morpho

Developer Guide
Version 0.6.0

January 13, 2024

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 the *morpho* source 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. Bug reports, suggestions for new features are welcome; please see `CONTRIBUTING.md` in the *morpho* git repository.

Modules and extensions can 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 a *package*, which is a folder containing files that collectively provide some enhancement or 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

Morpho hints

Despite its small syntax, *morpho* provides many powerful features in common with other modern dynamic languages. In this chapter, we discuss how to write good *morpho* code.

3.1 Language features

3.1 Importing modules

The `import` keyword allows you to load modules and extensions with the same simple syntax. Conceptually, each file defines a *namespace*, a collection of symbols, and we'd like ways to control how symbols from one namespace appear in another. When you use import, by default all symbols defined become visible. That is, if you include the line

```
import color
```

in your program, you can use all functions, classes and global variables defined by the color module. Some modules, however, may define a lot of things that aren't necessary for your program and indeed may use symbols that you would like to use yourself; this phenomenon is often referred to as *namespace polution*.

The first approach is to use the `for` keyword in the import statement, e.g.

```
import color for Red, Color
```

When this statement is encountered in your program, the `color` module is loaded, but only the symbols `Red` and `Color` become visible to your code. Symbols like Blue and Green that are also defined in the `color` module are not visible, so this code,

```
print Blue
```

would raise the *SymblUndf* compilation error. This approach is ideal for when you only want to use a small subset of symbols from a module, and you know them ahead of time.

The second approach is to use the `as` keyword, e.g.

```
import color as col
```

which loads the color module, but *none* of the symbols defined are directly visible to you. Instead, you must access them by referring to `col` in your code like this, e.g.

```
print col.Red
```

The symbol `col` refers to a specific namespace; when the compiler encounters this code, it looks up the symbol `Red` in the namespace `col`. Using the symbol directly like so

```
print Red
```

would raise an error. Hence, `Red` is now free for you to use!

You can even combine these two,

```
import color as col for Red, Color
```

which defines a namespace `col`, but only includes `Red` and `Color` in it.

When you're writing a library, you may want to prevent symbols being *exported* to a program that includes them. Some languages include keywords like `private` for this purpose, but this leads to verbose code. *Morpho* offers a simpler solution: any symbol that begins with an underscore is **never** exported to a program that imports it. It's recommended to use this for things like:

- definitions of errors that are only for use by your module e.g.

  ```
  var _myErr = Error("MyErr", "An Error occurred")
  ```

- functions that have an interface whose interface is liable to change, or that provide internal functionality secondary to your module's purpose.

3.2 Data structures

Morpho provides a wide variety of data structures for you to use. Most of these are standard fare in computer science, and hence we do not provide detailed theoretical background here. Rather, we focus on explaining the various circumstances each data structure may be of use.

3.2 Lists

Lists represent an ordered collection of items.

3.2 Dictionaries

Dictionaries or *hashmaps* as they are sometimes called.

3.3 Protocols

As a dynamic language, *morpho* makes heavy use of *protocols*. Protocols are collections of methods defined in different classes that aim to provide analogous class-dependent functionality. For example, although the implementation of the `clone()` method looks quite different within each class definition, it behaves the same way everywhere, namely making a shallow copy of an object. Hence, the user can rely on calling `clone()` without worrying about the specific type of an object it's attempting to clone. Protocols make code more reusable, reduce duplication, and reduce the need for type checking. Unlike languages such as Swift and Objective C, where protocols are defined with a specific keyword, *morpho* eschews this and simply adopts them informally.

3.3 Clone

The clone method makes a *shallow* copy of an object, i.e. any attached objects are duplicated rather than cloned themselves. If a deep clone is necessary for a particular class, the user should provide this as a separate method.

Chapter 4

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 9.

4.1 Debugging commands

4.1 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.

4.1 Continue

Continues program execution, leaving the debugger.

4.1 Disassemble

Displays disassembly for the current line of code.

4.1 Garbage collect

Forces a garbage collection.

4.1 Clear

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

4.1 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

4.1 List

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

4.1 Print

Prints the value of variables.

`p symbol`	Prints the value of a given symbol
`p`	Print all currently visible symbols

4.1 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.

4.1 Quit

Terminates program execution.

4.1 Step

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

4.1 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 5

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 creating 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 6

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 entirely stable; this will occur at v1.0.0. As we gain experience writing extensions and identify common needs, we anticipate improving the API. We welcome your feedback.

Chapter 7

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 any language that can link with C.

7.1 Basic data types

7.1 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 and functions provided by *morpho:*

- **Initialize a `value` with a literal.** Macros provided include `MORPHO_NIL`, `MORPHO_TRUE`, `MORPHO_FALSE`. You can also use `MORPHO_INTEGER()` and `MORPHO_FLOAT()` to create integers and floats respectively.
- **Convert a `value` to a C type.** `MORPHO_GETINTEGERVALUE`, `MORPHO_GETFLOATVALUE`, `MORPHO_GETBOOLVALUE`, `MORPHO_GETOBJECT`. In general, it's important to be sure that a value contains the type you expect before using these. Use `morpho_valuetoint` or `morpho_valuetofloat` for example to convert a generic value to an integer or double respectively; these functions return true if they succeed.
- **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.

7.1 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. For some types, 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 7.4.

You **must not** store a *generic* pointer in a value using MORPHO_OBJECT, only a suitably defined `struct` will work. Passing a generic pointer to the virtual machine will likely cause a segfault, and it may occur at a random point (i.e. when the garbage collector runs).

7.1 Varrays

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, `varray.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

```
void varray_integerinit(varray_integer *v); // Initialize a varray
bool varray_integeradd(varray_integer *v, int data[], int count); // Add a specified numbe
bool varray_integerresize(varray_integer *v, int count); // Resize the varray
int varray_integerwrite(varray_integer *v, int data); // Add one element, returns number o
void varray_integerclear(varray_integer *v); // Clear the varray
```

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);
...
```

Check the *morpho* source before defining a new varray type for a C type in an extension—it may cause conflicts. Note that `varray.h` defines varrays for `char`, `int`, `double` and `ptr_diff`; value.h defines varrays for `value`. You should not redefine these.

7.2 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*[1] 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;
}
```

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.

There are a few important things to note about defining your own *morpho* C-function. First, these 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 7.6. If you create any new objects in your function, you **must** bind them to the virtual machine as described in Section 7.7.

You can call *morpho* code from within your function with certain restrictions, see Section 7.8 for details.

[1] i.e. a pointer of type `void *`. This is specifically intended to prevent developers from relying on a particular implementation of the virtual machine.

7.2 Optional parameters

Your function may accept optional parameters just as regular *morpho* functions do. 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 extension's initialization function by calling `builtin_internsymbolascstring`; this is for performance reasons and enables the runtime to detect optional arguments. You must keep a record of the returned value; typically this is done in a global variable (`fnoption1` and `fnoption2` in the example above).

In your function implementation, you call `builtin_options` with, `v`, `nargs` and `args` as passed to you by the runtime, 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, cleaning up any memory you allocated that has not been bound to the virtual machine as per Section 7.7 and returning `MORPHO_NIL`.

7.3 Implementing a new class

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

1. The new class reuses `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 very lightweight. It does have some limitations: you may **not** use this strategy if you need to store C pointers 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 7.4) which can include arbitrary information including C pointers to non-objects. You then create what is referred to as a *veneer class,* (see Section 7.5), 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. The *morpho* source uses this pattern extensively: many examples are to be found in `src/classes/` in the git repository; the List or Range classes are good places to look.

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 definition, 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));
```

7.4 Implementing a new object type

Objects are heap-allocated data structures with a special format that enables the *morpho* runtime to infer type information. Other than the simple types that are held in a value, most *morpho* data structures including strings, lists, dictionaries, matrices, etc. are implemented as objects. The *morpho* object system is readily extensible, and it's easy to create new ones. Note that objects are not necessarily visible to the user, to facilitate this it's also necessary to define what is known as a *veneer class* (Section 7.5 which defines methods that can be called from *morpho* code).

To illustrate the definition and implementation of a new object type, 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. Many similar examples are to be found in `src/classes` in the morpho source.

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 **must** 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 contain arbitrary information; 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. Where these are listed as optional, they can be set to `NULL` in the definition.

- `void objectfoo_printfn (object *obj, void *v)` 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 7.5). *Note that in Morpho 0.6 and higher the virtual machine is passed as an argument, and you should use* `morpho_printf` *to print to morpho's output stream.*

  ```
  void objectfoo_printfn (object *obj, void *v) {
          morpho_printf(v, "<Foo>");
  }
  ```

- `void 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.

- `void 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 anything in a value that has been passed to you or bound to the virtual machine. 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 collector 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);
  }
  ```

- `hash objectfoo_hashfn(object *obj)` [Optional] Called by the dictionary data structure to compute a hash from the object. You **must only** define this if your object type is immutable, i.e. cannot be modified once created. If this were the case for an objectfoo, we could then hash the contents using one of the functions in `src/datastructures/dictionary.h`. Note that if you define this function, you must also define a comparison function that compares the contents of the object.

```
hash objectfoo_hashfn (object *obj) {
        return dictionary_hashvalue( ((objectfoo *) obj)->foo );
}
```

- `int objectfoo_cmpfn(object *a, object *b)` [Optional if no hash function] Called by the morpho runtime to test for equality between two objects. If this isn't defined, *morpho* assumes by default that two values are only equal if they are identical (i.e. refer to the same object). This function should return one of `MORPHO_EQUAL`, `MORPHO_NOTEQUAL`, `MORPHO_BIGGER` or `MORPHO_SMALLER` depending on whether an ordered comparison is meaningful or not. The library function `morpho_comparevalue` is available to compare two values.

```
int objectfoo_cmpfn(object *a, object *b) {
        objectfoo *af = (objectfoo *) a;
        objectfoo *bf = (objectfoo *) b;

        return morpho_comparevalue(af->foo, bf->foo);
}
```

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,
        .hashfn=objectfoo_hashfn, // Or NULL if we want a mutable type
        .cmpfn=objectfoo_cmpfn // Or NULL to prevent deep comparisons
};
```

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.

7.5 Veneer classes

A veneer class is a morpho class definition that the runtime uses 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. If the user calls `clone()` on a value that contains an `objectmatrix`, morpho uses the veneer class to select the method to call. This can sometimes happen implicitly: 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 in the same way as regular C classes. The programmer must provide method implementations, define the class and register it with the runtime as described in Section 7.3. The only difference, of course, is that `MORPHO_SELF` contains a reference to the specific object type. Most *morpho* data structures have veneer classes, and many examples can be found in `src/classes/` in the git repository; `list.c` and `range.c` are good places to look. These files follow a common pattern: they begin with the object definition, then provide C functions to work with the new object type, and then define the veneer class. We recommend extensions adopt a similar pattern for new object types and associated veneer classes.

To register a class as a veneer class, one additional step is required in initialization code. For example, in `list_initialize` in the file `src/classes/list.c` is the line:

```
object_setveneerclass(OBJECT_LIST, listclass);
```

which registers the class `matrixclass` (this was the return value of a previous call to `builtin_addclass`) as the veneer class for objects of type `OBJECT_LIST`. The user therefore sees an `objectlist` referred to in a value as a *morpho* `List` and can interact with it accordingly.

7.6 Error handling

When an error occurs, you can report this to the user by setting the virtual machine to an error state.

In order to do so, it's first necessary to define the error. *Morpho* errors are defined by a *tag*, a short label that is used to identify the error and a detailed *message* for the user. The separation of tags from messages permits locale-dependent display of error messages to be implemented in future versions of *morpho*. The tag and a default message are usually defined as macros in a header file. For example, this common error comes from `src/datastructures/error.h`:

```
#define VM_OUTOFBOUNDS                        "IndxBnds"
#define VM_OUTOFBOUNDS_MSG                    "Index out of bounds."
```

To define the error with the morpho runtime, you need a line like this in your initialization function:

```
morpho_defineerror(VM_OUTOFBOUNDS, ERROR_HALT, VM_OUTOFBOUNDS_MSG);
```

The first argument is the tag and the last is the message. The second argument is the error type, which can be any of:—

- `ERROR_WARNING` Indicates a warning, rather than an error.
- `ERROR_HALT` Indicates an error that should halt execution.
- `ERROR_LEX` Indicates a lexing error; used by lex.c/.h [Use only if extending the lexer]
- `ERROR_PARSE` Indicates a parser error; used by parse.c/.h [Use only if extending the parser]
- `ERROR_COMPILE` Indicates a compiler error. [Use only if extending the compiler]

The values `ERROR_USER`, `ERROR_DEBUGGER` and `ERROR_EXIT` are reserved for use by the runtime.

When the error occurs in your code, you can then report this fact by calling `morpho_runtimerror`:

```
morpho_runtimeerror(v, VM_OUTOFBOUNDS);
```

You provide the virtual machine reference that was passed to you and the error tag. Use any existing error tags defined in the morpho header files rather than duplicating the error.

Once you have reported the error to the VM, your function should return as swiftly as possible, returning `MORPHO_NIL`. You are responsible for freeing any newly created objects or other allocated memory, and undoing side-effects like open files unless you have already registered these with the virtual machine via `morpho_bindobjects`.

In come circumstances, your program may detect some condition that, while not an error that would prevent further progress, nonetheless should be raised with the user. An example scenario might be where a numerical problem is poorly conditioned, and hence the solution is likely to be of poor quality. Morpho provides an analogous function to raise a given error as a warning. You can use this for any error, even if it was not declared with `ERROR_WARNING`; errors declared with `ERROR_WARNING` are always raised as warnings.

```
morpho_runtimewarning(v, VM_OUTOFBOUNDS);
```

If your program detects a state that arises from incorrect programming, you may use the UNREACHABLE macro (see Section 8.2.5) to terminate *morpho* immediately. An example where this might be desirable is if an inconsistency or impossible condition is detected in one of your data structure*s*. Use this primarily for your own debugging purposes; the user should never see an internal consistency error.

7.7 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 a target of continuous 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 7.4. 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 your code allocates 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 responsible for freeing that object by calling `object_free`.

An important exception occurs when you create an object that is intended for the user to work with. You might return the object from a C function or method, or you might store the object in another data structure that the runtime or user provided you with. In this case, you must tell the virtual machine about the object(s) and the garbage collector then becomes responsible for their management.

To do so, you bind the object(s) 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 creating an object that contains other objects, you must take care to bind all objects at once (or at least bind the outermost objects first) so that the garbage collector can see the interdependency.

3. 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 7.8 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. There is also a macro `MORPHO_DEBUG_LOGGARBAGECOLLECTOR` that logs object creation and destruction that can be of assistance.

7.8 Re-entrancy

The morpho virtual machine is *re-entrant*, i.e. C function and method implementations can themselves call user code written in *morpho*, re-entering the virtual machine to execute it and then making use of the results open return. The API for this is described in this section. Re-entrant calls can be recursive: during such a call, the user's *morpho* code itself may call back C functions, only limited by the depth of the call stack, which is currently fixed by the macro `MORPHO_CALLFRAMESTACKSIZE` in `build.h`.

In order to use the API, you first need a value that contains a callable object. These could include:

- A C function, as returned by `builtin_addfunction,` and contained in an `objectbuiltinfunction`.
- A morpho function contained in an `objectfunction`.
- An invocation on an object, which contains both method and receiver, and is contained in an `objectinvocation`.
- A closure contained in an `objectclosure`.

All of these are called in the same way, and the runtime automatically takes care of any setup necessary to execute the call. You can check whether a value contains a callable object with the macro `MORPHO_ISCALLABLE()`.

Once you have a callable object, you can have the virtual machine execute it with given parameters by calling:

```
bool morpho_call(vm *v, value fn, int nargs, value *args, value *ret);
```

This function requires a list of arguments, given as a C array of values. You must check the return value: If the call is successful, `morpho_call` returns `true` and the value `ret` is filled out. Otherwise, the virtual machine is returning in an error state and you should return as quickly as possible.

Another common scenario is that you wish to invoke a method on a given object. You can do so with `morpho_invoke`:

```
bool morpho_invoke(vm *v, value obj, value method, int nargs, value *args, value *ret);
```

The method label should be given in the parameter method, and the parameters work similarly to `morpho_call`.

If you wish to call a method on the same object many times with different parameters, you can avoid the cost of method lookup by getting a callable invocation from the runtime using the function,

```
bool morpho_lookupmethod(value obj, value label, value *method);
```

and supplying the resulting value in `method` as the input to `morpho_call`. Note that you must check to ensure `morpho_lookupmethod` was successful; it will return `false` if the method wasn't found for example.

You can count the number of fixed parameters for a callable object by using,

```
bool morpho_countparameters(value f, int *nparams);
```

7.8 Restrictions on re-entrant code

When the virtual machine is entered, arbitrary code may be executed that can reconfigure the state of the runtime environment. As a result, functions that use the reentrant API are subject to a number of restrictions so that they do not refer to information that is no longer valid across a call to `morpho_call` or `morpho_invoke`.

- The value of the `args` pointer passed to your function may no longer be valid after a re-entrant call. You *must* obtain all parameters using `MORPHO_GETARG` before making the call.
- You must use `morpho_bindobjects` carefully. If you make a re-entrant call after using `morpho_bindobjects`, it is possible that the garbage collector will run without being able to "see" a valid reference to the object held in your C code (the garbage collector cannot check the C stack, for example) and will erroneously destroy the object leading, ultimately, to a segfault. To avoid this, either use `morpho_bindobjects` after making all re-entrant calls, or call `morpho_retainobjects` to ensure they're retained across a re-entrant call. If you do the latter, you **must** call `morpho_releaseobjects` with the handle returned by `morpho_retainobjects` before your code returns.

Part III *Morpho* internal documentation

Chapter 8

libmorpho source code

The *morpho* source code has been significantly reorganized in version 0.6 to improve its modularity and reusability. *Morpho* has been split into a dynamic library, *libmorpho*, that provides the compiler, virtual machine and runtime, and is designed to be embeddable in other applications. The command line interface is now provided by a separate program, *morpho-cli*, and other kinds of interface may be provided in future[1].

The libmorpho source is in the src folder within the git repository. Each file is intended to provide one piece of functionality, and is organized into subfolders as follows:—

Folder	Purpose
core	Compiler and virtual machine.
debug	Debugger and profiler.
classes	Morpho object types and veneer classes.
linalg	Support for dense and sparse linear algebra.
builtin	Support construction of builtin classes and functions.
support	Various utility packages
datastructures	Low level data structures used by *morpho*.

A description of files within each of these folders is provided in Table 8.1.

8.1 Compiling *morpho*

As of version 0.6, *morpho* is now compiled with cmake, an automated build utility, facilitating much improved cross-platform building. To install from source, clone the git repository to a convenient place,

```
git clone https://github.com/Morpho-lang/morpho-libmorpho.git
```

and then

```
cd morpho-libmorpho
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make install
```

You may need to use `sudo make install`.

You can also build project files for various IDEs. For example you can create an Xcode project by using

```
cmake -GXcode ..
```

[1]A prototype Jupyter interface exists, for example.

Folder	File	Description
core	compile.c/.h	Compiler
	core.h	Type definitions for VM core
	gc.c/.h	Garbage collector
	optimize.c/.h	Optimizer
	vm.c/.h	Virtual machine.
debug	debug.c/.h	Debugger
	profile.c/.h	Profiler
classes	classes.h/.h	List of classes
	array.c/.h	Array class
	closure.c/.h	Closure class
	clss.c/.h	class object type
	cmplx.c/.h	Complex number
	dict.c/.h	Dictionary class (see dictionary.c/.h for underlying implementation)
	file.c/.h	File and Folder classes
	function.c/.h	Function class
	instance.c/.h	Object class
	invocation.c/.h	Invocation class
	list.c/.h	List class
	range.c/.h	Range class
	upvalue.c/.h	upvalue object type
	strng.c/.h	Strings
	system.c/.h	System class
linalg	matrix.c/.h	Matrix class
	sparse.c/.h	Sparse class
builtin	builtin.c/.h	Support for builtin functions and classes
	functiondefs.c/.h	Builtin functions
support	common.c/.h	Contains utility functions
	extensions.c/.h	Extensions
	lex.c/.h	Lexer
	memory.c/.h	Basic memory allocation
	parse.c/.h	Parser
	random.c/.h	Random number generator
	resources.c/.h	Resource locator
	threadpool.c/.h	Thread pool
datastructures	dictionary.c/.h	Dictionary struct
	error.c/.h	Error messages
	object.c/.h	Fundamental object type
	program.c/.h	Program data structure
	syntaxtree.c/.h	Syntax tree struct used by compiler
	value.c/.h	Basic value type
	varray.c/.h	Variable length arrays

Table 8.1: Map of the *morpho* source code.

8.2 Coding standards

The *morpho* source, being somewhat large (around 40k lines as of 0.6), aims to obey the following coding standards, which may be refined as the project evolves.

8.2 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.

8.2 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.
 *
 * @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>
 */
```

8.2 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.

8.2 Scoping

Macros not intended for use outside a particular context should be `#undef`'d.

8.2 Unreachable code

Where appropriate, mark unreachable code with `UNREACHABLE(x)`. The parameter of this macro is a short `static char` that should explain to the reader where the unreachable code is to be found in the source. The morpho virtual machine immediately halts and reports an “Internal consistency error:” followed by your explanation.

8.2 C99 features

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

- We use the `bool` type and `true` and `false`.
- 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 9

The Virtual Machine

9.1 Virtual machine structure

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

9.2 *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. Parameters B and C are used to denote the input registers. Larger operands are possible in the last three instruction formats.

9.3 *Morpho* opcodes

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

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 9.1: *Morpho* instruction formats.

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, otherwise `MORPHO_NIL`.

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

9.4 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 in the new call frame.

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

Method calls are identical to function calls, except that *r0* contains the object. Invocations are unpacked before calling.

9.5 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 **must** 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 10

The compiler

10.1 Overview

The *Morpho* compiler takes a string of *Morpho* input and converts to bytecode by a three stage process (Fig. 10.1.1): 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 `lex.c/.h` and `parse.c/.h`. Both of these elements can adapted to parse things other than *morpho* code; see `json.c/.h` for an example of how this is done for the JSON interchange language.

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 9.

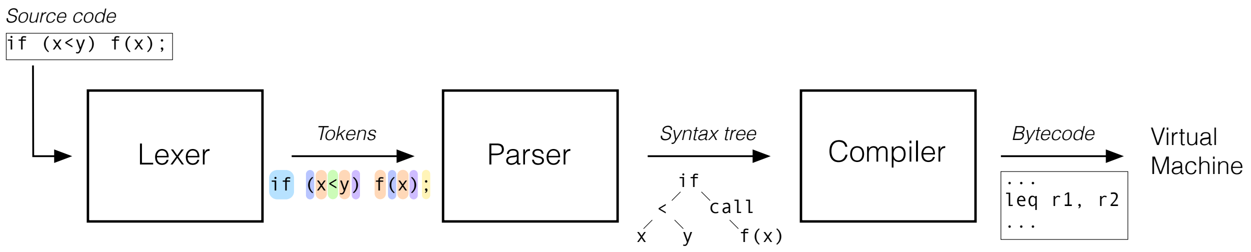

Figure 10.1.1: Structure of the *Morpho* compiler.

10.2 Extending the compiler

This section provides a very brief guide to how the lexer, parser and compiler may be extended. When modifying the compiler, it's a good idea to do so in the order suggested by Fig. 10.1.1: 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.

10.2 Lexer

The lexer takes a raw C source string as input and sequentially splits it into *tokens* demarcated by their starting point, length and type. It is defined in support/lex.c and support/lex.h and has been rewritten for v0.6 to be usable for lexing applications beyond the compiler.

To use a lexer, it must first be initialized by calling `lex_init` with a pointer to the source `start` and a starting line number `line`:

```
void lex_init(lexer *l, const char *start, int line)
```

You can obtain tokens by repeatedly calling `lex`:

```
bool lex(lexer *l, token *tok, error *err)
```

which fills out a `token` structure with the next token and returns `true` on success. If an error occurs, it will fill out the provided `error` structure and return `false`. Once the source string is exhausted, the lexer returns a special token (the token type can be configured) indicating end of file.

Once you're done with a lexer, you must call

```
void lex_clear(lexer *l)
```

to free any attached data.

New token types (for example, for a new operator or keyword) can be implemented by adding a new entry into the enum in `lex.h`. You **must** 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. You should then modify the lexer to generate the token. Tokens are identified and generated by the lexer in multiple ways:—

- A preprocessor function gets the first look at a token, and identifies things like symbols and numbers. You can replace this with a custom version. The current preprocessor checks whether symbols match keywords (you can disable this with `lex_setmatchkeywords`).
- If the preprocessor function doesn't claim the token, the lexer tries to identify the token from a table of `tokendefn`'s in `lex.c`. This is used to identify keywords and operators. You can easily extend the token definition table, or replace it with your own. Each token definition can specify a function to call after the token is identified to perform further processing; this is done to lex strings, for example.

You can create custom lexers, entirely replacing morpho tokens. See the JSON parser in `classes/json.c` for an example.

Note that syntax highlighting in the CLI also depends on a `linedit_colormap` between token types and colors; you would need to add these there.

10.2 Parser

The parser implements the Pratt parsing algorithm[1]. You typically 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

```
bool parse_MY(compiler *c, void *out)
```

and call:

1. `parse_advance` and `parse_consume` to obtain tokens.

[1]Pratt, Vaughan. "Top down operator precedence." Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (1973). A very good discussion is also to be found in `https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/`

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 definition 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`.

You can create a custom parser by replacing the parser definition table.

10.2 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.

10.2 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 11

Debugger

The debugger is a challenging component because it interacts both with the virtual machine—and hence needs to know about its inner workings—and also the user. The overall design of *morpho* exposes virtual machines as opaque pointers to as much of the code as possible. Code that needs to work directly with the virtual machine must define `MORPHO_CORE` and import `core.h` and is referred to as *privileged*.

To enforce the separation, we divide debugging into multiple components:—

1. debugannnotations, which are a data structure that encodes annotations about the bytecode, e.g. associating registers with symbols and other metadata. This data structure is in libmorpho in `datastructures/debugannotation`, and is not privileged.

2. debugger, defined in debug/debug.c. This data structure maintains the state of the debugger, and provides a debugging API for debugger interfaces. Requests for information are fulfilled using `morpho_printf`. This code is privileged.

3. clidebugger, defined in `morpho-cli`. This implements an interactive REPL interface that allows the user to work with the debugger. It is not privileged, and exclusively works through the debugger API. The interface is described above in part 1.

Chapter 12

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.

12.1 Command language

Morpho communicates with *morphoview* via an imperative scommand language designed to be simple and human readable. Valid commands are given in table 12.1 together with their syntax. Commands and their elements may be separated by arbitrary whitespace. Additional commands may be provided in the future.

12.1 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 12.1: **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.

12.1 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.