment of code is represented by a node. To observe the node structure it possible to either use **mconvert** with the *-t* option, or the python function `qtree()`. For example:

```
>>> print mc.qtree("a = 2+2")
        Program    program      TYPE    unamed
        Includes   program      TYPE
        | Include    program      TYPE    #include <armadillo>
        | Include    program      TYPE    using namespace arma ;
  1   1 Funcs      program      TYPE    unamed
  1   1 | Main       func_common  TYPE    main
  1   1 | | Declares   func_return  TYPE
  1   1 | | | Var        unknown      (int)   a
  1   1 | | Returns    func_return  TYPE
  1   1 | | Params     func_return  TYPE
  1   1 | | Block      code_block   TYPE
  1   1 | | | Assign     unknown      TYPE
  1   1 | | | | Var        unknown      (int)   a
  1   5 | | | | Plus       expression   int
  1   5 | | | | | Int        int          int
  1   7 | | | | | Int        int          int
        Inlines    program      TYPE    unamed
        Structs    program      TYPE    unamed
        Headers    program      TYPE    unamed
        Log        program      TYPE    unamed
        | Error      program      TYPE    Var:0
```

There is quite a lot going on in this picture. First of all, each line represents a node. The columns represents respectively

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

For more on node meta-information, see `node`.

In addition to include nodes that represents the translation, there is a bit of meta-information in the code. This will not be assigned neither line number nor cursor number. The remaining lines are the nodes used to do the actual code translation. The can be interpreted as follows:

- The program consists of a collection of functions `Funcs`
- The collection of funcs contains one main function `Main`
- The main function contains information about `declarations`, `return values` and `parameters` in the first three node children.
- The fourth child is the `code block` which contains the function content.
- The code block contains one code line, an `assignment`.
- The assignment contains a left hand side `variable` and an expression right hand side `addition operator`
- The `addition` contains the two `integers`.

The class of each node is determined as the are created as the Matlab code is translated. The translation handler and the datatype however varies a bit. Some are fixed, like that *Program* is handled by the *program* translation rule or that *Int* have the datatype *int*. Others, like the variable *Var* can change upon how the configuration is set up. Intuitively enough, if datatype is set to *int*, then the translation handler will follow and also be *int*:

```
>>> print mc.qtree("a = 2+2", suggest=True)
        Program    program      TYPE    unamed
        Includes   program      TYPE
        | Include    program      TYPE    #include <armadillo>
        | Include    program      TYPE    using namespace arma ;
  1    1 Funcs      program      TYPE    unamed
  1    1 | Main       func_common  TYPE    main
  1    1 | | Declares   func_return  int
  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     unknown      TYPE
  1    1 | | | | Var        int          int     a
  1    5 | | | | Plus       expression   int
  1    5 | | | | | Int        int          int
  1    7 | | | | | Int        int          int
        Inlines    program      TYPE    unamed
        Structs    program      TYPE    unamed
        Headers    program      TYPE    unamed
        Log        program      TYPE    unamed
```

In other words, there are for these nodes, multiple translation for depending on context. This is important to achieve the desired behavior.

Module	Description
builder	Contains the *Builder* class that is used to convert Matlab code into node tree representation
`constants`	A collection of usefull constants used by various interpretation rules
`assign`	Support functions for variable assignments
`branches`	Support functions for if-tests, loops, try-blocks
`codeblock`	Support functions for filling in codeblock content
`expression`	Support functions for filling in expression content
`findend`	Look-ahead functions for finding the end of various code structures
`functions`	Support functions for constructing Functions, both explicit and lambda, and program content
`identify`	Look-ahead functions for identifying ambigous contexts
`iterate`	Support functions for segmentation of lists
`misc`	Miscelenious support functions
`variables`	Support functions for constructing various variables

3.1 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_** 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 `tree` to build a full toke tree. The result is as follows:

```
>>> print builder
        Project    program      TYPE    project
        | Program    program      TYPE    file1.m
        | | Includes   program      TYPE
        | | | Include    program      TYPE    #include <armadillo>
        | | | Include    program      TYPE    using namespace arma ;
  1   1 | | Funcs      program      TYPE    file1.m
  1   1 | | | Main       func_common  TYPE    main
  1   1 | | | | Declares   func_return  TYPE
  1   1 | | | | | Var        unknown      TYPE    a
  1   1 | | | | Returns    func_return  TYPE
  1   1 | | | | Params     func_return  TYPE
  1   1 | | | | Block      code_block   TYPE
  1   1 | | | | | Assign     unknown      TYPE
  1   1 | | | | | | Var        unknown      TYPE    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
```

If 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 relevant data types configure. To configure datatypes, use the *configure* method:

```
>>> builder.configure(suggest=True)
>>> print builder
```

```
        Project    program      TYPE    project
        | Program    program      TYPE    file1.m
        | | Includes   program      TYPE
        | | | Include    program      TYPE    #include <armadillo>
        | | | Include    program      TYPE    using namespace arma ;
  1   1 | | Funcs      program      TYPE    file1.m
  1   1 | | | Main       func_common  TYPE    main
  1   1 | | | | Declares   func_return  int
  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     unknown      TYPE
  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:

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

class `matlab2cpp.`**`Builder`**(*disp=False*, *comments=True*, ***kws*)

Convert Matlab-code to a tree of nodes.

Method	Description
configure()	Use assigned values and suggestion engine to fill in datatypes
load()	Load code with a given name
syntaxerror()	Throw an apropriate SyntaxError for the Matlab code

`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    program      TYPE    project
         | Program    program      TYPE    unnamed.m
         | | Includes   program      TYPE
         | | | Include    program      TYPE    #include <armadillo>
         | | | Include    program      TYPE    using namespace arma ;
   1   1 | | Funcs      program      TYPE    unnamed.m
   1   1 | | | Main       func_common  TYPE    main
   1   1 | | | | Declares   func_return  TYPE
   1   1 | | | | | Var        unknown      TYPE    a
   1   1 | | | | | Var        unknown      TYPE    b
   1   1 | | | | | Var        unknown      TYPE    c
   1   1 | | | | Returns    func_return  TYPE
   1   1 | | | | Params     func_return  TYPE
   1   1 | | | | Block      code_block   TYPE
   1   1 | | | | | Assign     unknown      TYPE
   1   1 | | | | | | Var        unknown      TYPE    a
   1   3 | | | | | | Int        int          int
   1   6 | | | | | Assign     unknown      TYPE
   1   6 | | | | | | Var        unknown      TYPE    b
   1   8 | | | | | | Float      double       double
   1  12 | | | | | Assign     unknown      TYPE
   1  12 | | | | | | Var        unknown      TYPE    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
>>> builder.configure(suggest=True)
>>> print builder
         Project    program      TYPE    project
         | Program    program      TYPE    unnamed.m
         | | Includes   program      TYPE
         | | | Include    program      TYPE    #include <armadillo>
         | | | Include    program      TYPE    using namespace arma ;
   1   1 | | Funcs      program      TYPE    unnamed.m
   1   1 | | | Main       func_common  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     unknown      TYPE
   1   1 | | | | | | Var        int          int     a
   1   3 | | | | | | Int        int          int
   1   6 | | | | | Assign     unknown      TYPE
   1   6 | | | | | | Var        double       double  b
   1   8 | | | | | | Float      double       double
   1  12 | | | | | Assign     unknown      TYPE
   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
```

load(*name*, *code*)

Load a Matlab code into the node tree.

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()
>>> print builder
        Project    program      TYPE    project
>>> builder.load("unnamed.m", "")
>>> print builder
        Project    program      TYPE    project
        | Program    program      TYPE    unnamed.m
        | | Includes   program      TYPE
        | | | Include    program      TYPE    #include <armadillo>
        | | | Include    program      TYPE    using namespace arma ;
  1   1 | | Funcs      program      TYPE    unnamed.m
        | | Inlines    program      TYPE    unnamed.m
        | | Structs    program      TYPE    unnamed.m
        | | Headers    program      TYPE    unnamed.m
        | | Log        program      TYPE    unnamed.m
```

syntaxerror (*cur*, *text*)
Raise an SyntaxError related to the Matlab code.

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
```

CHAPTER

FOUR

CONFIGURING DATATYPES

One of the translation challenges is how each variable type 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.

4.1 Variable types

Even though not always relevant, all node has it's own datatype. It can be referenced by *node.type* and can be inserted as placeholder through *%(type)s*. Note however that there are many data types available. The options for valid variable types are listed in the supplement file. They can be roughly split into two groups: **numerical** and **non-numerical** types. The numerical types are as follows:

	unsigned int	*int*	*float*	*double*	*complex*
scalar	uword	int	float	double	cx_complex
vector	uvec	ivec	fvec	vec	cx_vec
row-vector	urowvec	irowvec	frowvec	rowvec	cx_rowvec
matrix	umat	imat	fmat	mat	cx_mat
cube	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
char	Single text character
string	Text string
struct	Struct container
structs	Struct array container
func_lambda	Anonymous function

4.1.1 Numerical datatypes

Most of the allowed datatypes are numerical values with varying type-space and dimensionality. So when addressing a numerical value, the nodes attributes *node.dim* and node.mem' can often be useful, since they contain direct information about that information.

Example

```
>>> node = mc.collection.Var(None, "name", type="ivec")
>>> print node.type
ivec
>>> print node.dim, node.mem
1 1
```

These nodes also support dynamic rewriting of the datatype. Continuing our example:

```
>>> node.dim = 3
>>> print node.type
imat
>>> node.mem = 4
>>> print node.type
cx_mat
```

The connection between the nummerical values and datatypes is as follows:

mem	Description		*dim*	Description
0	unsiged int		0	scalar
1	integer		1	(col-)vector
2	float		2	row-vector
3	double		3	matrix
4	complex		4	cubu

4.2 Manual assignment

If not addressed, the program will not assign datatype types to the variables. It is possible to navigate the node tree and assign the variables one by one (see `node`), but that sould be very cumbersome.

4.2.1 Variable types

The supplement file consists in practice of only variable *scope* which is a nested dictionary. The outer shell of scope has string keys that reference the name of each function, and declared struct and cells. The values are dictionaries that represents the inner shell. The inner shell has string keys that refer to the local variable names string values that represents the variable type.

4.2.2 Anonymous functions

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

```
>>> print mc.qhpp("function f(); g = @(x) x^2; g(4)", suggest=True)
#include <armadillo>
using namespace arma ;

void f()
{
  std::function<int(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)", suggest=True)
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.

4.2.3 Data structures

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 suppliment file, the local variable *a* will be assigned as a *struct*. In addition, since the struct has content, the suppliment 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 ;',
]
```

Given that the data structure is in the form of an array, the process is similar to a single element. There is only two differences. In the translation, 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, and the number should be increased. To adjust this number, the suppliment 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 ;',
]
```

4.2.4 Suggestions

4.2.5 Includes

4.3 Suggestion engine

Consider the following program where the datatypes are unassigned:

```
>>> 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.

The supplement file created by *mconvert* or :py::~*matlab2cpp.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 ;',
]
```

To the right of the type assignment, the program will add a suggestion to aid the user. The next time the *mconvert*-script is run, the inserted values will be imported and used.

The user can automatically populate the datatypes to some degree by using the *-s* or *–suggestions* flag (or using the *suggest=True* flag for *mc.qpy*):

```
>>> 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 suggestions are created through an iterative process. 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 and addition between *int* and *double*. To not loose 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.

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

CHAPTER

FIVE

NODE BEHAVIOR

General definition of a node representation of code segment.

During translation an instance of the current state in form of a Node will be provided. It has a set of properties and module functions that can be used to retrieve and manipulate the state of the nodes.

To illustrate both nodes and relationship we introduce the following example:

```
>>> builder = mc.Builder()
>>> program = builder.load("unnamed", "function y=f(x); y=x+4")
>>> mc.set_variables(program, {"f" : {"x": "int", "y": "double"}})
>>> builder.configure()
>>> program.translate()
>>> print mc.qtree(program)
        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  double  f
  1   1 | | Declares   func_return  double
  1   1 | | | Var        double       double  y
  1   1 | | Returns    func_return  double
  1  10 | | | Var        double       double  y
  1  13 | | Params     func_return  int
  1  14 | | | Var        int          int     x
  1  16 | | Block      code_block   TYPE
  1  18 | | | Assign     unknown      TYPE
  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
```

5.1 Navigating the tree

The program is the root of the tree. To move down from there can be done using indexing. All nodes are interable, allowing standard Python movement:

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

Moving upwards in the tree is done using the *parent* reference:

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

The *parent* reference is not the only reference available. These include:

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

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.

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*.

parent [Node] A reference to the direct node parent above the current one.

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.

5.2 Interactive node attributes

The following node attributes are interactive in the sense that they can both be observed and changed. Many of them are also interconnected such that change will automatically affect others. For example:

```
>>> print var_y.type
double
>>> print var_y.dim
0
>>> var_y.dim = 1
>>> print var_y.type
vec
```

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*.

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*.

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

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.

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*

5.3 Static node attributes

The following attributes are automatically generated by the software and are provided for convenience.

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

cls [str] A string representation of the class name. Avalable in the string format as *%(class)s*

code [str] The code that concived this node.

cur [int] The index to the position in the code where this node was concived. It takes the value 0 for nodes not created from code.

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*

line [int] The codeline number in original code where this node was concived. It takes the value 0 for nodes not created from code.

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.

ret [str, tuple] The raw translation of the node. Same as *node.str*, but on the exact form the tranlsation 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.

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

5.4 Context manipulation

auxillary [Push subtree up to its own code line and save it to an] auxillary variable.

resize Resize handler. Used to bring a cube structure down to a matrix and vector properly.

include warning error suggest_datatype

5.5 Tree processing

translate summary

5.6 References to other nodes

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

`auxiliary`(*type=None*, *convert=False*)
Create a auxiliary variablele and move actual calcuations to own line.

Parameters **`type`** (*str, None*) – If provided, auxiliary variable type will be converted

`flatten`(*ordered=False*, *reverse=False*, *inverse=False*)
Return a list of all nodes

Tree:

A |B C

/| |

D E 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

ri : 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

`summary`(*args=None*)
Generate a summary of the tree structure with some meta-information.

Returns Summary of the node tree

Return type str

See also:

mc.qtree

`translate`(*opt=None*, *only=False*)
Generate code

CHAPTER

SIX

NODE COLLECTION

All the various different kinds of nodes

`matlab2cpp.collection.`**`All`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Assign`**(*parent*, *name='‘*, *backend='unknown'*, *value='‘*, *type='TYPE'*, *pointer=0*, *line=None*, *cur=None*, *code=None*, *file=None*)

`matlab2cpp.collection.`**`Assigns`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Band`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Bcomment`**(*parent*, *value*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Block`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Bor`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Branch`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Break`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Case`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Catch`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Cell`**(*parent*, *backend='cell'*, ***kws*)

`matlab2cpp.collection.`**`Cget`**(*parent*, *name*, *backend='cell'*, ***kws*)

`matlab2cpp.collection.`**`Colon`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Cond`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Condline`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Counter`**(*parent*, *name*, *value*, *backend='structs'*, *type='structs'*, ***kws*)

`matlab2cpp.collection.`**`Cset`**(*parent*, *name*, *backend='cell'*, ***kws*)

`matlab2cpp.collection.`**`Ctranspose`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Cvar`**(*parent*, *name*, *backend='cell'*, ***kws*)

`matlab2cpp.collection.`**`Declare`**(*parent*, *name='‘*, *backend='unknown'*, *value='‘*, *type='TYPE'*, *pointer=0*, *line=None*, *cur=None*, *code=None*, *file=None*)

`matlab2cpp.collection.`**`Declares`**(*parent*, *name='‘*, *backend='unknown'*, *value='‘*, *type='TYPE'*, *pointer=0*, *line=None*, *cur=None*, *code=None*, *file=None*)

`matlab2cpp.collection.`**`Ecomment`**(*parent*, *value*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Elementdivision`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Elexp`**(*parent*, *backend='expression'*, ***kws*) `matlab2cpp.collection.`**`Elif`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Elmul`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Else`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`End`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Eq`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Error`**(*parent*, *name*, *value*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Exp`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Expr`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Fget`**(*parent*, *name*, *value*, *backend='struct'*, ***kws*)

`matlab2cpp.collection.`**`Float`**(*parent*, *value*, *backend='double'*, *type='double'*, ***kws*)

`matlab2cpp.collection.`**`For`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Fset`**(*parent*, *name*, *value*, *backend='struct'*, ***kws*)

`matlab2cpp.collection.`**`Func`**(*parent*, *name=''*, *backend='unknown'*, *value=''*, *type='TYPE'*, *pointer=0*, *line=None*, *cur=None*, *code=None*, *file=None*)

`matlab2cpp.collection.`**`Funcs`**(*parent*, *backend='program'*, *line=1*, ***kws*)

`matlab2cpp.collection.`**`Fvar`**(*parent*, *name*, *value*, *backend='struct'*, ***kws*)

`matlab2cpp.collection.`**`Ge`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Get`**(*parent*, *name*, ***kws*)

`matlab2cpp.collection.`**`Gt`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Header`**(*parent*, *name*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Headers`**(*parent*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`If`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Imag`**(*parent*, *value*, *backend='cx_double'*, *type='cx_double'*, ***kws*)

`matlab2cpp.collection.`**`Include`**(*parent*, *name*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Includes`**(*parent*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Inline`**(*parent*, *name*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Inlines`**(*parent*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Int`**(*parent*, *value*, *backend='int'*, *type='int'*, ***kws*)

`matlab2cpp.collection.`**`Lambda`**(*parent*, *name=''*, *backend='func_lambda'*, ***kws*)

`matlab2cpp.collection.`**`Land`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Lcomment`**(*parent*, *value*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Le`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Leftelementdivision`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Leftmatrixdivision`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Log`**(*parent*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Lor`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Lt`**(*parent*, *backend='expression'*, ***kws*) `matlab2cpp.collection.`**`Main`**(*parent*, *name='main'*, *backend='func_common'*, ***kws*)

`matlab2cpp.collection.`**`Matrix`**(*parent*, *backend='matrix'*, ***kws*)

`matlab2cpp.collection.`**`Matrixdivision`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Minus`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Mul`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Ne`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Neg`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Nget`**(*parent*, *name*, *backend='struct'*, ***kws*)

`matlab2cpp.collection.`**`Not`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Nset`**(*parent*, *name*, *backend='struct'*, ***kws*)

`matlab2cpp.collection.`**`Opr`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Otherwise`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Params`**(*parent*, *name='‘*, *backend='unknown'*, *value='‘*, *type='TYPE'*, *pointer=0*, *line=None*, *cur=None*, *code=None*, *file=None*)

`matlab2cpp.collection.`**`Paren`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Plus`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Program`**(*parent*, *name*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Project`**(*backend='program'*, *name='project'*, *cur=0*, *line=0*, *code='‘*, *file='unnamed'*, ***kws*)

`matlab2cpp.collection.`**`Resize`**(*parent*, *backend='cube_common'*, ***kws*)

`matlab2cpp.collection.`**`Return`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Returns`**(*parent*, *name='‘*, *backend='unknown'*, *value='‘*, *type='TYPE'*, *pointer=0*, *line=None*, *cur=None*, *code=None*, *file=None*)

`matlab2cpp.collection.`**`Set`**(*parent*, *name*, ***kws*)

`matlab2cpp.collection.`**`Sget`**(*parent*, *name*, *value*, *backend='structs'*, ***kws*)

`matlab2cpp.collection.`**`Sset`**(*parent*, *name*, *value*, *backend='structs'*, ***kws*)

`matlab2cpp.collection.`**`Statement`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`String`**(*parent*, *value*, *backend='string'*, *type='string'*, ***kws*)

`matlab2cpp.collection.`**`Struct`**(*parent*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Structs`**(*parent*, *backend='program'*, ***kws*)

`matlab2cpp.collection.`**`Switch`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Transpose`**(*parent*, *backend='expression'*, ***kws*)

`matlab2cpp.collection.`**`Try`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Tryblock`**(*parent*, *backend='code_block'*, ***kws*)

`matlab2cpp.collection.`**`Var`**(*parent*, *name*, ***kws*)

`matlab2cpp.collection.`**`Vector`**(*parent*, *backend='matrix'*, ***kws*)

`matlab2cpp.collection.`**`Warning`**(*parent*, *name*, *value*, *backend='program'*, ***kws*) `matlab2cpp.collection.`**`While`**(*parent*, *backend='code_block'*, ***kws*)

CHAPTER

SEVEN

TRANSLATION RULES

Given a fully configured node-tree, the job can start to make a translation. The translation is the application of a set of translation rules. The rules are collected in the folder *rules*. All files contained in the folder tarts with the prefix _ (to avoid conflicting names with the python interpreter) and a *.py* extension indicating that the code just a traditional python script.

Starting with the simplest form of translation is to define a simple string. For .. rubric:: example

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

The name *Int* (with capital letter) represents the node the rule is applicable for, 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 .. rubric:: example

```
>>> 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 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+1"
>>> print mc.qscript("5", Int=Int)
5+1 ;
```

There are also other place holder names. For example, variables *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 node 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. For example:

```
>>> print mc.qscript("2+3")
2+3 ;
```

Here we have an addition node *Plus*, with two children, both *Int*. They are respectively index 0 and 1. We can use this information to manipulate how the addition works:

```
>>> 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 node might be variable. 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 noting in front, in between each node child, a "+" should be used, 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 full extent of how the system uses string values. However, in practice, they are not used much. Instead functions are used. They are defined with the same name the class (the software figures the details out what is what). This function should always take a single *node* argument which represents the current node in the node tree. The function should return either a string or tuple in the same way as the directly defined string and tuple are define so far. 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 ;
```

CHAPTER

EIGHT

INSERTED CODESNIPPETS