UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

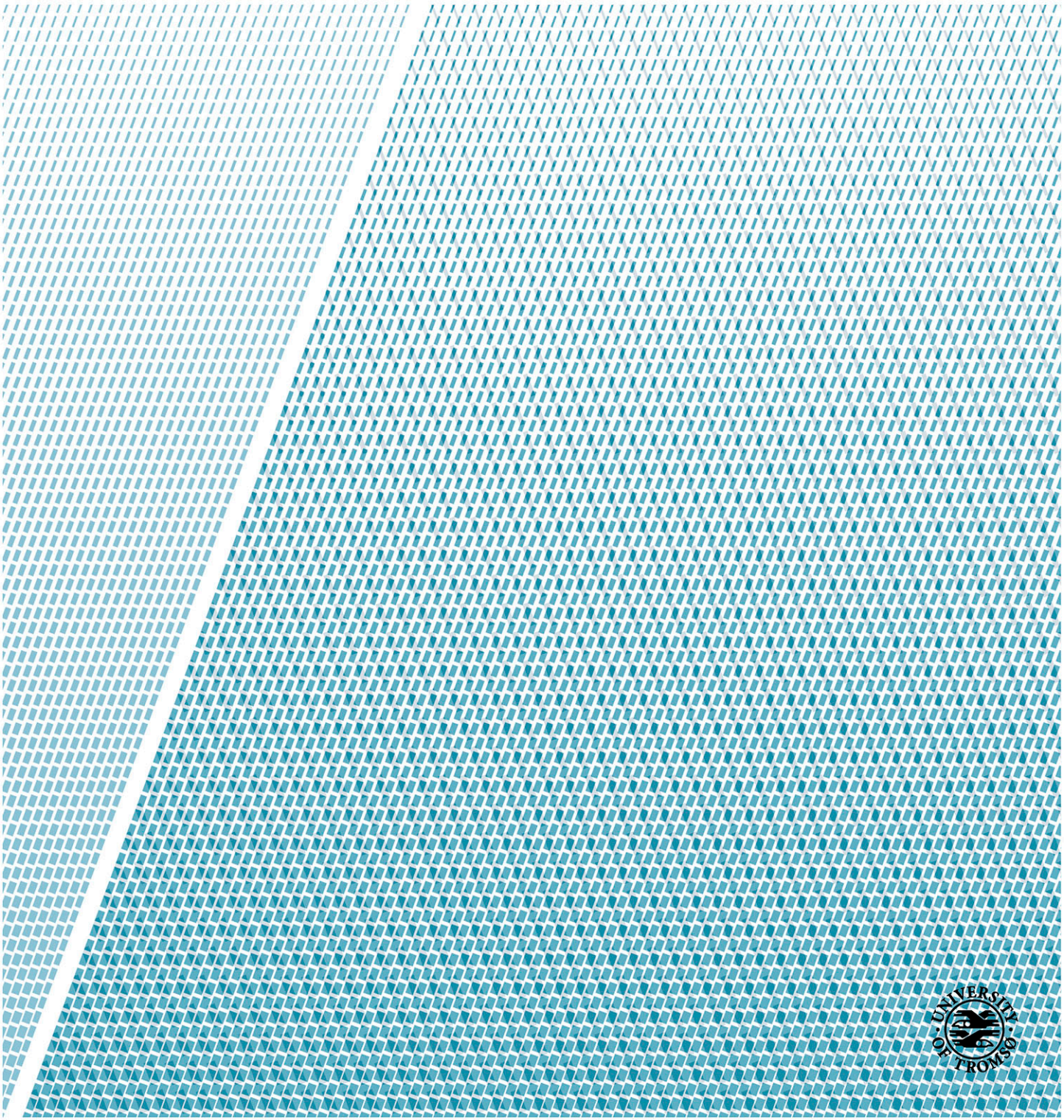
Faculty of Science and Technology
Department of Computer Science

Multiparadigm Optimizing Retargetable Transdisciplinary Abstraction Language

—

Ove Kåven

INF-3990 Master's Thesis in Computer Science - April 2015



Abstract

Scientists and engineers require ever more powerful software and hardware to analyze data and build models. Unfortunately, current solutions to the problem are often hard to use for scientists that are not software engineers. And software engineers often do not have the mathematical background to understand the scientific problem to solve.

This thesis describes MORTAL, a new general-purpose programming language and compiler for high-performance applications, which aims to bridge this gap by offering a multiparadigm programming environment that allows, for example, the mathematical formulae written by the scientist (perhaps using declarative programming) to be connected to the algorithms implemented by the software engineer (perhaps using object-oriented or functional programming) in a natural way, understood by both. The language will apply modern compiler and static analysis technology, along with contract programming, in new ways to both prevent bugs and improve runtime performance.

The implemented compiler is self-hosting and able to compile itself, showing that the language and its compiler, though not fully implemented yet, is already usable. The performance of MORTAL programs is also on par with the performance of C programs.

We believe MORTAL has the potential to become a useful language for solving many of the more demanding tasks of modern science.

Contents

1 Introduction

1.1 Background

Science is developing at an ever-accelerating pace, and so is the amount of science data collected by researchers all over the world [1]. Scientists and engineers require ever more powerful software and hardware to analyze data and build models. To help fulfill this demand, computer scientists have developed ever more sophisticated solutions, such as concurrency, parallel programming, distributed computing, supercomputing clusters, distributed storage, and so on.

Although these solutions work, it is hard for such scientists to implement them themselves [2], so they typically require trained computer scientists to implement effectively. As such, new kinds of computational problems need to be explained to qualified computer scientists, who will then try to write custom software for solving the problem, based on currently available technologies (and limited by their understanding of them). Unfortunately, not all computer scientists have the necessary mathematical background to understand the computational problem that needs to be solved – and worse, experienced computer scientists are usually in short supply. A solution is needed for either making the existing computer scientists more efficient, or reducing the need for them, or both.

In this thesis, we will explore a new general-purpose programming language that aspires to do both.

1.2 Why another programming language?

It is certainly true that many frameworks and programming languages have already been developed to help solve many of these problems. Unfortunately, most of them still require trained computer scientists to use effectively. And for those that don't (while still being expressive enough for most scientists), they are usually interpreted languages, which implies that many things that could be done to make the program run faster on

the available hardware (optimization) can't easily be done by the interpreter, but must be done by the programmer. (Examples of this may include common subexpression elimination, and loop optimizations such as loop-invariant code motion and strength reduction. Or even taking advantage of the vector (SIMD) instructions of modern CPUs and GPUs.) How to do this (or even knowing it's possible!) is not always intuitive to someone who isn't a computer scientist.

Conversely, many problems of a deeply mathematical or statistical nature can be difficult for many computer scientists to understand, and many of them will implement a computation without understanding what it does, and thus may be unaware of potential mathematical transformations of the problem that might increase numerical accuracy or reduce computation time.

We have investigated numerous programming languages (see Chapter 6), but none of them seem to solve the above issues adequately.

1.3 Primary requirements

For solving the above issues through developing a new language, we believe that it must have at least the following features:

1. *Multiparadigm:* it must be possible for the language to be used effectively by both software engineers (whose intuition tend to imperative or functional programming) and scientists (whose intuition tend to declarative programming). By having multiple paradigms in the same language, the mathematical formulae declared by the scientist can be connected to the computation frameworks and algorithm libraries implemented by the software engineer in a natural way, understood by both.

2. *Optimizing:* the language should be designed for minimal overhead, and for compiling to optimized machine code that can take full advantage of the fastest hardware available to the user (possibly supercomputing clusters).

3. *Retargetable:* the language should make it possible to abstract away platform specifics without losing performance. It should be possible to allow the same source code to compile for various desktop or mobile OSes, for GPUs (Graphics Processing Units), or, at some point, even FPGAs (Field-Programmable Gate Arrays). Ideally, the business logic should even be independent of the software libraries used to perform the computations.

4. *Transdisciplinary:* the language should be general-purpose, and make it possible to write programs that can combine knowledge from different domains. There should be no inherent limitations to what fields it can be used in.

5. *Abstracting:* the language should be easy and intuitive to use, and automate away as many implementation details as possible, allowing the programmer to focus on the concepts that are really of concern.

A promising way to achieve these goals without making the language impossibly complex is to have the language take full advantage of modern compiler technology, and more importantly, have it provide strong metaprogramming facilities. Metaprogramming can be roughly described as writing code that can generate or transform code. (Well-known facilities that can be used for this include C++'s templates [3], and Lisp's macros.) Sufficiently powerful metaprogramming systems even allow the construction of domain-specific languages (DSLs) within the host language. The new language should therefore strive to be flexible and provide a good metaprogramming system.

1.4 Contributions

- The design for the syntax of a new metaprogrammable language for use in high-performance applications.
  - To allow it to interoperate with existing libraries and frameworks, it initially implements the traditional procedural and object-oriented programming paradigms (including exception handling), with a few adaptations for metaprogrammability, automatic memory management, performance, and productivity. The corresponding declarative paradigms are not yet designed, but the syntax is left flexible enough to allow them to be integrated later. (This partially satisfies the *Multiparadigm* and *Retargetable* requirements.)
  - To make it customizable and be usable in different domains, it provides a number of overloadable operators and sufficient metaprogramming power to grant a natural syntax to the use of external libraries and frameworks. (This satisfies the *Transdisciplinary* and *Abstracting* requirements.)
  - To make it run at high performance on any available hardware, it is a compiled language that also makes provisions for runtime code generation and runtime algorithm specialization. (This satisfies the *Optimizing* requirement.)

  - To make it easy to learn for programmers familiar with other languages, it uses a classic curly-braces syntax, with a few adaptations to make the syntax ambiguity-free.
  - To facilitate writing bug-free programs, the language integrates the static analysis capabilities of modern compilers. In particular, the principles of contract-based programming (also known as «Design by Contract» [4]) are integrated as a compile-time static analysis and code optimization tool with zero runtime overhead, instead of its original role as a runtime testing aid. As far as we know, this approach has never been attempted before.

- The design and implementation of an optimizing compiler for said language.
  - It already implements a large part of the language design, and some of the memory management.
  - It is self-hosting (i.e., it is written in its own language, and compiles itself).
  - It currently generates C code, which can in turn be compiled to machine code. Other code generators are planned (needed for the *Retargetable* requirement).
  - It is released as open source.

Although the language's design and implementation is not yet complete, the current work already meets several of the major objectives, and work is ongoing to meet the remaining ones.

1.5 Thesis overview

- Chapter 2 describes the overall design of the language and its compiler.
- Chapter 3 describes the syntax of the language.
- Chapter 4 outlines what has been implemented so far.
- Chapter 5 evaluates the current implementation.
- Chapter 6 surveys a selection of other languages and frameworks.
- Chapter 7 is the conclusion.

2 Design

This chapter describes how the MORTAL language and its compiler is designed.

2.1 Principles

MORTAL is designed to be easy to learn and use, while providing powerful metaprogramming and resource management facilities that helps automate tasks that programmers may find tedious or confusing, as well as support for paradigms that allow scientists to express their problems in a simple, concise manner.

MORTAL aims to keep the core language intuitive, flexible, and customizable, leaving libraries and class frameworks able to define how the language should work with them, and to provide the necessary runtime support for MORTAL's more advanced programming models. The goal is to enable seamless, minimal-overhead integration between MORTAL programs and any number of external libraries, with the user-friendliness and expressivity of domain-specific languages, but within the framework of a general-purpose, optimizing language.

MORTAL tries to be a practical language, able to solve a wide range of real-world problems. Every MORTAL feature is designed to work together to build a language that balances expressive power, practicality, optimizability, machine verifiability, and user friendliness.

2.2 Static analysis

MORTAL aims to help programmers avoid bugs by performing extensive static analysis of the program. Static analysis is a difficult problem in general, in part because the analyzer does not really know what the programmer intended, and must infer it from the code. MORTAL approaches the problem by taking contract programming (also known as «Design by Contract» [4]), which in most other languages results in runtime checks, and turn it into a compile-time construct which allows the static analyzer to

know the programmer's intentions. The extra information allows the static analyzer to verify the correctness of the program more easily and efficiently than traditional static analysis, while potentially also catching more bugs. Types of bugs that can easily be caught this way include out-of-range arguments and buffer overflows.

In some cases, the extra information may also allow MORTAL to perform additional optimizations, resulting in faster code. An example of this would be in alias analysis, where contracts may be used to ensure that two parameters will not alias (refer to the same object). Parameters that may alias reduce optimization possibilities (such as reordering code and keeping values in registers), so telling the compiler that they won't alias may result in more optimal code.

2.3 Self-hosting

MORTAL's compiler is self-hosting. This means the MORTAL compiler is written in its own language. In addition to allowing MORTAL features to be used in the compiler itself, this makes it easier to embed parts of the compiler into MORTAL programs. A long-term goal of MORTAL is to make it simple to generate code at runtime (e.g., to speed up certain kinds of computations through runtime algorithm specialization), and being able to use the same code generator at compile time and runtime will make this easier to achieve.

Writing a compiler in its own language has certain challenges, such as the problem of how to compile the compiler from scratch, or how to recover from a broken compiler. Both of these problems are currently addressed by having the MORTAL compiler generate C code (Section 6.1.1.2), which is then checked into version control along with the MORTAL source code. A regular C compiler is then enough to compile a working MORTAL compiler. (The chicken-and-egg problem of how to write the first compiler was handled by first writing an initial (non-self-hosting) compiler in Python (Section 6.1.7.2), and using it to bootstrap the self-hosting compiler.)

2.4 Syntactic issues

MORTAL does not use header files, but allows modules to import each other directly. These imports may be circular, i.e., module A may import module B, and module B in turn may import module A. This has implications for MORTAL's syntax, because it means the compiler frontend must be able to parse module A without first parsing

module B, or vice versa. This is only possible if the syntax is sufficiently rigid and ambiguity-free. Thus, MORTAL uses a fairly fixed and non-customizable syntax, while leaving the corresponding semantics much more flexible and customizable.

Other advantages of using a fixed syntax is that it is relatively easy for IDEs to do syntax highlighting, and that the parser could run independently of the rest of the compiler (which happened to be useful when bootstrapping the self-hosting compiler).

The chosen syntax is described in Chapter 3.

2.5 Compiler frontend

Currently, the frontend parser is written in C. It uses a hand-written lexer (using the GLib library (Section 6.3.1) for string handling, error handling, and Unicode support), and a parser written in GNU Bison[1] (a tool which compiles context-free grammars described with a variant of Backus-Naur Form (BNF) to C code). The parser converts MORTAL source code to an Abstract Syntax Tree (AST), which the remainder of the compiler can work with.

The AST representation is designed to be serializable. This has the following advantages:

- For source files that does not change, their ASTs may be cached on disk, resulting in faster compilation times.
- For generating code at runtime, ASTs may be embedded into the generated executable.
- The compiler frontend could run independently of the rest of the compiler.

2.6 Initial passes

Since there are no header files, and objects may refer to each other circularly, at least two initial analysis passes must be used to augment the initial AST. The compiler's intermediate representation (IR) is similar to the original AST (uses the same classes), but is kept separate from the original AST, and contains additional annotations.

- All imported modules are loaded and parsed.

[1] https://www.gnu.org/software/bison/

- (Pass 1) For each loaded module, a skeletal IR is made from the AST. Special arguments, such as the `this` argument of methods, are added to the IR.
- (Pass 1) For each loaded module, each defined top-level object (such as types and functions) in the IR are entered into a symbol table.
- (Pass 2) Types of top-level objects and their parameters are determined by taking type annotations from the original AST, searching for the type definition in the symbol tables in scope, and adding the resulting types to the corresponding IR.
- (Pass 2) Implicit objects, such as vtables, are added to the IR.

2.7 Main passes

The main passes only need to be run on the module to be compiled, not on imported modules. They transform the IR into a form suitable for code generation.

- All statements and expressions in the original AST are evaluated and added to the IR.

When declarative paradigms are implemented, they may transform declarative programs into imperative ones at this point, by applying rewrite rules from imported modules.

- A control flow graph is constructed from the IR. (The control flow graph can immediately be used to find errors such as not returning a value.) [5]
- The dominators of the control flow graph are found, using the Lengauer-Tarjan algorithm [6] (with some optimizations described by L. Georgiadis et al [7]). Dominators and dominance frontiers are of interest in some types of analysis, in particular to transform code to SSA form (see below).
- Live variable analysis is performed, using the IR and the control flow graph. [8, 9]

From here, the IR can be transformed into Static Single Assignment (SSA) form using the algorithm described by R. Cytron et al [10], but this has not yet been implemented. SSA form is used in many modern compilers, as it facilitates advanced optimization and static analysis. In SSA form, every variable is assigned to exactly once. (If the original program assigns to a variable more than once, the original variable will map to multiple SSA variables, each being a different «version» of the original variable. Special

functions, called ϕ-functions, are inserted wherever it's necessary to select a version based on control flow. Dominance frontiers determine where this is necessary.) Due to SSA variables being effectively immutable after assignment, SSA form is comparable to functional programming paradigms, which makes SSA easy to analyze and optimize for many of the same reasons functional languages are. However, neither transforming into or back out of SSA form are trivial operations, and research is still being done in this area.

- Transparent structs and classes (Section 3.4.19) are rewritten away.
- Memory management (Section 2.9) is performed, using live variable information to minimize runtime overhead.
- The control flow graph and IR is serialized into a final form useful for code generation.

2.8 Backend

MORTAL is designed to support several possible backends (code generators). Since the IR given to the backend has the structure of an AST, it is fairly easy for backends to output source code for other languages, and have it resemble the original MORTAL program. Eventually, MORTAL aims to support generating C (Section 6.1.1.2), C++ (Section 6.1.2.3), OpenCL/CUDA (Section 6.1.8.1), and VHDL (Section 6.1.8.2). However, only the C backend is currently implemented.

A backend to generates machine code directly is also planned, by using LLVM (Section 6.3.2). The backend would convert MORTAL IR to LLVM IR, which can then be passed to LLVM for code generation. LLVM uses SSA form internally, which means there may be no need for the main passes to transform out of SSA in this case.

2.9 Memory management

MORTAL is designed to be able to manage memory without the need for a tracing garbage collector. Although using a tracing garbage collector with MORTAL should be possible, MORTAL primarily focuses on reference-counting-based memory management. This allows memory to be released as early as possible, in a deterministic way, and also allowing resources other than memory to be managed using the same mechanisms (i.e., MORTAL supports the RAII (Resource Acquisition Is Initialization) idiom).

The programming model used by MORTAL is inspired by the Vala programming language (Section 6.1.2.9), allowing user-defined types to choose between three memory-management strategies. MORTAL's type system enforces the chosen strategy, to the extent possible.

2.9.1 Struct types

Structs are value types. A variable or field of struct type holds the struct instance directly, and the struct instance data is destroyed when the corresponding variable or field goes out of scope.

- If a local variable is of struct type, the struct is allocated on the stack, and destroyed when the local variable goes out of scope.
- If a function parameter is of struct type (and is not an output parameter), the struct is copied onto the stack when the function is called, and destroyed when the function returns.
- If an instance data field is of struct type, the field is allocated as part of the instance allocation, and is destroyed when the instance is destroyed.

2.9.2 Non-reference-counted class types

Classes are reference types. A variable or field of class type holds a reference to a class instance. More than one variable or field may refer to the same class instance. However, a non-reference-counted instance may only have one owner at any time.

- The type of the owner's reference should be qualified with `owned`. When the owner ceases to exist (which includes being overwritten), the owned instance is also automatically destroyed, by calling its deallocator/destructor.
- The type of other references may be qualified with `unowned`. Unowned references do not affect the lifetime of the referenced instance. (Note that there may be a risk of dangling references, if the instance is destroyed while unowned references still exist. If this is possible, weak references may be a better option, if the class supports it. In the future, static analysis may be used to detect potentially dangling references.)

- If the class supports weak references, variables or fields (that are not the owner) may be qualified with `weak`. If the referenced instance ceases to exist, then weak references will be set to null. Local variables cannot be weak. (Also note that `weak` is an object qualifier, while `owned`/`unowned` are type qualifiers. This is because classes must store the memory address of weak references, but type qualifiers would also affect values that are not necessarily stored in memory, such as function return values. Weak references should be of unowned type.)

- Instances of class type are always allocated on the heap. Newly created instances must be assigned to an `owned` variable or field, or be returned from a function with an owned return type.

- Attempting to assign a reference that's not newly created (and is not an owned return value) to an owned reference (which includes returning it from a function with an owned return type) may result in the instance being copied, and the copy being assigned to the latter reference. If the instance cannot be copied (e.g., if the class does not have a copy constructor), this will result in a compilation error. (Exception: If the source reference is a local variable that's no longer used after the assignment, then MORTAL may do an ownership transfer instead of creating a copy. This will always succeed.)

- The following objects are `owned` by default:

  - Global variables that are not weak
  - Struct/class fields that are not weak
  - Function/operator return values, other than property getters
  - Local variables that has at least one newly created instance, or owned return value, assigned to it

- The following objects are `unowned` by default:

  - Weak variables/fields
  - Function/operator parameters
  - Property getter return values
  - Local variables that do not have any newly created instance, or owned return value, assigned to it

2.9.3 Reference-counted class types

Classes support reference counting if they define the appropriate methods. MORTAL will call the appropriate reference-counting methods automatically, but requires class frameworks to provide their own method implementations. Thus, the reference count itself is under the control of the frameworks themselves, and there is also no need for a special «smart pointer» type.

In general, thread-safe reference counting can be slow (atomic operations may stall the CPU pipeline). For this reason, MORTAL goes to lengths to minimize the number of reference-counting operations.

- Most of the rules about `owned`, `unowned`, and `weak` still applies to reference-counted class types. However, there can now be more than one owned reference. To minimize reference-counting method calls, only owned references affect the reference count.
- A newly created instance should be given a reference count of one.
- When an owner ceases to exist, the reference count is decremented. The class should destroy itself when the reference count becomes zero.
- Unlike non-reference-counted class types, attempting to assign a reference that's not newly created (and is not an owned return value) to an owned reference does not result in a copy, but in incrementing the reference count. (In case of an ownership transfer, the reference count does not change.)
- Reference cycles can usually be avoided by making references in one direction (e.g., from child to parent) `unowned` or `weak`. Static analysis may be used to find and warn about potential reference cycles.

3 Syntax

This chapter describes the syntax of MORTAL's imperative (procedural and object-oriented) paradigm, as currently envisioned and partially implemented. In the future, MORTAL also plans to support certain declarative paradigms, but they are not described here (see Section 8).

3.1 General

MORTAL's syntax is a variant of the classic free-form curly-braces syntax, similar to C++ and Java. (This is a widely used style, which might help programmers familiar with other languages feel more at home in MORTAL, and the curly brace is a reasonable way of delimiting various language constructs.) Normally, imperative statements must be terminated by either a semicolon, or, where allowed, a curly-brace-delimited block. It is, in general, not necessary to use a semicolon after a closing curly brace.

MORTAL uses block scoping. Variables defined within an inner block are not accessible from outer blocks.

The syntax may change. For example, it may be possible for a future revision of the language to include an option to make semicolons completely optional, relying on line breaks instead.

3.2 Top-level syntax

The preferred file extension for MORTAL source code is `.mtl`. Each source file represents an independently compiled module. This section gives an overview of the constructs that are valid in top-level (module) scope.

3.2.1 Comments

To write a comment that runs to the end of the line, use a double slash (`//`). Otherwise, a comment block should start by a slash-asterisk (`/*`) and end with an asterisk-slash (`*/`). Comment blocks may be nested.

3.2.2 Identifiers

Identifiers are used to name things, such as types and variables. They may use ASCII or Unicode alphanumeric characters, plus ASCII underscore. The first character of an identifier cannot be a numeric digit, but any subsequent character may be.

3.2.3 Imports

Source modules will usually need to access symbols from other source modules. To do this, other source modules can be «imported» using the `import` statement. The `import` statement can have any number of string arguments, where the strings represent file paths (absolute or relative). The `.mtl` extension may be omitted.

Example:

```
import "stdio"; // imports the stdio module
```

3.2.4 Namespace blocks

Code may be placed inside namespace blocks. All symbols defined within that block will then be accessible from other blocks and modules by using explicitly qualified references, or by importing that namespace with the `using` statement (Section 3.2.5).

Example:

```
namespace ABC {
  // all symbols defined here are placed in the "ABC" namespace
  class foo;
}

bar: ABC.foo; // qualified reference to "foo" of "ABC"
```

3.2.5 Namespace imports

To avoid the need for qualifications, the symbols of a namespace can be imported into the current block with the `using` statement. (All imported namespaces are searched in import order, after searching the symbols of the current block.)

Example:

```
using ABC;

bar: foo; // unqualified reference to "foo" within "ABC"
```

3.2.6 Variables

Ordinary variables can be defined in module, namespace, function, or method scope. (If they are defined in `struct` or `class` scope, they're called fields instead.) Their lifetime is limited to the lifetime of the scope they're defined in. All variables have a type, and may optionally have an initializer.

If no initializer is provided, a default initializer will be used. (Integers and floating-point values are initialized by zero by default. References are initialized to `null` by default. For user-defined value types, the constructor is called.)

When a variable is defined within a function/method, the compiler may be told to infer its type from its initializer, to make defining temporary variables easier.

Examples:

```
v1: int; // v1 is a variable of type "int", initialized with zero
v2: int = 2; // v2 is initialized with 2
// assume that "foo" is a class
v3: foo; // v3 is an (initially null) reference to "foo"
v4: foo = foo(); // v4 refers to a new "foo" object
v5: foo = func(); // v5 refers to the object returned by func
v6: foo(); // same as v4 (syntactic sugar)
// assume that "bar" is a struct
v7: bar; // bar's default constructor is used
v8: bar = func(); // v8 copies the object returned by func
v9: bar(); // same as v7
// type inference is possible within functions/methods
v10 ::= foo(); // same as v4
```

```
v11 ::= func(); // same as v5/v8
```

3.2.7 Typedefs

The `typedef` statement allows making types from other types.

Examples:

```
// Define Counter as an alias for "long"
typedef Counter: long;
// Define CounterList as a specialization of the
// generic container List for the Counter type
typedef CounterList: List<:Counter>;
// Define ConstCounter as an immutable long, and
// let external C code refer to it as "c_counter_t"
typedef ConstCounter: const long => __c_type.c_counter_t;
```

3.2.8 Enums and flags

The `enum` and `flag` statements can be used to define certain kinds of value types. Internally, enums and flags are integers, except that particular values have names. Enums hold one value at a time, while flags allow bitwise combinations of multiple values (or none). Because of the bitwise nature of flags, named flag values should, in general, be powers of 2. However, there's often reason to give names to particular bit combinations.

Examples:

```
enum ErrorCode {
  NoError = 0,
  PEBKAC  = 1
}
results: ErrorCode = ErrorCode.NoError;

flag ProcessOptions {
  WithBacon  = 1,
  WithCheese = 2,
  WithFries  = 4,
  WithEverything = 7 // all of the above
}
```

```
requirements: ProcessOptions =
  ProcessOptions.WithBacon |
  ProcessOptions.WithCheese;
```

3.2.9 Structs

The `struct` statement is used for defining new value types. (Value types are, in general, allocated on the stack, and passed by value (i.e. copied) when calling other functions/methods. However, it's still possible to pass structs by reference; see Section 3.5.9.) Structs may contain data fields, methods, operators, and other type definitions. Structs can inherit from other structs, but does not, in general, support subtype polymorphism (subclassing). They do, however, support parametric polymorphism (generics).

Example:

```
struct Counter {
  public count: long;

  tick() {
    count = count + 1;
  }
}
...
count: Counter;
count.tick();
```

Structs are described in more detail in Section 3.4.

3.2.10 Classes

The `class` statement is used for defining new reference types. (Reference types are, in general, allocated on the heap, and passed by reference when calling other functions/methods.) Classes can do everything structs can, but are more flexible, and supports subtype polymorphism (subclassing). However, class lifetime management is more complex, which may make them slower, depending on how they're used.

In order to be compatible with as many object-oriented class frameworks as possible, MORTAL leaves it to the programmer to implement certain aspects of the meta-object protocol, such as Run-Time Type Information (RTTI) and reference-counted memory

management. This is done by defining certain special methods and operators. Hence, users who want to use RTTI are expected to inherit from a base class which implements it, and each class framework is expected to provide its own base class that implements its own style of RTTI.

Example:

```
abstract class Geometry {
  public center_x: int;
  public center_y: int;

  public constructor(center_x: int, center_y: int);
  public abstract virtual draw();
}

class Circle: Geometry {
  public radius: int;

  public constructor(center_x: int, center_y: int, radius: int);
  public override draw() { ... }
}
...
obj: Geometry = Circle(50, 50, 20);
obj.draw(); // calls Circle's draw method
// The following would be legal if Geometry implemented RTTI
if obj is Circle { ... }
```

Classes are described in more detail in Section 3.4.

3.2.11 Interfaces

The `interface` statement can be used to define a special type of class that do not contain code or data, only abstract methods and operators. That is, they function as protocols, not as implementations. Although abstract classes can do the same, using interfaces instead avoids certain issues with subtype polymorphism (subclassing) and multiple inheritance. Since interfaces, unlike abstract classes, cannot contain code or data, implementations are never inherited from them, which avoids any conflicts or ambiguities that may arise when a class inherits from multiple base classes. When

classes can only inherit implementations from one base class, and all other bases must be interfaces, such conflicts can never occur.

However, MORTAL does not restrict multiple inheritance in this way, meaning it's possible for such conflicts to occur in MORTAL. MORTAL allows the use of interfaces to avoid them, but also allows conflicts to be resolved in other ways (see Section 3.4.12). However, some class frameworks may restrict multiple inheritance, requiring the use of interfaces.

Interfaces are described in more detail in Section 3.4.

3.2.12 Functions

Ordinary functions can be defined in module or namespace scope. (If they are defined in `struct`, `class`, or `interface` scope, they're called methods instead.) They can currently not be nested, but this might change in the future.

A function is intended to contain a piece of code that operates on its arguments (if any), and (optionally) returns a value. (Functions that operate on other objects than their (explicit or implicit) arguments are possible, but such functions are said to have «side effects», and some programming paradigms may restrict their use.) The argument types (if any), and the return type (if any), must be explicitly declared. (Type inference for the function signature is not currently possible.) On the other hand, functions support parametric and ad hoc polymorphism (generics and overloading). Furthermore, if a function is declared `transparent`, it can work much like a macro.

Example:

```
// Function that squares an integer
square(x: int): int {
  return x * x;
}
// Can be written shorter, like this
square(x: int): int { x * x }
// Generic max function for any type
// (declared transparent, so it works like a macro)
transparent max<T>(x: T, y: T): T { x >= y ? x : y }
```

Functions are described in more detail in Section 3.5.

3.2.13 Delegates

The `delegate` statement can be used to define types of references to functions or methods, along with any contextual information. The function/method signature (parameter and return types) is defined as part of the delegate type. Actual references can then be created by defining variables of the delegate type. Any function/method that match the delegate's signature can be assigned to such a variable, and the variable can then be called in order to call the referenced function/method. A delegate variable can also store contextual information, typically the instance to use if the variable refers to an instance method. Once anonymous (lambda) functions are implemented, delegate variables should also be able to store references to their closures.

If the delegate statement is qualified with `static`, it defines a delegate type that does not store contextual information. Static delegates are thus equivalent to C function pointers.

Example:

```
static delegate Xform(x: int): int;
// two functions with matching signatures
identity(x: int): int { x }
square(x: int): int { x * x }
...
func: Xform; // define delegate variable
func = identity;
y = func(5); // calls identity, y = 5
func = square;
y = func(5); // calls square, y = 25
```

3.2.14 C compatibility

MORTAL tries to make it straightforward to write wrappers for external libraries with a C interface. Note that such external libraries need not actually be implemented in C, as long as the public interface is compatible with C (which most are).

When writing a module that is intended to wrap an existing C library, the C header files to use can be declared with the `__c_include` statement. The definitions from that C header file can then be used when prefixed with `__c_lib`, `__c_type`, `__c_ptr`, or `__c_ptrtype`.

Example:

```
__c_include "time.h"

// define t as a variable of C type "time_t"
t: __c_type.time_t;
```

Wrapper modules can be completely inlined (i.e., users will link directly to the C library, not the compiled MORTAL module). To declare such a wrapper module, all `__c_include` statements must be at the top of the module, and be qualified as `transparent`. Also, all definitions in the file must be declared in an inlineable way (e.g., as `transparent`, see Section 3.4.19), or as aliases (which, in cases of addressable objects, such as functions, must be qualified with `extern`).

Example:

```
transparent __c_include "time.h"

// define "UnixTime" as an alias of the C type "time_t",
// which behaves like a signed long integer
typedef UnixTime: long => __c_type.time_t;

// define "time" as an alias of the C function "time",
// which takes a (possibly null) write-only reference
// to an argument of type UnixTime, and returns a UnixTime.
extern time(out? timer: UnixTime): UnixTime => __c_lib.time;

// define "Time" as a wrapper for the C structure "struct tm"
// (the generated code will use struct tm directly)
transparent class Time: __c_ptr.struct.tm {
  // struct members can be declared using the original name,
  // or with different names
  tm_sec: int; // declare original name
  sec: int => __c_lib.tm_sec; // declare shorter name
  ...
  static extern localtime(timer: UnixTime): Time => __c_lib.localtime;
  // for non-static methods, the first argument is a reference
  // to the struct/class itself
  extern asctime(): CString => __c_lib.asctime;
  // since the class is transparent, all methods will be inlined
```

```
    strftime(fmt: CString): CString {
        ... __c_lib.strftime(..., fmt, this) ...
    }
}
```

The prefixes for accessing C types and objects are:

Prefix	C	MORTAL	Examples
`__c_lib`	Addressable object		`stdout`, `printf`
`__c_type`	Non-pointer type	Value type	`off_t`, `struct in_addr`
`__c_ptr`	Non-pointer type	Reference type	`FILE`, `struct tm`
`__c_ptrtype`	Pointer type	Reference type	`iconv_t`, `gpointer`

3.3 Types

MORTAL uses a nominal, static type system. (It is the most common type system among compiled languages, including C/C++, which MORTAL aims to be compatible with.)

Every addressable object in MORTAL must have a type. This section is about how to use types; defining new types is covered in other sections. Note that high-level data types such as matrices, strings, and associative arrays are intended to be provided by libraries (such as the GLib wrapper in MORTAL's standard library), not by the core language, and are not discussed here.

3.3.1 Variables and fields

For data objects, such as variables and fields, the type must be declared after a colon.

Example:

```
v: Circle; // v is a variable of type "Circle"
```

For local variables inside functions, it's also possible to use type inference.

Example:

```
v ::= 5; // since the type of 5 is "int", v is inferred to be "int".
```

3.3.2 Nullable (maybe) types

Reference types (classes) can be nullable, i.e., variables of such types can refer to `null`. To declare that a variable can refer to `null`, add a question mark after its type.

Example:

```
v: Circle?;
...
v = null;
```

For local variables inside functions, this is not necessary. Local variables are always formally nullable, and static analysis is then used to determine whether a local variable can actually be null at any given point.

3.3.3 Const types

Types can be declared as `const`, which basically makes the corresponding objects impossible to modify. For example, it will not be possible to call methods that are not qualified with `const`, as non-const methods might modify the object.

Note that constness applies to the object reference, not to the object itself. There may be both const and non-const references to the same object. The object cannot be modified through the const references, but can be modified through the regular references.

Example:

```
v: const Circle;
// A non-const reference can be assigned to
// a const reference, but not vice versa.
v = Circle();
```

3.3.4 Type parameters

Parametric types may take arguments in the form of either other types, or as expressions. (See Section 3.4.14.) The argument list is enclosed in angle brackets, and arguments are delimited by commas. Type arguments must be prefixed by colons, and expression arguments enclosed in square brackets.

Example:

```
import "glib";
// Create a GLib hash table using the type
// "CString" as the key type, and "int" as
// the value type.
table: GLib.HashTable<:CString, :int>;
```

3.3.5 C arrays

It is possible, but intentionally somewhat inconvenient, to define C arrays in MORTAL. Defining C arrays directly is discouraged because they do not have strict bounds checking, making overflows possible. Instead, class frameworks are encouraged to define their own safe array types that add bounds checking (possibly using contract programming (Section 3.5.14), which in MORTAL need not add any runtime overhead), and programmers should use such types instead. Thus, C arrays should only be used in the internal implementation of safe array types, or for compatibility with external libraries written in C.

To define a C array in MORTAL, use the builtin parametric type `__c_array`. It takes two parameters, a type and an expression. The type may be any type of constant size (including another array). The expression represents the array size. The array size may be omitted in function parameters, as the size does not need to be known in this case. However, it's still good practice to specify it whenever it is known.

Example:

```
// Define array holding 10 integers.
arr: __c_array<:int, [10]>;
// C arrays indices start at 0.
arr[0] = 1; // Set the first entry to 1.
arr[9] = 5; // Set the last entry to 5.
```

3.3.6 Type qualifiers

Types may have the following qualifiers applied to them.

`const`	The reference is immutable. See Section 3.3.3.
`owned`	The reference is owned. See Section 2.9.
`unowned`	The reference is unowned. See Section 2.9.

3.4 Structs, classes, and interfaces

Structs, classes, and interfaces allow defining the various entities within a system in an abstract, object-oriented way. A typical struct or class can have any number of instances, called objects. Structs and classes may contain data fields, methods, operators, and other type definitions. Interfaces may contain abstract methods, operators, and other

type definitions.

The following sections mostly discuss classes. It should be understood that anything said about classes also applies to structs and interfaces, unless otherwise specified. Anything said about class instances also applies to struct interfaces, unless otherwise specified. Interfaces cannot have instances.

3.4.1 Instance data

Per-instance data fields are defined using the same syntax as variables. Initializers are allowed; they are evaluated when the instance is constructed, just before the user-defined constructor code (if any) is executed. If no initializer is provided, a default initializer will be used, just as with variables. Any particular instance's instance data is automatically deleted when the instance ceases to exist.

Example:

```
class SampleData {
  public data: int = 5; // initialized to 5 on construction
  public count: int; // initialized to 0 on construction

  public tick() { count += 1; } // increase count
}
```

3.4.2 Static (class) data

If a data field is qualified with `static`, then the field becomes class data, instead of per-instance data. Class data exists independently of any class instances, and instances do not get their own copy of it; all instances access the same field. Class data fields are typically initialized whenever the module containing them is loaded, and deleted when the module is unloaded. Typically, this means on program startup and program shutdown, respectively.

Example:

```
class GlobalData {
  public static count: int; // initialized to 0 on program startup

  public static tick() { count += 1; } // increase count
}
```

3.4.3 Constructors

Constructors are used to initialize struct or class instances after memory has been allocated for them. They may be user-defined or automatically defined, or a combination of both. If MORTAL's automatically-generated code suffices, the constructor's body may be omitted. Like ordinary methods, constructors can be overloaded. However, they cannot return a value.

Two special types of constructors exist:

- The default constructor. This is the constructor that can be called without arguments. For struct types, it is automatically used when a variable or field is defined without an initializer.

- The copy constructor. This is the constructor that can be called with a single argument, with a type of the struct/class itself. For struct types, it is automatically used when a variable or field is defined with an initializer. For class types, it is automatically used when a non-reference-counted object needs to be copied (see Section 2.9).

MORTAL constructors do the following:

- Calls the constructors of all base classes. By default, their default constructors are called, but it's possible to specify other constructors.

- For any instance fields that have the same name as a constructor parameter, initializes the instance field using the corresponding argument.

- If a constructor parameter is named _ (an underscore), initializes any remaining instance fields that have the same name as some field in the object referenced by the corresponding argument.

- Runs the initializers of all remaining non-inherited instance data fields, if any.[1]

- Executes any user-defined constructor code.

Example:

[1] This may be redundant if the user-defined constructor code also initializes the field. If so, the optimizer would remove the redundant initialization from the final code.

```
abstract class Geometry {
  public center_x: int;
  public center_y: int;

  // Default constructor. Auto-initializes fields to 0.
  public constructor();
  // Copy constructor.
  public constructor(_: Geometry);
  // Auto-initializes fields to constructor's arguments.
  public constructor(center_x: int, center_y: int);
}

class Circle: Geometry {
  public radius: int;

  // Default constructor. Auto-initializes fields to 0.
  public constructor();
  // Copy constructor.
  public constructor(_: Circle);
  // Calls Geometry's non-default constructor,
  // and auto-initializes radius field.
  public constructor(c_x: int, c_y: int, radius: int) :
    Geometry(c_x, c_y);
  // Calls Geometry's non-default constructor,
  // and "manually" initializes radius field.
  public constructor(c_x: int, c_y: int, c: Circle) :
    Geometry(c_x, c_y)
  {
    radius = c.radius;
  }
}
```

3.4.4 Destructors

Destructors are used to clean up struct and class instances before their memory is freed. They may be user-defined or automatically defined, or a combination of both. If MORTAL's automatically-generated code suffices, the destructor's body may be omitted. Like ordinary methods, destructors can be overloaded. They can also return a value.

A special type of destructor exist: the default destructor. This is the destructor that can be called without arguments. For non-reference-counted objects, it is automatically used when the object goes out of scope, or its owner is deleted.

MORTAL destructors do the following:

- Executes any user-defined destructor code.
- Runs the default destructors of all non-inherited instance data fields.
- Calls the default destructors of all base classes.
- Returns the value returned by the user-defined destructor code, if any.

Example:

```
class SampleData {
  public count: int;

  // Default destructor. Returns the final value of the counter.
  public destructor(): int { count }
}
```

3.4.5 Allocators and deallocators

Allocators and deallocators only apply to classes, not to structs. Struct memory allocation cannot be overridden.

Constructors and destructors do not allocate or free the memory on their own. By default, MORTAL allocates class objects on the heap using `malloc` and frees them using `free`, but if desired, this can be overridden by defining allocators and deallocators. This is of particular interest for wrapper modules, which often want to wrap external allocators. (An example is the C library's `fopen`, which allocates memory for the object it constructs.)

To override MORTAL's default allocation strategy, every public constructor should have a corresponding allocator, and the allocator should call it explicitly. Alternatively, the class can define only allocators, and not depend on constructors at all. (However, if the class is subclassed, the subclasses call up to the base constructors, not the base allocators.)

If an allocator or deallocator is defined without a body, MORTAL generates a default body that does the same thing that MORTAL would do if no allocator/deallocator was defined. This can be useful if such methods need to be made available to external C code.

Example:

```
class SampleData {
  // Default allocator with default body,
  // wrapping the default constructor.
  public new();
  // Non-default allocator.
  public new(init: int) {
    data: SampleData = __c_lib.malloc(sizeof(SampleData));
    data.constructor(init);
    return data;
  }
  // Default deallocator with default body,
  // wrapping a default destructor that
  // returns an integer.
  public delete(): int;
}
```

3.4.6 Initializers and deinitializers

Similar to static constructors and destructors in other languages, initializers and deinitializers can be used to initialize and clean up the classes themselves, rather than their instances. For example, initializers can be used to initialize the static fields of a class, or auto-register the class in a class factory. Initializers and deinitializers cannot have arguments or return values.

Example:

```
class GlobalData {
```

```
  public static count: int;

  initializer() {
    count = load_from_file();
  }
  deinitializer() {
    save_to_file(count);
  }
}
```

3.4.7 Instance methods

Instance methods are defined using the same syntax as functions. Within instance methods, a special variable, `this`, always refers to the current instance. Using it directly is rarely necessary, however; it is implicitly used when referring to any instance data field or method.

Example:

```
class SampleData {
  public count: int;

  // Implicit reference to "this"
  public get_count(): int { count }
  // Explicit reference to "this"
  public inc_count() { this.count += 1; }
}
```

Methods are described in more detail in Section 3.5.

3.4.8 Static (class) methods

If a method is qualified with `static`, then the method becomes a class method, instead of an instance method. Class methods work independent of any class instances, and cannot implicitly refer to any instance data field or method. In particular, the special variable `this` does not exist within a static method. Static data, however, is accessible from static methods.

Example:

```
class GlobalData {
  public static count: int;

  public static tick() { count += 1; }
}
```

In some languages, class methods get a `this` variable that refers to the class object itself, rather than to a class instance. (In such languages, class objects are instances of a metaclass type.) This kind of class method do not currently exist in MORTAL. but metamethods offer comparable functionality (Section 3.4.15).

3.4.9 Properties

Properties allow objects to expose state in a safe way. They offer the same encapsulation safety that accessors and mutators do in many object-oriented languages (such as C++ and Java), but are easier to use, since they are used just like regular data fields. When a property is read from, the getter is called, and when it's written to, the setter is called. The getter is expected to return a value. The setter gets a single argument, named `value`, which it can use to update the object's state. If no setter is defined, then the property is read-only.

Example:

```
class SampleData {
  private real_count: int;

  public my_count: int {
    get { real_count } // returns real_count
    set { real_count = value; } // updates real_count
  }
}
...
data: SampleData();
data.my_count = 10; // calls the setter (with value 10)
x = data.my_count; // calls the getter
```

3.4.10 Indexers

Indexers allow objects to act as key-value stores. (If the key is an integer, then the object can act as an array.) Indexers work like properties, except that they have additional arguments in the form of keys. Indexers can have more than one key (e.g., for matrices and multidimensional arrays), and they can also be overloaded. Indexer keys are given in square brackets.

Example:

```
class SampleData {
  private data: int;

  public [key: int]: int {
    // returns data if key is 0, otherwise returns -1
    get { key == 0 ? data : -1 }
    // updates data if key is 0, otherwise does nothing
    set { if key == 0 then data = value; }
  }
}
...
data: SampleData();
data[0] = 10; // calls the setter (with key 0, value 10)
x = data[0]; // calls the getter (with key 0)
```

3.4.11 Operators

By overloading operators, objects can make it possible to work with them in familiar and comprehensive ways. Modern science makes use of many high-level operators (e.g., matrix multiplication and division), and so MORTAL defines a large number of operators that can be overloaded by libraries that implement the functionality that the programmer needs. (For a list, see Section 3.6.3.)

Example:

```
struct ComplexNumber {
  public r: double;
  public i: double;
  public constructor(r: double, i: double);
```

```
  // Adding two complex numbers together
  // will create a new complex number.
  public + (x: ComplexNumber): ComplexNumber {
    ComplexNumber(r + x.r, i + x.i)
  }
}
```

Operators are described in more detail in Section 3.5.

3.4.12 Inheritance

In object-oriented programming, there are two aspects to inheritance. Implementation inheritance allows child classes (subclasses) to extend or modify the functionality of one or more parent classes (superclasses or base classes), without needing to reimplement everything. The fields, methods, and operators of the base classes become part of the child class, or can be overridden as necessary. Protocol inheritance, on the other hand, allows for subtyping. When inheriting from another class, you generally do both, but this section will focus on implementation inheritance. (Protocol inheritance is covered in the next section.)

MORTAL supports multiple inheritance; structs and classes may inherit from any number of other structs and classes. Also, classes may inherit from structs and interfaces. However, whenever multiple inheritance is available, there's generally a risk of conflicts between the members of the base classes. Furthermore, if these base classes have a common base class, then that base class may need to be inherited more than once, and its methods and operators may be overridden in different ways (the «Diamond Problem»). MORTAL aims to allow conflict resolution using the following approaches:

- If all bases (possibly except one) are interfaces (or mixins, see Section 3.4.19), then there can be no conflicts. If a member of a base interface has the same name as a member of a base class, then that interface member is simply implemented by that base class. Also, although interfaces can themselves inherit, they can't contain code, so they can't refer to their base classes in a particular way. Hence, their base classes do not need to be inherited in a particular way, so duplicates can safely be removed.

- Class frameworks may optionally forbid multiple inheritance of non-interface types, thus making the above rule the only allowed rule for that class framework.

- Abstract members cannot conflict, for the same reason that interface members cannot conflict. (However, bases of abstract classes can.)
- Referring to a particular base class's version of a member is possible by explicitly qualifying the reference with the appropriate base class (whether or not a conflict exists). For difficult situations (e.g., accessing a particular version of a common base class), multiple qualifiers may be necessary.
- Classes may indicate which member versions they wish to inherit with `using` statements.
- Otherwise, unqualified member references can be resolved by ordering all base classes in some reasonable order of preference (e.g. breadth-first search). C3 linearization [11] offers a consistent way to do this, and is gaining in popularity among other languages. MORTAL will thus also use this ordering.
- At some point, to solve the issue of a common base class, MORTAL might also support C++-like virtual inheritance. However, it is hoped that interfaces are a good enough solution for the problems that virtual inheritance would solve, so that MORTAL might not need it.

For overloaded methods and operators, there are additional complications.

- All overloads for a particular method name are, by default, taken from the same class. That is, if a child class defines a method with the same name (but different parameters) than a method in a base class, then the base class's version is hidden, and only accessible through qualified references. Similarly, if two base classes defines methods with the same names, then only the versions in one of the base classes (chosen using the rules above) will be visible.
- Classes that want to combine overloads from their base classes (possibly with their own overloads) may indicate this with `using` statements.

Example:

```
abstract class Vehicle {
  public abstract drive();
  public abstract wash();
}
```

```
// LandVehicle and WaterVehicle inherits from Vehicle
// and implements the "drive" and "wash" methods.
class LandVehicle: Vehicle {
  public drive() { ... }
  public wash() { ...}
}

class WaterVehicle: Vehicle {
  public drive() { ... }
  public wash() { ...}
}

class AmphibiousVehicle: LandVehicle, WaterVehicle {
  // Explicitly indicate which "wash" to inherit
  using WaterVehicle.wash;
}
...
craft: AmphibiousVehicle();
// Since LandVehicle is the first base class,
// LandVehicle.drive() is chosen here.
craft.drive();
// WaterVehicle.wash() is used here.
craft.wash();
```

3.4.13 Subtype polymorphism

In object-oriented programming, protocol inheritance allows for subtyping by allowing subclass instances to be used as if they were superclass instances, i.e., the subclass exposes the same methods and operators as the superclass, and can be used in place of the superclass, but may have a different implementation.

In order for a class to support subtype polymorphism, it must allow its methods and operators to be overridden (reimplemented) by subclasses. This can be done by qualifying them with `virtual`. Subclasses can then override virtual methods by defining its own implementation of them, qualified with `override`.

MORTAL interfaces are implicitly polymorphic; all the methods and operators of an

interface are implicitly virtual and can (and usually must) be overridden by its subclasses. However, non-interface classes must explicitly qualify its methods and operators with `virtual`. Structs do not support subtype polymorphism at all.

Declaring a member as virtual means that its implementation may need to be selected at runtime, based on the instance's dynamic (runtime) type. For this reason, calling virtual methods may be slightly slower than calling non-virtual methods.

A subclass may declare that a virtual method cannot be overridden further by qualifying its implementation with `final`. In some cases, this may allow the compiler to select the implementation at compile time instead of runtime.

Example:

```
interface Washable {
  wash();
}

class LandVehicle: Washable {
  public override wash() { ... }
  public virtual drive() { ... }
}

class Automobile: LandVehicle {
  public override drive() { ... }
}
...
car1: LandVehicle = Automobile();
car2: Washable = Automobile();
// Automobile.drive() is used here.
car1.drive();
// LandVehicle.wash() is used here.
car2.wash();
// Note: since "drive" is not in Washable,
// car2.drive() is not valid.
```

3.4.14 Parametric polymorphism

Parametric polymorphism allows a class to operate on data of arbitrary types, without significantly losing type safety. In many languages, this is called generics. For example, a generic array class can provide a single array implementation that will work for any data type. Although subtype polymorphism (or even untyped pointers) can be used for this purpose, this may involve undesirable typecasting (note the inability to call `car2.drive()` in the previous section). With parametric polymorphism, the compiler can ensure type safety and remove the need for explicit typecasts.

By default, MORTAL generics employ type erasure (much like Java). This implies that the class implementation itself does not have direct knowledge of the type used, and may internally operate on untyped pointers (unless the class restricts the allowable types, e.g. if it's a specialization). The actual type to use is known only to the user of the class (although there are ways around this, such as implicit parameters (Section 3.5.5)). Between the class and its users, the compiler will automatically insert typecasts where necessary, preserving type safety. If the actual type is a value type, such typecasts may perform conversion into a reference type by allocating memory («boxing»). Many third-party data structure libraries (such as GLib) have interfaces based on untyped pointers, and thus work well with type erasure.

Example:

```
// T represents the parametric type, here unrestricted.
// This container stores a single object of type T.
class Container<T> {
  static delegate Destroy(data: T);
  sz: int;
  destroy: Destroy;
  data: T?;
  // Implicit parameters offers a way around
  // type erasure. Here the constructor
  // receives the size of T, as well as
  // its deallocator as a static delegate,
  // used as a callback in the destructor.
  constructor(implicit sz: int = sizeof(T),
              implicit destroy: Destroy = T.delete);
  destructor() { if data !== null then destroy(data); }
```

```
  set_val(new_data: T) { data = new_data; }
  get_val(): T { data }
  get_size(): int { sz }
}
...
// Create a container to store an integer.
// Type arguments must always be preceded by a colon.
intstore: Container<:int>();
intstore.set_val(42); // boxes and stores the integer
x = intstore.get_val(); // retrieves and unboxes the integer
```

Parameters can have defaults. Example: `class Container<T = int>`

Polymorphic classes can be partially or fully specialized. To partially specialize, define a class implementation with stronger restrictions than the more general implementation. To fully specialize, define a class implementation for a particular type. MORTAL will choose the most specialized implementation that matches the parameters. (In cases of ambiguity, a compilation failure may result.) Types can be restricted/specialized in the following ways:

- The type can be qualified using `class`, `struct`, or `delegate`. This restricts the type to either a reference type, a value type, or delegate type, respectively. Example: `class Container<class T>`
- The type can be restricted to subclasses of a particular superclass or interface. Example: `class Container<T: Base>`
- A particular type can be given (full specialization). Example: `class Container<:int>` (In this case, there's no named parameter; the implementation would use `int` instead of `T`.)

It is also possible for parameters to be values instead of types (for example, to specify the size of a fixed-size array type). In this case, both the class implementation and the class user must enclose the parameter in square brackets. Note that type erasure also affects such parameters, and it may thus be necessary to use implicit parameters to gain access to these values.

Example:

```
// Here, the "len" parameter defaults to 1.
class FixedArray<T, [len: int = 1]> {
```

```
  ...
}
...
vector: FixedArray<:int, [3]>();
```

In the event that type erasure is undesirable, it's possible to implement parametric polymorphism by placing the implementation into a transparent class (Section 3.4.19), and then subclassing it as a non-transparent class for every type to be supported. Each such subclass may be given the same name, and be defined as specializations. This will result in compiling a separate implementation for each type (like C++, but more explicit). This is a tradeoff; while this may in some cases result in more efficient code, it may also significantly increase the size of the resulting executables.

Example:

```
// Place implementation into a transparent class
// to prevent type erasure.
transparent class ContainerImpl<T> {
  ...
}
// Create normal classes derived from it.
// This implements it for integers and booleans.
class Container<:int>: ContainerImpl<:int>;
class Container<:bool>: ContainerImpl<:bool>;
...
// Create a container to store an integer.
intstore: Container<:int>();
```

3.4.15 Metamethods

In MORTAL, metamethods are methods that are implicitly regenerated in subclasses (unless explicitly overridden), in a way that's useful for metaprogramming. Many class frameworks require a significant amount of «boilerplate» code to be written for every class to be used with the framework. With metamethods, such code only needs to be written once, in some common base class, and then the required code is automatically generated for every class that derives from this base class. In some cases, metamethods can also be used to facilitate subtyping-like code reuse without virtual methods (static

polymorphism), which may result in faster code, at the cost of increasing the size of the resulting executable.

Metamethods can be compared to metaclasses in dynamic languages (in which case MORTAL metamethods are slightly less powerful), or to the «curiously recurring template pattern» in C++ (in which case MORTAL metamethods are slightly more powerful).

Example:

```
class Vehicle {
  // Here, clsname() returns the name of the class
  // as a string. Vehicle.clsname() returns "Vehicle".
  public static meta clsname(): CString { @#owner.name }
  // turn_on() is also regenerated for every subclass,
  // so that it will use the subclass's ignition method.
  public meta turn_on() { ignition(); }
  protected ignition() { ... }
}

class Automobile: Vehicle {
  // The clsname() and turn_on() methods are implicitly
  // regenerated for Automobile. Thus, Automobile.clsname()
  // will return "Automobile", and Automobile.turn_on will
  // call Automobile.ignition (even though Vehicle.ignition
  // isn't virtual). (If turn_on had not been regenerated,
  // Vehicle.turn_on would be inherited as-is, i.e., it
  // would call Vehicle.ignition, not Automobile.ignition.)
  protected ignition() { ... }
}

class Boat: Vehicle {
  // Metamethods can be overridden.
  public turn_on() { ... }
}

car: Automobile();
// Note: since clsname and turn_on aren't virtual methods,
```

```
// the result depends on the (static) type of the
// "car" variable itself, not on the (dynamic) type of
// the object it is a reference to.
name = car.clsname(); // returns "Automobile"
car.turn_on(); // uses Automobile.ignition
```

3.4.16 Runtime Type Information

In object-oriented programming, a Runtime Type Information (RTTI) system allows any piece of code to identity the runtime (dynamic) type of some (polymorphic) object that's typically only known to be a subclass of some base class. Typically, a program will want to downcast the reference to the proper subclass type to gain access to its functionality, and RTTI makes it possible to identify which objects are instances of a particular subclass and which are not.

MORTAL is designed to be compatible with third-party class frameworks (such as Gtk+) that implement their own type information systems. For this reason, MORTAL does not have a standard RTTI system, but requires class frameworks to provide their own, if they need one.

To provide RTTI, two operators must be overloaded:

- The `typeid` operator. This operator should take no arguments, and the return type may be chosen by the class framework, but the result should be some unique value representing the object's type. (For Gtk+, for example, it could return a GType, an integer ID representing a registered type.)

- The `is` operator. This operator should take a single argument (the result of the `typeid` operator), and return a Boolean value, which says whether the class is of, or is a subclass of, the type identified by the argument.

Both of these operators should be provided as both static and non-static versions. The static version is used if the operators are used on the class itself, and the non-static version is used on class instances. Since RTTI is meant to work for polymorphic objets, the non-static version of these operators should typically either be, or call, a virtual method, unless the framework has other means of determining an object's dynamic type (such as storing the type ID in a field). Furthermore, in implementations where new typechecking code needs to be generated for every subclass, the operators may also need to be metamethods.

When a class framework implements RTTI, MORTAL will use it whenever appropriate. For example, the `as` operator (Section 3.6.6) can use it to check whether a typecast is safe, and multimethod dispatch (Section 3.5.11) can check argument types to determine which implementation of an overloaded method to use.

Example:

```
// This simple RTTI implementation uses the class name
// as a string to identify the type.
// It does not support multiple inheritance.
class BaseClass {
  // The helper methods
  protected meta static cls_name(): CString { @#owner.name }
  // the non-meta cls_is overrides the meta cls_is
  // for this class, but not for its subclasses
  protected static cls_is(n: CString): bool { cls_name() == n }
  protected meta static cls_is(n: CString): bool {
    if cls_name() == n return true;
    (@super).cls_is(n)
  }
  protected meta virtual obj_cls_name(): CString { cls_name() }
  protected meta virtual obj_is(n: CString): bool { cls_is(n) }
  // The operators
  public meta static typeid(): CString { cls_name() }
  public meta typeid(): CString { obj_cls_name() }
  public meta static is(n: CString): bool { cls_is(n) }
  public meta is(n: CString): bool { obj_is(n) }
}
```

3.4.17 Reference counting

Reference counting is a memory management scheme that allows an object to be strongly referenced («owned») by more than one user. Each object has a counter that tracks the number of owners. When the last owner of the object goes away, the object can be deallocated. Reference counting only applies to classes, not to structs.

MORTAL is designed to be compatible with third-party class frameworks (such as Gtk+) that implement their own memory management schemes. For this reason, MOR-

TAL does not have a standard reference counting system, but requires class frameworks to provide their own, if they need one.

To provide reference counting, the class must define the special methods `referencer` and `unreferencer`, neither of which may take an argument or return a value. The referencer should increase the instance's reference count, and the unreferencer should decrease it, and deallocate the instance when the count reaches zero. The increases and decreases should be atomic, and newly created instances should have a reference count of 1. The compiler will insert calls to these special methods automatically whenever it sees that an object gains a new strong reference (e.g., through assignment), or an old strong reference vanishes (e.g., through going out of scope). However, the compiler may use static analysis to minimize the number of such calls.[2]

Example:

```
// This simple refcount implementation uses the
// regular increment/decrement operators.
// Thus, it is not threadsafe.
class Counted {
  refcount: int = 1;
  referencer() { refcount++; }
  unreferencer() { if !--refcount delete this; }
}
...
obj1: Counted(); // initial refcount is 1
{
  obj2: Counted = obj1; // refcount is now 2
  obj1 = null; // refcount is now 1
} // obj2 goes out of scope
// refcount is now 0, so the object is deleted
```

Classes may optionally support weak references (references that are not counted, but still are tracked and automatically set to `null` if the referenced object is destroyed). This can be done by defining the special methods `addweakref` and `removeweakref`, which each should take a reference to a reference (i.e., the parameter should be of class type, and should be qualified with `ref`). The object should remember the weak

[2]Thread-safe refcounting implementations may stall the CPU pipeline every time they touch the counter, which may have a significant performance impact. This makes minimizing the number of such operations a worthwhile effort.

references registered using these methods, and set them to `null` when the object is destroyed. As with strong references, the compiler will insert calls to these special methods automatically; the programmer just needs to use the `weak` qualifier where appropriate.

Reference counting is optional in MORTAL. If a class does not provide reference counting, then MORTAL instead enforces the rule that any instance of that class may only have one owner at any time. Any other reference to the same instance must be explicitly or implicitly qualified as `weak` or `unowned`. This is less convenient for the programmer, but can be more efficient, and is nearly as safe.

Memory management in MORTAL is described in more detail in Section 2.9.

3.4.18 Abstract classes

Abstract classes are simply classes that cannot be instantiated. As this restriction does not apply to its subclasses, abstract classes are usually used as superclasses for a class hierarchy where the superclass itself is too generic to be meaningful to use on its own. Unlike interfaces, they can contain code that can be inherited by subclasses (see Section 3.4.12 for an example).

Another potential use for abstract classes may be to create a class with only static members, in which case the class does not need to be instantiated. However, the same functionality could also be achieved using regular functions in a namespace.

Abstract classes may contain abstract methods (methods without implementations). Since non-abstract classes cannot contain abstract methods, non-abstract classes that want to inherit from an abstract class must override its abstract methods.

3.4.19 Transparent classes

Transparent classes are among MORTAL's most important metaprogramming features. Transparent classes work a lot like macros. No code is generated for the class itself; rather, the relevant parts of the class definition is substituted into the code wherever the class is used. This can be extra useful when combined with MORTAL's other metaprogramming facilities, allowing the generated code to be altered in various ways.

Transparent classes can be used in many ways. Here are some of them:

- If a transparent struct that does not derive from another struct is used directly, then each of its fields will be instantiated independently in the generated code. The class's methods will be treated as transparent methods (Section 3.5.13), meaning

they will be inlined. This can, for example, be used to write wrapper classes for external libraries with zero runtime overhead. This use is only allowed for structs, not classes.

- If a transparent class/struct that does derive from another class/struct is used directly, then the generated code will use the base class/struct. Any fields defined in the transparent class are treated as aliases for the corresponding fields in the base class/struct. The class's methods will be treated as transparent methods, as above. This can, for example, be used to write wrapper classes for externally defined structures with zero runtime overhead. This use is allowed for both classes and structs.

- If a non-transparent class derives from a transparent class, then the transparent class's fields and methods will be generated as part of the derived class, as if they were metamethods (Section 3.4.15). This can, for example, be used to write mixins (pieces of behaviour that do not themselves form an independent class, but can be «mixed into» other classes). Multiple mixins can be inherited, even with class frameworks that do not otherwise support multiple inheritance.

- Deriving from a generic transparent class can also be used to get parametric polymorphism without type erasure (Section 3.4.14).

- With additional metaprogramming glue (to be implemented in future versions of MORTAL), it would also be possible to use transparent classes to add support for other OOP-based programming paradigms, such as aspect-oriented programming.

3.4.20 Inner classes

Inner classes are classes that are nested inside another class (the outer class), and whose instances retain a reference to an instance of the outer class. This allows inner class instances to refer to all fields and methods of the outer class as if they were part of the inner class itself. A typical use of inner classes is to implement iterators.

It is possible to nest classes without making them inner classes, by qualifying the nested class with `static`.

Example:

```
class Outer {
  data: int;
```

```
  public constructor(data: int);
  public class Inner {
    // direct access to "data"
    get(): int { data }
  }
  public static class NotInner {
    // no direct access to "data"
  }
}
...
obj ::= Outer(42);
// Inner classes must be constructed
// from an instance of the outer class.
inn ::= obj.Inner();
x = inn.get(); // returns 42
// Static classes does not have to be.
notinn ::= Outer.NotInner();
```

3.4.21 Interfaces

Interfaces are a kind of abstract class that cannot contain code or data, only abstract virtual methods and operators. All methods and operators defined within an interface definition are implicitly abstract and virtual. The primary use of interfaces is in subtype polymorphism (Section 3.4.13), where the interface represents the (abstract) type and any classes that inherit from it represent subtypes. Although an ordinary abstract class could do the same, the advantage of interfaces is that they avoid some of the issues concerning multiple inheritance (Section 3.4.12).

3.4.22 Member access control

As is common among OOP languages, class/struct members can be public, protected, or private. The default access level is `private`; any member that should be visible outside the class itself must be explicitly qualified with `public` or `protected`.

- If a member is private, then only members of the class and its nested/inner classes can access that member directly.

- If a member is protected, then members of subclasses can also access that member directly.
- If a member is public, then everyone that can access the class itself can also access that member.

Access control for the class/struct itself is covered in Section 3.4.24.

3.4.23 Invariants

As part of MORTAL's support for contract programming, classes may define invariants. Class invariants are, essentially, conditions that all methods are obliged to preserve. Note that methods only need to conform to an implication relation; the invariants only need to be true on exit if they were true on entry. The only exception is the allocator or constructor, who may not assume that the invariants are true on entry. (By mathematical induction, this would suffice to prove that the invariants will always be true after construction.) Furthermore, methods should not need explicit runtime checks for whether the invariants are true. (Indeed, MORTAL's optimizer might remove such checks, under the assumption that they are redundant.)

Defining class invariants helps prevent bugs, by allowing MORTAL to detect and report methods that may violate them. Unlike many other languages that support contract programming, MORTAL contracts are meant to be checked at compile time through static analysis, not at runtime. This may affect the way complex contracts should be written, but it also increases the chances of detecting and understanding rarely-occurring and hard-to-find bugs, and reduces the need for testing to do so.

Example:

```
class Storage {
  value: int;
  bias: int;
  invariant {
    value >= 0;
    bias >= -5 && bias <= 5;
    value + bias < 10;
  }
  public set_value_unsafe(v: int) {
    // this should fail to compile since the
```

```
    // argument is unrestricted and unchecked.
    value = v;
  }
  public set_value_safe(v: int) {
    // this should compile since the
    // argument is checked.
    if v >= 0 && v + bias < 10 {
      value = v;
    }
  }
}
```

For applying conditions to individual methods, see Section 3.5.14.

3.4.24 Class Qualifiers

Structs, classes, and interfaces may have the following qualifiers applied to them.

`abstract`	Abstract class. See Section 3.4.18.
`final`	The class cannot be subclassed. See Sections 3.4.12 and 3.4.13.
`public`	The class is accessible from other modules.
`private`	The class is not accessible from other modules. This is the default.
`sealed`	The class can only be subclassed from the same module, not from other modules.
`static`	If used on a nested class, the class is not an inner class. See Section 3.4.20. Currently no effect on top-level classes.
`transparent`	The class is transparent. See Section 3.4.19.
`__c_throw_arg`	If used on an exception base class, functions/methods that may throw exceptions of this type uses an extra parameter for returning such exceptions. This mechanism may be used for compatibility with external class frameworks like Gtk+.
`__rtinit`	If used on a vtable type (see below), the vtable object is to be allocated and initialized by program code, not by the compiler. This may be used to customize the vtable initialization, to be compatible with external class frameworks like Gtk+.

`__vtable`	If used on a nested class, MORTAL uses it as the enclosing class's vtable (table of virtual methods) type, instead of generating such a type itself. This may be used to customize the vtable layout, to be compatible with external class frameworks like Gtk+.

3.4.25 Field Qualifiers

Data fields within structs and classes may have the following qualifiers applied to them.

`public`	See Section 3.4.22.
`private`	See Section 3.4.22. This is the default.
`protected`	See Section 3.4.22.
`ref`	The field is an indirect reference. May be used with struct types to store a reference instead of a copy, or with class types to store a reference to a reference. Should only be used for compatibility with external libraries, as MORTAL's memory management system cannot track such references. Where possible, create new classes instead.
`static`	The field is class data, not instance data. See Section 3.4.2.
`weak`	The field is a weak reference. May be used if the referenced class can track weak references. If so, weak references become `null` when the referenced object ceases to exist. See Section 3.4.17.
`__outer`	If used in an inner class, MORTAL uses the field as the outer class reference, instead of generating such a field itself.
`__super`	If used in a subclass, MORTAL stores the superclass instance data in the field, instead of generating such a field itself. This may be used to customize the instance layout, to be compatible with external class frameworks like Gtk+.
`__vtable`	If used, MORTAL uses the field as the class's vtable (table of virtual methods) reference, instead of generating such a field itself. This may be used to customize the instance layout, to be compatible with external class frameworks like Gtk+.

3.4.26 Method Qualifiers

Methods and operators within structs and classes may have the following qualifiers applied to them.

`abstract`	Abstract method. May only be used in abstract classes. See Section 3.4.18.
`const`	The method does not modify the instance it is called on. For property getters, this is the default. See Section 3.3.3.
`extern`	The method implementation is in an external library. See Section 3.2.14.
`final`	The method cannot be overridden. See Section 3.4.13.
`meta`	Metamethod. See Section 3.4.15.
`multimethod`	Multimethod. See Section 3.5.11.
`override`	The method overrides a superclass method. See Section 3.4.13.
`public`	See Section 3.4.22.
`pure`	The method has no side effects.
`private`	See Section 3.4.22. This is the default.
`protected`	See Section 3.4.22.
`static`	The method is a class method. See Section 3.4.8.
`transparent`	The method is transparent. See Section 3.5.13.
`virtual`	The method supports subtype polymorphism. See Section 3.4.13.

3.5 Functions, methods, and operators

Functions, methods, and operators are self-contained pieces of code. All of the program's code must be inside a function, method, or operator. (Property getters and setters are considered a special type of method.) The program's entry point should be a public function (not a method) called `main` (Section 3.5.15).

The following sections mostly discuss functions. It should be understood that anything said about functions also applies to methods, operators, and property getters/setters, unless otherwise specified.

3.5.1 Parameters

Functions may take parameters. In this text, the word *parameter* is used to denote the formal argument, i.e., the named object used in the definition of the function. The word *argument* is used to denote the actual argument, i.e., the value that a caller passes to the function. A function call may be considered to bind the arguments to the parameters (or, more accurately, the initial values of the parameters).

All parameters must have an explicitly declared name and type. Since MORTAL allows callers to bind arguments to parameters by name (not only by position), programmers should use meaningful parameter names.

Parameters may have default expressions. If the caller does not bind an argument to a parameter that has a default expression, then the default expression is evaluated, and its result bound to the parameter instead. (If there's an unbound parameter without a default expression, a compilation error results.)

Example:

```
// The function "divide" has two parameters,
// named "dividend" and "divisor",
// both of type "int". The divisor has a
// default expression, which evaluates to one.
// Its return value is of type "int".
divide(dividend: int, divisor: int = 1): int
{
  return dividend / divisor;
}
...
// bind by position
x = divide(12, 3); // 12 / 3 = 4
// using default expression
x = divide(5); // 5 / 1 = 5
// bind by name
x = divide(divisor = 2,
           dividend = 6); // 6 / 2 = 3
```

3.5.2 Return type

Functions may return values. To do so, the return type must be explicitly declared. If no return type is declared, then the function cannot return a value. If a return type is declared, then the function must always return a value (unless it throws an exception). Failure to do so will result in a compilation error.

Example:

```
// The function "find_answer" has a
// return type of "int".
find_answer(): int
{
  return 42;
}

// The function "do_nothing" has no
// return type.
do_nothing()
{
}
```

3.5.3 Exception specifications

MORTAL uses checked exceptions. That is, functions that may throw exceptions must declare the kind of exceptions they may throw (or a superclass of those exceptions). Callers that do not plan to catch the exceptions must themselves declare that they can throw those exceptions, and so on. Failure to do so will result in a compilation error. (The primary reason is that MORTAL does not have a standard exception class, but the exception handling system needs to know what type of exception you wish to use in order to generate the appropriate code. See Section 3.8.1.)

Example:

```
import "glib";
// open_file reports errors using GLib's GError system.
open_file(name: CString): File throw GLib.Error
{
  f ::= File(name, "r");
```

```
    if !f {
        throw GLib.FileError(GLib.FileErrorCode.NOENT);
    }
    return f;
}
```

3.5.4 Abbreviated return syntax

If the last statement in a function is a `return` statement, then it can be abbreviated by omitting the `return` keyword and the final semicolon. This can be convenient for very short functions. (Omitting the semicolon is important. If there is a final semicolon, the expression will not get interpreted as an abbreviated `return` statement, and so a compilation error will result because the function does not return a value.)

Example:

```
// These two functions are equivalent.
find_answer_long(): int
{
    return 42;
}
find_answer_short(): int { 42 }
```

3.5.5 Implicit parameters

If a function parameter is qualified with `implicit`, then the caller cannot bind an argument to it. For the caller, the parameter is effectively invisible. An implicit parameter must have a default expression. This may be used to automatically pass information available to the caller to the function. This is typically used to work around type erasure when using parametric polymorphism.

Example:

```
// Due to type erasure, sizeof(T) is meaningless inside
// this function, but is meaningful to the caller.
// Thus, it can be evaluated as an implicit parameter.
get_size_of<T>(implicit sz: int = sizeof(T), count: int): int
{
    return sz * count;
```

```
}
...
x = get_size_of<:int>(5); // x = sizeof(int) * 5
```

3.5.6 «this» reference

Non-static methods have a special invisible parameter called `this`, which is a reference to the class instance the method was called on. The `this` reference is automatically used when referring to other non-static class members, so using it explicitly isn't normally needed, unless it's necessary to refer to the instance itself.

3.5.7 Variadic functions

Variadic functions are functions that can accept an unlimited number of arguments. For this purpose, it's possible to add special parameters to collect extra arguments (arguments that cannot be bound to any of the other parameters). These parameters should be of some container type that can store the extra arguments.

Arguments provided by position and arguments provided by name are stored in different containers. The container for by-position parameters should be a sequential data structure, such as a list or an array, and the container for the by-name parameters should be a key-value data structure, such as a hash table.

The parameter that should hold the by-position arguments must be prefixed by one asterisk, and the parameter that should hold the by-name arguments must be prefixed by two asterisks. If either is not provided, then extra arguments of that type is not accepted.

Example:

```
import "glib"; // use List and HashTable from GLib.

// Here, all extra arguments are of "int" type.
extra_args(arg: int,
           *pos_args: GLib.List<:int>,
           **key_args: GLib.HashTable<:CString, :int>)
{
   ...
}
```

```
...
// "arg" is set to 24, and "pos_args"
// gets the two entries 25 and 26.
// "key_args" gets an entry with
// key "extra" and value 42.
extra_args(24, 25, 26, extra=42);
```

3.5.8 C-style varargs

The C varargs mechanism can be used by importing the `stdarg` module. With this module, a function can support either by-position or by-name arguments (but not both). The container type to use is `va_args`. Functions must have at least one non-varargs parameter before the `va_args` parameter (but that parameter may be implicit).

If `va_args` is used for by-position arguments, then the argument values are passed unchanged. The argument list is not automatically terminated (though it's possible to use implicit arguments to do so; see below).

If `va_args` is used for by-name arguments, then each argument will be passed as a pair of values (the argument name in the form of a C string, followed by the argument value). The argument list is automatically terminated with a null pointer.

Using C-style varargs is inherently unsafe, since the compiler has no general way of checking whether the input is of the correct type or has the correct number of arguments. MORTAL only supports C-style varargs for compatibility reasons. However, there are two ways of mitigating the risks.

- If the function takes a list that's to be terminated with a special value, then that special value can be added as an implicit parameter after the `va_args` parameter. This way, callers do not need to remember to terminate the list themselves. (Note that parameters listed after the `va_args` parameter can only be used for this purpose. Attempting to access such parameters directly will not work.)

- If the function takes a C format string, then the format string argument can be qualified with `__c_printf`. In this case, MORTAL may be able to check that the extra arguments conform to the format string (provided the format string is a constant).

Example:

```
import "stdio", "stdarg";

// Prints error message to standard error
print_error(__c_printf msg: CString, *args: va_args)
{
  stderr.printf("ERROR: ");
  stderr.vprintf(msg, args.iterator());
  stderr.printf("\n");
}
```

3.5.9 Output parameters

If a parameter is qualified with `ref` or `out`, it becomes an output parameter. This means it can be used to return values to the caller (in addition to the return value). If, as is the case in many libraries, the return value is used to indicate success or failure, it may even be necessary to use output parameters to return results.

An output parameter is always passed by reference, never by value (even for struct types). Thus, if the caller has bound the parameter to a variable or field, then writing to the parameter will also change that variable or field for the caller. If the parameter type is already a reference type, then a reference to the reference is passed; this means that the function may change not only the object being referenced, but the reference itself (e.g., to make it refer to a different object).

The `ref` and `out` qualifiers may be followed by a question mark (making them `ref?` and `out?`, respectively). This makes the argument reference nullable, allowing the parameter to not be bound to anything. (This is different from having a default expression; if a default expression is used, the result of the default expression is bound to the parameter, and then, in the case of output parameters, passed by reference.) To check whether an argument reference is null, the `ref` operator can be used, e.g. `ref arg1 == null`. For struct types, the `===` operator will also work for this purpose.

A parameter qualified with `ref` is an input-output parameter. It can be used for passing information both into and out of the function. The function may access the original value of the argument before changing it, or decide not to change it.

A parameter qualified with `out` is an output-only parameter. It can only be used for passing information out of the function, not into it. MORTAL automatically initializes output-only parameters the same way variables without explicit initializers are initialized

(see Section 3.2.6). The function can not access the original value of the argument; any original value will be lost.[3]

Output-only parameters may not have any obvious functional benefits over input-output parameters, but there are still several reasons for supporting them:

- They can have documentation benefits, making it clearer how certain parameters should be used.
- They can be used for compatibility with external libraries that expect a pointer to an uninitialized structure to be passed in as an argument.
- When a function may be called remotely using RPC (remote procedure call) mechanisms, declaring a parameter output-only may help reduce the amount of data that needs to be sent to make the call.

3.5.10 Ad hoc polymorphism

Ad hoc polymorphism, or overloading, allows multiple function definitions to share the same name, and a definition to be automatically chosen depending on the arguments. Each definition must have different parameter lists. To minimize ambiguities when binding arguments by position, MORTAL requires that each parameter list must either differ in at least one parameter position without a default expression, ignoring names and most qualifiers. Alternatively, in case of methods, the method definitions may differ in staticness.

When an overloaded function is called, the compiler will choose one of the available definitions, depending on the arguments of the call. The overload resolution proceeds as follows:

- All definitions that do not fit the arguments are rejected. Non-static methods are rejected if there's no class instance.
- The remaining definitions are given either «Exact Match» rank or «Conversion» rank. If at least one argument requires a type conversion other than an upcast to a superclass, the rank is «Conversion», otherwise it is «Exact Match».
- If definitions with «Exact Match» rank exist, then all definitions with «Conversion» rank are rejected.

[3] In other words, the function assumes that arguments bound to output-only parameters are uninitialized, and that it must initialize them itself.

- Definitions that are strictly more general are rejected, leaving only the most specific definitions that match. A definition is strictly more general than another if all positional parameter types of the other method match (can be upcast to) the types of the more general method, but not vice versa.

- In case of methods, if both static and non-static definitions remain, the static definitions are rejected.

- If only one definition remains, it is chosen. Otherwise, the call is ambiguous or invalid, and a compilation error results.

Example:

```
describe(v: Vehicle) { ... }
describe(v: LandVehicle) { ... }
...
car: Automobile();
// Automobile is a subclass of LandVehicle,
// which is a subclass of Vehicle.
// Thus, while both has "exact match" rank,
// LandVehicle is more specific than Vehicle.
describe(car); // calls describe(v: LandVehicle)
```

3.5.11 Multimethods

By default, overload resolution (Section 3.5.10) is done at compile time, using the static types of the arguments. Multimethods extend this by also allowing overload resolution to be done at runtime, using the dynamic (actual) types of the arguments. Note that this is only possible for parameter types that implement RTTI (Section 3.4.16).

To enable runtime overload resolution for a function, function definitions should be qualified with `multimethod`. Runtime overload resolution proceeds as follows:

- Compile-time overload resolution yields the most specific definition that matches the static types of the arguments.

- If the function chosen at compile time is not qualified with `multimethod`, no runtime overload resolution is performed.

- Currently, only definitions that are strictly more specific than the function chosen at compile time, and accessible from the scope this function is defined in, are considered at runtime.
- Parameter types that implement RTTI, and that are different from the parameters of the function chosen at compile time, are checked one by one. Function definitions are rejected if an argument's dynamic type does not match (is not a subclass of) the corresponding parameter.
- The most specific remaining definition is chosen. If no other definition matches, the function chosen at compile time is used.

3.5.12 Parametric polymorphism

Functions support the same kind of parametric polymorphism that classes do (see Section 3.4.14), and with the same caveats (such as type erasure being the default). However, in the case of functions, type parameters do not always need to be explicitly given, but can sometimes be inferred from the arguments.

Example:

```
find_in_list<T>(list: List<:T>, obj: T): int
{
  idx: int;
  for x in list { if x == obj return idx; idx++; }
  return -1;
}
...
list: List<:Automobile>;
car: Automobile();
list.append(car);
// Here, T is Automobile, which can be inferred from
// List<:Automobile> or from the type of car.
x = find_in_list(list, car);
```

Type erasure can be avoided using transparent functions (Section 3.5.13). For small functions that can be inlined, simply making the polymorphic function transparent may be enough. For large functions that should not be inlined, the transparent function can be wrapped by a non-transparent function for every type to be supported. These

functions may then be overloaded (Section 3.5.10). This will result in compiling a separate implementation for each type (like C++, but more explicit). This is a tradeoff; while this may in some cases result in more efficient code, it may also significantly increase the size of the resulting executables.

Example:

```
// This implementation of the "max" function
// would be incompatible with type erasure,
// as the comparison operator >= needs the
// type of the operands to be known.
// Thus, it should be transparent.
transparent tmax<T>(a: T, b: T): T
{
  a >= b ? a : b
}

// Although the above implementation is small
// enough to be useful as-is, we may choose
// to wrap it into a bunch of overloaded
// non-transparent functions, one for each
// type we want to support.
max(a: int, b: int): int { tmax(a, b) }
max(a: double, b: double): double { tmax(a, b) }
```

3.5.13 Transparent functions

Transparent functions work a lot like macros. No code is generated for the function itself; rather, the function body is substituted into the code wherever the function is called. This can be extra useful when combined with MORTAL's other metaprogramming facilities, allowing the generated code to be altered in various ways.

Transparent functions can be used in several ways. Here are some of them:

- A transparent function can simply be a function that's always inlined. If a library consists entirely of transparent functions, then everything needed will be inlined into the executable, with no runtime dependency on the library. (However, not all functions can be made transparent. Virtual methods or functions to be used

as delegates, for example, cannot be transparent. Typically, neither can recursive functions.)

- Transparent functions can also be used to write wrapper functions for external libraries with zero runtime overhead.

- Generic transparent functions can also be used to get parametric polymorphism without type erasure (Section 3.5.12).

3.5.14 Contracts

As part of MORTAL's support for contract programming, functions may define preconditions and postconditions. Preconditions are conditions that must be true before calling the method (must be ensured by the caller), and postconditions are conditions that must be true when the method returns (must be ensured by the callee). Note that functions only need to conform to an implication relation; the postconditions only need to be true if the preconditions are true. Furthermore, functions should not need explicit runtime checks for whether the preconditions are true. (Indeed, MORTAL's optimizer might remove such checks, under the assumption that they are redundant. Where runtime checks are needed, preconditions should not be used.)

Preconditions and postconditions are considered part of a function's public interface. In case of methods, if a subclass overrides a virtual method with conditions, then the subclass implementation of the method is also required to obey the original implication relation (i.e., any new conditions may only relax the preconditions and strengthen the postconditions).

Defining preconditions and postconditions helps prevent bugs, by allowing MORTAL to detect and report method calls and method implementations that may violate them. Unlike many other languages that support contract programming, MORTAL contracts are meant to be checked at compile time through static analysis, not at runtime. This may affect the way complex conditions should be written, but it also increases the chances of detecting and understanding rarely-occurring and hard-to-find bugs, and reduces the need for testing to do so.

In some cases, preconditions can also help MORTAL's optimizer. For example, if two parameters have the same type, and it is a reference type, then it would be possible for the corresponding arguments to alias (refer to the same object), and this possibility would limit the freedom of the optimizer when both parameters are accessed frequently.

If a precondition says that the two arguments may not alias, then the optimizer can use this information to perform more aggressive optimizations.

Example:

```
class Storage {
  value: int;

  // The caller is required to ensure v is non-negative
  // (using runtime checks if necessary). The caller
  // is assured that the "value" field will be updated,
  // and that the method will return its argument.
  public set_value(v: int): int
    requires v >= 0
    ensures value == v
    ensures @result == v
  {
    value = v; // Satisfy first postcondition
    return v; // Satisfy second postcondition
  }
}
```

For applying conditions to entire classes, see Section 3.4.23.

3.5.15 The main function

If a public function (not a method) is called `main`, it is treated as the program's entry point. It maps directly to the C-language main function, and should have a compatible signature. The return type should be `int`. If it accepts arguments (it does not need to), the first parameter should be an `int`, and the second a C array of C strings. The first parameter represents the length of this C array. The first entry of the array is the file path of the program's executable (not necessarily an absolute path), the rest are the command-line arguments, if any.

Example:

```
// Basic signature
public main(argc: int, argv: __c_array<:CString, [argc]>): int
{
  // argc = number of arguments + 1
```

```
  // argv[0] = executable of current program
  // argv[1 ... argc-1] = arguments
  return 0;
}

// Class frameworks may simplify things by
// creating a more convenient structure.
public transparent struct MainArgs {
  argc: int;
  argv: __c_array<:CString, [argc]>;
  // Can define methods and operators here
  // to make MainArgs look like any other
  // container type, with indexers,
  // iterators, etc.
  // The following makes the "size" property
  // and the indexer skip argv[0], but allows
  // using the "exe" property to get at argv[0].
  public size: uint { get { argc - 1 } }
  public [idx: int]: CString { argv[idx + 1] }
  public exe: CString { get { argv[0] } }
}

// Then the main function can look like this.
public main(args: MainArgs): int
{
  // use args as a container type
  return 0;
}
```

3.5.16 Function Qualifiers

Functions outside structs and classes may have the following qualifiers applied to them.

`extern`	The function implementation is in an external library. See Section 3.2.14.
`multimethod`	Multimethod. See Section 3.5.11.

`public`	The function is accessible from other modules.
`pure`	The function has no side effects.
`private`	The function is not accessible from other modules. This is the default.
`transparent`	The method is transparent. See Section 3.5.13.

For methods and operators inside structs and classes, refer to Section 3.4.26.

3.5.17 Parameter Qualifiers

Function parameters may have the following qualifiers applied to them.

`implicit`	The parameter cannot be explicitly bound. See Section 3.5.5.
`out`	The parameter is output-only. See Section 3.5.9.
`out?`	The parameter is output-only and optional. See Section 3.5.9.
`ref`	The parameter is input-output. See Section 3.5.9.
`ref?`	The parameter is input-output and optional. See Section 3.5.9.
`__c_printf`	The parameter is a C format string. Any extra arguments should be checked against the format string bound to this parameter. See Section 3.5.8.
`*`	The parameter is used to collect extra by-position arguments. See Section 3.5.7.
`**`	The parameter is used to collect extra by-name arguments. See Section 3.5.7.

3.6 Expressions

Expressions allow memory accesses and computations to be expressed in a reasonably natural way. MORTAL's large number of overloadable operators and Unicode support allow a wide range of scientific computations to be expressed in a comprehensive fashion.

3.6.1 Numbers

In MORTAL code, number literals may be written in any of the following ways:

Type	Base	Example
Decimal integer	10	42
Hexadecimal integer	16	0x2a
Octal integer	8	0o52
Binary integer	2	0b101010
Decimal floating-point	10	4.2E+1
Hexadecimal floating-point	16	0x1.5P+5

For type inference purposes, integers currently default to type `int`, and floating-point numbers currently default to type `double`. However, the conversion to a binary number is delayed, in case of a cast to a different numeric type. Once the target type has been decided, the number literal is then converted directly to that type.

3.6.2 Strings

String literals must be enclosed in either single or double quotes (' or "). As in C, single quotes are used to denote the value of a single character, while double quotes are used to denote an addressable string of any length. (However, this use of single quotes may change in the future.)

String literals may contain escape sequences, which start with a backslash. MORTAL uses many of the same escape sequences as C. Escape sequences include:

Name	Sequence	Character
Alarm	`\a`	ASCII code 7 (BEL)
Backspace	`\b`	ASCII code 8 (BS)
Formfeed	`\f`	ASCII code 12 (FF)
Newline	`\n`	ASCII code 10 (LF)
Return	`\r`	ASCII code 13 (CR)
Tab	`\t`	ASCII code 9 (HT)
Vertical Tab	`\v`	ASCII code 11 (VT)
Single quote	`\'`	'
Double quote	`\"`	"
Backslash	`\\`	\
8-bit code	`\x`hh	Hex code hh
16-bit code	`\u`hhhh	Hex code hhhh
32-bit code	`\U`hhhhhhhh	Hex code hhhhhhhh

For type inference purposes, single-quoted literals currently default to type `int` (with UTF-32 encoding), and double-quoted literals currently default to type `CString` (with UTF-8 encoding). However, the conversion is delayed, in case of a cast to a different string type. Once the target type has been decided, the string literal is then converted directly to that type.

3.6.3 Operators

MORTAL allows both ASCII and Unicode operators to be used. The following ASCII operators are currently defined.

Prec.	Assoc.	Operator	Overloadable	Description
1	Left	`.`	See note 3	Member reference
		`?.`	See note 3	Null-safe member reference
		`??.`	See note 3	Auto-allocating member reference
		`(...)`	Yes	Call
		`[...]`	See note 4	Index
		`++`	Yes	Post-increment
		`--`	Yes	Post-decrement
2	Right	`@`	No	Metaexpression
		`*`	No	Expansion

Prec.	Assoc.	Operator	Overloadable	Description
3	Left	`<...>`	No	Parametric type argument
4	Right	`(:...)`	No	Static typecast
		`ref`	No	Make reference
		`sizeof`	No	(Static) Size of
		`typeid`	Yes	(Dynamic) Type ID of
5	Left	`as`	No	Dynamic typecast
6	Left	`^`	Yes	Power
7	Right	`+`	Yes	Copy
		`-`	Yes	Negation
		`~`	Yes	Bitwise NOT
		`++`	Yes	Pre-increment
		`--`	Yes	Pre-decrement
8	Left	`*`	Yes	Multiplication (inner product, matrix product)
		`**`	Yes	Multiplication (cross product)
		`<>`	Yes	Multiplication (outer product)
		`/`	Yes	Right division
		`\`	Yes	Left division
		`%`	Yes	Remainder
		`*.`	Yes	Elementwise multiplication
		`/.`	Yes	Elementwise right division
		`\.`	Yes	Elementwise left division
		`div`	Yes	Floored division
		`mod`	Yes	Modulo
9	Left	`+`	Yes	Addition
		`-`	Yes	Subtraction
10	Left	`<<`	Yes	Bit shift left
		`>>`	Yes	Bit shift right
11	Left	`&`	Yes	Bitwise AND
12	Left	`><`	Yes	Bitwise XOR
13	Left	`|`	Yes	Bitwise OR
14	Left	`??`	No	Null coalescing
15	None	`::`	No	Slice
		`..`	No	Range
16	None	`==`	Yes	Equal to

Prec.	Assoc.	Operator	Overloadable	Description
		`!=`	Yes	Not equal to
		`===`	No	Alias of
		`!==`	No	Not alias of
		`<`	Yes	Less than
		`>`	Yes	Greater than
		`<=`	Yes	Less than or equal to
		`>=`	Yes	Greater than or equal to
		`<=>`	Yes	Combined comparison
		`~`	Yes	Pattern match
		`in`	Yes	Contained in
		`not in`	Yes	Not contained in
		`is`	See note 5	(Dynamic) Derived type of
		`is not`	See note 5	(Dynamic) Not derived type of
17	Right	`!`, `not`	Yes	Logical NOT
18	Left	`&&`, `and`	Yes	Logical AND
19	Left	`||`, `or`	Yes	Logical OR
20	Right	`?...:...`	No	Conditional
21	Right	`->`	No	Anonymous function
22	Right	`:=`	Yes	Inline assignment
23	None	`throw`	No	Throw exception

Notes:

1. «Prec.» is precedence. Lower numbers mean higher precedence. Operators with higher precedence are evaluated first. For example, `2 + 3 * 4` equals `14`, because `*` is evaluated before `+` since `*` has a higher precedence.

2. «Assoc.» is associativity. Associativity determines order of evaluation among operators of equal precedence. For example, `3 * 4 % 8` equals `4`, because `*` is evaluated before `%` since these operators are left-associative. Also, `2 < 3 < 4` is not allowed because `<` has no associativity. (Such constructs may be allowed in the future.)

3. The member reference operator itself cannot be overloaded, but references to certain member names can be converted to method calls by defining the members as properties (Section 3.4.9).

4. References to indexes can be converted to method calls by defining indexers (Section 3.4.10). Indexers can be overloaded.

5. While the `is` operator can be overloaded, `is null` and `is not null` is exempt from any overloading, and can thus be used to check if a reference is null.

For Unicode operators, only Unicode characters classified as «Symbol, Math» will ordinarily be allowed as operators. Precedence for them may be defined in the future.

Many of the operators listed above work the same in MORTAL as in other languages. Furthermore, some of them are not yet implemented, or exist solely for the purpose of being overloaded by libraries (e.g., the matrix operators are only meaningful when overloaded by some library that implements matrices). Thus, only a few of the operators will be described here.

3.6.4 Members and dereferences

The period, `.`, is used to refer to named members of objects, classes, namespaces, etc. For example, if `universe` is a reference to an object with a field called `answer`, then you can say

```
universe.answer = 42;
```

Following a reference in order to access the actual object is usually called dereferencing. Referencing and dereferencing is automatic, but depends on the types used. Thus, typecasts may affect the way this is done.

However, if an object reference is `null` (doesn't refer to anything), and the member is not static, then doing the above unconditionally would probably cause problems (most likely crash). This kind of bug is common in other languages and may cause both stability and security issues. Thus, if you're dealing with nullable references, then you'll have to check for null (and MORTAL's static analysis will enforce this). To make the job easier, MORTAL provides a number of operators for handling nulls.

- To explicitly check for null, the recommended way is to use either `is null` or `=== null`, since these constructs cannot be overloaded.

- The null coalescing operator `??` allows you to substitute another object when the one you want is null. In the expression `a ?? b`, `a` is used if it's non-null, otherwise `b` is used.

- The null-safe member reference operator `?.` only follows the reference if it is not null. If it is null, then the expression (data access or method call) evaluates to the default value for the inferred expression type, usually null or zero. In the expression `car?.drive()`, the `drive` method is only called if `car` isn't a null reference.

- As a special case, if the null-safe member reference operator is used on an allocator, then it is invoked if the reference is null. In the expression `car?.new()`, a new car object is allocated and assigned to `car` if it was null before this.

- The auto-allocating member reference operator `??.` will, if the reference is null, allocate a new object of the required type and assign it to the reference. Then it follows the reference as normal. In the expression `car??.drive()`, a new car object is allocated (using the default allocator) and assigned to `car` if it was null, and then the `drive` method is called, regardless of whether `car` was previously null or not.

3.6.5 Calls

Calling a function, method, or delegate is done by providing an argument list (which may be empty) in parentheses. A future version of MORTAL might lift the need for parentheses in some cases, but for now, the parentheses are required, even for empty argument lists. Arguments can be bound by position, by name, or both.

Example:

```
drive(); // no arguments
drive(50, 10); // arguments bound by position
drive(speed=50, km=10); // bound by name
car.wash(); // method call, no arguments
```

Functions that aren't methods can still be called using method call syntax. In this case, the object stand-in is bound as the first by-position argument. (In the D language (Section 6.1.2.6), this is called Uniform Function Call Syntax. Elsewhere, it's typically called Unified Call Syntax, and is a proposed C++ improvement.)

Example:

```
50.drive(10); // same as drive(50, 10)
```

3.6.6 Typecasts

MORTAL has two typecast operators. The least safe, but most versatile, is the static typecast operator. For example, to downcast a `Vehicle` to an `Automobile`, you can say

```
car = (:Automobile)vehicle;
```

This is unsafe if `vehicle` is not actually a reference to an `Automobile` or a subclass of it, and should be done with care. However, if `Vehicle` implements RTTI (Section 3.4.16), then the dynamic typecast operator `as` can be used instead, which is safer. The above then becomes

```
car = vehicle as Automobile;
```

If `vehicle` is not actually a reference to an `Automobile`, then this will yield a null reference (which can be handled using the operators in Section 3.6.4).

Typecasting between value types and owned reference types may result in «boxing» and «unboxing», where memory is automatically allocated (and freed) by the compiler, and the value stored there, in order to create a persistent reference to it. This will usually happen as a result of type erasure (when attempting to store a value type in a container that internally operates on untyped pointers), but can also be done explicitly. Typecasting between value types and unowned reference types, on the other hand, results in referencing and dereferencing without allocating new memory. (Memory management in MORTAL is described in Section 2.9.)

By default, either operator only allows upcasts, downcasts, type conversions, constness/ownedness changes, boxing/unboxing, and other well-behaved typecasts (equivalent to C++'s `static_cast`, `dynamic_cast`, and `const_cast`). If a typecast more like C++'s `reinterpret_cast` is called for, it's usually necessary to do a static cast to a C untyped pointer and back. For example

```
floatvar = (:float)(:__c_ptr.void)intvar;
```

This could even be wrapped in a transparent polymorphic function. However, using such techniques is discouraged, as static analysis may not be able to verify the correctness of such operations.

3.6.7 Metaexpressions

Metaexpressions are evaluated at compile time, and can refer to information available to the compiler that wouldn't normally be available at runtime. A simple example is the

source code filename (similar to the `__FILE__` macro in C/C++). Metaexpressions are useful when used in metamethods or transparent methods, or in implicit parameters to ordinary methods.

The form `@expression` results in either a numeric value, an object reference, or an identifier. If the expression creates a string, then it is converted to an identifier (Section 3.2.2), and treated as such.

The form `@#expression` results in a string. If the expression refers to an object, then the name of that object is converted to a string.

To manipulate an object's name as a string inside the metaexpression itself, refer to the object's `name` property, e.g., `owner.name`. This is useful for constructing new identifiers, e.g., `@(owner.name + "Child")`.

Special information that can be used in metaexpressions include:

Name	Type	Description
`function`	Function object	Enclosing function or method
`owner`	Class object	Enclosing class or struct
`super`	Class object	Base class or struct
`module`	String	Module name
`file`	String	Source file name
`line`	Integer	Source line number

The metaexpression facility is likely to be expanded in future versions of MORTAL, including making it possible to call pure functions from metaexpressions, in order to evaluate them at compile time.

3.6.8 Expansions

The expansion operator `*` is intended for use in transparent variadic functions (Section 3.5.7). When extra arguments are collected into a special parameter, this operator can expand that parameter back into an argument list for the purpose of calling another function. This operator is not available in non-transparent functions.

3.6.9 Slices and ranges

Ranges may be specified using the slice operator (`::`) or range operator (`..`). The difference between them is that a range includes the endpoint, while a slice does not. For example, `1::3` means the numbers `1` and `2`, while `1..3` means the numbers `1`, `2`, and `3`.

Currently, slices and ranges are not independent objects, and the slice and range operators can thus only be used in very specific circumstances. They can be used in for loops, or as indices for containers (provided the container defines an indexer which accepts slices and ranges).

Example of a for loop:

```
for x in 1..5 {
  // this loop will run 5 times
}
```

Example of an index:

```
import "glib";
str: GLib.String = "abcdef";
substr ::= str[2::4]; // yields "cd"
```

In the future, it will also be possible to specify non-default step sizes. For example, `1::2::5` will mean the numbers `1` and `3`, and `1::2..5` will mean the numbers `1`, `3`, and `5`.

3.6.10 Anonymous functions

Anonymous functions are functions that are defined inline inside a function, and that can be assigned to delegates (Section 3.2.13). Argument and return types do not need to be specified explicitly, as they can be inferred from the delegates they're assigned to.

It is planned that anonymous functions should eventually be able to access the local variables of the function they're defined in, i.e., they will have the functionality of lexical closures. Note that only non-static delegates will be able to store closures.

Example:

```
delegate MathOp(x: int, y: int): int;
...
op: MathOp;
op = (x,y) -> { x + y };
```

3.7 Statements

In the imperative paradigm, every function is made up of statements. Most statements involve expressions. (In fact, an expression is by itself a valid MORTAL statement,

see Section 3.7.4.) In general, statements must be terminated with a semicolon, except for those which terminate with a closing curly brace instead, or those which use the abbreviated return syntax (Section 3.5.4).

3.7.1 Variables

A variable definition (Section 3.2.6) is a valid statement. Variables may be declared anywhere within a function body, and type inference may be used. The scope and lifetime of a local variable is the remainder of the scope it is defined in.[4]

If a local variable is qualified with `static`, then the variable's lifetime is extended to the module it's defined in. Like a global variable, a static variable only has one instance, and its value is preserved across function calls.

3.7.2 Assignments

Assigning to a variable copies the value or reference to it. Any expression can be assigned to the variable; the value of the expression is computed, and then copied to the variable.

```
num = 10;
```

Note that an assignment is a statement, not an operator. This use of the equals sign is not allowed inside expressions. [5] (Inside expressions, the inline assignment operator `:=` can be used instead.)

Since the value of the expression is always computed before the assignment is done, it is possible to use the old value of a variable to compute a new value for the same variable.

```
num = num + 1;
```

This would increment the value of `num` by 1. This type of assignment also has a shorthand form:

```
num += 1;
```

Conditions can be added to assignment statements.

```
num = 10 if num < 10;
```

[4] However, the compiler is free to destroy the variable after its last use, instead of waiting until the scope ends.

[5] This stops inexperienced programmers from doing assignments when they mean to use the equality operator `==`, and also allows the symbol to be used for other things, such as binding arguments by name in function calls.

This will only assign 10 to `num` if it was smaller than 10 before the assignment.

3.7.3 Delete statements

`delete` statements can be used to destroy objects of class type. It takes a reference and, if it's non-null, calls the referenced object's deallocator.

```
delete obj; // Equivalent to obj?.delete();
```

3.7.4 Expressions

A bare expression can be a statement. This is mainly useful if the expression has side effects, such as if it's a call.

```
printf("Hello World\n");
```

Just as with assignments, conditions can be added to expression statements.

3.7.5 Compound statements

Any place where a single statement is accepted, it's possible to use multiple statements if they are enclosed in curly brackets. This also creates a new scope, which can hold its own local variables.

```
{
  num: int = 10;
  printf("Hello\n");
}
```

3.7.6 If statements

The `if` statement allows statements of any kind to be executed conditionally. There are several ways to write `if` statements. The standard way is to use compound statements.

```
if num < 10 {
  num = 10;
}
```

An `if` statement may have an `else` clause, which is executed if the condition was false.

```
if check_status() {
   printf("All OK\n");
} else {
   printf("Failed\n");
}
```

`if` statements can be chained (though, in many cases, it may be better to use a `switch` statement (Section 3.7.7) instead).

```
if num == 5 {
   printf("Found a 5\n");
} else if num == 6 {
   printf("Found a 6\n");
} else {
   printf("Found neither\n");
}
```

Compound statements are not mandatory. Most statements that start with a keyword are recognized.

```
if num < 10 return num;
```

Otherwise, you can use the `then` keyword.

```
if num < 10 then num = 10;
```

In such cases, however, you can usually add conditions to statements instead.

```
num = 10 if num < 10;
```

3.7.7 Switch statements

`switch` statements allow statements to be executed depending on the value of a variable or expression. While an `if` statement can only check one value at a time, a `switch` statement can check many. Thus, though `switch` statements could be emulated using `if` chains, `switch` statements are usually both more efficient and more readable.

```
switch num {
case 5:
   printf("Found a 5\n");
case 6:
   printf("Found a 6\n");
```

```
case 10, 20:
   printf("Found a 10 or 20\n");
default:
   printf("Found neither\n");
}
```

A `switch` statement can be exited with a `break` statement. However, there's no implicit fallthrough (i.e., execution will exit the `switch` rather than continuing into another `case`), so this is usually not needed. For situations where it's necessary to continue execution in another `case`, the `goto` statement (Section 3.7.13) can be used.

Each `case` is its own scope. There's no need for extra curly brackets if you need to define local variables inside a particular `case`.

3.7.8 While loops

The `while` loop allows statements to be executed repeatedly as long as a condition is true. The condition is tested before every loop iteration (but not during them). The standard way to write `while` loops is to use compound statements.

```
while num < 10 {
   num = num + 1;
}
```

As with the `if` statement, compound statements are not mandatory. Some statements that start with a keyword are recognized (but not as many as for the `if` statement). Otherwise, you can use the `do` keyword.

```
while num < 10 do num = num + 1;
```

3.7.9 Do-while loops

`do while` loops are like `while` loops, except that the condition is first tested after the first loop iteration, instead of before. This means that the statements are always executed at least once.

```
do {
   num = num + 1;
} while num < 10;
```

Or, without compound statements:

```
do num = num + 1 while num < 10;
```

3.7.10 For loops

`for` loops come in two variants. The simple one iterates over all elements in a container or range.

```
for x in the_list {
  do_something(x);
}
```

In this case, `the_list` is a variable referring to a container object holding a list of objects. The statements are executed once for each such object, with the local variable `x` referring to that object. The scope of the variable is only the `for` loop itself.

The other variant is the C-style `for` loop.

```
for (num=0; num<10; num+=1) {
  do_something(num);
}
```

Here, the parentheses are required. They contain three statement parts separated by semicolons. Each part is optional, but the semicolons are mandatory. The first part is a comma-separated initializer list, which can contain definitions of new variables and/or assignments to already existing variables. If a new variable is defined here, its scope is only the `for` loop itself. The second part is a condition. As in a `while` loop, it is tested before the first iteration. The third part is a comma-separated list of assignments to perform or expressions to evaluate after every iteration, just before testing the condition again.

3.7.11 Control flow statements

The `break` statement breaks out of a loop or `switch` statement. Example:

```
do {
  if need_to_abort() {
    break;
  }
  num = num + 1;
} while num < 10;
```

The `continue` statement breaks out of the current iteration of a loop, but not the loop itself. The next iteration is prepared as normal (including any `for`-loop assignments, and testing of any loop condition). Example:

```
do {
  if need_to_try_again() {
    continue;
  }
  num = num + 1;
} while num < 10;
```

The `return` statement ends execution of a function and returns the provided value, if any. Example:

```
return num;
```

As with assignments, conditions can be added to control flow statements.

3.7.12 Try statements

`try` statements are used to recover from exceptions (Section 3.8). Exceptions are usually thrown when some error occurs. When exceptions are thrown, normal execution aborts. If the exception was thrown in scope of a `try` statement which has an exception handler that matches the exception, then execution jumps to that handler. Example:

```
try {
  risky_operation();
  finish_operation();
} catch err: IOError {
  handle_io_error(err);
}
```

The above will catch any `IOError` exceptions thrown by `risky_operation` or `finish_operation`, and recover from them by calling `handle_io_error`. (If the exception was thrown by `risky_operation`, then `finish_operation` is not executed.) Other types of exceptions are not caught.

```
try {
  risky_operation();
  finish_operation();
} catch {
```

```
    handle_any_error();
}
```

The above will catch any exception, no matter what type. However, there's no way to determine the type of exception, so this should only be used as a last-resort handler. For example:

```
try {
    risky_operation();
    finish_operation();
} catch err: IOError {
    handle_io_error(err);
} catch {
    handle_any_error();
}
```

The above handles `IOError` by calling `handle_io_error`, and also handles all other types of exceptions by calling `handle_any_error`.

If the `try` statement has a `finally` clause, then that clause is executed regardless of whether an exception was thrown. Example:

```
try {
    risky_operation();
} finally {
    finish_operation();
}
```

The above forces `finish_operation` to be called even if an exception is thrown by `risky_operation`. However, it does not actually handle the exception. If this `try` statement is executing within another `try` statement, then the exception is passed to the outer `try` statement after calling `finish_operation`.

It's possible to have both exception handlers and a `finally` clause. In this case, the `finally` clause executes last. Example:

```
try {
    risky_operation();
} finally {
    finish_operation();
} catch err: IOError {
    handle_io_error(err);
```

```
} catch {
  handle_any_error();
}
```

If an exception is thrown from `risky_operation`, then `finish_operation` is called anyway, after the appropriate exception handler. (However, exceptions thrown by `finish_operation` will not be handled by this `try` statement.)

3.7.13 Goto statements

`goto` statements can be used to cause execution to jump directly to some given point inside a function. They are subject to several restrictions. For example, you are, in general, not allowed to jump directly into a scope you're not already in. Most of the time, `goto` statements should be avoided in favour of other language features (like control flow statements, or exceptions and `try` statements), but sometimes they are useful.

`goto` statements come in two variants. The regular one requires that you explicitly declare jump targets using `label`:

```
label my_label;
num = num + 1;
goto my_label;
```

The other variant can be used inside `switch` statements:

```
switch num {
case 5:
  printf("Handle 5\n");
  goto case 6;
case 6:
  printf("Handle 5 or 6\n");
}
```

3.8 Exceptions

Exceptions are used for runtime error handling. Exceptions have proven useful because the point at which an error is detected may not be the point at which the error should be handled. Thus, exceptions offer a structured and safe way of propagating an error up the call chain to a point where it can be handled.

Exceptions are not intended for handling programming errors such as invalid arguments. To detect such errors, contracts (Section 3.5.14) should be used instead.

3.8.1 Checked exceptions

Checked exceptions are exceptions that, if left unhandled, results in a compilation error. Exceptions may be handled either with `try`/`catch` statements (Section 3.7.12), or by allowing the function to propagate (rethrow) the exceptions, by listing them in the function's exception specification (Section 3.5.3). In MORTAL, all exceptions are checked by default, but unlike other languages that enforce them, their primary purpose in MORTAL is not to discipline the programmer, but to make MORTAL's exception handling compatible with external libraries and class frameworks (which may not be written in a language that supports exceptions).

In languages that use checked exceptions to discipline the programmer, programmers are typically expected to be specific about the exceptions they may want to throw. However, as this is tedious and also not very future-proof, it is often simpler and safer for programmers to simply specify the base exception class in every exception specification. Thus, in MORTAL, the latter is considered a perfectly acceptable way of specifying that something may throw an exception.

There are three reasons MORTAL requires specifying which exception class to use (even if only the base exception class):

- It declares whether something can throw an exception at all. For the programmer, there's no need to put `try` blocks around code that is known to not throw exceptions. For the compiler, not having to handle exceptions usually results in smaller and faster code.

- It avoids compatibility issues with external libraries that may have limited or no awareness of exceptions. For example, an external library function might take a delegate that cannot throw exceptions. Then attempting to use a reference to a function that can throw exceptions as the delegate would result in an error.

- MORTAL does not define a standard base exception class, but rather tries to support using multiple exception handling mechanisms (which may be defined by class frameworks). Each base exception class belong to a particular exception handling mechanism, and specifying which base exception class to use tells the

compiler which mechanism to use (and thus, what kind of exception handling code to generate).

It also remains an option to use checked exceptions the traditional way (being specific about the exceptions a piece of code may throw).

In the future, MORTAL may allow defining a default exception class, so that functions, methods, and operators that do not have an explicit exception specification will be considered able to throw exceptions of that type. By setting an appropriate base exception class as the default, programmers would be able to use MORTAL as if it had unchecked exceptions.

3.8.2 Argument-based exceptions

To be compatible with C libraries like GLib, MORTAL supports argument-based exceptions. The base class of such exceptions (GError, for example) should be qualified with `__c_throw_arg` (and should also support RTTI). If a function is declared to throw exceptions derived from this base class, then an extra implicit parameter is added to the function. This parameter is an output parameter (Section 3.5.9) that returns a reference to the exception class. If no exception is thrown, then the returned reference is null, otherwise it signifies that an exception was thrown (in which case the function's regular return value may not be meaningful).

In the future, MORTAL may also support a form of exception handling based on the standard C library's `errno` variable.

4 Implementation

This chapter describes what has been implemented so far.

4.1 Language features

Procedural and object-oriented programming works. The type system works, though some features like const types, non-nullable types, and implicit type conversions are not fully implemented.

All statements are implemented, except `goto` statements. Exceptions are partially implemented.

Roughly half of the operators are implemented. The operators that are implemented are: all kinds of member references, call, index, pre/post-increment/decrement, metaexpression, parametric arguments, all typecasts, sizeof, typeid, negation, ordinary multiplication, right division, remainder, addition and subtraction, bit shifts, bitwise and/or/not, null coalescing, slice/range (partially), all comparison operators (except pattern match), logical and/or/not, conditional, and inline assignment.

Function calls are mostly implemented, including binding arguments by position or name, implicit arguments, and the Unified Call Syntax, but variadic functions are only partially implemented (they only work for C-style varargs). The expansion operator is not implemented. Transparent functions are partially implemented (nontrivial control flow is not supported yet).

Object-oriented programming with structs and classes is mostly implemented, including operator overloading, properties/indexers, inheritance, virtual methods, metamethods, multimethods, RTTI, and all kinds of polymorphism. Not implemented are type conversions, slice/range support, some inheritance features (e.g., conflict resolution), some transparent class/struct features (e.g., mixins), some construction features (e.g., implicit copying), member access control, and special handling of interfaces.

4.2 Library features

MORTAL's standard library current implements transparent wrappers for parts of the C library (standard I/O and number formatting), and many GLib data types (including dynamic arrays, linked lists, strings, queues, hash tables, and I/O channels). These wrappers provide an object-oriented interface, and are easy to use from MORTAL.

4.3 Compiler features

The MORTAL compiler is self-hosting (apart from the lexer and parser), and able to generate C code from MORTAL code. Memory management is partially implemented.

Transformation to SSA (Static Single Assignment) form is not yet implemented. Most forms of static analysis (including contract programming) is not implemented.

4.4 Source code

The MORTAL compiler is open source and released under the MIT license. It is hosted on SourceForge. The main project URL is https://sourceforge.net/projects/mortal/.

The current build system is based on autoconf/automake. Thus, after retrieving or unpacking the sources, it is probably necessary to run `autoreconf`, then `./configure`, and finally `make`, to build the MORTAL compiler. This should result in a `mtlc` executable.

The main directory contains the sources of the compiler itself, both as source `.mtl` files, and as generated `.c` and `.h` files (which are needed to compile the compiler for the first time).

The `lib` subdirectory contains MORTAL's equivalent of a standard library. When compiling MORTAL programs, the path to the `lib` directory should always be given to `mtlc` using the `-I` option, e.g., `mtlc -I/path/to/lib program.mtl`.

The `tests` subdirectory contains the unit tests. They use GLib's unit test framework, and can be run with `make check`.

4.5 Syntax highlighting

For syntax highlighting, experimental language description files are available for GtkSourceView-based editors/IDEs (such as gedit and Anjuta), and for the Kate editor. In the files accompanying this thesis, they are called `mortal.lang` and `mortal.xml`, respectively.

5 Evaluation

The implementation of the MORTAL language is far from complete, and thus not yet as expressive or as safe as it needs to be. However, the current implementation can still be evaluated with respect to correctness, usability, and performance.

5.1 Correctness

The MORTAL compiler's source code includes a suite of unit tests designed to test various features for correctness (see Section 4.4). The tests use GLib's unit test framework. Running `make check` from the compiler source directory will run the test suite. There are currently 25 test groups, organized in 6 source files. All features currently implemented pass the current unit tests.

However, the currently biggest test for correctness is whether the MORTAL compiler is able to compile itself correctly. This is checked by running `make verify` from the compiler source directory. This effectively compiles the compiler thrice (first a reference compiler is built from the C sources in version control, then that reference compiler is used to build a test compiler, and then the test compiler is told to build itself), and at the end, the resulting compiler must pass the unit tests. However, if there's a problem in the compiler, the test compiler will usually fail to compile itself before it gets to the unit tests.

The compiler is able to compile itself, so the compiler appears to work correctly.

5.2 Usability

No significant program has yet been written in MORTAL, but we may judge the language's usability by whether it can be used to build a compiler. Since the MORTAL compiler is written in MORTAL, this is evidently possible.

MORTAL's RTTI system and its multimethods, for instance, have proven quite useful in the construction of the compiler. Multimethods have allowed AST (Abstract Syntax

Version	Measurements [s]	Fastest [s]	Average [s]
Optimized C	3.348 3.192 3.192 3.195 3.193 3.193	3.192	3.219
Simple MORTAL	4.534 4.464 4.362 4.346 4.402 4.455	4.346	4.427
Optimized MORTAL	3.347 3.188 3.189 3.192 3.189 3.187	3.187	3.215

Table 5.1: Performance of «fasta» benchmark, with $n = 10000000$

Tree) manipulation to be done with an ease comparable to that of functional languages.

5.3 Performance

MORTAL aims to provide performance comparable to C/C++. To test this, a benchmark was selected from the «Computer Languages Benchmark Game»[1], and implemented in MORTAL. The benchmark used is «fasta», which generates three genome sequences in FASTA format (one repeating sequence, and two random sequences that need to follow particular probability distributions). The MORTAL implementation was compared to «fasta.gcc», a reasonably well-optimized C implementation. (There is a faster C implementation, but it uses Unix file descriptors and its own buffering, instead of standard I/O. This may also be possible to do in MORTAL, but for the purpose of this evaluation it was decided to use standard I/O.)

Two MORTAL implementations were written: one without I/O optimizations, and one with a level of I/O optimization comparable to the C version. (Note that the optimizations done in the latter version should ordinarily not be done in a MORTAL program, as they use MORTAL's C compatibility features to do direct pointer manipulation, which may be difficult to verify with static analysis. However, pointer manipulation was mainly necessary because MORTAL does not yet fully implement slices, which could have provided a safer and easier alternative.)

The measurements were done on a laptop with an AMD A10-5757M quad-core CPU running at 2.5GHz, and 8 GB of RAM, running Debian jessie (gcc version 4.9.2).[2] The results (Table 5.1) show that a MORTAL program can achieve the performance of C. The difference between MORTAL and C is mostly because of minor implementation differences between the C and MORTAL programs, in part due to MORTAL's standard library using different C library functions, so that the loops had to be structured slightly

[1] http://benchmarksgame.alioth.debian.org

[2] To rerun the benchmark using the files accompanying this thesis, enter the `benchmarksgame/bencher` directory, edit `makefiles/my.linux.ini` to adjust the MORTAL compiler path, then run `python bin/bencher.py fasta`

differently. Since MORTAL's compiler currently generates C code itself, it would otherwise not be possible for MORTAL to outperform well-written C code. It might become possible once the LLVM backend is implemented.

6 Related work

6.1 Programming languages

Thousands of programming languages already exist, several of which try to solve some of the same problems that MORTAL does. Describing all of them individually would be a huge undertaking, but this chapter provides a survey of the most important ones, the ways in which MORTAL has been inspired by them, and the ways in which MORTAL is different.

6.1.1 Procedural languages

Procedural languages are imperative, structured languages that allow code to be grouped into procedures and functions. The grandfather of all such languages was ALGOL (not described here since it's no longer popular).

6.1.1.1 Fortran

[12] The Fortran language is more than 60 years old. Although the first version would not be considered structured, the language has been revised several times over the years, introducing many of the features of modern languages. It is primarily designed for scientific computation, and is heavily used in many research institutions, in part because it has a higher performance and more flexible math operations than the otherwise more popular C language.

6.1.1.2 C

[13] One of the most influential programming languages ever, C is more than 40 years old and still in widespread use worldwide, especially in systems programming, and has been an inspiration for numerous other languages. It is a fairly low-level language, limited to the procedural paradigm, and is considered by some as a «portable assembler». This is

also how MORTAL uses it; MORTAL code can be compiled to C code, which can then be compiled by a C compiler to native machine code.

MORTAL's syntax has much in common with C syntax, and contains many constructs intended for C compatibility, but MORTAL aims to be a more high-level language than C is. See also Section 6.1.2.3.

6.1.1.3 Pascal

[14] Pascal predates C by about two years. It has a more verbose and high-level syntax than C, but was originally designed for teaching, and did not have support for modules and other features useful to large projects. Many Pascal dialects with such features have been created over the years, including Borland's Turbo Pascal (which later evolved into Delphi, Section 6.1.2.10) and Free Pascal. However, the various dialects are generally not compatible, and this lack of portability has limited Pascal's success.

MORTAL has borrowed some elements from Pascal syntax. MORTAL's inline assignment operator looks like Pascal's assignment operator, MORTAL's `div` and `mod` operators are also similar to Pascal's, and MORTAL's parameter and variable declarations have a similar style to Pascal's (name, colon, type).

6.1.2 Object-oriented languages

Object-oriented languages improve on procedural languages by allowing code and data to be grouped together into objects. Pure object-oriented languages require doing this. The grandfather of all such languages was Simula (developed in the 1960s at the Norwegian Computer Center in Oslo, Norway).

6.1.2.1 Smalltalk

[15] Smalltalk was developed in the 1970s. The last version, Smalltalk-80, is a dynamic language where everything is an object, and pretty much everything (even control flow) is done by passing messages (calling methods). It introduced the concept of metaclasses, which allowed classes themselves to be objects that could be manipulated. Smalltalk has been a major influence on later OOP languages.

6.1.2.2 Eiffel

[16] Eiffel was developed in the 1980s. It is an object-oriented language which introduced a number of innovations regarding code safety, including modern contract-based

programming («Design by Contract» [4]) and null pointer safety («void safety»). Eiffel's strong safety features has been a major influence on later languages.

6.1.2.3 C++

[17] C++ is an evolution of the C language, introducing an object-oriented paradigm, powerful metaprogramming (in the form of very versatile templates), operator overloading, a powerful standard library, and many other improvements. However, despite C++'s expressive power, its C roots continue to limit its intuitiveness, usability, safety, and programmer productivity to this day, making many programmers shy away from it. Those that do use it exhibit a great deal of self-discipline, often in the form of numerous coding style guidelines.

MORTAL's syntax has much in common with C++ syntax, but also several important differences. Some of the ways MORTAL differs from C++ are:

- MORTAL has a more regular (less ambiguous) syntax.
- MORTAL does not have forward declarations, header files, or a preprocessor.
- MORTAL does not need the programmer to know about pointers.
- MORTAL can pass function arguments by name instead of by position.
- MORTAL has broader support for operator overloading, with many operators useful to scientists and engineers, and even allows user-defined RTTI (Run-Time Type Information).
- MORTAL's resource management is similar in capability to C++ smart pointers, but MORTAL makes such mechanisms easier and more efficient by making them part of the language itself, while keeping them user-definable.
- MORTAL generics allow type erasure by default, resulting in less code bloat than C++ templates.
- MORTAL supports contract-based programming.
- MORTAL aims to support declarative programming.

6.1.2.4 Objective-C

[18] Objective-C is an independent evolution of the C language, introducing a style of object-oriented programming much closer to Smalltalk (Section 6.1.2.1) than to C++. It provides dynamic typing and reflection, but no templates or operator overloading. Objective-C is the preferred programming language for Apple operating systems (OS X, iOS). Newer versions of Objective-C supports ARC (Automatic Reference Counting) for memory management. MORTAL's memory management is conceptually similar to ARC.

6.1.2.5 C#

[19] C# is a pure object-oriented evolution of the C language. Its OOP feature set is comparable to C++'s, with a few extensions, but is a very different language, neither compatible with C nor C++. C# compiles to Microsoft .NET bytecode, instead of directly to native machine code. The use of the .NET framework gives the language a number of advantages over C++, such as automatic garbage collection, ability to support things like aspect-oriented programming and code contracts, and a generally safer programming environment. A disadvantage is that the .NET framework is fairly heavyweight, and only runs on Windows platforms. There's a free .NET clone, Mono, which runs on other platforms and supports most of .NET, but not all.

MORTAL's syntax has much in common with C#. For example, MORTAL supports C#-like properties and indexers, and makes the same struct/class distinction that C# does. A few keywords, like `sealed`, has C#-like semantics.

Some of the ways MORTAL differs from C# are:

- MORTAL is portable, compiles to native machine code, and does not require a runtime.
- MORTAL aims to be compatible with ordinary C, C++, and Fortran code. (This includes a lot of scientific software.)
- MORTAL does not force programmers into the OOP paradigm.
- MORTAL supports multiple inheritance.
- MORTAL aims to support declarative programming.

6.1.2.6 D

[20, 21] D is a complete redesign of the C++ language. It fixes many of the problems of C++, has a less ambiguous syntax, more powerful metaprogramming, and automatic memory management, using a tracing garbage collector (though the programmer can also manage individual memory blocks explicitly). It also supports contracts, functional programming, and concurrent programming, and it compiles to native machine code.

MORTAL shares many of D's goals, and supports things like D's «Uniform Function Call Syntax», but in general, has a quite different syntax from D.

Some of the ways MORTAL differs from D are:

- MORTAL supports automatic memory management using reference counting, and does not require a tracing garbage collector. (Tracing garbage collectors has certain disadvantages, such as potentially much larger working sets, and momentary freezes during garbage collection. Although you can disable D's garbage collector, you would then have to manage memory manually.)
- MORTAL does not require a particular standard library.
- MORTAL's contracts are checked at compile-time, rather than runtime, thus potentially catching more bugs.
- MORTAL can pass function arguments by name instead of by position.
- MORTAL aims to be compatible with C++ code.
- MORTAL aims to support declarative programming.

6.1.2.7 Java

[22] Java is a pure object-oriented language with capabilities comparable to C++ and C#. Java compiles to JVM (Java Virtual Machine) bytecode, instead of directly to native machine code. The use of a portable virtual machine gives the language a number of advantages over C++, such as automatic garbage collection, the ability to run a Java program on a wide range of platforms without recompiling them, and a generally safer programming environment. Java is often used for web applications (Java applets), where it allows programs embedded in web pages to run on the user's computer with good performance. The use of a virtual machine adds a degree of security when running untrusted Java applets this way.

The JVM has also become popular in its own right, with several third-party implementations. Several other languages, including Clojure (Section 6.1.4.1), Groovy (Section 6.1.2.8), and Scala (Section 6.1.9.2), compile to JVM bytecode.

MORTAL's syntax has much in common with Java syntax. For example, MORTAL classes look and behave a lot like Java classes, including support for inner classes. Also, by default, MORTAL generics use type erasure the same way Java generics do. Exception handling is also similar.

Some of the ways MORTAL differs from Java are:

- MORTAL is portable, compiles to native machine code, and does not require a runtime.
- MORTAL aims to be compatible with ordinary C, C++, and Fortran code.
- MORTAL does not force programmers into the OOP paradigm.
- MORTAL can pass function arguments by name instead of by position.
- MORTAL supports operator overloading.
- MORTAL supports multiple inheritance.
- MORTAL aims to support declarative programming.

6.1.2.8 Groovy

[23] Groovy is an offshoot of the Java language. It adds numerous features, including operator overloading and functional programming features, while still compiling to JVM bytecode and thus being compatible with ordinary Java code. Its metaprogramming capabilities and flexible syntax gives it a certain ability to build domain-specific languages.

MORTAL has borrowed some elements from Groovy syntax. The null-safe member operator `?.` is inspired by Groovy. Other similar operators are the «Elvis» operator `?:` and the «spaceship» operator `<=>`.

6.1.2.9 Vala

[24] Vala is an object-oriented programming language with C#-like syntax, but specifically designed for the GLib/GObject class framework (Section 6.3.1). The Vala compiler generates plain C code, which can then be compiled to native machine code.

Vala provides automatic memory management on top of GObject's native reference counting, without the need for a garbage collector [25]. MORTAL memory management is strongly influenced by Vala memory management.

6.1.2.10 Delphi

[26] Delphi is a proprietary evolution of Borland Pascal with object-oriented features, and a visual component library that allows both user interfaces to be created and functionality to be added through dragging, dropping, and linking components. Delphi is considered a Rapid Application Development (RAD) tool.

MORTAL aims to support declarative programming in a way that allows components to be linked with the same ease as in Delphi.

6.1.3 Array languages

Array languages are dynamic languages that specialize in computations on multidimensional data sets, by providing a selection of predefined functions and operators for this purpose. Each operation can typically transform the entire data set all at once, and builtin operations are typically also implemented as native machine code. This can result in quite good performance, but only to the extent the provided operations are a good match to the problem.

6.1.3.1 Matlab

[27] Matlab is a proprietary procedural and object-oriented language. It is built around vector and matrix operations. It has numerous libraries (toolboxes) for visualization, analysis, and simulation. It can also work alongside the dataflow-based Simulink (Section 6.1.6.5).

MORTAL's collection of overloadable mathematical operators is somewhat inspired by Matlab's operators.

There is an open source language, Octave, that's mostly compatible with Matlab [28].

6.1.3.2 R

[29, 30] R provides procedural, object-oriented, and functional programming. It has numerous libraries for visualization and statistical analysis, many of which are quite sophisticated. Unfortunately, the size of the data sets R can work with are usually limited to how much can be loaded into memory.

Due to the popularity of R among scientists, MORTAL aims to be easy to learn for R programmers, and to have similar productivity-enhancing features. For example, like R, MORTAL allows arguments to be bound to parameters by name. (Many of R's functions have numerous optional parameters, making this a quite useful feature.)

6.1.4 Functional languages

Functional languages are fundamentally based on lambda calculus (possibly allowing other programming paradigms to be layered on top). They strive to express computations not by specifying (imperative) actions, but by describing transformations. (Thus, in pure functional languages, variables are immutable, and can only be transformed into new variables or return values.) Such languages are very expressive, as well as very safe, but lambda calculus does not map directly to how real hardware works, which makes it difficult for functional languages to achieve the same runtime performance that imperative languages do. The various functional languages in existence tend to make different tradeoffs between purity, expressivity, and performance, and most of them use tracing garbage collectors.

6.1.4.1 Lisp

[31, 32] The original Lisp was developed in the 1950s, and has since spawned a whole family of functional languages based on the Lisp (S-expression) syntax. The Lisp family includes languages such as Common Lisp, Scheme, and Clojure. Some of them, including Common Lisp, also supports object-oriented programming. Lisp's homoiconicity (representational equivalence between code and data) gives it very powerful metaprogramming capabilities, with code being able to transform and rewrite itself in arbitrary ways, including the ability to build domain-specific languages.

6.1.4.2 Standard ML

[33] Standard ML is an impure functional language. It is statically typed, and supports algebraic data types. In general, types are inferred and do not need to be specified explicitly. Standard ML and its derivatives are used in programming language research.

6.1.4.3 OCaml

[34] OCaml is an impure functional language that supports both functional and object-oriented programming. Like Standard ML, OCaml is statically typed and uses type

infererence. OCaml is compiled to native machine code, resulting in decent performance.

6.1.4.4 Haskell

[35] Haskell is a pure functional language with elements from object-oriented languages, including operator overloading. It is statically typed, uses type inference, and supports algebraic data types. Despite being a pure functional language, it allows a certain degree of imperative programming through the use of monads (which are essentially representations of state in a way that does not violate the principles of lambda calculus, along with syntactic sugar that makes them easy to use in an imperative style). It uses lazy evaluation extensively, allowing even infinite sequences to exist and be computed on-demand.

6.1.5 Declarative languages

Unlike imperative and functional languages, declarative languages allow the programmer to focus on the «what» instead of the «how». Programmers need only describe the exact problem to be solved in a formal, machine-readable way, and can then let some automatic inference engine or solver search for a solution. Declarative languages are typically restricted to particular domains, and are not necessarily Turing complete. However, declarative programming is fairly intuitive and easy to learn for people that aren't computer scientists.

6.1.5.1 AMPL

[36] AMPL is a proprietary language for solving various mathematical problems, especially constrained optimization problems. It provides both declarative and imperative programming. It supports numerous solvers, each able to solve a certain class of problems.

6.1.5.2 Prolog

[37] Prolog is a logic programming language. Prolog programs mainly consist of statements expressing Horn clauses. (Horn clauses are a subset of first-order logic.) Execution of a Prolog program consists of the user providing some query, which Prolog's inference engine (which uses SLD resolution) will then try to satisfy. Prolog programs need not be purely declarative, as clauses may have side effects when the inference engine evaluates them, or cuts, which force the engine to skip parts of the search space.

6.1.5.3 Datalog

[38] Datalog is a fully declarative subset of Prolog. While it is logically complete, it is not Turing complete, and is thus mostly used as an embedded language in larger systems. It is possible for compilers and static analyzers to use Datalog to check various properties of a program [39].

6.1.6 Concurrent languages

Concurrent languages are languages explicitly designed to support multiple threads of execution, and allow them to communicate in safe ways.

6.1.6.1 Ada

[40] Ada is an early procedural language with support for concurrency as well as very strong safety features. It has a strict static type system, and supports both exception handling and a form of contract-based programming. Modern versions of the language also support object-oriented programming.

6.1.6.2 Erlang

[41, 42] Erlang is a functional language with support for Actor-based concurrency. It is a dynamic language with garbage collection, and quite robust and fault-tolerant. To help avoid service interruption, it even supports code hotswapping. Although it is based on a virtual machine concept, it is also possible to compile it to native machine code for good performance.

6.1.6.3 Go

[43] Go is an object-oriented language built around the Communicating Sequential Processes (CSP) [44] model of concurrency. It is a statically typed, garbage collected language that compiles to native machine code. Go provides a form of structural typing, where the compiler can automatically infer which interfaces are implemented by a structure. Go has a minimalistic philosophy, avoiding many features that exist in other object-oriented languages, such as inheritance, exceptions, and generics.

6.1.6.4 LabVIEW

[45] LabVIEW is a proprietary dataflow language, designed for interfacing with hardware devices such as laboratory instruments and sensors. It allows programs to be constructed graphically by dragging, dropping, and linking «virtual instruments». Data (possibly originating from hardware sensors) will then flow through the resulting dataflow graph, transformed or visualized as appropriate by the virtual instruments.

6.1.6.5 Simulink

[46] Simulink is a proprietary dataflow language, designed for analyzing and simulating dynamic systems. It integrates with Matlab (Section 6.1.3.1). It allows systems to be modeled by building graphical block diagrams. In addition to simulating the resulting systems within Simulink, it is possible to generate C, VHDL, and Verilog code from the models.

6.1.7 Dynamic languages

Dynamic languages are languages that are generally not compiled to native machine code, but rely on an interpreter (or JIT) and take advantage of the flexibility this gives them. They typically have dynamic type systems with duck typing, automatic memory management, seamless introspection, the ability for code to modify itself, and other powerful metaprogramming features. They can usually also be used as embedded languages in larger systems.

6.1.7.1 JavaScript

[47] JavaScript is an object-oriented language with functional features. It does not use classes, and objects instead inherit from other objects (prototypes). JavaScript is usually used as an embedded language, particularly on web pages.

6.1.7.2 Python

[48] Python is an object-oriented language that also provides procedural and functional programming. It does not use the traditional curly brace style, but uses indentation to delimit blocks. Python emphasizes readability, and imperative statements can not be used inside expressions. In Python, assignment is a statement (i.e., there's no in-line assignment). Python provides automatic memory management based on reference

counting. Python's expressivity and ease of use has made it popular also among scientists.

MORTAL has borrowed some elements from Python syntax. For example, in MORTAL, assignment using the equal sign `=` is a statement, not an operator, similar to Python. Also, arguments can be bound to parameter by name, as in Python. The `in` and `is` operators (and their negations) are inspired by Python. And, like Python, MORTAL is based on reference counting.

6.1.7.3 Ruby

[49] Ruby is an object-oriented language that also provides procedural and functional programming. Its functional programming support is more extensive than Python's. Ruby has a syntax that encourages (but does not require) a functional programming style, with everything being an expression, and blocks of code being first-class objects.

6.1.8 Non-CPU languages

While the compilers for these languages may run on the CPU, the programs themselves are intended to run on other hardware.

6.1.8.1 OpenCL

[50] OpenCL is a procedural language that lets certain types of heavy computations be offloaded to the computer's GPU (Graphics Processing Unit), which normally drives the computer's display. Modern computers have quite powerful GPUs designed for advanced 3D graphics, and their floating-point computational capacity can be orders of magnitude better than the main CPU's. Through languages like OpenCL, this power can be used for things other than 3D graphics.

MORTAL aims to eventually provide the ability to generate OpenCL code from MORTAL source code.

6.1.8.2 VHDL

[51] VHDL is a dataflow language (with some imperative features for state machine design) for designing and simulating digital circuits. VHDL programs can be compiled and downloaded to FPGAs (Field Programmable Gate Arrays). Since such programs are implemented in hardware, they can be orders of magnitude faster than software programs, depending on the task.

MORTAL aims to eventually provide the ability to generate VHDL code from MORTAL source code.

6.1.9 Multiparadigm languages

Though many of the languages in the other sections also combine elements from multiple paradigms, this section is dedicated to languages whose core design is fundamentally multiparadigm, resulting in very high expressivity.

6.1.9.1 Oz

[52] The Oz language implements pretty near every programming paradigm there is (including concurrent programming), resulting in an extremely expressive language. Its primary drawback is that its design makes it difficult to achieve performance comparable to other compiled languages, so it is mainly used for programming language research.

MORTAL attempts to avoid this performance trap by, as much as possible, limiting its own expressive power to things amenable to static analysis, thus allowing efficient machine code to be generated.

6.1.9.2 Scala

[53] The Scala language provides object-oriented, functional, and concurrent programming. It is compatible with Java (Section 6.1.2.7), and runs on the Java virtual machine (though work is also ongoing for using LLVM (Section 6.3.2) to generate native machine code). With its robustness, expressive power, and good performance, it is becoming a popular language for web services and high-performance distributed computing applications.

MORTAL has many similarities to Scala, but for performance reasons, often does things in different ways. For example, to add extension methods to classes, MORTAL prefers the D language's (Section 6.1.2.6) «Uniform Function Call Syntax» over the implicit type conversions of Scala's «enrich my library».

Some of the other ways MORTAL differs from Scala are:

- MORTAL compiles to native machine code, and does not require a runtime. (Scala's upcoming LLVM backend would also compile to native machine code, but probably still require a runtime.)

- MORTAL requires full type annotation on functions/methods, which reduces compile time.
- MORTAL supports contract-based programming.
- MORTAL can pass function arguments by name instead of by position.
- MORTAL aims to support declarative programming.

6.1.9.3 Rust

[54] The Rust language is a language that's still under development, but already usable. It implements procedural, object-oriented, functional, and concurrent programming, and is designed for distributed computing applications. Its self-hosting compiler uses LLVM (Section 6.3.2) to create native machine code, and provides very good performance. Its expressive power approaches (and may eventually even surpass) Scala's, except for not being compatible with Java. And its type system, static analysis, and resource ownership rules makes the language safer to use than many other natively-compiled languages, making it easy to avoid invalid pointers and race conditions.

MORTAL has many of the same goals and features as Rust, but with a different philosophy and syntax. MORTAL has a similar concept of resource ownership, but uses a less restrictive and more explicit model, primarily inspired by Vala (Section 6.1.2.9). For example, MORTAL allows multiple mutable references to an object to exist at any time, even for non-reference-counted objects.

Some of the other ways MORTAL differs from Rust are:

- MORTAL supports C++/Java-style object-oriented programming (including constructors, destructors, implicit conversions, class inheritance, and method and operator overloading).
- MORTAL supports exception handling (try/catch statements).
- MORTAL aims to ensure safety through type annotations and contract-based programming, not through forcing particular design patterns.
- MORTAL aims to support declarative programming.

6.1.9.4 Nim

[55] The Nim language (formerly known as Nimrod) is still under development, but has been usable for a while. It implements procedural, object-oriented, functional, and concurrent programming, as well as some level of declarative programming through AST-based term rewriting. Its self-hosting compiler generates C, C++, or Objective-C code, which can be compiled to native machine code with very good performance. Its powerful term-rewriting metaprogramming system can be used both for high-level code optimization and for creating domain-specific languages, resulting in a very expressive programming language. And its type system, static analysis, per-thread garbage-collected memory pools, and message queue system makes the language safer to use than many other native-compiled languages. Nim's garbage collector is based on deferred reference counting, making it faster than tracing garbage collectors.

MORTAL has many of the same goals and features as Nim, but with a different philosophy and syntax. Nim uses an indentation-based syntax (like Python), unlike MORTAL's curly brace style. MORTAL's transparent classes and functions may be used to achieve effects similar to Nim's templates and macros, and MORTAL metaexpressions may be used to achieve compile-time evaluation.

Some of the other ways MORTAL differs from Nim are:

- MORTAL supports automatic memory management using reference counting, without the use of a garbage collector. (MORTAL also reduces typical reference counting overhead in several ways, such as through variable liveness analysis and unowned references.) MORTAL also explicitly supports weak references.
- MORTAL does not need the programmer to know about pointers.
- MORTAL's metaprogramming system follows a more object-oriented style than Nim's.
- MORTAL uses a nominal type type system, while Nim mainly uses a structural type system (though there's a qualifier that allows types to be nominally distinct). MORTAL also has a more customizable Runtime Type Information system.
- MORTAL supports contract-based programming.
- MORTAL aims to support declarative programming in other ways than AST-based term rewriting.

6.1.9.5 Wolfram

[56] The Wolfram language (the language behind Wolfram Research's «Mathematica» and the «Wolfram Alpha» website) is a mature, proprietary multiparadigm language for scientific computing. It implements procedural and functional programming, as well as symbolic computation based on term rewriting (including things like symbolic integration and differentiation). It has a very large library of mathematical and scientific functions and visualizations, some of which are able to fetch and operate on various information published on the Internet.

Since Wolfram is a dynamic, non-compiled language, it is difficult for compiled languages like MORTAL to compete on expressiveness. That said, MORTAL aims to eventually support declarative programming with a symbolic computation system that may use similar principles as Wolfram. The intention is to allow MORTAL to rewrite mathematical equations into forms that solves the declared problem and can be efficiently computed by a compiled program, with as little programming effort as possible.

Some of the other ways MORTAL differs from Wolfram are:

- MORTAL is intended to be a general-purpose language.
- MORTAL supports C++/Java-style object-oriented programming.
- MORTAL supports contract-based programming.

6.2 Other languages

6.2.1 XML

[57] XML (Extensible Markup Language) is a markup language, typically used for representing information in a structured, machine-readable fashion, while still being readable to humans. XML allows the information and its schema (if any) to be stored and maintained separately.

MORTAL aims to make working with such languages straightforward.

6.2.2 OWL

[58] OWL (Web Ontology Language) is an ontology language based on Description Logics (DL), a family of subsets of first-order logic. OWL allows information and knowledge to

be represented in a machine-readable form, and in a distributed fashion. OWL ontologies are typically encoded using XML, but does not need to be.

MORTAL aims to be compatible with such languages in its support for declarative programming.

6.2.3 OBO

[59] OBO (Open Biomedical Ontologies) is an ontology language often used in bioinformatics. It is used to represent knowledge about things like biological functions and their relationships. It is derived from OWL, with a few extensions.

MORTAL aims to be compatible with such languages in its support for declarative programming.

6.2.4 SQL

[60] SQL (Structured Query Language) is a domain-specific language for database queries. It is used to retrieve data from, or write data into, relational databases.

MORTAL aims to make it easy to issue SQL queries, and process their results.

6.3 Libraries and frameworks

6.3.1 GLib

[61] GLib is an open source library that provides a number of portable functions and high-level data structures. It also provides a portable, language-agnostic, and full-featured object-oriented class framework, GObject. The Gtk+ widget library is built on top of GObject. Both GLib and GObject uses reference counting for their memory management.

MORTAL is designed to be compatible with GLib and GObject, and MORTAL's compiler uses GLib extensively.

6.3.2 LLVM

[62] LLVM is a collection of libraries and tools for creating compilers (both ordinary and JIT). A number of fully functional compilers based on LLVM exists for various languages, including «Clang», a C compiler.

MORTAL intends to use LLVM for compiling directly to native machine code (Section 2.8), and for generating code at runtime.

6.3.3 Spark

[63] Spark is a high-performance distributed computing framework implemented in Scala (Section 6.1.9.2). In addition to its native Scala API, it provides APIs for Java and Python. It achieves good performance by pipelining several transformations together into a single work unit, and then recovering from any loss of cluster nodes by recomputing the lost parts from the beginning. This allows Spark to minimize disk access, which can be a significant factor in a distributed system [64]. Spark can also be used to build distributed database systems [65].

MORTAL aims to be able to interface with frameworks such as this, and then implement declarative paradigms on top to make it easy to express complex scientific computations.

7 Conclusion

This thesis introduces and describes MORTAL, a new metaprogrammable programming language for high-performance applications, and its compiler. MORTAL's multi-paradigm approach is intended to bridge the gap between software engineers and scientists, its flexibility and metaprogramming is intended to make it suitable for a wide range of applications, and its static analysis features (such as its statically checked contract programming constructs) is intended to help make MORTAL programs bug-free.

The language currently has procedural and object-oriented programming, and provides many important features such as RTTI, function and operator overloading, subtype and parametric polymorphism, multimethods, and exceptions. The language design satisfies most of its original goals, but declarative paradigms still need to be designed and integrated in order to satisfy them all.

The compiler is self-hosting and able to compile itself, showing that the language and its compiler, though not fully implemented yet, is already usable. The performance of MORTAL programs is also able to match the performance of C programs.

MORTAL is open source, released under the MIT license, and available from https://sourceforge.net/proje

We believe that as MORTAL and its compiler matures, it will become a useful language for solving many of the demanding computational tasks of modern science.

8 Future work

Although the language is already usable, much work remains. Some of the most important remaining tasks are:

- Finish the memory management implementation. (To compute the optimal reference counting strategy in all cases, it may be beneficial to implement transformation into SSA form first.)
- Finish exception handling. The various control flow contingencies must be handled.
- Coroutines/generators (resumable functions). MORTAL could implement these by placing their «local» variables in a heap structure instead of on the stack, and let the caller hold a reference to the structure. The heap structure might also hold state such as which resume point to use.
- Anonymous functions and closures. Anonymous functions without closures could simply be transformed into regular functions with an automatically generated name. Closures may be best implemented by transforming the anonymous function into a coroutine (see above). If the anonymous function is saved to a delegate, the delegate can hold the reference to the coroutine's heap structure.
- Fully support the GObject class framework. Because of the complexities of GObject construction, MORTAL's metaprogramming facilities must first be significantly improved. It may also be an advantage to implement metaclasses in MORTAL, to be able to accurately express the relationships between the various GObject data types.
- More metaprogramming, including compile-time function evaluation. An embedded interpreter may have to be written to safely compute arbitrary expressions at compile time.
- Declarative programming (e.g., logic and constraint programming), and integration of external inference engines to solve declarative problems. MORTAL wrappers for

such inference engines would need advanced metaprogramming facilities to be able to rewrite declarative functions into imperative functions that call the inference engine (if it can't find a compile-time solution).

- Full static analysis. Static analysis could take advantage of the same inference engines that would be used logic programming, by transforming a function's code and its contracts into logical statements, and asking the inference engine to look for contradictions.
- More backends, such as C++, OpenCL, and the LLVM code generator.
- Runtime code generation. For runtime algorithm specialization, MORTAL bytecode representing the algorithm implementation could be embedded into the compiled executable. Assuming the executable is linked to the MORTAL compiler as well as LLVM, the compiler could then optimize the MORTAL bytecode as appropriate, then use LLVM to generate machine code that can be executed.
- Investigate making MORTAL syntax more permissive. It might be nice not to have use parentheses when calling functions in unambiguous ways, and also not to have to use semicolons after every imperative statement.
- Evaluate MORTAL's expressivity and safety. MORTAL programs should be shorter and easier to write than programs written in other languages, is this the case? Does MORTAL reliably detect common programming errors?
- Tutorials and sample programs, to help people learn the language.

References

[1] A. Hey, S. Tansley, and K. Tolle, *The fourth paradigm: Data-intensive scientific discovery*. Microsoft Research, 2009.

[2] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, pp. 21:1–21:44, Apr. 2011.

[3] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.

[4] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, pp. 40–51, Oct 1992.

[5] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, pp. 1–19, July 1970.

[6] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, pp. 121–141, Jan. 1979.

[7] L. Georgiadis, R. E. Tarjan, and R. F. Werneck, "Finding dominators in practice," *Journal of Graph Algorithms and Applications*, vol. 10, no. 1, pp. 69–94, 2006.

[8] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, (New York, NY, USA), pp. 194–206, ACM, 1973.

[9] K. D. Cooper, T. J. Harvey, and K. Kennedy, "Iterative dataflow analysis, revisited," *Proceedings of the PLDI'02*, 2002.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, Oct. 1991.

[11] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington, "A monotonic superclass linearization for Dylan," *SIGPLAN Not.*, vol. 31, pp. 69–82, Oct. 1996.

[12] J. C. Adams, *Fortran 95 handbook: complete ISO/ANSI reference*. MIT press, 1997.

[13] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C programming language*, vol. 2. prentice-Hall Englewood Cliffs, 1988.

[14] N. Wirth, "The programming language Pascal," *Acta Informatica*, vol. 1, no. 1, pp. 35–63, 1971.

[15] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[16] B. Meyer, "Eiffel: A language and environment for software engineering," *Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, 1988.

[17] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.

[18] B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[19] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, *The C# programming language*. Addison-Wesley Professional, 3rd ed., 2008.

[20] "The D Programming Language." http://dlang.org/.

[21] A. Alexandrescu, *The D programming language*. Addison-Wesley Professional, 2010.

[22] K. Arnold, J. Gosling, and D. Holmes, *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.

[23] "The Groovy programming language." http://www.groovy-lang.org/.

[24] "Vala - Compiler for the GObject type system," https://wiki.gnome.org/Projects/Vala.

[25] "Vala Memory Management Explained," https://wiki.gnome.org/Projects/Vala/ReferenceHa

[26] "Embarcadero Delphi." http://www.embarcadero.com/products/delphi.

[27] "MathWorks MATLAB." http://www.mathworks.com/products/matlab/.

[28] "GNU Octave." https://www.gnu.org/software/octave/.

[29] "The R Project for Statistical Computing." http://www.r-project.org/.

[30] R. Ihaka and R. Gentleman, "R: a language for data analysis and graphics," *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[31] G. L. Steele, *Common LISP: the language*. Digital press, 1990.

[32] G. L. Steele, Jr. and R. P. Gabriel, "The evolution of LISP," *SIGPLAN Not.*, vol. 28, pp. 231–270, Mar. 1993.

[33] R. Milner, *The definition of standard ML: revised*. MIT press, 1997.

[34] "OCaml." http://ocaml.org/.

[35] "Haskell Language." https://www.haskell.org/.

[36] "AMPL Streamlined modeling for real optimization." http://ampl.com/.

[37] W. Clocksin and C. S. Mellish, *Programming in Prolog*. Springer Science & Business Media, 2003.

[38] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 1, no. 1, pp. 146–166, 1989.

[39] J. Whaley, D. Avots, M. Carbin, and M. Lam, "Using datalog with binary decision diagrams for program analysis," in *Programming Languages and Systems* (K. Yi, ed.), vol. 3780 of *Lecture Notes in Computer Science*, pp. 97–118, Springer Berlin Heidelberg, 2005.

[40] N. H. Cohen, *ADA As a Second Language*. McGraw-Hill Higher Education, 2nd ed., 1995.

[41] "Erlang Programming Language." http://www.erlang.org/.

[42] J. Armstrong, R. Virding, C. Wikström, and M. Williams, "Concurrent programming in Erlang," 1993.

[43] "The Go Programming Language." https://golang.org/.

[44] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[45] "National Instruments LabVIEW." http://www.ni.com/labview/.

[46] "MathWorks Simulink." http://www.mathworks.com/products/simulink/.

[47] D. Flanagan, *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2002.

[48] "python." https://www.python.org/.

[49] "Ruby Programming Language." https://www.ruby-lang.org/en/.

[50] Khronos Group, "OpenCL - The open standard for parallel programming of heterogeneous systems." https://www.khronos.org/opencl/.

[51] P. J. Ashenden, *The designer's guide to VHDL*, vol. 3. Morgan Kaufmann, 2010.

[52] "The Mozart Programming System." http://mozart.github.io/.

[53] "The Scala Programming Language." http://www.scala-lang.org/.

[54] "The Rust Programming Language." http://www.rust-lang.org/.

[55] "Nim Programming Language." http://nim-lang.org/.

[56] "Wolfram Language." https://www.wolfram.com/language/.

[57] W3 Consortium, "Extensible Markup Language (XML)." http://www.w3.org/XML/.

[58] W3 Consortium, "OWL Web Ontology Language Overview." http://www.w3.org/TR/owl-features/.

[59] "The Open Biological and Biomedical Ontologies." http://obofoundry.org/.

[60] C. J. Date and H. Darwen, *A Guide to the SQL Standard*, vol. 3. Addison-Wesley New York, 1987.

[61] "GLib." https://wiki.gnome.org/Projects/GLib.

[62] "The LLVM Compiler Infrastructure Project." http://llvm.org/.

[63] "Apache Spark - Lightning-Fast Cluster Computing." https://spark.apache.org/.

[64] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[65] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, SIGMOD '13, (New York, NY, USA), pp. 13–24, ACM, 2013.