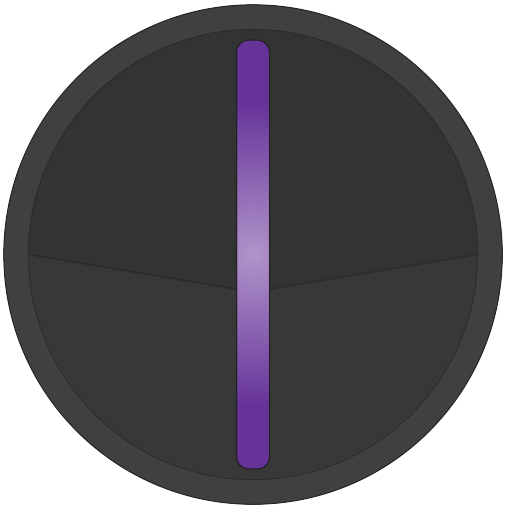
Bondrewd

Release 0.0

abel1502

Sep 08, 2023

CONTENTS



Bondrewd is a programming language that aims to provide powerful compile-time metaprogramming capabilities, while still being as fast as C++ when compiled. The language is inspired primarily by Python and Rust. The title itself is a reference to Made in Abyss, a Japanese manga and anime series.

The source code of the project can be found on GitHub.

Note: At the moment, the language is under active development. Anything in this documentation may change at any time.

Warning: As a matter of fact, a lot **has** changed, and some parts of this documentation are now outdated. Excercise caution, and refer to the blog for more up-to-date information.

CHAPTER

ONE

LANGUAGE IDEAS

This document features a list of ideas for key language features. These should give a general idea of the direction the language will evolve in.

- **Compile-time metaprogramming**. Ideally, I'm striving for flexibility similar to Python's, but entirely at compile-time. As few things as possible should be implemented as language features, and instead most things should be done through the standard library. If compiler support is required, it should be concealed behind a trivial standard library implementation, so that from within the language, it should be indistinguishable from something implemented in it.
- **Argument collectors**. In modern languages, function arguments aren't limited to a sequence of values. Some support variadic arguments, some support keyword ones… I believe this shouldn't be a language feature, but rather a library one. For that reason I'm considering an abstraction of an argument collector: an object responsible for handling the arguments passed to a function. This would also have the added benefit of simplifying the process of writing function wrappers, as one could simply reuse the argument collector of the wrapped function in the wrapping one. The idea is still rather vague, but I'm considering it as one of the crucial features of the language.

  … or not. The more I think about it, the more it seems to just overcomplicate things. I guess it would be better to just introduce keyword and variadic arguments as language features… But I really do like the idea of saying "take the same arguments as this function"…

  Maybe instead of collectors I should introduce "argument acceptors": objects responsible for turning an AST expression into some sort of value. Can `unused` be implemented through this?
- **Macros**. I'm pretty certain I want to be able to influence ASTs with (procedural) macros. Token stream-based macros seem cool, but might cause issues with namespace encapsulation and stuff like that. Maybe it would be fine if macros had a way to specify a limit of what grammatical constructs they could generate. That way the compiler can be sure that the macro won't create new scopes, for instance. Alternatively, we could just limit all token-based macros to creating exclusively expressions. The only things it would prevent them from creating are `impl`s, namespaces, variable declarations (in the immediate scope) and assignments. Maybe we could even allow statements as well, as long as the result is balanced (i.e. can be parsed completely). Also not sure how I feel about macros defining new macros. Overall, still got things to think about here.
- **Generalized namespaces**. Various attribute access is ubiquitous in modern programming languages. But in all cases known to me, it boils down to a string-keyed mapping. Coupled with some thoughts on traits, this gave me an idea to generalize namespaces: a key can be any (`ctime`) object. I'm considering two possible syntaxes for this: either `a::(74)` or `a::for 74`. (Note that this isn't about `.` vs `::` — I intend to have both with slightly different meanings). The first one probably would be more clear (for example, what would `a::for b::c` mean? Introducing new unapparent precedence rules goes against my design intentions). Normal attribute access would then be equivalent to `a::("attr")` or `a::for "attr"`. Come to think of it, a hybrid syntax might be a good idea: `a::(for 74)`.

  It is important to note that both objects play a part in generalized attribute resolution (which should be reflected in the attribute access override mechanism). Probably the left-hand side object would be queried first, and if it doesn't provide the attribute, the right-hand side object would be queried (in a distinct way, though — I don't

want attribute access to be symmetric by any means). This way, for example, the 'trait-for-type' namespace could be expressed as `Foo::(for Barable)`. I'm hoping with this and a few other changes, I could even make traits a library feature…

- **Generalized types**. It's desirable to be able to specify, for instance, a trait as the constraint for an argument of a function, automatically creating a template. However, at this point, it would require special support from the compiler. An alternative I'm considering is to call structs and builtin types something like "specific types", and to allow type annotations to be "generalized types". A generalized type should be able to tell is another type matches it or not (Still undecided about whether it should only concern specific types, or also generalized ones). This is a very raw idea though, I still have a lot to think about here.

  I guess a "specific type" is something that can be used to compile value manipulations. (Maybe this implies that there should be a separate metaclass for `ctime`-types). It could actually have separate methods for compile-time and run-time manipulations, allowing to make all `ctime` values to be passed by-reference regardless of attributes…

- **Mutable types**. This is still a very vague idea, but I'd like to have a way to change a variable's type 'dynamically' at compile time. For example, it would be nice to have the File type automatically track that it's been opened before it's used, and closed before it's destroyed. This can already be implemented, but comes at a run-time cost. My idea for a solution is to allow the object to store some information about itself at compile time. I'm not yet sure about the details, but I'm considering either something like mutable template parameters, or, alternatively, some sort of `ctime` fields (for non-`ctime` classes). One thing to note is that this cannot always be properly computed at compile time, so providing the only available File API in this way would be detremental. The better approach would be a wrapper around normal File that would implement the checks. If the further use cases I come up with could also be decomposed this way, I'll consider enforcing this specific mechanism in some way.

  Actually, this has lead me to consider the compile-time representation of objects. From what I've come up with so far, the object, from the perspective of the compiler, seems to consist of the type (a reference to a `ctime`-available object), an optional value (represented how?), which is only present for successfully constant-folded expressions, and a reference to some `ctime` object responsible for implementing manipulations with the object (perhaps only present for runtime objects, but maybe not). This is actually a little depressing in a way — to represent an object, we need 1 to 3 other objects… But something like this has already been done in other languages (like Python), so I guess it's not that bad. Maybe the value could be an AST tree…? No, I don't think so, actually. It's already done the other way — objects can be embedded into the AST in the form of `Constant` nodes.

- **Asynchronous compilation**. This isn't, strictly speaking, a language feature, in that it isn't exposed to the user. But it's a crucial concept behind forward reference resolution in complex contexts. Essentially, the idea is to make compilation of every function asynchronous, and whenever a ctime operation cannot be resolved, to suspend the compilation until the corresponding object is available. Note that, to support recursion between functions and other similar constructs, objects would have to be defined in parts, as soon as they become available. So, for a function, the ctime status and the signature would appear before the body (unless the return type is to be deduced from the body, this case I'm not sure about how to handle…). This system probably won't eliminate the need for incomplete types (*Generalized types*), though…

- **Cartridges**. I'm still very much undecided on how I'd like to implement the module ('cartridge') system. The current stub way of specifying the cartridge in the first line of the file really doesn't appeal to me. But, crucially, I don't want to tie cartridges to the filesystem. I really like how C++ namespaces separate the logical and physical organization of code.

- **Implicit `ctime`, explicit `runtime`**. I'm beginning to think that `ctime` should be the default, and runtime code should be explicitly marked as such. This is due to the fact that, starting at the root of the file, it actuallyccontains `ctime` code. All sorts of definitions (classes, traits, functions, namespaces, impls, vars and so on) are actually compile-time code, which may incur something for run-time (like a reserved space or function bytecode in memory), but nothing that would be compiled into any sort of assembly code. The first (and only) place where runtime code could appear is inside a runtime function, so it makes a lot more sense to mark those, as opposed to everything else. Maybe we could even omit the explicit marking, and instead assume that a function is runtime only if it contains a runtime-exclusive operation (i.e. calling another runtime function, or accessing a runtime

variable). One potential problem is that it might be counter-intuitive to some users, but I guess I'm willing to make that sacrifice for the sake of principle.

- ...

1.1 Code samples

This section features some code samples that I'm considering for the language. I use these to make better decisions about the language design.

Listing 1: `class` and `impl` considerations

```
cartridge test;


class Foo[T: type] {
    bar: int32,
    baz: &uint64,
    field: T
};


// Bad, I guess, because that would imply sequential evaluation of methods
// ... unless I make it sequential for references within the impl scope,
// but allow to use everything in method scopes. Like Python does.
impl[T: type] Foo[T] {
    if (T != int32) {
        public func sum(&self): uint64 => {
            bar + *baz
        };
    };
};
```

Listing 2: Function definition syntax options

```
// 1. Optional trailing return type in the form of `-> type`;
//    required `=>` before the body; any expression for the body.
// func foo(a: int32, b: int32) -> int32 => {
//     a + b
// };

// I guess I prefer this one
// 2. Same as 1, but with `: type` for the return type instead.
func foo(a: int32, b: int32): int32 => {
    a + b
};
```

Listing 3: Class definition in a function-like manner

```
// Essentially, you specify the constructor signature instead of the fields.
// This means that static fields must be declared in an `impl` block, along
// with methods. I'm considering allowing one `impl` block to be implicitly
```

(continues on next page)

(continued from previous page)

```
// declared as a braced block immediately after the class definition...
class Foo[T: type] (
    bar: int32,
    baz: &uint64,
    field: T
);
```

Listing 4: `ctime` declarations

```
// `ctime` functions could be declared in one of two ways:
// (I should pick one of these)

// 1. With a `ctime` keyword before the function body
//    (Essentially, just by making its result a compile-time expression)
func foo(a: int32, b: int32): int32 => ctime {
    a + b
};

// 2. With a `ctime` keyword before the declaration
ctime func foo(a: int32, b: int32): int32 => {
    a + b
};

// `ctime` classes are unambiguous:
ctime class Foo(int32 a, int32 b);

// Same thing goes for `ctime` traits:
ctime trait Bar {
    func sum(&self): int32;
};

// `ctime` impls are (probably) the same thing:
// (A `ctime` impl is only applicable to `ctime` classes and `ctime` traits)
ctime impl Bar for Foo {
    // All vars and methods are implicitly `ctime`
    func sum(&self): int32 => self.a + self.b;
};
```

Listing 5: `.` vs `::` for attribute access

```
class MyInt(@private int32 value);

impl PrivateCtor for MyInt {};

impl MyInt {
    // A static variable
    var total: int32 = 0;

    // Also a static variable, but with a special attribute
    @std::as_type_field
    var total_2: int32 = 0;
};
```

(continues on next page)

(continued from previous page)

```
var my_int_val = MyInt(5);

// To access a static variable of a class, you use `::`
MyInt::total += 1;
my_int_val::total += 1;

// `.` accesses instance attributes. That includes instance attributes of types,
// defined in the corresponding metatype
my_int_val.value += 1;
std::dbg::ctime_assert(MyInt.name == "MyInt");

// However, I think I'd like to have some cool methods on some types, like:
type.of(my_int_val) my_int_val_2;  // Actually, with this specific one there's
    // another problem: `of` must be a ctime function, so it cannot accept
    // non-ctime arguments...
    // One possible solution could be to allow constant-folding functions with
    // non-ctime but unused arguments... Note: we should probably either still
    // compute their expressions at runtime, just to enforce side-effects, or
    // check that no side-effects occur in them. Alternatively, we could allow
    // adding an `unused` marker to an argument, suggesting that its evaluation
    // can be dropped but forbidding its use in the function body. Maybe just
    // having no name could act as such a marker, but that could be confusing.
int32 c = int32::max;
int32 d = int32.parse("123");

// To have static variables accessible through `.`, I'm considering adding a
// special annotation to tell the type factory that it should be that way.
// With it, you have no guarantees agains current or possible future name
// collisions, so it might be a good idea to mark the annotation with a leading
// underscore...
// By the way, static attributes are stored in `MyInt.statics`
MyInt.total_2 += 1;
```

Listing 6: Argument collectors

```
func increment(a: int32): int32 => {
    a + 1
};

func logged_increment(args: increment.args_def) => {
    log("Before");
    var res = increment(args);
    log("After");
    // TODO: How do I allow to say `return res;` and then not complain about an
    //       implicit unit return? I guess I'd like this block to have a `Never`
    //       for its return expression type, and then have it implicitly cast to
    //       `int32`
    return res;
};

func foo(
```

(continues on next page)

(continued from previous page)

```
    a: int32,
    b: int32 = 5,
    @std::varargs c: int32[],
    @std::unused_arg d: int32,
): int32 => {
    a + b + c.sum()  /* + d  /* nope - can't use an unused arg's value */ */
};
```

Listing 7: Example implementation of `type.of(...)`

```
trait type {
    // A lot of other things...

    // I'm not sure how template type deduction should be implemented, actually
    func of[T: type](@std::unused_arg value: T) -> type {
        T
    };

    // No, you know what, this is bad. I'd much prefer to provide type.of as a
    // compiler builtin, and then implement type deduction in the library with
    // it.
};
```

Listing 8: Imposing trait requirements on a type value

```
trait Foo {
    func foo(&self) -> Unit;
};

// These two definitions are essentially the same thing, with two exceptions:
//  - The second case checks that `T` implements `Foo` at compile-time
//    (which you can also do manually in the first case, though)
//  - The second case doesn't allow to explicitly specify the type of `value`
//    (although I'm considering adding a mechanism to get the result of
//     overload resolution as an object...)

func bar[T: type](value: T) => {
    value.foo();
};

func baz(value: Foo) => {
    value.foo();
};
```

Listing 9: Forward references

```
// In functions: will work, because function body is interpreted lazily
func f1() => {
    f2();
};

func f2() => {};
```

(continues on next page)

(continued from previous page)

```
// In classes: will not work as is...
// Maybe we should interpret the class fields lazily as well...
class C1(
    // By the way, this syntax for references might turn out problematic, since
    // it is ambiguous whether this means the refernce type, or a reference to
    // the type object...
    b: &C2,
);

class C2();
```

Listing 10: Immediate impls

```
// These are perfectly okay without any special grammar needed:
impl class Foo(
    int32 a,
    int32 b,
) {
    // ...
};

impl ns std::something {
    // ...
};

// However, it gets a bit messy with templates:
// (We need to specify template parameters thrice!)
impl[T: type] class Foo[T: type]()[T] {
    // ...
};

// I need some better solution for this...
// Actually, even without immediate impl, this is still problematic
// Maybe I should allow to somehow provide an impl for a template type
// without explicitly redeclaring the template parameters?
// I certainly don't want to repeat Rust's practice of having to repeat
// all type constraints every time...
```

Listing 11: Explicit template declarations

```
// Class template
template[T: type] class Foo(
    // ...
);

// Impl template
template[T: type] impl Foo {
    // ...
};

// Function template
```

(continues on next page)

(continued from previous page)

```
template[T: type] func bar() => {
    // ...
};

// Inline class impl template
// (No grammatical exception needed here. All this does is declare a template
//  within which lays an inline impl (impl with a class declaration inside).
//  Since impl blocks return the class object, this makes the class templated
//  automatically.)
template[T: type] impl class Baz() {
    // ...
};

// Could we use any expression here?
Abomination = template[T: type] (T, int32);

// I guess we could allow explicit names instead:
template Abomination[T: type] (T, int32);

// ... and deduce them from the object, if omitted:
template[T: type] class Smth();

// But then weird usecases arise:
template[T: type] std::int32;
// It already exists, but from the template's perspective, it's just a class
// with a qualname like any other. So, would it overwrite std::int32?
// And should it?

// What if we name the outermost entity?
template Smth2[T: type] class ();

// Maybe demand a => like with functions?
template Smth3[T: type] => impl class () {};

// This looks horrible, though...
template do_smth4[T: type] => func () => {};
```

Listing 12: Reference implementation for the core traits

```
impl ns std {

// A type is, essentially, just a marker trait. Specific type-related behavior
// is implemented through different traits. The primary reason for this to exist
// is that to impl object a for object b, Trait must be implemented for a's
// type, and Type should be implemented for b's type.
trait Type: Hash + Eq + Copy {
};

trait Trait {
    func get_impl_for(&self, type: &Type): Option[TraitImpl[Self]];

    func get_slots(&self): TraitSlots;
```

(continues on next page)

(continued from previous page)

```
};

trait ManualTrait : Trait {
    func do_impl_for(&self, type: &Type, impl: TraitImpl[Self]): Unit;
};

// TODO: TraitImpl, TraitSlots, ...?

}
```

Listing 13: Potential syntax for file-level configuration

```
// 1. A file statement with some attributes
@std::use_reference_type(std::permission_ref)
file;

// 2. A file block?
file (
    reference_type = std::permission_ref;
);

// 3. 'Outer' attributes?
@!std::use_reference_type(std::permission_ref);
// or
@^std::use_reference_type(std::permission_ref);

// Note: I'd potentially like these to be able to influence even the parser
// used for the file, but I'm not sure how that should work across different
// syntaxes...
```

CHAPTER

TWO

GRAMMAR SPECIFICATION

This document contains the most up-to-date grammar specification for the Bondrewd programming language. The grammar is written in a modified version of Pegen's format. The parser produced is a PEG parser with packrat caching.

The lexical specification is still to be documented.

Note: The grammar is still a work in progress. Use for general reference only.

2.1 Grammar

```
# PEG grammar for the Bondrewd language

# TODO: add lookaheads and cuts where applicable; add `(memo)` to the most common rules
# TODO: Sequence helpers!
# TODO: Helpers to change expr_context. Then also add expr_context to the AST
# TODO: Empty rules cause issues with how left-recursion is handled. Fix it!

@subheader '''\
#include <string>
'''

@extras '''\
std::string _concat_strings(const std::vector<lex::Token> &strings) const {
    if (strings.empty()) {
        return "";
    }

    std::string result{};
    std::string_view quotes = "";
    bool first = true;

    for (auto &s: strings) {
        if (first) {
            quotes = s.get_string().quotes;
            first = false;
        }

        result += s.get_string().value;
```

(continues on next page)

(continued from previous page)

```
        if (s.get_string().quotes != quotes) {
            // TODO: Custom error type!
            throw std::runtime_error("String literals must have the same quotes");
        }
    }

    return result;
}

template <typename T>
ast::maybe<T> _opt2maybe(std::optional<ast::field<T>> opt) {
    if (opt) {
        return std::move(*opt);
    } else {
        return nullptr;
    }
}

template <typename T>
ast::sequence<T> _prepend1(ast::field<T> item, ast::sequence<T> seq) {
    assert(item);
    seq.insert(seq.begin(), std::move(*item));
    return seq;
}

'''

# This means all ast::* types are automatically wrapped into ast::field<>
@wrap_ast_types

start: file

#region file
file[ast::file]:
    | b=stmt* $  { ast::File(std::move(b)) }
#endregion file

#region stmt
stmt[ast::stmt] (memo):
    | cartridge_header_stmt
    | assign_stmt
    | expr_stmt
    | pass_stmt

cartridge_header_stmt[ast::stmt]:
    | 'cartridge' n=name ';'  { ast::CartridgeHeader(std::move(n)) }

assign_stmt[ast::stmt]:
    | a=expr op=assign_op b=expr ';'  { ast::Assign(std::move(a), std::move(b),␣
↪std::move(op)) }

assign_op[ast::assign_op]:
```

(continues on next page)

(continued from previous page)

```
    | '='    { ast::AsgnNone() }
    | '+='   { ast::AsgnAdd() }
    | '-='   { ast::AsgnSub() }
    | '*='   { ast::AsgnMul() }
    | '/='   { ast::AsgnDiv() }
    | '%='   { ast::AsgnMod() }
    | '<<='  { ast::AsgnLShift() }
    | '>>='  { ast::AsgnRShift() }
    | '&='   { ast::AsgnBitAnd() }
    | '|='   { ast::AsgnBitOr() }
    | '^='   { ast::AsgnBitXor() }

expr_stmt[ast::stmt]:
    | a=expr ';'  { ast::Expr(std::move(a)) }

pass_stmt[ast::stmt]:
    | ';'  { ast::Pass() }
#endregion stmt

#region defn
defn[ast::defn] (memo):
    | f=xtime_flag a=raw_defn  { ({ a->flag = std::move(f); a; }) }

raw_defn[ast::defn]:
    | var_def
    | func_def
    | struct_def
    | impl_def
    | ns_def

# TODO: Allow 'let' too
var_def[ast::defn]:
    | 'var' n=name t=type_annotation? v=['=' expr] ';'  { ast::VarDef(std::move(n), _
↪opt2maybe(std::move(t)), _opt2maybe(std::move(v)), true) }

func_def[ast::defn]:
    | 'func' n=name? '(' a=args_spec ')' t=type_annotation? b=func_body  {␣
↪ast::FuncDef(std::move(n), std::move(a), _opt2maybe(std::move(t)), std::move(b)) }

func_body[ast::expr]:
    | '=>' expr
    | block_expr

impl_def[ast::defn]:
    | 'impl' c=expr b=defn_block  { ast::ImplDef(std::move(c), std::nullopt,␣
↪std::move(b)) }
    | 'impl' t=expr 'for' c=expr b=defn_block  { ast::ImplDef(std::move(c), std::move(t),
↪ std::move(b)) }

defn_block[ast::sequence<ast::stmt>]:
    | '{' stmt* '}'

```

(continues on next page)

(continued from previous page)

```
# TODO: Forbid 'class' here?
struct_def[ast::defn]:
    | ('class' | 'struct') n=name? a=args_spec  { ast::StructDef(std::move(n),␣
↪std::move(a)) }

# TODO: Allow actual names!
ns_def[ast::defn]:
    | 'ns' ns_spec

#region ns_spec
# TODO: Represent "cartridge::" somehow other than a string?
ns_spec[ast::defn]:
    | 'cartridge' '::' a=ns_spec_raw  { ast::NsDef(_prepend1(std::move("cartridge"),␣
↪std::move(a))) }
    | a=ns_spec_raw  { ast::NsDef(std::move(a)) }

ns_spec_raw[ast::sequence<ast::identifier>]:
    | a='::'.name+  { std::move(a) }
#endregion ns_spec

#region args_spec
# TODO: *args, **kwargs - or templated that, perhaps?
# TODO: support for explicit argspec objects, if necessary
args_spec[ast::args_spec]:
    | a=args_spec_nonempty ','?  { std::move(a) }
    | { ast::args_spec(ast::make_sequence<ast::arg_def>(), false) }

args_spec_nonempty[ast::args_spec]:
    | "self" a=(',' arg_spec)*  { ast::args_spec(std::move(a), true) }
    | a=','.arg_spec+  { ast::args_spec(std::move(a), false) }

# TODO: unused and fixed args?
arg_spec[ast::arg_spec]:
    | n=name t=type_annotation d=('=' expr)?  { ast::arg_spec(std::move(n), std::move(t),
↪ _opt2maybe(std::move(d))) }
#endregion args_spec
#endregion defn

#region flow
flow[ast::flow] (memo):
    | 'unwrap' a=raw_flow  { ({ a->unwrap = true; a; }) }
    | raw_flow

raw_flow[ast::flow]:
    | if_flow
    | for_flow
    | while_flow
    | loop_flow

if_flow[ast::flow]:
    | 'if' c=expr t=flow_block e=('else' flow_block)?  { ast::If(std::move(c),␣
↪std::move(t), _opt2maybe(std::move(e))) }
```

(continues on next page)

(continued from previous page)

```

for_flow[ast::flow]:
    | 'for' v=name 'in' s=expr b=flow_block e=('else' flow_block)?  {␣
↪ast::For(std::move(v), std::move(s), std::move(b), _opt2maybe(std::move(e))) }

while_flow[ast::flow]:
    | 'while' c=expr b=flow_block e=('else' flow_block)?  { ast::While(std::move(c),␣
↪std::move(b), _opt2maybe(std::move(e))) }

loop_flow[ast::flow]:
    | 'loop' b=flow_block  { ast::Loop(std::move(b)) }

flow_block[ast::expr]:
    | block_expr
    | flow_expr
    | flow_control_expr
#endregion flow

#region expr
expr_or_unit[ast::expr]:
    | expr
    | { ast::Constant(std::monostate()) }  # TODO: Implement Unit!

# TODO: Support constants!
expr[ast::expr] (memo):
    | defn_expr
    | flow_expr
    | expr_0

#region wrappers
defn_expr[ast::expr]:
    | a=defn  { ast::Defn(std::move(a)) }

flow_expr[ast::expr]:
    | a=flow  { ast::Flow(std::move(a)) }
#endregion wrappers

#region operators
#region expr_0
expr_0[ast::expr] (memo):
    | and_expr
    | or_expr
    | expr_1

and_expr[ast::expr]:
    | a=expr_2 b=('and' expr_1)+  { ast::BoolOp(ast::And(), _prepend1(a, b)) }

or_expr[ast::expr]:
    | a=expr_2 b=('or' expr_1)+  { ast::BoolOp(ast::Or(), _prepend1(a, b)) }
#endregion expr_0

#region expr_1
```

(continues on next page)

(continued from previous page)

```
expr_1[ast::expr]:
    | not_expr
    | expand_expr
    | pass_spec_expr
    | flow_control_expr
    | expr_2

not_expr[ast::expr]:
    | 'not' a=expr_1  { ast::UnOp(ast::Not(), std::move(a)) }

# TODO: Maybe add other expand rules?
#       For statements, at least?
expand_expr[ast::expr]:
    | 'expand' a=expr_1  { ast::Expand(std::move(a)) }

pass_spec_expr[ast::expr]:
    | 'ref' a=expr_1  { ast::PassSpec(ast::ByRef(), std::move(a)) }
    | 'move' a=expr_1  { ast::PassSpec(ast::ByMove(), std::move(a)) }
    | 'copy' a=expr_1  { ast::PassSpec(ast::ByCopy(), std::move(a)) }

flow_control_expr[ast::expr]:
    | return_expr
    | break_expr
    | continue_expr

return_expr[ast::expr]:
    | 'return' a=expr_or_unit  { ast::Return(std::move(a)) }

break_expr[ast::expr]:
    | 'break' a=expr_or_unit  { ast::Break(std::move(a)) }

continue_expr[ast::expr]:
    | 'continue'  { ast::Continue() }
#endregion expr_1

#region expr_2
expr_2[ast::expr]:
    | comparison_expr
    | bidir_cmp_expr
    | expr_3

comparison_expr[ast::expr]:  # TODO: Extract from sequence somehow (without a 1000-char␣
↪rule, preferably)
    | f=expr_3 n=comparison_followup_pair+  { ast::Compare(
        std::move(f),
        ast::make_sequence<ast::cmp_op>(),
        ast::make_sequence<ast::expr>())
    }

comparison_followup_pair[std::pair<ast::field<ast::cmp_op>, ast::field<ast::expr>>]:
    | o=comparison_op a=expr_3  { std::make_pair(std::move(o), std::move(a)) }

```

(continues on next page)

(continued from previous page)

```
comparison_op[ast::cmp_op]:
    | '=='  { ast::Eq() }
    | '!='  { ast::NotEq() }
    | '<'   { ast::Lt() }
    | '<='  { ast::LtE() }
    | '>'   { ast::Gt() }
    | '>='  { ast::GtE() }
    | 'in'  { ast::In() }
    | 'not' 'in'  { ast::NotIn() }

bidir_cmp_expr[ast::expr]:
    | a=expr_3 '<=>' b=expr_3  { ast::BinOp(ast::BidirCmp(), std::move(a), std::move(b))␣
↪}
#endregion expr_2

#region expr_3
expr_3[ast::expr]:
    | arithm_expr
    | bitwise_expr
    | expr_4

arithm_expr[ast::expr]:
    | sum_expr
    | product_expr
    | modulo_expr

sum_expr[ast::expr]:
    | a=(sum_expr | product_expr) o=sum_bin_op b=product_expr  { ast::BinOp(std::move(o),
↪ std::move(a), std::move(b)) }

sum_bin_op[ast::binary_op]:
    | '+'  { ast::Add() }
    | '-'  { ast::Sub() }

product_expr[ast::expr]:
    | a=(product_expr | expr_4) o=product_bin_op b=expr_4  { ast::BinOp(std::move(o),␣
↪std::move(a), std::move(b)) }

product_bin_op[ast::binary_op]:
    | '*'  { ast::Mul() }
    | '/'  { ast::Div() }

modulo_expr[ast::expr]:
    | a=expr_4 '%' b=expr_4  { ast::BinOp(ast::Mod(), std::move(a), std::move(b)) }

bitwise_expr[ast::expr]:
    | bitor_expr
    | bitand_expr
    | bitxor_expr
    | shift_expr

bitor_expr[ast::expr]:
```

(continues on next page)

(continued from previous page)

```
    | a=(bitor_expr | expr_4) '|' b=expr_4  { ast::BinOp(ast::BitOr(), std::move(a), std::move(b)) }

bitand_expr[ast::expr]:
    | a=(bitand_expr | expr_4) '&' b=expr_4  { ast::BinOp(ast::BitAnd(), std::move(a), std::move(b)) }

bitxor_expr[ast::expr]:
    | a=(bitxor_expr | expr_4) '^' b=expr_4  { ast::BinOp(ast::BitXor(), std::move(a), std::move(b)) }

shift_expr[ast::expr]:
    | a=(shift_expr | expr_4) o=shift_bin_op b=expr_4  { ast::BinOp(std::move(o), std::move(a), std::move(b)) }

shift_bin_op[ast::binary_op]:
    | '<<'  { ast::LShift() }
    | '>>'  { ast::RShift() }
#endregion expr_3

#region expr_4
expr_4[ast::expr] (memo):
    | unary_expr
    | power_expr
    | expr_5

unary_expr[ast::expr]:
    | o=unary_op a=(unary_expr | expr_5)  { ast::UnOp(std::move(o), std::move(a)) }

unary_op[ast::unary_op]:
    | '+'  { ast::UAdd() }
    | '-'  { ast::USub() }
    | '~'  { ast::BitInv() }
    | '&'  { ast::URef() }
    | '*'  { ast::UStar() }

power_expr[ast::expr]:
    | a=expr_5 '**' b=expr_5  { ast::BinOp(ast::Pow(), std::move(a), std::move(b)) }
#endregion expr_4

#region expr_5
expr_5[ast::expr]:
    | dot_attr_expr
    | colon_attr_expr
    | call_expr
    | macro_call_expr
    | subscript_expr
    | expr_6

dot_attr_expr[ast::expr]:
    | a=expr_5 '.' b=name  { ast::DotAttribute(std::move(a), std::move(b)) }

```

(continues on next page)

(continued from previous page)

```
colon_attr_expr[ast::expr]:
    | a=expr_5 '::' b=name  { ast::ColonAttribute(std::move(a), std::move(b)) }

call_expr[ast::expr]:
    | a=expr_5 '(' b=call_args ')'  { ast::Call(std::move(a), std::move(b)) }

macro_call_expr[ast::expr]:
    | a=expr_5 '!' b=token_stream_delim  { ast::MacroCall(std::move(a), std::move(b)) }

subscript_expr[ast::expr]:
    | a=expr_5 '[' b=call_args ']'  { ast::Subscript(std::move(a), std::move(b)) }

# TODO: Actually implement
call_args[ast::call_args] (memo):
    | { ast::call_args(ast::make_sequence<ast::call_arg>(), nullptr, nullptr) }

token_stream[ast::expr]:
    | token_stream_delim
    | token_stream_no_parens

token_stream_delim[ast::expr]:
    | '(' ~ a=token_stream* ')'  { ast::TokenStream(/* ??? */) }
    | '[' ~ a=token_stream* ']'  { ast::TokenStream(/* ??? */) }
    | '{' ~ a=token_stream* '}'  { ast::TokenStream(/* ??? */) }

token_stream_no_parens[ast::expr]:
    | (!any_paren any_token)+  { ast::TokenStream(/* ??? */) }

any_paren:
    | '(' | ')'
    | '[' | ']'
    | '{' | '}'

any_token[lex::Token]:
    | NAME
    | NUMBER
    | STRING
    | KEYWORD
    | PUNCT
#endregion expr_5

#region expr_6
expr_6[ast::expr]:
    | primary_expr
#endregion expr_6
#endregion operators

#region primary
primary_expr[ast::expr]:
    | a=NUMBER  { ast::Constant(util::variant_cast(a.get_number().value)) }
    | &STRING a=strings  { ast::Constant(std::move(a)) }
    | '...'  { ast::Constant(/* Ellipsis, somehow... */) }
```

(continues on next page)

(continued from previous page)

```
    | var_ref_expr
    | group_expr
    | tuple_expr
    | array_expr
    | ctime_block_expr
    | block_expr

var_ref_expr[ast::expr]:
    | a=name  { ast::VarRef(std::move(a)) }

# TODO: _concat_strings!
strings[std::string] (memo):
    | a=STRING+  { _concat_strings(a) }

group_expr[ast::expr]:
    | '(' weak_expr ')'

tuple_expr[ast::expr]:
    | '(' ')'  { ast::Tuple(ast::make_sequence<ast::expr>()) }
    | '(' a=','.expr+ ','? ')'  { ast::Tuple(std::move(a)) }

array_expr[ast::expr]:
    | '[' ']'  { ast::Array(ast::make_sequence<ast::expr>()) }
    | '[' a=','.expr+ ','? ']'  { ast::Array(std::move(a)) }

# TODO: Maybe allow runtime blocks too?
ctime_block_expr[ast::expr]:
    | 'ctime' b=block_expr  { ast::CtimeBlock(std::move(b)) }

block_expr[ast::expr] (memo):
    | '{' b=stmt* v=expr_or_unit '}'  { ast::Block(std::move(b), std::move(v)) }

# To allow for both a::b::c and a::(123)::("abra" concat "cadabra")
attr_name[ast::expr]:
    | n=name  { ast::Constant(std::move(n)) }
    | group_expr
#endregion primary

#region weak
weak_expr[ast::expr]:
    | infix_call_expr
    | expr

infix_call_expr[ast::expr]:
    | a=expr_4 o=name b=expr_4  { ast::InfixCall(std::move(o), std::move(a),␣
↪std::move(b)) }
#endregion weak
#endregion expr

#region utils
name[std::string]:
    | a=NAME  { a.get_name().value }
```

(continues on next page)

(continued from previous page)

```
xtime_flag[ast::xtime_flag]:
    | 'ctime'  { ast::CTime() }
    | 'rtime'  { ast::RTime() }
    | { ast::DefaultTime() }

type_annotation[ast::expr]:
    | ':' a=expr  { std::move(a) }
#endregion utils
```

CHAPTER

THREE

DATA MODEL

The three fundamental concepts behind Bondrewd's data model are *object*, *type* and *trait*. Everything in Bondrewd is an object (including types and traits). Every object has a type. Types may have traits implemented for them.

A type may be considered as the minimal structural unit with which the static type checking may operate. For runtime objects, in compiled code, polymorphism is inherently limited to values within a single type.

A trait may be considered as a category of types, as well as some functionality associated with them. A trait has a set of *slots* (which respresent the associated functionality).

An object is said to satisfy a trait if the trait is implemented for its type. An object is a type if it satisfies the `Type` trait. An object is a trait if it satisfies the `Trait` trait.

Note: If you want to create a custom type or a trait, pay attention that the `Type` or `Trait` traits should NOT be implemented for the actual object you're creating, but for its type. To avoid confusion, the term *metatype* is used to refer to the type of a type or a trait.

3.1 Core traits

`Trait`
: This trait signifies that an object is a trait.

`Type`
: This trait signifies that an object is a type.

`Any`
: This trait is implemented for all types. It has no slots. It's used to represent the most general type constraint.

Unfinished...

CHAPTER

FOUR

COMPILE-TIME MODEL

One of the primary ideas of the language is to describe much of it as compile-time entities, as opposed to language-level features. To support that, the language should provide a powerful compile-time metaprogramming. This document aims to describe it in detail.

4.1 Introduction

To better understand the operation of Bondrewd's compile-time, think of your program not as of a definition in a static format, but as a script responsible for generating an executable. It has the compiler at its disposal, utilizing it as a helper library. This 'compilation script' is, overall, an imperative program, but you have a lot of declarative constructs at your disposal.

What is different from the 'script and library' scenario, and what might cause some confusion at first, is that the language of the 'script' and the language used for the declarative constructs largely coincide, and even interoperate to an extent. However, in exchange this provides a lot of power and flexibility.

4.2 Compile- and runtime scopes

At the top level, your program starts with compile-time code. Any class, function, or variable definition is always a compile-time statement. The only places where runtime code is allowed (runtime scopes) are inside a runtime function and in the initializer of a runtime variable.

Note: A function is runtime by default (implicitly) or if it has the `rtime` keyword. To make a function compile-time, use the `ctime` keyword.

In a compile-time scope, all code must be executed at compile-time. In a runtime scope, some code may be executed at compile-time, and some at runtime. This involves, for example:

- Constant-folding (e.g. `1 + 2` may be evaluated to `3` at compile-time);
- Type annotations (e.g. in `var foo: int32`, `int32` is a compile-time entity);
- Explicitly `ctime` code (e.g. `ctime { ... }`, `ctime func foo() { ... }`, `ctime var bar = 42`, `ctime if (true) { ... }`, etc.);
- Behavior of some code can be redefined at compile-time (e.g. attribute resolution, operator overloading, etc.).

The most important thing to remember is that a runtime function is merely a compile-time object containing an AST of runtime code and some metadata. Its compilation is handled, to an extent, as an ordinary compile-time operation, which means it may be influenced by compile-time code. Pretty much the same holds for runtime variables.

4.3 Execution order

Compile-time code is executed in the direct, straight-forward order. No forward references are available (because there's no feasible way to continue compiling until their definition without a preceding statement). Think of it like what Python does. This also means that from within functions, you may use yet undefined items, so long as they are defined by the time the function is first called. This holds for both compile-time and runtime functions, with the catch that runtime functions aren't 'called' at compile-time, but are 'compiled' at some point. For them, the 'first call' is the first (and only) time they are compiled. Usually, this is one of the last phases of compilation, so you may mostly rely on all compile-time definitions being available. Note, however, that an unused runtime function may never be compiled, so you should not rely on side effects of compile-time code in its definition.