Beast

PROGRAMMING LANGUAGE

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 ]+
```

$$\tag{3.1}$$

3.1 Identifiers

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

$$\tag{3.2}$$

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.

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

$$\tag{3.3}$$

3.3 Literals

```
Literal  ::=  IntLiteral
```

$$\tag{3.4}$$

3.3.1 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.

3.3.2 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.

3.3.3 Integer literals

Integer literal is always of type `Int32`.

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

$$\text{(3.5)}$$

3.4 Comments

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

$$\text{(3.6)}$$

Additionally, **comments can be nested**.

```
1  void main() {
2    /* This is a comment
3      /* This is a comment, too. */
4      Still a comment */
5    thisIsACode();
6  }
```

$$\text{(3.7)}$$

4 | Expressions

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

<div align="center">(4.1)</div>

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
```

<div align="center">(4.2)</div>

4.1.1 Operator precedence

Priority	Operator	Semantics	Assoc.
1	`x(args)`	Function call	
	`x.ident`	Member access	
	`x!`	Mutable type	
	`x?`	Reference type	L→R
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	
	`x <= y`	Less than or equal to	
	`x == y`	Equal to	L→R
	`x >= y`	Greater than or equal to	SPEC
	`x > y`	Less than	
7	`x && y`	Logical AND	L→R,
	`x || y`	Logical OR	SAME
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:

```
1 // (Almost) equivalent with (a > b) && (b >= c) && (c == d) && (d >= e) && (e > f)
2 a > b >= c == d >= e > f;
3 // (Almost) equivalent with (a <= b) && (b < c) && (c <= d) && (d == e) && (e < f)
4 a <= b < c <= d == e < f;
5 // (Almost) equivalent with (a == b) && (b == c)
6 a == b == c;
7 // Syntax error
8 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.

```
1    enum Enum {
2      a, b, c, d
3    }
4
5    Void f( Enum e ) {}
6
7    Void main() {
8      // Following lines are semantically identical
9      f( Enum.a );
10     f( :a );
11   }
```

$$(4.6)$$

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 Casting

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

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!`	`suffNot`	`x.#opSuffix( op )`
`x?`	`suffRef`	
`!x`	`preNot`	`x.#opPrefix( op )`
`x * y`	`binMult`	
`x / y`	`binDiv`	
`x + y`	`binPlus`	
`x - y`	`binMinus`	
`x != y`	`binNeq`	`x.#opBinary( op, y )`
`x < y`	`binLt`	`y.#opBinaryR( op, x )`
`x <= y`	`binLte`	
`x == y`	`binEq`	
`x >= y`	`binGte`	
`x > y`	`binGt`	
`x && y`	`binLogAnd`	
`x || y`	`binLogOr`	
`x = y`	`assign`	`x.#assign( y )`
`x := y`	`refAssign`	`x.#refAssign( y )`

(4.7)

4.4 Variable declarations

```
VarDeclStmt  ::=  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.

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

(4.9)

5 | Functions

```
FunctionDecl          ::=  TypeExpr Identifier ParentCommaExpr ( FunctionDefPart | ';' )
FunctionDefPart       ::=  [ FunctionDefConstraint ] ( CommandBlock | '=' Expression )
FunctionDefConstraint ::=  'if' '(' Expression ')'

CommandBlock          ::=  '{' { Command } '}'
Command               ::=  VarDeclCmd
                           | CommaExpression ';'
```

$$(5.1)$$

5.1   Definition constraint

5.2   The 'auto' keyword

6 | Types

6.0.1 Type casting

6.0.2 Pointers vs. references

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

$$(6.1)$$

7 | Modules

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


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

(7.1)

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

```
1│ 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.

7.1   Imports

```
ImportStmt  ::=  { Decoration } 'import' ModuleIdentifier ';'
```

(7.3)

Using the '**import**' statement, you can make symbols other modules accessible for the current scope.

Imports are not modified using standard access modifier decorators (@public, @private, etc.), but with @global and @local, @local being default.

7.1.1   Local imports

Local import makes the symbols from the specified module accessible only for the current scope (and its subscopes).

```
 1 | module a;
 2 |
 3 | Void aFunc() {
 4 |   // Do something
 5 | }
 6 |
 7 | /* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
 8 | module b;
 9 |
10 | import a; /* Equivalent with @local import a; */
11 |
12 | Void bFunc() {
13 |   aFunc(); // Ok
14 | }
15 |
16 | /* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
17 | module c;
18 |
19 | import b;
20 |
21 | Void cFunc() {
22 |   import a;
23 |   aFunc(); // Ok
24 | }
25 | Void cFunc2() {
26 |   aFunc(); // Error
27 | }
```

(7.4)

7.1.2 Global imports

Symbols that are imported using the global import are accessible from the current scope (and its subscopes) and all scopes that import the current scope (and their subscopes).

```
 1 | module a;
 2 |
 3 | Void aFunc() {
 4 |   // Do something
 5 | }
 6 |
 7 | /* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
 8 | module b;
 9 |
10 | @global import a;
11 |
12 | Void bFunc() {
13 |   aFunc(); // Ok
14 | }
15 |
16 | /* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
17 | module c;
18 |
19 | @global import b;
20 |
21 | Void cFunc() {
22 |   bFunc(); // Ok
23 |   aFunc(); // Ok
24 | }
25 |
26 | /* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
27 | module d;
```

```
28
29  import c;
30
31  Void Func() {
32    cFunc(); // Ok
33    aFunc(); // Ok
34  }
35
36  /* - - - - - - - - DIFFERENT MODULE - - - - - - - - - */
37  module e;
38
39  import d;
40
41  Void eFunc() {
42    dFunc(); // Ok
43    cFunc(); // Error
44    aFunc(); // Error
45  }
```

(7.5)

8 | Decorators

```
Decoration  ::=  '@' Identifier [ ParentCommaExpr ]
```

$$(8.1)$$

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:

```
 1  module a;
 2
 3  class C {
 4
 5  // Scope decoration
 6  @decoratorA @decoratorB:
 7    Int32 d;
 8    Int64 c;
 9
10    // Block decoration
11    @decoratorE @decoratorF {
12      Int8! b;
13      Int16! c;
14    }
15
16    // Statement decoration
17    @decoratorC @decoratorD Int8! a;
18
19  }
```

$$(8.2)$$

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):

```
1  enum DecorationContext {
2    importModifier, // system only
3    parameterModifier, // system only
4    accessModifier, // system only for now
5    metadata, // concept, not fully documented
6    fieldWrapper, // concept, not fully documented
7    typeWrapper, // concept, not fully documented
8    typeMixin, // concept, not fully documented
9    functionModifier, // system only
10   classDeclarationModifier, // system only
11   enumDeclarationModifier, // system only
12   variableDeclarationModifier, // system only
13   controlStatementModifier, // system only
14   codeBlockModifier // system only
15 }
```

$$(8.3)$$

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.
```
1  @final class C {
2
3  @public:
4    @private Int8 a; // Error - @public and @private decorators are incompatible
5
6  }
```

$$(8.4)$$

8.2 Predefined decorators

8.2.1 Import locality modifiers (@global, @local)

The decorators only support `importDecoration` context. See Imports.

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 )`

```
 1 | class C {
 2 |   @private @friend( func3 ) Int8 x;
 3 |   @protected Int16 func() {
 4 |     return 32000;
 5 |   }
 6 |   @public Int32 y;
 7 | }
 8 |
 9 | @final class D : @public C {
10 |
11 | @public:
12 |   Void func2() {
13 |     x = 6; // Error
14 |     func(); // Ok
15 |   }
16 |
17 | }
18 |
19 | Void func3() {
20 |   C c;
21 |   c.x = 5; // Ok - func3 is friend with C.x
22 |   c.func(); // Error
23 |   c.y = 10; // Ok
24 | }
```

$$(8.5)$$

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.

```
1  // 'y' and 't' parameters are compile-time
2  Void f( Int x, @ctime Int y, @ctime Type t ) {
3
4  }
```

$$(8.6)$$

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.

```
1  @ctime Type! t = Int; // t is a mutable @ctime variable that can hold types.
2
3  t x = 5; // Now we declared a variable x of type t (which is Int). It is a runtime
      variable.
4
5  t = String; // We're changing the @ctime variable here
6  t str = "asd"; // y is of type String
7
8  @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.

```
 1  @ctime Type TypeIntersection( Type t1, Type t2 ) {
 2    // if t2 is parent of t1
 3    if( t1 is t2 )
 4      return t2;
 5
 6    // if t1 is parent of t2
 7    else if( t2 is t1 )
 8      return t1;
 9
10    else
11      return Void;
12  }
```

(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).

```
 1  @final @ctime class MyFunctionInfo {
 2
 3  @public:
 4    Void #ctor!( Function F ) {
 5      returnType = F.#returnType;
 6      parameterTypes = F.#parameters.map( x => x.type );
 7    }
 8
 9  @public:
10    Type returnType;
11    Type[] parameterTypes;
12
13  }
```

(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.

```
 1  @ctime Type! t = Int8;
 2
 3  if( 3 == 5 )
 4    t = Int16; // Warning: compile-time variable modification inside a runtime control
 5        statement
 6  // t == Int16 here
```

(8.10)

```
 1  @ctime for( Int16 x = 0; x < 20; x ++ ) {
 2    // x is a compile-time variable
 3    writeln( "lol" );
```

```
 4     // The code generated from this would be twenty writeln("lol") calls
 5   }
 6
 7   for( Int16 x = 0; x < 20; x ++ ) {
 8     // x is a runtime variable, the loop is executed at runtime
 9   }
10
11   @ctime for( Int16 x = 0; x < 20; x ++ ) @ctime {
12     // x is a compile-time variable and this entire block is also executed at
           compile-time (the "lol" is written into compiler console)
13     writeln( "lol" );
14   }
```

$$(8.11)$$

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.

```
 1   @final class HashTable( Type Key, Type Value ) {
 2
 3   @public:
 4     // You may ask, that the hell is "T!?!" ?! Don't be scared, T! means a mutable
           type T, T!? means non-mutable reference to mutable type and T!?! means mutable
           reference to mutable type (mutable reference == you can change where the
           reference points to using the ':=' operator)
 5     T!?! #operator!( Operator("x[args]"), @autoCtime Key key ) {
 6       Index hash = key.#hash() % tableSize_;
 7
 8       // Locating the record here
 9     }
10
11   }
12
13   Void main() {
14     HashTable( String, Int )! table;
15
16     table[ stdin.read() ] = 5;
17
18     table[ "key" ] = 10;
19     // On the previous line the #key parameter is known at compile time, so the
           function is optimized (the hash calculation is performed at compile time)
20   }
```

$$(8.12)$$

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 | 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 | Language reflection

10.1 Overloadsets

```
1 Overloadset #overloadset
2 Symbol #matchOverload( args )
3
4 class Overloadset {
5 }
```

$$(10.1)$$

10.2 Symbols

```
1 String #identifier
2 Type #type
3 SymbolMetadata #symbol
4
5 class SymbolMetadata {
6
7 }
```

$$(10.2)$$

10.3 Classes

```
1 String #className // Can be null for anonymous classes
2 ClassMetadata #class
3 Overloadset #member( String identifier )
4 Size #instanceSize
5
6 class ClassMetadata {
7 }
```

$$(10.3)$$

10.4 Function

```
 1 Type #returnType
 2 Type #returnType( args )
 3 FunctionMetadata #function
 4 Type #argType( Index index )
 5 String #argIdentifier( Index index )
 6 Bool #isVariadic
 7
 8 class FunctionMetadata {
 9
10 @public:
11    String identifier;
12
13
14 }
```

(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.

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

12.2   Random thoughts scrapbook

```
 1  auto max( auto x, x.#type y ) = ( x > y ) ? x : y;
 2
 3  write( "Asd_\(varx)_and_\(vary)" );
 4
 5  T!? #new( @ctime Type T ) {
 6    return malloc( T.#size ).to( T!? );
 7  }
 8
 9  auto Int = Int32;
10  auto Float = Float32;
11
12  auto I1 = Int8;
13  auto I2 = Int16;
14  auto I4 = Int32;
15  auto I8 = Int64;
16  auto I16 = Int128;
17
18  Int8.#identifier == "Int8"
19  I8.#identifier == "I8"
20  I8.#name = "Int8"
21
22  decorator.ctime or decorator( ctime )
23  module.straw.beast.test or module( straw.beast.test )
24
25  class null {
26    @public @static T #implicitCast( Type T )
27      if( T is Pointer )
28    {
29      module.this
30      module.straw.beast.functionInAnotherModule();
```

```
31       module( "asd.test" ).functionInAnotherModule();
32       return T(0);
33   }
34 }
35
36 class Void {}
37
38
39 // X class
40 class X( Int32 i ) {
41 }
42
43 X(5).#type == Class
44 X(5).#template == X
45 X.#type == ClassTemplate
46 X.#identifier == "X"
47 X(5).#name == "X"
48 X(5).#identifier == null
49
50 // Overloadsets
51 I8
52
53 // Constantization
54 Void foo() {
55   Int! x = 5;
56   x += 3;
57   constantize x;
58
59   x += 3; // Error: x is constant
60 }
61
62 !!!
63 class C( @ctime Int x, Int y ) { // y IS A RUNTIME VARIABLE
64
65 }
66
67
68 @noscope @ctime if( true ) // if does not create scope, so all variables declared in
     it are available outside its body
69   Int! x;
70
71   while else jako v Pythonu
```

$$(12.1)$$

Multiple inheritance

- A base class cannot be inherited twice (in any way, considering the entire inheritance tree); if it is, all inheritances but one must be decorated with `@virtual`. This rule is not applied if the inherited class is an interface (doesn't contain variables)

- Classes don't handle constructing of their virtual base classes

12.2.1   Possible lists

List type	Allocation method	Reasonable item count	FIFO	LIFO	Index access	Fast insert	Fast delete	Constant item address	Keeps order
Array	2^n	1k		*	*				*
Lazy array (swap delete)	2^n	10k				*	*		
Linked list	n or 2^n	∞	*	*		*	*	*	*
Tree	n or 2^n	∞	*	*		*	*	*	*

$$(12.2)$$

Container names   container, array, list, collection, tree, table, hash, associativeXX, set