Beast programming language

Language specification/reference

Daniel 'Danol' Čejchan

2016-2017

1 | Table of contents

2 | Introduction

Beast is an imperative, structured, modular programming language supporting OO and functional paradigms. Currently, only transcompilation to C is available. In future, a LLVM backend integration is planned.

Most notably, the language implements a new concept called *code hatching* (also a subject of this paper) that unifies templating, compile-time function execution, compile-time reflection and metaprogramming in general.

2.1 Inspiration

Inspiration from C++

- Most of the syntax

Inspiration from D

- Module system (not implemented yet)
- Compile-time function execution

Ideas not directly inspired by other languages

- Code hatching concept
- Language reflection
- The '`:`' accessor
- The constant value function parameters
- Decorators (not implemented yet)

3 | Lexical

In a normal code, Beast accepts only standard ASCII characters. Non-ascii characters are allowed only in comments.

```
Whitespace:     [\n \t \r ]+
```

(3.1)

3.1 Identifiers

```
Identifier:    [#a-zA-Z_][a-zA-Z_0-9]*
```

(3.2)

An identifier consists of any combination of lower and upper case ASCII letters, numbers and undescores, with these additional rules:

- An identifier cannot begin with a number.
- An identifier can begin with the hasthag (#) character. These identifiers are used in various language constructs and can have restrictions of where and how they can be used in declarations.

3.2 Keywords

More keywords will be added in the future.

```
auto, break, class, delete, else, if, module, new, return, while
```

(3.3)

3.3 Operators

```
+ - * / ? ! == != > >= < <= && || ! = :=
```

(3.4)

3.4 Special tokens

```
. , @ ( ) { }
```

(3.5)

3.5 Literals

```
Literal  ::=  IntLiteral
```

(3.6)

3.5.1 Boolean 'literals'

Beast does provide `true` and `false`, however those are not keywords or literals, but just `@ctime` variables defined from the runtime library. That means that they can be redefined.

3.5.2 The 'null' 'literal'

The `null` is of an unspecified type that is statically implicitly castable to pointer or reference of any type. It is not a keyword; it is defined in runtime library and thus can be redefined.

3.5.3 Integer literals

Integer literal is always of type `Int32`.

```
IntLiteral:    -?[0-9]+
```

(3.7)

3.6 Comments

```
LineComment:    //[^\n $][\n $]
Comment:        /\*.*?\*/
```

(3.8)

Additionally, **comments can be nested**.

```
void main() {
  /* This is a comment
    /* This is a comment, too. */
    Still a comment */
  thisIsACode();
}
```

(3.9)

4 | Expressions

```
Expression            ::=  AssignExpr
                           | DecoratedExpression
ParentCommaExpr       ::=  '(' [ Expression { ',' Expression } ] ')'
DecoratedExpression   ::=  { Decoration }+ AssignExpr
```

(4.1)

4.1 Operators

```
AssignExpr   ::=  LogicExpr
                  | LogicExpr '=' LogicExpr
                  | LogicExpr ':=' LogicExpr
LogicExpr    ::=  CmpExpr
                  | CmpExpr { '&&' CmpExpr }+
                  | CmpExpr { '||' CmpExpr }+
CmpExpr      ::=  SumExpr
                  | SumExpr '!=' SumExpr
                  | SumExpr { ( '==' | '>=' | '>' ) SumExpr }+
                  | SumExpr { ( '==' | '<=' | '<' ) SumExpr }+
SumExpr      ::=  MultExpr
                  | MultExpr { ( '+' | '-' ) MultExpr }+
MultExpr     ::=  VarDeclExpr
                  | NewExpr { ( '*' | '/' ) NewExpr }+
NewExpr      ::=  PrefixExpr
                  | 'new' PrefixExpr [ ParentCommaExpr ]
PrefixExpr   ::=  SuffixExpr
                  | '!'  SuffixExpr
SuffixExpr   ::=  AtomExpr
                  | 'auto'
                  | AtomExpr { SuffixOp }+
AtomExpr     ::=  [ ':'  ]  Identifier
                  | Literal
                  | Identifier
                  | ParentCommaExpr
SuffixOp     ::=  ParentCommaExpr
                  | '.'  Identifier
                  | '?'  | '!'
                  | ParentCommaExpr
```

(4.2)

4.1.1 Operator precedence

Priority	Operator	Semantics	Assoc.
1	`x(args)`	Function call	
	`x.ident`	Member access	
	`x!`	Mutable type	L→R
	`x?`	Reference type	
2	`!x`	Logical NOT	NOCH
3	**new** `x( args )`	Dynamic construction	NOCH
4	`x * y`	Multiplication	L→R
	`x / y`	Division	
5	`x + y`	Addition	L→R
	`x - y`	Subtraction	
6	`x != y`	Not equal to	NOCH
	`x < y`	Less than	L→R SPEC
	`x <= y`	Less than or equal to	
	`x == y`	Equal to	
	`x >= y`	Greater than or equal to	
	`x > y`	Less than	
7	`x && y`	Logical AND	L→R, SAME
	`x || y`	Logical OR	
8	`x = y`	Assignment	NOCH
	`x := y`	Reference assignment	

(4.3)

Associativity – explanation

- **L→R** Operations from the same group are processed from left to right, meaning `x + y + z` is processed as `(x + y) + z`.
- **NOCH** Operations cannot be chained with any operator of the same priority, meaning expressions like `x >> y >> z`, `x >> y << z` or `++!x` are syntactically incorrect.
- **SAME** Only the same operators can be chained, meaning `x & y & z` and `x | y | z` is correct, but `x & y | z` is not.
- **SPEC** Specific chaining rules, described in following sections.

Evaluation order Operands are evaluated left to right, if not specified otherwise.

4.1.2 Comparison operators chaining

Comparison operators can be chained in monotonous order:

```
// (Almost) equivalent with (a > b) && (b >= c) && (c == d) && (d >= e) && (e > f)
a > b >= c == d >= e > f;
// (Almost) equivalent with (a <= b) && (b < c) && (c <= d) && (d == e) && (e < f)
a <= b < c <= d == e < f;
// (Almost) equivalent with (a == b) && (b == c)
a == b == c;
// Syntax error
a > b < c;
```

(4.4)

When chaining comparison operators, the expressions in form `x == y > z` are internally rewritten to `(x == y) && (x > z)` (supports user overloaded operators), except the `x`, `y` and `z` expressions are evaluated only once.

4.2 Overloadsets (symbols, symbol lookup, identifier resolution, overloading)

Overloadset is a language structure, a container with all symbols that match currently processed expression. Consider following example:

```
1 Void f() {}
2
3 Void main() {
4   f();
5 }
```

(4.5)

Here, when processing the function call on line 4, the compiler starts with identifier `f`. It constructs an overloadset which contains the function `f` defined on line 1. Then, it reads parentheses, so it will look up for operator `x(args)` in the previous overloadset and put everything it has found to a new overloadset. Then, a process which selects the best matching overload is performed (will be described later in this chapter).

Now let's explain that process step by step.

4.2.1 Recursive identifier resolution

When there's an expression that starts with an identifier (`AtomExpr`, the *`Identifier`* version), a recursive identifier resolution is performed for it. This is done by searching symbols with the desired identifier in the current scope. If there is no match in the current scope, the compiler looks into parent scope, and so on. As soon as a match is found, the resolution ends, returning an overloadset with all matching symbols in the currently searched scope. If no match is found, an error is shown.

4.2.2 Local identifier resolution

The scoped identifier resolution is similar to the full identifier resolution, except it doesn't look into parent scopes at all.

4.2.3 The '`:ident`' accessor

This language construct can be used for example in parameter lists. When used, a **scoped identifier resolution is run** instead of the full identifier resolution. Also, **the resolution is not performed for the current scope but for the scope of an expected data type**. In function calls, the expected data type would be parameter data type. Please mote that operators are also translated into function calls, so this construct works with operators, too.

This of course does not work with '**`auto`**' parameters.

```
enum Enum {
  a, b, c, d
}

Void f( Enum e ) {}

Void main() {
  // Following lines are semantically identical
  f( Enum.a );
  f( :a );
}
```

(4.6)

4.2.4 Overload resolution

Overload resolution is performed every time the compiler needs to match arguments to a function overload. The process is performed over an overloadset where all items must be *callables* (meaning they support overload resolution; error is shown otherwise).

Each overload is assigned a "match level" number. Then, the overload with lowest match level is selected; if no overload matches given arguments, an error is shown; if there are more overloads with the same lowest match level, an error is shown. Overload match level is determined as a sum of 2^i of the following:

1. Implicit cast needed for at least one argument
2. Inferration needed for at least one anrgument (used `:ident`)
3. Function is static
4. Function is compiler-defined
5. General fallback (compiler-defined)

4.2.5 Type conversion

Beast introduces implicit and explicit cast. Implicit cast are used whenever there's mismatch between expected type and the type provided (for example when passing an argument to a function), explicit casting is done using function `to( @ctime Type targetType )` (which also tries implicit casting if there is no explicit cast available).

Implicit casting is realized by calling `var.#implicitCast( TargetType )`, where `TargetType` being the type the cast is intended to. The function has to return value of `TargetType`. Explicit cast is realized by calling `var.#explicitCast( TargetType )` in the same manner.

4.3 Operator overloading

Generally, all operators can be overloaded. Most operators are assigned an item in the `Operator` enum. Operator overloading rules are described in the table below; the "Enum" column corresponds with names of the `Operator` enum items (and the `@ctime Operator op` argument in the "Resolution" column). The "Resolution" column describes how operators are resolved. If there are multiple function resolutions mentioned in the column, the first one is tried first, if it fails (no matching overload is found), the second one is tried, and so on.

Operator	Enum	Resolution
`x(args)`		`x.#call(args)`
`x!` `x?`	`suffNot` `suffRef`	`x.#opSuffix( op )`
`!x`	`preNot`	`x.#opPrefix( op )`

`x * y`
`x / y`
	`binMult`
	`binDiv`
		`x.#opBinary( op, y )`
		`y.#opBinaryR( op, x )`

`x + y`
`x - y`
	`binPlus`
	`binMinus`	

`x != y`
`x < y`
`x <= y`
`x == y`
`x >= y`
`x > y`
	`binNeq`
	`binLt`
	`binLte`
	`binEq`
	`binGte`
	`binGt`	

`x && y`
`x || y`
	`binLogAnd`
	`binLogOr`	

`x = y`	`assign`	`x.#assign( y )`
`x := y`	`refAssign`	`x.#refAssign( y )`

(4.7)

4.4 Variable declarations

```
VarDeclStmt  ::=  { Decoration } TypeExpr Identifier
                      [ ( '=' | ':=' ) Expression ] ';'

TypeExpr     ::=  PrefixExpr
```

(4.8)

Declaring a variable is a thing well known from other programming languages. Declaring a variable consists of two tasks:

1. Allocating space for it (heap or stack)
2. Calling the constructor – evaluating `var.#ctor( args );`

There are multiple ways of how to construct a variable:

1. `Type var;` allocates a variable on the stack and calls `var.#ctor()`
2. `Type var = val;` allocates a variable on the stack and calls `var.#ctor( val )`
3. `Type var := val;` allocates a variable on the stack and calls `var.#ctor( Operator.refAssign, val )`
4. `Type( arg1, arg2 )` allocates a temporary variable on the stack, calls `tmpVar.#ctor( arg1, arg2 )` and returns the variable
5. `new Type( arg1, arg2 )` allocates a variable on the heap, calls `heapVar.#ctor( arg1, arg2 )`, and returns pointer to the variable

auto keyword When using **auto** instead of type, variable type is inferred from the value provided (this applies to variants 2 and 3). For dynamic construction (variant 5), type is inferred from type the expression is expected to be of.

```
auto var = true; // true is Bool, so var is of type Bool
Int? ref := new auto(); // ref constructor expects the Int? type, so auto
   is inferred as Int
```

(4.9)

@ctime variables Variable declarations can be decorated with the `@ctime` decorator. `@ctime` variables are completely evaluated at compile time. For more information, see chapter 9.

Variable lifetime (extent) Variable lifetime is same as in C++ or D – local variables exist until end of the scope (or **return**, **break**, exception, ...), where the are destroyed in reverse order they were defined. Dynamically constructed variables exist until they are manually destroyed (usually done via `DeleteStmt`). Static variables are constructed during application start and are never destroyed (probably will change in the future).

5 | Functions

```
FunctionDecl  ::=  TypeExpr Identifier ParentCommaExpr CodeBlock

CodeBlock     ::=  { Decoration } '{' { Statement } '}'

Statement     ::=  CodeBlock
                   | ReturnStmt
                   | BreakStmt
                   | IfStmt
                   | WhileStmt
                   | DeleteStmt
                   | Expression
                   | VarDeclStmt


ReturnStmt    ::=  'return' Expression ';'

BreakStmt     ::=  'break' ';'

IfStmt        ::=  { Decoration } 'if' '(' Expression ')' Statement
                        [ 'else' Statement ]

WhileStmt     ::=  'while' '(' Expression ')' Statement

DeleteStmt    ::=  'delete' Expression ';'
```

(5.1)

5.1 Function definitions

Functions in Beast are similar to those in the D programming language. You do not have to write declaration before definition.

```
Void foo() {}

Void foo2() {
  foo();
}
```

(5.2)

When declaring a parameter, you can access all parameters already declared in the parameter list (to declare a parameter, you can utilize all parameters to the left). In return type expression, you can access all function parameters.

```
b.#type foo( Int a, a.#type b, a.#type c ) {
  // code
}
```

(5.3)

5.1.1 **auto** return type

Function return type can be declared as **auto**. In that case, return type is inferred as a data type of the first **return** statement expression in the function code. If there are no return statements, return type is `Void`.

```
auto foo() {
  return 5; // return type is deduced to be Int
}
```

(5.4)

5.1.2 Constant-value parameters

It is possible to declare a parameter that accepts one exact value of a defined type (implicit casting is allowed). This is achieved by instead of using variable declaration syntax, pure expression is inserted to the parameter list. The value must me known at compile time, same as argument value when calling the function. Const-value parameter value and provided argument value are compared bit-by-bit.

This construct is useful for a `@ctime` parameter specialization or to differentiate between two overloads with otherwise same parameter types. It is also used when overloading operators.

```
class Stream {
  // CreateFrom.fromFile and .fromFile are const-value parameters
  Void #ctor( CreateFrom.fromFile, String filename ) { /* ... */ }
  Void #ctor( CreateFrom.fromString, String str ) { /* ... */ }
}

Void main() {
  // :ident construct is supported with const-value parameters
  Stream str1 = Stream( :fromFile, "file.txt" );
  Stream str2 = Stream( :fromString, "asdfgh" );
}
```

(5.5) Example of constant-value parameters. Please note that there is no `String` in Beast so far, neither user-defined enums

```
class C {

  // Operator.binPlus is a const-value parameter
  C #opBinary( Operator.binPlus, C? other ) {
    ...
  }

}
```

(5.6)

5.1.3 **@ctime** parameters

Static functions (class member functions currently not) can also have `@ctime` parameters – parameters decorated with the `@ctime` decorator (see chapter 9). Effectively, `@ctime` parameters are similar to template parameters in languages like C++, D or Java.

```
auto max( @ctime Type T, T a, T b ) {
  if( a > b )
    return a;
  else
    return b;
}

Void main() {
  print( max( Int, 5, 3 ) );
}
```

(5.7)

5.1.4 The 'auto' keyword

It is possible to declare a parameter as **auto** type. In that case, type of the parameter is deduced from provided argument type. This effectively creates a hidden `@ctime Type` variable, so functions with **auto** parameters cannot be declared as member functions (for now).

```
auto max( auto a, a.#type b ) {
  if( a > b )
    return a;
  else
    return b;
}

Void main() {
  print( max( 5, 3 ) );
}
```

(5.8)

5.2 Compiler-defined functions

There are currently these functions static defined by the compiler:

`print` functions

```
Void print( Bool data )
Void print( Int32 data )
Void print( Int64 data )
```

(5.9)

These functions print provided argument to the stdout as signed decimals (`true` is printed as `1` and `false` as 0). Calling these functions at compile time results in an error.

`assert`

```
void assert( Bool expr )
```

(5.10)

Does nothing if provided argument is `true`. If `expr` is `false`, creates an error (compiler error at compile time, prints text to stderr and exits the program at runtime).

`malloc`, `free`

```
Pointer malloc( Size bytes )
Void free( Pointer ptr )
```

(5.11)

Allocates/deallocates a memory on the heap. This function is also callable at compile time.

6 | Types

```
ClassDecl  ::=  'class' Identifier '{' DeclScope '}'
```

(6.1)

Currently, Beast only supports classes. There are also compiler-defined enums, but there's currently no syntactic support for user-defined enumerations.

6.1 Class declarations

Classes do not support inheritance at all (is to change in the future). No automatic constructor or destructor generation is implemented either, a programmer must manually write class constructors and destructors and call member constructors and destructors in them. It is important to call member constructors/destructors, especially for reference types. Constructors are declared as `Void #ctor( args )` and destructors as `Void #dtor()`.

For more info about constructor and destructor calls, see section 4.4.

For more info about operator overloading, see section 4.3.

```
class C {

  // Implicit constructor
  Void #ctor() {
    a.#ctor();
    b.#ctor();
  }

  // Copy constructor
  Void #ctor( C? other ) {
    a.#ctor( other.a );
    b.#ctor( other.b );
  }

  // Destructor
  Void #dtor() {
    a.#dtor();
    b.#dtor();
  }

  Int a;
  Int b;

}
```

(6.2)

Classes have following fields/functions/properties automatically defined/implemented by the compiler:

1. `T?` suffix operator that returns reference type (static call only)

2. `T!` suffix operator that currently does nothing (returns the class itself; static call only); in future, it will be used as a mutability declarator (Beast will be const-by-default)
3. Implicit cast to reference
4. `T( args )` constructing an instance of `T` with provided arguments (static call only; see section 4.4)
5. `var.to( Type )` for explicit/implicit casting (see subsection 4.2.5)
6. `var.#addr` that returns `Pointer` referencing the variable `var`
7. `T.#instanceSize` of type `Size` that returns instance size of given type (equivalent to `sizeof` in C++/D)
8. Comparison operators `T == T2` and `T != T2` for comparing type variables (static call only)

Classes can be considered a `@ctime` variables of type `Type` (see subsection 6.2.3).

Nested class declarations You can declare classes in classes, however you have to decorate the inner class with `@static`. Nested classes do not implicitly store pointer to parent class instance. They can be accessed from outside using `OuterClass.NestedClass`.

6.2 Compiler-defined types

Beast offers a limited amount of pre-defined types.

6.2.1 Numeric types

Beast provides 32-bit and 64-bit signed integer types – `Int32` (also aliased as `Int`) and `Int64`. Basic arithmetic operators are supported (`+`, `-`, `*`, `/`) and are overloaded in both left and right variants (see section 4.3), meaning both expressions like `a + 5` and `5 + a` are supported if `a` is of type implicitly convertible to `Int` (see subsection 4.2.5). `Int32` is implicitly castable to `Int64`.

Beast also has type `Size` which has size of 4 or 8 bytes, depending on a platform. This is a Beast alternative to `size_t`.

6.2.2 Reference types

Beast offers reference type `T?` (where `T` is the referenced type) and pointer type `Pointer`. `Pointer` is not bound to a type, referenced data is manipulated via `T ptr.data( @ctime Type T )`. You cannot have reference of a reference.

For most cases, reference is the way to go. It acts like referenced type, except for following cases:

- `ref.#ctor` and `ref.#dtor` calls reference constructor, not referenced type constructor.
- `ref := var` operator is overloaded and is used for changing the referenced variable
- Reference is implicitly castable to `Pointer`
- A `isNull` property is added (use as `ref.isNull`, not `ref.isNull()`) which returns if a reference references a variable or not
- A `@ctime Type ref.#baseType` property is added that stores the referenced type
- A `ref.#data` alias is added, which allows direct access to referenced variable namespace. This is useful for example when a programmer needs to manually call referenced value constructor/destructor (please note that it is usually dangerous) – he would use `ref.#data.-#dtor()`, because `ref.#dtor()` would call the reference destructor, not the referenced value one.

Property	Reference	Pointer
Pointer arithmetics	no	(in future)
Can be `null`	yes	yes
Rebindable	yes	yes
Binding	`ref := val` or `ref := ref2`	`ptr = val.#addr`
Member access	`ref.mem`	`ptr.data( Type ).mem`
Dereference	(implicit cast)	`ptr.data( Type )`

(6.3)

```
Int x = 5;
Int y = 6;

// ref now references variable x
Int? ref := x;
print( ref ); // prints 5
ref = 10; // x is set to 10

xref := y; // ref now references variable y
print( ref ); // prints 6
ref = 7; // y is is set to 7
```

(6.4)

```
Int x = 5;
Int y = 6;

// ptr now references variable x
Pointer ptr = x.#addr;
print( ptr.data( Int ) ); // prints 5
ptr.data( Int ) = 10; // x is set to 10

ptr = y.#addr; // ptr now references variable y
print( ptr.data( Int ) ); // prints 6
ptr.data( Int ) = 7; // y is is set to 7
```

(6.5)

6.2.3 Type type

Types (classes, enums, ...) are considered `@ctime` variables of type `Type`. Type variables can only be used as `@ctime` variables.

```
@ctime Type T! = Int;
T x = 5;

@ctime T := Bool;
T y = true;
```

(6.6)

7 | Modules

```
ModuleEntry           ::=  'module' ModuleIdentifier ';' { ModuleLvlDecl }
ModuleIdentifier      ::=  Identifier { '.'  Identifier }

ModuleLvlDecl         ::=  ModuleLvlDeclBlock
                           | ImportStmt
                           | FunctionDecl
                           | ScopeDecorationStmt
                           | ModuleLvlDeclBlock
ScopeDecorationStmt   ::=  { Decoration }+ ':'
ModuleLvlDeclBlock    ::=  { Decoration } '{' ModuleLvlDecl '}'
```

(7.1)

The program is divided into modules. Each module begins with a module declaration statement.

```
module package.package.moduleName;
```

(7.2)

The `ModuleIdentifier` in the `ModuleEntry` then works as an identifier for the module. Multiple modules with the same identifier are not allowed.

Filesystem representation Module identifier has to correspond with the directory structure the source file is in and the source file name has to be same as the module name (case sensitive). For example, having set up `project/src` and `project/include` source file directories, module `straw.beast.main` has to be in file `project/src/straw/beast/main.beast` or `project/include/straw/beast/main.beast`.

Naming convention Modules should be named in lower-CamelCase. This is not enforced, however disobeying this rule results in a warning.

8 | Decorators

```
Decoration  ::=  '@' Identifier
```

(8.1)

Generally, decorators provide syntax support for altering properties of types, declarations or even blocks of code.

8.1 Decorator application

At module level, there are three ways of how to apply a decorator:

```
module a;

class C {

// Scope decoration
@decoratorA @decoratorB:
  Int32 d;
  Int64 c;

  // Block decoration
  @decoratorE @decoratorF {
    Int8! b;
    Int16! c;
  }

  // Statement decoration
  @decoratorC @decoratorD Int8! a

}
```

Scope decorations Decorators are applied to all statements following the decoration up to next scope decoration or current scope end. You cannot use scope decorations directly in the module root scope.

Block decorations Decorators are applied to all statements in the block.

Statement decorations Decorators are applied to the statement that follows the decoration.

8.1.1 Decoration contexts

The concept of decorators wraps up lot of possibilities and functionality. In order to make things work, it is necessary to define different types of decorators, each altering the program in a different, unique way. In Beast, these decorator subtypes are called *decorator contexts*. Each decorator is defined for one particular context. However decorator identifiers are overloadable, so it is possible to define multiple decorators for different contexts with the same name.

The contexts are following (decorators of contexts commented with "system only" cannot be defined by the programmer):

```
enum DecorationContext {
  importModifier, // system only
  parameterModifier, // system only
  accessModifier, // system only for now
  metadata, // concept, not fully documented
  fieldWrapper, // concept, not fully documented
  typeWrapper, // concept, not fully documented
  typeMixin, // concept, not fully documented
  functionModifier, // system only
  classDeclarationModifier, // system only
  enumDeclarationModifier, // system only
  variableDeclarationModifier, // system only
  controlStatementModifier, // system only
  codeBlockModifier // system only
}
```

(8.3)

The 'importModifier' decoration context

8.1.2 Decorator overloading

Decorators can take parameters, just like functions (`@decoration` is equivalent to `@decoration()`). The behavior is following:

1. Overloads that do not match given arguments are ignored
2. Find overload that has lowest-index context (as ordered in the `DecorationContext` enum) and remove overloads that are of different context
3. Remove those overloads whose list of required implicit casts is superset of any other overload's required implicit cast list in the overloadset
   - **`auto`** arguments are treated as implicit casted
   - Arguments containing `:ident` accessor are also treated as implicit casted
4. Apply the only overload remaining (or error)

Decorator ordering by context In order to improve code readability and programmers' awareness, it is enforced by the compiler that decorations are ordered by the context they're used in (the order is specified in the `DecorationContext` enum).

8.1.3 Decorator conflicts

You cannot apply two decorators that are incompatible. The most common case are the access modifier decorators.

```
@final class C {

@public:
  @private Int8 a; // Error - @public and @private decorators are
     incompatible

}
```

(8.4)

8.2 Predefined decorators

8.2.1 Import locality modifiers (`@global`, `@local`)

The decorators only support `importDecoration` context. See **??**.

8.2.2 Access modifiers (`@public`, `@private`, `@protected` and `@friend`)

Access modifier decorators specify where a symbol is accessible from. They only support the `moduleLevelDeclaration` context.

`@public` Symbols with the `@public` access modifier are accessible from everywhere.

`@private` Symbols with the `@private` access modifier are accessible only from the current scope (and it's subscopes).

`@protected` The `@protected` only makes sense when used in classes. It makes symbols accessible from the current scope, its subscopes and any scope that derives from any of its subscopes.

`@friend` The `@friend` decorator exclusively allows access to one scope symbol, specified as the parameter. This rule has higher priority than `@protected` or `@private`.

Syntax: `@friend( Symbol friend )`

```
class C {
  @private @friend( func3 ) Int8 x;
  @protected Int16 func() {
    return 32000;
  }
  @public Int32 y;
}

@final class D : @public C {

@public:
  Void func2() {
    x = 6; // Error
    func(); // Ok
  }

}

Void func3() {
  C c;
  c.x = 5; // Ok - func3 is friend with C.x
  c.func(); // Error
  c.y = 10; // Ok
}
```

(8.5)

Inter-compatibility The `@public`, `@protected` and `@private` decorators are not compatible with any decorator from the three (this also means you can't apply them twice). Also, `@friend` and `@public` decorators are not compatible.

8.2.3 The `@ctime` (and `@autoCtime`) decorator

Generally, the `@ctime` decorator is related with compile time execution. It supports multiple contexts and depending on the context, its semantics can slightly change.

Axioms overview

- A compile-time variable's value is always known at compile time (=> cannot change during runtime).
- A compile-time function is always evaluated at compile time. That implies that parameters must be known at compile time. The return type is also compile-time.
- Some runtime functions can be executed at compile time.
- There is a difference between a compile-time function and a function executed at compile time.
- A compile-time variable is implicitly castable to const runtime variable.
- A compile-time class contains compile-time variables.
- A compile-time class can contain runtime functions, but those cannot modify class' fields (they can only be const).

Compile-time parameters A function parameter decorated with `@ctime` is a compile-time parameter.

Compile-time parameter values must always be known at compile time. Compile-time parameters can be mutable.

Technically, a function with compile-time parameter is a function template.

```
// 'y' and 't' parameters are compile-time
Void f( Int x, @ctime Int y, @ctime Type t ) {

}
```

(8.6)

For more info, see **??**.

Compile-time variables A variable decorated with `@ctime` is a compile-time variable. Compile-time variables defined in function bodies **can be mutable** (global compile-time variables cannot be mutable). A compile-time variable can be of a compile-time type (a non-compile-time variable cannot be of a compile-time type). You cannot define a compile-time variable as a dynamic member of a non-compile-time class.

In order to prevent confusion, it is prohibited to work with runtime variables and manipulate (change values of) @ctime variables in the same expression.

```
@ctime Type! t = Int; // t is a mutable @ctime variable that can hold
   types.

t x = 5; // Now we declared a variable x of type t (which is Int). It is a
   runtime variable.

t = String; // We're changing the @ctime variable here
t str = "asd"; // y is of type String

@ctime t str2 = "lol"; // str2 is compile time variable of type String
```

(8.7)

For more info, see **??**.

Compile-time functions A function decorated with `@ctime` is a compile-time function. A compile-time function is always executed at compile time. It returns a compile-time type. Its parameters and all variables used in its body are compile-time.

The `@ctime` decorator should be omitted in parameter, variable and return type declaration; a warning is shown otherwise.

```
@ctime Type TypeIntersection( Type t1, Type t2 ) {
  // if t2 is parent of t1
  if( t1 is t2 )
    return t2;

  // if t1 is parent of t2
  else if( t2 is t1 )
    return t1;

  else
    return Void;
}
```

(8.8)

For more info, see **??**

Compile-time classes A class decorated with `@ctime` is a compile-time class. Any instance of a compile-time class is a compile-time instance/variable. All its mutable member functions must be compile-time functions, all its member variables are compile-time variables (the `@ctime` decorator should be omitted).

```
@final @ctime class MyFunctionInfo {

@public:
  Void #ctor!( Function F ) {
    returnType = F.#returnType;
    parameterTypes = F.#parameters.map( x => x.type );
  }

@public:
  Type returnType;
  Type[] parameterTypes;

}
```

(8.9)

For more info, see **??**.

Compile-time control statements A control statement decorated with `@ctime` is a compile-time control statement. In a compile-time control statement, **expressions in the statement are evaluated in compile-time, but the statement bodies are evaluated at runtime**.

All variables declared in the statement expressions are compile-time. They don't need to be decorated with the `@ctime` decorator.

```
@ctime Type! t = Int8;

if( 3 == 5 )
  t = Int16; // Warning: compile-time variable modification inside a
     runtime control statement

// t == Int16 here
```

(8.10)

```
@ctime for( Int16 x = 0; x < 20; x ++ ) {
  // x is a compile-time variable
  writeln( "lol" );
  // The code generated from this would be twenty writeln("lol") calls
}

for( Int16 x = 0; x < 20; x ++ ) {
  // x is a runtime variable, the loop is executed at runtime
}

@ctime for( Int16 x = 0; x < 20; x ++ ) @ctime {
  // x is a compile-time variable and this entire block is also executed
     at compile-time (the "lol" is written into compiler console)
  writeln( "lol" );
}
```

(8.11)

Compile-time code block A code block decorated with the `@ctime` is executed at compile time. All variables declared in it are compile time and they should not be decorated with the `@ctime` decorator.

The `@autoCtime` decorator This decorator works similar to the `@ctime`, except it only makes things compile-time when it is possible.

In `classDeclaration` and `enumDeclaration`, it makes the type compile-time if it derives from a compile-time type or if it has a nonstatic compile-time member.

In the `parameterDeclaration` context, the parameter is made compile-time whenever the function is called with a parameter value that is known at compile time.

```
@final class HashTable( Type Key,

@public:
  // You may ask, that the hell t
     mutable type T, T!? means no
     T!?! means mutable reference
     can change where the referen
  T!?! #operator!( Operator("x[ar
    Index hash = key.#hash() % ta

    // Locating the record here
  }

}

Void main() {
  HashTable( String, Int )! table

  table[ stdin.read() ] = 5;

  table[ "key" ] = 10;
  // On the previous line the #ke
     the function is optimized (t
     compile time)
}
```

In the `functionDeclaration` context, the function is made compile-time when all of its parameters (all of them must be decorated with `@autoCtime` or `@ctime`) and/or return type are compile time.

In the `variableDeclaration` context, the variable is made compile time if its type is compile time.

The decorator cannot be used in any other contexts.

8.2.4 The `@noscope` decorator

9 | Code hatching concept (`@ctime`)

Ctime is a powerful concept introduced in Beast. It provides a form of metaprogramming, but that's not all what it does.

Basic idea The ctime concept has one simple idea: to introduce a **type of variables of which value can be deduced at any line of code** (without having to run the program).

These variables will are called ctime variables in Beast.

Consequences Consequences of this one simple rule are following:

10 | Reflection

10.1 Overloadsets

```
Overloadset #overloadset
Symbol #matchOverload( args )

class Overloadset {
}
```

(10.1)

10.2 Symbols

```
String #identifier
Type #type
SymbolMetadata #symbol

class SymbolMetadata {

}
```

(10.2)

10.3 Classes

```
String #className // Can be null for anonymous classes
ClassMetadata #class
Overloadset #member( String identifier )
Size #instanceSize

class ClassMetadata {
}
```

(10.3)

10.4 Function

```
Type #returnType
Type #returnType( args )
FunctionMetadata #function
Type #argType( Index index )
String #argIdentifier( Index index )
Bool #isVariadic

class FunctionMetadata {

@public:
  String identifier;


}
```

(10.4)

11 | Beast practices & styling guide

- Class and enum names are in `UpperCamelCase`.
- Enum members are in `lowerCamelCase`. They do not contain any enum-related prefixes.
- Decorator names are in `lowerCamelCase`.
- Variable (and parameter) names are in `lowerCamelCase`, type variables are in `UpperCamelCase`.
- Function names are in `lowerCamelCase`.
- The `_` symbol can be used in identifiers as a separator (for example class `PizzaIngredient_Cheese`).

11.1 Further recommended code style

- Indent with tabs (so anyone can set up tab size based on his preferences)
- Spaces in statements like this: **`if`**`( expr ) {`, opening brace on the same line
- Spaces around operators: `x + y`
- Decorators on the same line with decorated symbols (if the decoration list is too long).

12 | Plans for the future

- Aliased imports
- Namespaces
- User decorators
- Blueprints – "mixin classes" – bad idea?
- Mixins
- Mixins but no mixins (not mixing a string, mixing a declaration)
- Lambdas
- Singletons
- Extern functions – cooperation with other programming languages
- Compiler support for documentation comments?
- Compiler outputs intellisense data?
- Compiler caching

12.1 Documentation to-do

- Virtual functions, when a class is virtual
- `@noWarning(W103)`
- `@label` (for break, continue nested)
- `@notNull`
- # prefixed identifiers (rules, restrictions) + `to`
- **`class`** `X : @public @final Y`
- Array literals