Developer Guide
Version 0.5.7

June 16, 2023

Contents

Chapter 1

Introduction

This Developer Manual aims to assist users interested in extending or improving *morpho.* Developers interested in adding new features to *morpho* can utilize one of the following mechanisms for expansion:

- **Modules** are written in the *morpho* language and are loaded with the `import` keyword. Creating a module is no different than writing a *morpho* program!
- **Extensions** are written in C or C++ using the *morpho* C API. These are also loaded with the `import` keyword; the distinction between modules and extensions is purposefully not visible to the user; a module could be reimplemented as an extension with the same interface, for example
- **Contributing to the *morpho* source code.** Changes to the core data types, improvements to the compiler, etc. could be incorporated into morpho directly. We highly recommend connecting to the *morpho* developers before doing this to check if the idea is already being worked on, or whether there is guidance or advice on how specific features should work.

Modules and extensions should be distributed in their own git repository as a package. See Chapter 2 for further information.

We also recommend contributers look at the `CONTRIBUTING.md` and `CODE_OF_CONDUCT.md` documents in the *morpho* git repository for information and advice about contributing to *morpho*, as well as how ethical standards for participation in our community.

This developer guide is only partially complete and represents a work in progress: We are gradually adding in helpful information to assist developers.

Chapter 2

Morpho packages

To facilitate convenient distribution, *morpho* supports a lightweight notion of *packages*, which comprise some functionality. A package is simply a git repository that contains some or all of the following file structure:

/share/modules for `.morpho` files that define a module

/share/help for `.md` files containing interactive help (see below).

/lib for compiled extensions (these may be produced during installation).

/src for source files

/examples for examples

/test for test files

/manual if a package is sufficiently complex to require a manual

README.md information about the package, installation etc.

Morpho searches both its base installation and all known packages when trying to locate resources. We anticipate that at some future point a package management system similar to pip will be created; the structure above is intended to be sufficiently simple that different installation approaches could be supported.

We recommmend naming your package with the *morpho-* prefix. Please let the morpho development team know about interesting packages!

2.1 *Morpho* help files

Morpho's interactive help system utilizes a subset of the Markdown plain text formatting system. Help files should be put in the `/share/help` folder of your package so that morpho can find them.

Each entry begins with a heading, for example:

```
# Entry
```

Using different heading levels indicates to *morpho* that a topic should be included as a subtopic. Here, the two heading level 2 entries become subtopics of the main topic, which uses heading level 1:

```
# Main topic

## Subtopic 1

## Subtopic 2
```

Morpho's help system supports basic formatting, including emphasized text:

```
This is *emphasized* text.
```

and lists can be included like so:

```
* List entry
* Another list entry
```

Code can be typeset inline,

```
Grave accents are used to delimit ‘some code‘
```

or can be included in a block by indenting the code:

```
    for (i in 1..10) print i
```

The terminal viewer will syntax color this automatically.

Morpho (ab)uses the Markdown hyperlink sytax to encode control features. To specify a tag or keyword for a help entry, create a hyperlink where the label begins with the word `tag`, and include the keyword you'd like to use in the target as follows:

```
## Min
[tagmin]: # (min)

Finds the minimum value...
```

This unusual syntax is necessary as Markdown lacks comments or syntax for metadata, and we use hyperlinks to encode text in a way that is valid Markdown but remains transparent to regular Markdown viewers. The # is a valid URL target, and the construction in effect 'hides' the text in parentheses. Since hyperlink labels (the part in square brackets) must be unique per file, add any text you like, typically the name of the tag, after the word `tag`.

Similarly, to tell the help viewer to show a table of subtopics after an entry, add a line like this:

```
[showsubtopics]: # (subtopics)
```

Any characters after `showsubtopics` in the label are ignored, so you can add additional characters to ensure a unique label.

Part I Coding in *morpho*

Chapter 3

The *morpho* debugger

Morpho provides a debugger that allows you to pause execution of a program, examine the state of variables and registers, and continue. To enable it run morpho with the `-debug` command line switch. Note that there is a performance penalty for running with debugging enabled.

You may set hard breakpoints in your code—places where morpho will always pause—by placing an `@` symbol. For example:

```
@
print "Hello World"
```

will break immediately before executing the print statement. When morpho reaches one of these breakpoints, it enters debugging mode:

```
---Morpho debugger---
Type '?' or 'h' for help.
Breakpoint in global at line 5 [Instruction 15]
@>
```

The `@>` prompt reminds you that you're in the debugger rather than in interactive mode. You can then perform a number of commands to understand the current state of the virtual machine, set additional breakpoints, examine the contents of variables and registers, etc. Most commands have a long form, e.g. "break" or "clear" and a short form "b" or "x" respectively. Debugger commands are largely consistent with those for the `gdb` tool.

Some of the debugging features require knowledge of how *morpho's* virtual machine works, which is documented in Chapter 8.

3.1 Debugging commands

3.1 Break

The `b` command sets a breakpoint:

`b lineno` Break at a given line number.

`b *instruction` Break at a given instruction.

`b functionname` Break at a given function.

`b Class.methodname` Break at a given method.

3.1 Continue

Continues program execution, leaving the debugger.

3.1 Disassemble

Displays disassembly for the current line of code.

3.1 Garbage collect

Forces a garbage collection.

3.1 Clear

Clears a breakpoint. The syntax is the same as for `b`. Note the abbreviation is `x` not `c`.

3.1 Info

Info reports on various features of the virtual machine.

`i address n` Displays the physical address of the object in register *n*. *[This is primarily useful when debugging morpho itself]*

`i break` Displays all active breakpoints

`i globals` Displays the value of all globals

`i global n` Displays the contents of global *n*

`i registers` Displays registers for the current function call

`i stack` Displays the current stack

`i help` Displays a list of valid info commands

3.1 List

Prints a program listing of the lines around the current execution point.

3.1 Print

Prints the value of variables.

`p symbol` Prints the value of a given symbol

`p` Print all currently visible symbols

3.1 Set

Prints the value of a variable or register.

`set r n = <expr>` Sets the value of register *n* to be <*expr*>.

`set <symbol> = <expr>` Sets the value of variable <*symbol*> to be <*expr*>.

Expressions must be simple constant values.

3.1 Quit

Terminates program execution.

3.1 Step

Continues execution, but returns to the debugger at the next line.

3.1 Trace

Shows the current execution trace, i.e. the list of functions and method calls that the program has made to get to the current point.

Chapter 4

The *morpho* profiler

Morpho provides a simple profiler to help identify bottlenecks in the program. To use it run morpho with the `-profile` command line switch. As the program runs, a separate monitor thread runs independently and samples the state of the morpho virtual machine at regular intervals, deducing at each time which function or method is in use. At the end of program execution, the profiler prints a report. A sample run[1] might produce something like:

```
===Profiler output: Execution took 51.019 seconds with 272450 samples===
issame                              32.98% [89866 samples]
Delaunay.dedup                      15.63% [42580 samples]
(garbage collector)                 13.41% [36528 samples]
List.ismember                       7.16% [19518 samples]
Delaunay.triangulate                6.40% [17450 samples]
List.enumerate                      3.58% [9750 samples]
Show.trianglecomplexobjectdata      2.59% [7065 samples]
Circumsphere.init                   2.24% [6091 samples]
OneSidedHookeElasticity.integrandfn 1.77% [4834 samples]
Matrix.column                       1.25% [3412 samples]
(anonymous)                         1.25% [3406 samples]
List.count                          1.01% [2758 samples]
Range.enumerate                     0.90% [2451 samples]
...
```

On the first line, the profiler reports the time elapsed between the start and end of executing the program (which does not include compilation time) and the total number of samples taken. In subsequent lines, the profiler reports the name of a function or method, the number of samples in which the virtual machine was observed to be in that function, and the overall fraction of samples as a percentage. The list is sorted so that the most common function is reported first. The profiler reports on both user-implemented functions and morpho functions and methods that are implemented in C (but visible to the user).

There are some special entries: anonymous functions are reported as `(anonymous)`; time in the global context, i.e. outside of a function or method is reported as `(global)`; time spent in the garbage collector is reported as `(garbage collector)`, here on the third line. Garbage collection in this example is frequently $\sim 10\%$ of execution time; if it becoms significantly higher, this may suggest your program is created too many temporary objects.

How to interpret and act on profiler data is something of an art form. In the above example, the largest

[1] on the *morpho* example `examples/meshgen/sphere.morpho`

fraction of execution time was spent in a relatively function, `issame`, that compared two objects. An obvious strategy would have been to simply reimplement the function in C, which would have undoubtedly improved the performance. However, on inspecting the code it was realized that `issame` was actually being called by `Delaunay.dedup` to remove entries from a data structure, and that by using a different data structure this step could be entirely eliminated providing a significant performance gain.

Hence, optimization involves not only thinking about the performance of individual pieces of code, but also the data structures and algorithms being used. The profiler simply directs the programmer's attention to the most time consuming bits of code to avoid optimizing sections of code that aren't called frequently.

Part II Coding in C

Chapter 5

Writing an extension

Morpho extensions are dynamic libraries that are loaded at runtime. From the user's perspective, they work just like modules through the `import` statement:

```
import myextension
```

When the compiler encounters an import statement, it first searches to see if a valid extension can be found with that name. If so, the extension is loaded and compilation continues.

Extensions are implemented in C or any language that can be linked with C. A minimal extension looks like this:

```
// myextension.c

#include <stdio.h>
#include <morpho/morpho.h>
#include <morpho/builtin.h>

value myfunc(vm *v, int nargs, value *args) {
    printf("Hello world!\n");
    return MORPHO_NIL;
}

void myextension_initialize(void) {
    builtin_addfunction("myfunc", myfunc, BUILTIN_FLAGSEMPTY);
}

void myextension_finalize(void) {
}
```

All *morpho* extensions **must** provide an initialize function, and it **must** be named `EXTENSIONNAME_initialize`. In this function, you should call the morpho API to define functions and classes implemented by your extension, and set up any global data as necessary. Here, we add a function to the runtime that will be visible to user code as `myfunc`.

Morpho extensions **may** but are not required to provide a finalize function, with a similar naming convention to the initializer. This function should deallocate or close anything created by your extension that isn't visible to the *morpho* runtime. Here, the function data structures are handled by the morpho runtime so there's no finalization to do.

The remaining code implements your extension. Here, we implement a very simple function that conforms to the interface for a "builtin" function. The function just prints some text and returns `nil`.

Compiling an extension

To compile the above code, it's necessary to ensure that the morpho header files are visible to your compiler. They could be copied from the morpho git to `/usr/local/include/morpho` for example *[we intend to automate this as part of installation in future releases]*.

You need to compile this code as a dynamic library. For example on the macOS with clang,

```
cc -undefined dynamic_lookup -dynamiclib -o myextension.dylib myextension.c
```

The `-dynamiclib` option indicates that the target should be a dynamic library. The `-undefined dynamic_lookup` option indicates to the linker that any undefined references should be resolved at runtime.

Packaging an extension

As for *morpho* modules, we advise hosting your extension in a git repository with *morpho-* as the prefix and with the file structure as suggested in chapter 2. We recommend including the C source files in `/src` and compiling your extension to `/lib`, where it can be found by *morpho*. We highly recommend including interactive help files in `/share/help` and examples as well. All extensions should have a `README.md` explaining what the extension is for and how the user should install it.

In the imminent future, we anticipate providing an automated way to build extensions that should help with installation. For now, the above recommendations should ensure your basic file structure is future-proof.

We also note that the C API is not yet stable. As we gain experience writing extensions and identify common needs, we anticipate improving the API. We welcome your feedback.

Chapter 6

The *morpho* C API

Morpho, like many languages, is implemented in C for performance reasons. Extensions to morpho can also be written in C, or in a language that can link with C.

6.1 Basic data types

6.1 Value

A `value` is the most basic data type in *morpho*. At any time, a value can contain any *one* of:

- A signed integer, equivalent to an `int32`.
- A `double` precision floating point number.
- A pointer to an object.
- A boolean value indicating `true` or `false`.
- The value `nil` representing no information.

The structure of a value is kept opaque for performance reasons; setting and getting a value must be done through macros provided by *morpho:*

- **Initialize a `value` with a literal.** Macros provided include `MORPHO_NIL`, `MORPHO_TRUE`, `MORPHO_FALSE`. You can also use
- **Convert a C type to a `value`.** Use the macros `MORPHO_INTEGER`, `MORPHO_FLOAT`, `MORPHO_OBJECT`, `MORPHO_BOOL`. to do this.
- **Convert a `value` to a C type.** `MORPHO_GETINTEGERVALUE`, `MORPHO_GETFLOATVALUE`, `MORPHO_GETBOOLVALUE`, `MORPHO_GETOBJECT`.
- **Test whether a value is of a certain type.** Use `MORPHO_ISNIL`, `MORPHO_ISINTEGER`, `MORPHO_ISFLOAT`, `MORPHO_ISOBJECT`, `MORPHO_ISBOOL`. Do **not** use a direct comparison with a literal, because the value implementation is intentionally opaque and such comparisons may fail. In other words, do this

  ```
  if (MORPHO_ISNIL(val)) ... // Correct
  ```

  and not

```
if (val==MORPHO_NIL) ... // Incorrect
```

Additionally, a number of utility functions exist to compare values:

- `MORPHO_ISEQUAL(a, b)` tests if two values are equal. For strings, etc. this involves a detailed comparison.
- `MORPHO_ISSAME(a, b)` tests if two values refer to the same object (or are equal if they are not objects). This macro is intended to be faster than ISEQUAL.
- `morpho_ofsametype(value a, value b)` returns true if a and b have the same type.
- `MORPHO_ISNUMBER(value a)` returns true is a is a number (i.e. integer or float).
- `morpho_valuetoint`, `morpho_valuetofloat`, `MORPHO_INTEGERTOFLOAT`, `MORPHO_FLOATTOINTEGER` provide conversion between types.
- `MORPHO_ISFALSE`, `MORPHO_ISTRUE` test if a value is true or false.

6.1 Objects

Objects are data types that require memory allocation, and are implemented as C `struct`s that always begin with a field of type `object`. This design enables *type munging*, i.e. casting any object to a generic `object` type, but with the ability to infer the type of an object at a later point. To store a pointer to an object in a `value`,

```
value v = MORPHO_OBJECT(objectpointer)
```

Many macros are provided to detect what kind of object is present, for example `MORPHO_ISSTRING`, `MORPHO_ISLIST`, `MORPHO_ISMATRIX`, `MORPHO_ISSPARSE`. Once you have determined the type of an object, you can then use macros like `MORPHO_GETSTRING`, `MORPHO_GETLIST` or similar to retrieve a pointer of the correct type. Convenience macros such as `MORPHO_GETCSTRING` are provided to enable easy access to object fields from a `value`.

New types of object can be defined; see Section 6.4.

6.1 Varray

Variable length arrays are arrays that dynamically adjust in size as new members are added. They're a very useful type that differs only by the type contained in them. Hence, `common.h` provides two convenient macros to create them for a specific type. Suppose we want to define a varray of integers: to do so, we would include in an appropriate `.h` file the statement:

```
DECLARE_VARRAY(integer, int)
```

and then

```
DEFINE_VARRAY(integer, int)
```

in our `.c` file. These definitions would create

```
varray_integerinit(varray_integer *v);
varray_integeradd(varray_integer *v, int data[], int count);
varray_integerwrite(varray_integer *v, int data);
varray_integerclear(varray_integer *v);
```

Where we want to use an integer varray, we would write something like

```
varray_integer v;
...
/* Initialize the varray */
varray_integerinit(&v);
...
/* Write an element to the varray */
varray_integerwrite(&v, 1);
...
/* Deinitialize the varray */
varray_integerclear(&v);
...
```

6.2 Implementing a new function

Creating a new builtin morpho function requires the programmer to write a function in C with the following interface:

```
value customfunction(vm *v, int nargs, value *args);
```

Your function will be passed an opaque reference to the virtual machine `v`, the number of arguments that the function was called with `nargs`, and a list of arguments `args`. You must **not** access the argument list directly, but rather use the macro `MORPHO_GETARG(args, n)` to get the `n`th argument. You must return a `value`, which may be `MORPHO_NIL`. For example, a simple implementation of the `sin` trigonometric function might look like this:

```
value sin_fn(vm *v, int nargs, value *args) {
        if (nargs!=1) /* Raise error */;
        double input;
        if (morpho_valuetofloat(MORPHO_GETARG(args, 0)) {
                return MORPHO_FLOAT(sin(input));
        } else /* Raise error */
        return MORPHO_NIL;
}
```

Morpho C-functions are naturally variadic; if your function is called with the wrong number of arguments, you are responsible for raising an error as described in Section 6.6.

If you create any new objects in your function, you **must** bind them to the virtual machine as described in Section 6.7.

To make the function visible to morpho, call `builtin_addfunction` in your extension's initialization function, e.g.

```
builtin_addfunction("sin", sin_fn, BUILTIN_FLAGSEMPTY);
```

From morpho, the user can then use the new `sin` function as if it were a regular function.

6.2 Optional parameters

Your function may accept optional parameters just as for regular morpho functions. The library function `builtin_options` enables you to retrieve these values, as in the below example:

```
static value fnoption1;
static value fnoption2;

init(void) { // Initialization function called when your extension is run
        fnoption1=builtin_internsymbolascstring("opt1");
        fnoption2=builtin_internsymbolascstring("opt2");
}

value opt_fn(vm *v, int nargs, value *args) {
        int nfixed; // Number of fixed args.
        value optval1 = MORPHO_NIL;        // Declare values to receive
        value optval2 = MORPHO_INTEGER(2); // optional parameters
        if (!builtin_options(v, nargs, args, // Pass through
                             &nfixed,        // Number of fixed parameters is returned
                        2,              // Number of possible optional args
                        fnoption1, &optval1, // Pairs of symbols and values to receive th
                        fnoption2, &optval2) return MORPHO_NIL;

        // ...
}
```

Symbols for the optional parameters must be declared in your initialization function by calling `builtin_internsymbolascst`. These are typically tracked in a global variable.

You call `builtin_options` with the arguments passed to your function, `v`, `nargs` and `args`, an pointer of type int* to receive the number of fixed parameters detected, and then a list of optional parameters and their associated symbols. If `builtin_options` detects that a particular optional argument has been supplied by the user, the corresponding `value` is updated.

You must check the return value of `builtin_options`, which returns `false` on failure. Where this occurs, you must return as quickly as possible from your function, returning `MORPHO_NIL`.

6.3 Implementing a new class

There are two implementation patterns to define a new `morpho` class:

1. The object uses an `objectinstance` and all information is stored in properties of the object. These are visible to the user, can be edited by the user using the property notation, and are accessible from within C code using `objectinstance_getproperty` and `objectinstance_setproperty`. The user may subclass a class implemented in this way and override method definitions. Many functionals use this strategy, as it's lightweight. It does have some limitations: you may **not** use this strategy if you need to store bare pointers (i.e. those that refer to something that isn't an object) or if you want the very fastest possible performance since property access is relatively expensive.

2. You create a new object type (see Section 6.4) which can include arbitrary information. You then create what is referred to as a *veneer class,* (see Section 6.5), a *morpho* class that that defines user-accessible methods. While more cumbersome, this pattern provides the fastest possible performance, but object properties are not visible to the user and the resulting class cannot be subclassed by the user.

Both of these use a similar approach to define the class, which is in effect simply a collection of method implementations. Indeed, methods have exactly the same interface as C-functions.

```
value mymethod(vm *v, int nargs, value *args);
```

and are written in the same way. From within a method, the macro `MORPHO_SELF(args)` returns the object itself as a `value`.

Once you have defined your method implementations, you must tell the *morpho* runtime about your class. First, a set of macros are provided to create the appropriate class definition, e.g. for the Mesh class,

```
MORPHO_BEGINCLASS(Mesh)
MORPHO_METHOD(MORPHO_PRINT_METHOD, Mesh_print, BUILTIN_FLAGSEMPTY), MORPHO_METHOD(MORPHO_S
/* ... */
MORPHO_ENDCLASS
```

You call the `MORPHO_BEGINCLASS` macro with a name for your class (this need not be the user-facing name). You then use `MORPHO_ METHOD` repeatedly to specify each method. The first argument is the user-facing method label, which is often a macro. The second argument is the C function that implements the method. The final argument is a list of flags that can be used to inform morpho about the method. These are reserved for future use and `BUILTIN_FLAGSEMPTY` is sufficient. Finally, you use `MORPHO_ENDCLASS` to finish the class definition.

Having defined the available methods you must then call `builtin_addclass` in your initialization code to actually define the class. For the Length functional, this would look like:

```
builtin_addclass(LENGTH_CLASSNAME, MORPHO_GETCLASSDEFINITION(Length), objclass);
```

The first argument, `LENGTH_CLASSNAME`, is the user-visible name for the class. The second argument is the class definition. Use the macro `MORPHO_GETCLASSDEFINITION` to retrieve this, supplying the name you used with `MORPHO_BEGINCLASS`. The final argument is the parent class. Often, we want this to be Object, and we can retrieve this like so:

```
objectstring objclassname = MORPHO_STATICSTRING(OBJECT_CLASSNAME);
value objclass = builtin_findclass(MORPHO_OBJECT(&objclassname));
```

6.4 Implementing a new object type

The morpho object system is readily extensible. Many data structures, strings, lists, dictionaries, matrices, etc. are implemented as objects. To illustrate what's necessary to create a new one, in this section, we'll implement a new objectfoo type. It's a good idea to declare new object types in a separate pair of implementation (.c) and header (.h) files.

In the header file, we'll begin by declaring a global variable as `extern` that will later contain the objecttype. We'll also declare a macro to refer to the objecttype later.

```
extern objecttype objectfootype;
#define OBJECT_FOO objectfootype
```

Now we can declare an associated structure and type for objectfoos. Let's make them as simple as possible, simply storing a single `value`.

```
typedef struct {
        object obj;
        value foo;
} objectfoo;
```

Note that we start the structure declaration with a field of type `object` that is reserved for *morpho* to use. All object structures are **required** to have the first field be an `object`, because *morpho* uses this field to detect the object type. The remainder of the structure can be anything; here we just declare the value.

It's a good idea to implement convenience macros to check if a value contains a particular type of object, and to retrieve the object from a value with the correct pointer type. For example:

```
/** Tests whether an object is a foo */
#define MORPHO_ISFOO(val) object_istype(val, OBJECT_FOO)

/** Gets the object as a foo */
#define MORPHO_GETFOO(val)   ((objectfoo *) MORPHO_GETOBJECT(val))
```

You may also declare other macros to retrieve fields

```
/** Gets the foo value from a foo */
#define MORPHO_GETFOOVALUE(val) (((objectfoo *) MORPHO_GETOBJECT(val))->foo)
```

In the implementation file, we will create a global variable to hold the objectfootype; this will be filled in by initialization code.

```
objecttype objectfootype;
```

We can then begin defining an objectfoo's functionality. The first step is to implement a constructor function, which should allocate memory for the object and initialize it. This will involve calling,

```
object *object_new(size_t size, objecttype type)
```

to create a new object with a specified size and type. If `object_new` returns a non-NULL pointer, allocation was successful and you can initialize the object. The constructor for an `objectfoo`, for example, might be:

```
objectfoo *object_newfoo(value foo) {
        objectfoo *new = (objectfoo *) object_new(sizeof(objectfoo), OBJECT_FOO);
        if (new) new->foo=foo;
        return new;
}
```

You could provide more than one constructor to create your object from different kinds of input. For example, we could declare `object_foofromllist` to create a foo from an objectlist. Prototypes for the constructors should be added to the header file.

To interface our new object type with the morpho runtime, we need to define several functions:

- `objectfoo_printfn (object *obj)` Called by *morpho* to print a brief description of the object, e.g. <Object>. If the object's contents are short and can be conveniently displayed (as for a string), printing the contents is allowable. Detailed information or printing of complicated objects (e.g. a matrix) should *not* be implemented here; it should go in a veneer class (see Section 6.5).

  ```
  void objectfoo_printfn (object *obj) {
          printf("<Foo>");
  }
  ```

- `objectfoo_freefn(object *obj)` [Optional] Called when the object is about to be free'd, providing an opportunity to free any private data, i.e. data that is otherwise invisible to the virtual machine. Almost always, objects that are referred to by a `value` are not required to be free'd. For example anything that has been passed to you by the virtual machine, or that you have created and bound to the

virtual machine with `morpho_bindobjects`, should **not** be free'd. Rather, this is for memory that your object has allocated independently with `MORPHO_ALLOC`. Since our objectfoo doesn't have any private data, we can actually skip this function.

- `objectfoo_markfn(object *obj, void *v)` [Optional] Called by the garbage collector to find references to other objects. You should call `morpho_markobject`, `morpho_markvalue`, `morpho_markdictionary` or `morpho_markvarrayvalue` as appropriate to inform the garbage collector of these references. Since we have a reference to a value in the foo field, we just need to call `morpho_markvalue`. Failing to inform the garbage collector correctly of references your object holds can cause random crashes; to help identify these compile with `MORPHO_DEBUG_STRESSGARBAGECOLLECTOR`.

```
void objectfoo_markfn (object *obj) {
        morpho_markvalue( ((objectfoo *) obj)->foo );
}
```

- `size_t objectfoo_sizefn(object *obj)` Should return the size of the object. You **should** include the size of any private data you hold, but **should not** include the size of any references. Hence, we simply return the size of the struct. If the estimates returned by this function are incorrect, *morpho* programs using your object will still most likely run correctly, but the garbage collection may run either too frequently, impacting performance, or not frequently enough, potentially causing the program to run out of memory.

```
void objectfoo_sizefn (object *obj) {
        return sizeof(objectfoo);
}
```

These functions should be collected together in a `objecttypedefn` structure, which is normally declared statically:

```
objecttypedefn objectfoodefn = {
        .printfn=objectfoo_printfn,
    .markfn=objectfoo_markfn,
    .freefn=NULL,
    .sizefn=objectfoo_sizefn
};
```

Now all these functions have been defined, we must add the following line to initialization code,

```
objectfootype=object_addtype(&objectfoodefn);
```

which registers the objectfoodefn with the morpho runtime and returns an objectype, which we record for use elsewhere.

Nothing requires us to expose a new object type to the user; we can use such an object purely for internal purposes. Most objecttypes, however, provide a veneer class as we'll discuss in the following section.

6.5 Veneer classes

A veneer class is a morpho class definition that the runtime refers to whenever the corresponding object is encountered. Such a class provides a "veneer" over a regular morpho object that enables the user to interact with it like any other object. For example, if morpho tries to add a float to an `objectmatrix`, morpho looks up the veneer class for an `objectmatrix` then tries to invoke the `add` or `addr` method as appropriate.

Veneer classes are defined as for regular classes (see the preceding section). The programmer must provide method implementations, define the class and register it with the runtime. To register a class as a veneer class, one more step is required in initialization code:

```
object_setveneerclass(OBJECT_MATRIX, matrixclass);
```

which registers the class object `matrixclass` (this is the return value of `builtin_addclass`) as the veneer class for objects of type `OBJECT_MATRIX`.

6.6 Error handling

6.7 Memory management

The morpho runtime provides garbage collection: the user need not worry about deallocating any object. The actual garbage collector implementation is intentionally opaque and is likely a target of future improvement. The C programmer typically interacts with the garbage collector in two ways: First, new object types must provide a markfunction to enable the garbage collector to see any `value`s stored within an object as described in Section 6.4. Second, where new objects are created and returned to the user, these should typically be bound to a virtual machine as will be described below.

Morpho uses the following model for memory management. Generic blocks of memory can be allocated, free'd and reallocated using the following macros:

```
x = MORPHO_MALLOC(size)
MORPHO_FREE(x)
MORPHO_REALLOC(x, size)
```

If you allocate memory using `MORPHO_MALLOC`, you are responsible for freeing it. For example, if you allocate additional memory when an custom object is created, you should free it in the appropriate freefn.

If you create an object, you are resposible for freeing that object by calling `object_free` unless that object is bound to a virtual machine by calling:

```
morpho_bindobjects(vm *v, int nobj, value *obj);
```

with a list of objects. A simple example:

```
objectfoo *foo = objectfoo_new();
// Check for success and raise an error if allocation failed
value new = MORPHO_OBJECT(foo)
morpho_bindobjects(v, 1, &new);
```

Once an object is bound to a virtual machine, the garbage collector is responsible for determining whether it is in use and can be safely deallocated.

The "weak" garbage collection model used by morpho has a number of advantages: It reduces pressure on the garbage collector by reducing the number of blocks that need to be tracked, because data associated with the runtime environment does not have to be managed. It facilitates a number of virtual machine features such as re-entrancy, because morpho objects can be created, used and even returned to the user without the garbage collector being involved. Nonetheless, because of the non-deterministic nature of garbage collection, various classes of subtle bugs can arise:

1. An incorrectly programmed custom object may fail to inform the garbage collector about `value`s or other structures it has access to in its `markfn`. It is **essential** that this function works correctly, otherwise the garbage collector may think that an object is no longer in use (and hence deallocate it) when in fact it is.

2. When calling morpho code from C it is necessary to be careful to ensure that bound objects remain visible to the garbage collector or they may be incorrectly free'd. See Section 6.8 for further information.

In both cases, when an incorrectly free'd object is next used, it will cause a segmentation fault. To help debug such errors, it's possible to build morpho with the option `MORPHO_DEBUG_STRESSGARBAGECOLLECTOR`. This forces garbage collection on every bind operation, and will tend to cause segmentation faults to occur much sooner after the problematic code has executed.

6.8 Re-entrancy

The morpho virtual machine is *re-entrant*, i.e. C function and method implementations can call morpho code and re-enter the virtual machine.

```
bool morpho_call(vm *v, value fn, int nargs, value *args, value *ret);
bool morpho_lookupmethod(value obj, value label, value *method);
bool morpho_countparameters(value f, int *nparams);
bool morpho_invoke(vm *v, value obj, value method, int nargs, value *args, value *ret);
```

Part III *Morpho* internal documentation

Chapter 7

The *morpho* source code

The morpho source code is in the *morpho5* folder of the git repository and is modularized into files each of which is intended to provide one piece of functionality. The source is organized into folders:

Folder	Purpose
.	Global header files and main.
builtin	Support for builtin classes and functions
datastructures	Data structures and functions to manipulate them
geometry	Meshes, fields, etc.
interface	User interface code
utils	Utility functions
vm	Virtual machine and compiler
help	Interactive help files
modules	Modules written in morpho.

Files that provide various functionality are displayed in Table 7.1.

7.1 Coding standards

7.1 File contents

- Header files (`.h`) should include type definitions and function prototypes, and only include other header files necessary for the correct *declaration* of the contents, not necessarily their *implementation*. This helps reduce interdependencies.
- Implementation files (`.c`) must `#include` all necessary header files for successful compilation.

7.1 Comments

Functions, typedefs and other constructs in *Morpho* should be accompanied by comments in doxygen format describing the purpose of the construct and its interface, for example:

```
/** myfunc
 * @brief      What the function does
 *
 * @details    A more detailed description.
 *
```

Category	Folder	File	Functionality
Headers		`morpho.h`	Public interface to *Morpho*
		`build.h`	Build constants
Core	`vm`	`core.h`	Header file for core datatypes
		`compile.c/.h`	Compile strings to Morpho instructions
		`opcodes.h`	Define opcodes
		`vm.c/.h`	The virtual machine
Data structures	`data`	`dictionary.c/.h`	Implements dictionaries
		`object.c/.h`	Objects
		`matrix.c/.h`	Dense matrices
		`sparse.c/.h`	Sparse matrices
		`syntaxtree.c/.h`	Syntax trees
		`value.c/.h`	Values
		`varray.c/.h`	Variable length arrays (buffers)
Interface	`interface`	`cli.c./h`	The CLI
		`linedit.c/.h`	The line editor
		`help.c/.h`	Interactive help system
Builtin	`builtin`	`builtin.c/.h`	Support for functions and classes
		`functions.c/.h`	Built in functions
		`file.c/.h`	File class
		`veneer.c/.h`	Veneer classes onto built in objects
Utils	`utils`	`common.c/.h`	Common utility functions
		`error.c/.h`	Error handling
		`memory.c/.h`	Memory handling
		`parse.c/.h`	Lexer and parser
		`random.c/.h`	Random numbers
Main		`main.c`	Main function

Table 7.1: Map of the *morpho* source code.

```
 * @param[in]  x An input parameter
 * @param[out] y An output parameter
 * @return     What the function returns.
 */
```

Occasionally, it is useful to include ASCII diagrams in the source code to illustrate algorithms or data structures. They can formatted so that doxygen preserves the spacing as in the below example:

```
/** @detail
 * <pre>
 *      A
 *     / \
 *    B   C
 * </pre>
 */
```

7.1 Naming conventions

1. Types should be named lower case all one word, e.g `dictionaryentry`.
2. Functions should be named `module_functionname` where `module` refers to the conceptual unit that the function belongs to (typically the same as the filename).
3. Method definitions should be of the form `Classname_methodname` noting the capitalization.

7.1 Scoping

To improve the modularity of *Morpho*, all functions not intended to be used by other translation units should be declared `static`. Macros not intended for use outside a particular context should be `#undef`'d.

7.1 Unreachable code

Where appropriate, mark unreachable code with `UNREACHABLE(x)`. The parameter of this macro is a short static char describing where the unreachable code is.

7.1 C99 features

In this section we note specific C99 features used in the implementation of *Morpho*.

- Compound literals are used to implement `value` literals.
- Flexible array members are used to implement some object types. These look like this:

  ```
  struct mystruct {
          int len;
          double arr[]; /* Note lack of size here */
  };
  ```

  where the flexible array member must be at the end of the `struct` definition.
- Pointers are converted into `uintptr_t` to hash them. This type is defined so that conversion from a pointer and back yields the same value.

Chapter 8

The Virtual Machine

8.1 Virtual machine structure

The virtual machine operates on a progam which comprises a sequence of instructions, described in subsequent sections, that perform various functions. While running the program, the VM maintains the following state:

- **Program counter.** This points to the next instruction to be executed.
- **Data stack**. The data stack contains information the program is acting on. It is an ordered list of values that can grow as needed. A subset of these values are visible to the VM at any one time, referred to as the *register window*. Because morpho instructions can refer to any available register, not just the top of the stack, morpho's VM is a *register machine*.
- **Call stack**. The call stack keeps track of function calls and grows or shrinks as functions are called or return. Each entry, called a *call frame*, contains information associated with a function call: which parts of the data stack are visible, the value of the program counter when the call took place, a table of constants, etc.
- **Error handler stack**. Programs may provide code to handle certain kinds of errors. This stack keeps track of error handlers currently in use.
- **Garbage collector information**. As objects are created at runtime, the VM keeps track of them and their size and periodically removes unused objects.

8.2 *Morpho* instructions

Each *Morpho* VM instruction is a packed `unsigned int`, with the following possible formats
This permits up to 64 separate opcodes and up to 256 individually addressable registers per frame. Parameter A denotes the register that is used to store the result of the operation, or more generally the register that is affected. Operations B and C are used to denote the input registers. Additionally, a single bit flag (Bc and Cc) can be set for each input register. These are used, for example, to select between loading from a register or from the current frame's constant table.

Type	Bits				Description
	0-7	8-15	16-23	24-31	
1	Opcode	A	B	C	Operation with three byte parameters; B & C can be marked as constants.
2	Opcode	A	Bx		Operation with one byte and one short parameter & two flags.
3	Opcode	A	sBx		Operation with one byte and one signed short parameter & two flags.
4	Opcode	Ax			Operation with one 24 bit unsigned parameter

Table 8.1: *Morpho* instruction formats.

8.3 *Morpho* opcodes

Each instruction has 1-3 operands. Lower case letters indicate registers, upper case represents literals or constant ids.

Category	Opcode	Operands	Explanation
	`nop`		No operation.
	`mov`	a, b	Moves reg. b into reg. a
	`lct`	a, Bx	Moves constant B into reg. a
Arithmetic	`add`	a, b, c	Adds register b to c and stores the result in a.
	`sub`	a, b, c	Subtracts register c from b and stores the result in a.
	`mul`	a, b, c	Multiplies register b with register c and stores the result in a.
	`div`	a, b, c	Divides register b with register c and stores the result in a.
	`pow`	a, b, c	Raises register b to the power of register c and stores the result in a.
Logical	`not`	a, b	Performs logical not on register b and stores the result in a.
Comparison	`eq`	a, b, c	Sets reg. a to boolean b==c
	`neq`	a, b, c	Sets reg. a to boolean b!=c
	`lt`	a, b, c	Sets reg. a to boolean b<c
	`le`	a, b, c	Sets reg. a to boolean b<=c
Branch	`b`	sBx	Branches by (signed) B instructions
	`bif`	a, sBx	Branches by (signed) B instructions if a is true.
	`biff`	a, sBx	Branches by (signed) B instructions if a is false.
Function calls	`call`	a, B	Calls the function in register a with B arguments
	`return`	A, b	Returns from the current function If parameter A>0, register b is returned

Objects	`invoke`	a, b, C	Invokes method b on object a with C arguments. Arguments are stored in register a+1 onwards.
	`lpr`	a, b, c	Looks up property c in object b, storing the result in a.
	`spr`	a, b, c	Stores value c in object a with property b.
Closures	`closure`	a B	Encapsulates the function in register a into a closure using prototype number B from the enclosing function object.
	`lup`	a, B	Loads the contents of upvalue number B into register a.
	`sup`	A, b	Stores the contents of b in upvalue number A
	`closeup`	A	Closes upvalues beyond register number A
Indices	`lix`	a, b, c	Loads an element from array a. Indices to use are stored in registers b..c, and the result is stored in register b.
	`six`	a, b, c	Stores value c in array a with indices stored in registers b..c-1.
	`array`	a, b, c	Creates an array with dimensions in registers b..c and stores it in register a. *(Should be deprecated)*
Globals	`lgl`	a, Bx	Loads global number Bx into register a.
	`sgl`	a, Bx	Stores the contents of register a into global number Bx.
Error handlers	`pusherr`	Bx	Pushes the error handler in constant Bx onto the error handler stack
	`poperr`	sBx	Pops the current error handler off the error handler stack and branch by (signed) B instructions
	`print`	a	Prints the contents of register a.
	`cat`	a, b, c	Concatenates the contents of registers b-c and stores the result in register a. *(Should be deprecated?)*
	`break`		Breakpoint
	`end`		Denotes end of programs

8.4 How function calls work

When a function or method call takes place, the VM:

1. Records the program counter, register index and stack size in the current call frame.
2. Advances the frame pointer.
3. If the called object is a closure, pulls out the function to be called and records the closure in the new call frame.
4. Records the function to be called.

5. Sets up the constant table.
6. Advances the register window and clears the contents of registers to be used by the function.
7. Register r0 contains either the function object OR the value of `self` if this is a method call.
8. Registers r1 onwards contain the arguments of the function.
9. Moves the program counter to the entry point of the function.

8.5 Methods

Methods are similar to functions, except that *r0* contains the object.

8.6 How error handling works

Ordinarily, when a runtime error is generated execution immediately stops and the error is reported to the user. Sometimes a possible error can be forseen by the programmer and the program can be written to take an alternative course of action. To achieve this, the Morpho VM provides for error handlers.

Morpho keeps track of error handlers on a special stack. As execution proceeds, the program may add an error handler to the stack using the `pusherr` opcode; it can then be removed again by using `poperr`. Only the most recent error handler can be removed in this way.

When an error occurs, the VM searches the error handlers currently in use from the top of the error handler stack downwards to find an error handler that matches the ErrorID tag. If none is found, execution terminates and the error is reported to the user as normal. If suitable handler is found, however, execution resumes at a point specified by the handler. The callframe is reset to that of the error handler and any open upvalues beyond that frame are closed.

Re-entrancy

In searching for an error handler, the VM checks whether an intermediate call frame requires it to return. This happens if the VM has been re-entered from a C function (using `morpho_call` for example). In such a case, the VM returns from the intermediate frame rather than handling the error (and so `morpho_call` returns false). The calling C function should check for this case and either handle the error itself or exit as quickly as possible. If the error isn't handled, once the intermediate C function returns, the outer VM that called it will detect the error and resume the search for an error handler.

Chapter 9

The compiler

9.1 Overview

The *Morpho* compiler takes a string of *Morpho* input and converts to bytecode by a three stage process (Fig. 9.1.1): The source code is first divided into *tokens*, basic units like number, identifier etc., by the *lexer*. Tokens are then converted into a *syntax tree*, an abstract representation of the programs syntactic structure, by the *parser*. These two components are in `parse.c/.h`.

Finally, the syntax tree is converted to bytecode by the bytecode compiler (referred to hereafter simply as the compiler) in `compile.c/.h`. The resulting bytecode can then be run by the virtual machine described in chapter 8.

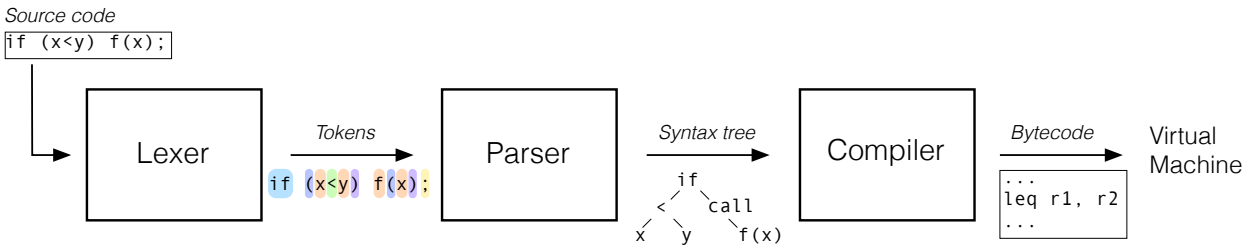

Figure 9.1.1: Structure of the *Morpho* compiler.

9.2 Extending the compiler

This section provides a very brief guide to how the compiler may be extended. When modifying the compiler, it's a good idea to do so in the order suggested by Fig. 9.1.1: First modify the lexer to produce any new token types required, then the parser to parse new syntax correctly, then finally the compiler. The file `build.h` contains a number of macros that can be defined that cause morpho to generate output to help debug compiler features, for example `MORPHO_DEBUG_DISPLAYSYNTAXTREE` displays a syntax tree after compilation.

9.2 New token types

New token types (for example, for a new operator or keyword) can be implemented by adding a new entry into the `tokentype` enum in `parse.h`. It is imperative to make a corresponding entry into the parser definition table (see `parserule rules[]` in `parse.c`), even if the token type is marked `UNUSED` for now. The overall order of token types isn't significant, but it's essential that these two structures match. You will also need to

modify `cli.c` as there is a parallel table `cli_tokencolors` describing how to syntax color different token types.

You should then modify the lexer to generate the token. Tokens are identified from the initial character in `lex()` and functions that it calls. New reserved words require modifying `lex_symboltype`. Modifications to literals require modifying `lex_string` or `lex_number`. Once a token is identified, it is recorded by calling `lex_recordtoken`.

9.2 Parser

You now need to create a parse rule, or edit an existing one if appropriate, to parse the new token type. Parse functions all have the form

```
syntaxtreeindx parse_MY(compiler *c)
```

and call:

1. `parse_advance` and `parse_consume` to obtain tokens.
2. `parse_expression` or `parse_precedence` to parse subexpressions.
3. `parser_addnode` to add nodes to the syntax tree.
4. `parse_error` to record errors.

Once the parse rule is created, it may be included in the parser definition table (see `parserule rules[]` in `parse.c`) or called from another function. New operators, for example, are typically inserted directly into the defintion table, and macros are available to denote the token as `PREFIX`, `INFIX` or `MIXFIX`. Keywords that introduce a statement, e.g. `var` or `fn`, require inserting an appropriate text and call into `parse_statement`.

9.2 Syntax tree

Once the parser has been modified, it may be necessary to create new syntax tree node types:

1. Create any new syntaxtreenodetypes necessary. A node type is defined by adding a new entry into the `syntaxtreenodetype` enumerated type definition. The order *does* matter: new node types should be grouped with leafs, unary operators or operators as is documented in the source. It is imperative to make a corresponding entry in the compiler definition table (see `compiler_nodefn noderules[]` in `compile.c`); it can be marked `NODE_UNDEFINED` for now. As for the parser definition table, it's essential that these structures match. It's also important to add a corresponding entry into the `nodedisplay` array in `syntaxtree.c`; this is used to display syntax trees for debugging purposes.
2. Modify the parse rule generated in the previous section to emit appropriate syntax tree nodes.

9.2 Compiler rule

1. Create a compiler rule for any new node types. Insert it into the compiler definition table at the corresponding place. Compiler rules call `compiler_nodetobytecode` to compile child nodes, and `compiler_writeinstruction` to create bytecode. Macros are available to encode instructions.
2. Creation of new instruction types is possible (by modifying the VM) but strongly discouraged; contact the developers if you have ideas about improved VM functionality.

Chapter 10

Morphoview

Morpho 0.5 separates out UI code into a separate application, *morphoview*. This enables, for example, running a program on a cluster and seeing the results on another computer, or having different client dependent implementations (e.g. a Metal client for macOS, an OpenGL client for Linux, etc.). The *morpho* language runtime communicates with morphoview via temporary files, but this will be replaced by a client/server model via ZeroMQ or similar.

10.1 Command language

Morpho communicates with *morphoview* via an imperative scommand language designed to be simple and human readable. Valid commands are given in table 10.1 together with their syntax. Commands and their elements may be separated by arbitrary whitespace. Additional commands may be provided in the future.

10.1 Usage

A typical file will begin will begin by specifying a scene,

```
S 0 2
```

and (completely optionally) setting the window title

```
W "Triangle"
```

Following this, objects in the scene may be defined,

```
o 1
```

which include vertex data and elements of the object. For instance, a simple triangle could include

```
v x
1 0 0
0 1 0
1 1 0
f 0 1 2
```

providing a list of three vertices and then specifying that a single facet is to be drawn from these vertices.

After objects have been specified, the scene can be drawn. Objects are positioned in the scene by using transformation matrices: Morphoview maintains a current object transformation matrix at all times. Initially, this is set to the identity matrix (so that the object is placed in the scene using the coordinates at which it is defined), but the matrix can be modified by the `i`, `r`, `s` and `t` commands. `e`

For example, to scale, rotate translate and then draw object 1,

Command	Syntax	Description
c	`c id r g b ...`	Color buffer declaration
C	`C id`	Use color or map object *id.* If no *id* is provided, clears current color object.
d	`d id`	Draw object *id* using current transformation matrix and color *id*.
o	`o id`	Object `id` is an integer that may be used to refer to the object later; unique per scene.

v	`v format`
	`f1 ...`
	
	`... fn`
		Vertex array
		`format` is a string that contains at least any or all of the letters x, n and c
		and is used to specify the information present and the order, e.g.
		`xnc` — vertex entries contain position, normal and color
		`x` — only position information is present
		`xc` — indicates position and color information
		This is then followed by the appropriate number of floats

p	`p` `v1 ...`	Points A list of integer indices into the object's vertex array to be drawn as points
l	`l` `v1 v2 ...`	Lines A list of integer indices into the object's vertex array to be drawn as a continuous sequence of lines
f	`f` `v1 v2 v3`	Facets A list of triplets of integer indices into the object's vertex array to be drawn as facets.
i	`i`	Set the current transformation to the identity matrix
m	`m m11 m21 .. m44`	Multiply the current transformation by the given 4x4 matrix, given in column major order
r	`r phi x y z`	Rotate by phi radians about the axis (x, y, z) [(x, y) in 2d]
s	`s f`	Scale by factor f
S	`S id dim`	Scene description `id` is an integer that may be used to refer to the scene later `dim`$\in \{2, 3\}$ is the dimension of the scene
t	`t x y z`	Translate by *(x,y, z)*
T	`T fontid string`	Draw text "string" with font `id` using current transformation matrix and color id
F	`F fontid file size`	Declare font `file` the filename and path of the desired font `size` the size in points `fontid` an integer that will be used by T commands to refer to the font
W	`W title`	Window features `title` is the window title

Table 10.1: **Morphoview command language.**

```
s 0.5
r 1 0 0 1
t 0.5 0.5 0
d 1
```

You can draw multiple objects using the same transformation matrix just by issuing subsequent draw commands.

Once the scene is specified, *Morphoview* will open a viewer window displaying the specified scene.

10.1 Morphoview internal structure

1. **Parser.** The parser processes the command file and builds up a Scene object from the program.
2. **Scene.** Morphoview programs describe a Scene. Once the scene is described, it is then prepared for rendering.
3. **Renderer.** This module takes a scene and prepares OpenGL data structures for rendering.
4. **Display.** Manages windows, user interface etc.