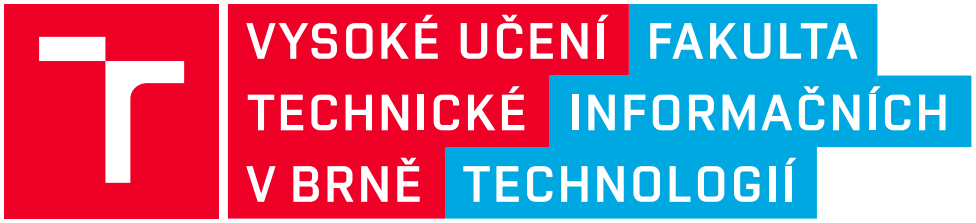

NATI (Not All That Innovative) programming language

Language specification/reference

Daniel 'Danol' Čejchan

2016

1 | Table of contents

2 | Introduction

NATI 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.

NATI is a practical tool for practical programmers. With NATI, you can write effective and readable code quite fast. Although the language itself builds on a significant abstraction, the functionality behind it is still close to HW. If you know the language, you can pretty well determine what assembly each line of your code generates.

Basic language classification:

- Imperative
- Classes are instances of `Type` class
- Statically typed

Pronounciation NATI is to be pronounced nɔːti, that means same as 'naughty'.

2.1 Why choose NATI

- **Good programming practices** NATI gently pushes you into writing your code readable and understandable.
- **Compile-time magic** In NATI, you can do many things much more comfortably using compile time function execution.
- **Language reflection** NATI has easy ways to get information about symbols, list class members, etc.
- **Modules** In NATI, you don't have to write separate header and source files and struggle with keeping them consistent.

2.2 Inspiration

Inspiration from C++

- Class system, multiple inheritance
- Most of the syntax
- User-defined literals

Inspiration from D

- Module system
- Compile time function execution

Ideas not directly inspired by other languages

- Overall collaboration of compile-time function execution
- Language reflection
- The '`:`' accessor
- The constant value function parameters
- Decorators

3 | Lexical

In a normal code, NATI accepts only standard ASCII characters. Non-ascii characters are allowed only in comments and string literals.

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

(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

```
alias, auto, break, case, class, continue, else, enum, for, if, import, in, is,
   module, return, switch, this, while
```

(3.3)

3.2.1 Reserved words

```
asm, auto, blueprint, debug, declaration, declare, decorator, delegate, do, except,
   finally, foreach, function, generic, goto, loop, scope, singleton, template,
   throw, try, with, yield
```

(3.4)

3.3 Literals

```
Literal      ::=  UnitLiteral [ Identifier ]
UnitLiteral  ::=  StringLiteral
                | IntLiteral
                | BinLiteral
                | OctLiteral
                | HexLiteral
                | FloatLiteral
```

(3.5)

3.3.1 Boolean 'literals'

NATI 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 a `Type` which 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

```
IntLiteral:     -?[0-9]+
BinLiteral:     -?0b[01]+
OctLiteral:     -?0o[0-7]+
HexLiteral:     -?0x[0-9a-fA-F]+
```

(3.6)

Data type Type of a literal is always the smallest signed type possible. If the number doesn't fit in the signed variant but fits in the unsigned variant with the same size, the unsigned variant is used (doesn't apply to negative literals). If the required bit count exceeds 128, a `BigInt` type is used.

```
1 10 // Int8
2 128 // UInt8
3 -10 // Int8
4 -150 // Int16
5 60000 // Int16
6 0xaa // UInt8
7 0xa01 // UInt16
```

(3.7)

3.3.4 Float literals

```
FloatLiteral:     -?[0-9]*\.[0-9]+([eE][+-]?[0-9]+)?
```

(3.8)

Data type Float literal is always of type `Float64`. It is recommended to use the `f` for casting to `Float32` (alternatively, you can use the explicit cast).

3.3.5 User-defined literals

NATI supports a syntactical construct that allows users to define their own literals. The user defined literal is then represented by a standard NATI literal followed by an identifier.

An appropriate function with prototype

```
1 @ctime auto #literal( String suffix, String value, Literal type );
```

(3.9)

is then called where the first argument is the *Identifier* used, the second is the literal value (stringified) and the third is the type of the literal.

```
enum Literal {
  stringLiteral,
  intLiteral,
  binLiteral,
  octLiteral,
  hexLiteral,
  floatLiteral;

@public:
  enum : Set {
    number = intLiteral | floatLiteral;
  }
}
```

(3.10)

```
@ctime Float32[ String ] lengthUnits = [
  "km": 1e3, "m": 1, "dm": 1e-1, "cm": 1e-2, "mm": 1e-3
];

@ctime Length #literal( String suffix, String value, Literal type )
  if( type is :number && suffix in lengthUnits.keys )
{
  return Length( value.to( Float32 ) * lengthUnits[ unit ] );
}

// Here, the #literal function is called with arguments #literal( "cm", "32.5",
   Literal.floatLiteral )
Length len = 32.5 cm;
```

(3.11)

TODO string literals, char literals

3.4 Comments

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

(3.12)

Additionally, **comments can be nested**.

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

(3.13)

4 | Expressions

```
Expression          ::=  AssignExpr
                         | Identifier '->' AssignExpr
CommaExpression     ::=  Expression { ',' Expression }

ParentCommaExpr     ::=  '(' [ PrntCommaExprItem { ',' PrntCommaExprItem } [ ',' ] ] ')'
PrntCommaExprItem   ::=  { Decoration } Expression [ '...' ]
```

(4.1)

4.1 Operators

```
AssignExpr          ::=  RangeExpr
                         | RangeExpr AssignOpOp RangeExpr
                         | RangeExpr '=>' RangeExpr
                         | RangeExpr { '=' RangeExpr }+
                         | RangeExpr { ':=' RangeExpr }+
RangeExpr           ::=  LogicExpr
                         | LogicExpr '..' LogicExpr
LogicExpr           ::=  CmpExpr
                         | CmpExpr { '&&' CmpExpr }+
                         | CmpExpr { '||' CmpExpr }+
CmpExpr             ::=  BitExpr
                         | BitExpr ( 'is' | '!is' | 'in' | '!in' | '!=' ) BitExpr
                         | BitExpr { ( '==' | '>=' | '>' ) BitExpr }+
                         | BitExpr { ( '==' | '<=' | '<' ) BitExpr }+
BitExpr             ::=  SumExpr
                         | UnaryExpr ( '>>' | '<<' | '>>>' | '<<<' ) UnaryExpr
                         | UnaryExpr { '&' UnaryExpr }+
                         | UnaryExpr { '|' UnaryExpr }+
                         | UnaryExpr { '^' UnaryExpr }+
SumExpr             ::=  MultExpr
                         | MultExpr { ( '+' | '-' ) MultExpr }+
MultExpr            ::=  VarDeclExpr
                         | VarDeclExpr { ( '*' | '/' ) VarDeclExpr }+
                         | VarDeclExpr '%' VarDeclExpr
VarDeclExpr         ::=  UnaryExpr
                         | UnaryExpr Identifier [ ParentCommaExpr ]
UnaryExpr           ::=  P1Expr
                         | ( '!' | '-' | '~' | '++' | '--' ) P1Expr
                         | P1Expr ( '++' | '--' )
                         | P1Expr { '?' | '!' }+
                         | InlineVarDecl
                         | '$'
P1Expr              ::=  AtomExpr
                         | AtomExpr { P1Op }
```

```
AtomExpr          ::=  [ ':' ] Identifier
                       | Literal
                       | '(' Expression ')'
                       | '[' [ ArrayLiteralExpr { ',' ArrayLiteralExpr } [ ',' ] ] ']'
ArrayLiteralExpr  ::=  RangeExpr [ '=>' RangeExpr ]
AssignOpOp        ::=  '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<='
                       | '>>=' | '<<<=' | '>>>='
P1Op              ::=  ParentCommaExpr
                       | '[' [ Expression { ',' Expression } [ ',' ] ] ']'
                       | '.' Identifier
```

(4.2)

4.1.1 Operator precedence

Priority	Operator	Enum	Semantics	Associativity
1	`x(args)`	`funcCall`	Function call	Left to right
	`x[args]`	`brackets`	Array subscripting	
	`x.ident`	`dispatch`	Attribute access	
2	`!x`	`preNot`	Logical NOT	Cannot be chained
	`-x`	`preMinus`	Sign negation	
	`~x`	`preTilde`	Bitwise NOT	
	`++x`	`preInc`	Prefix increment	
	`--x`	`preDec`	Prefix decrement	
	`x++`	`suffInc`	Suffix increment	Cannot be chained
	`x--`	`suffDec`	Suffix decrement	
	`x!`	`suffNot`	Mutable type	Left to right
	`x?`	`suffRef`	Reference type	
	`$`	`dollar`	Array size (can only be used inside the `brackets` operator)	
3	`x ident( args )`	(not overloadable)	In-expression variable declaration	Cannot be chained
4	`x * y`	`binMult(R)`	Multiplication	Left to right
	`x / y`	`binDiv(R)`	Division	
	`x % y`	`binMod(R)`	Modulo	Cannot be chained
5	`x + y`	`binPlus(R)`	Addition	Left to right
	`x - y`	`binMinus(R)`	Subtraction	
6 (cannot use P3,P4)	`x & y`	`binAnd(R)`	Bitwise AND	Left to right, only the same
	`x | y`	`binOr(R)`	Bitwise OR	
	`x ^ y`	`binXor(R)`	Bitwise XOR / power	
	`x >> y`	`binRSft(R)`	Bitwise right shift	Cannot be chained
	`x << y`	`binLSft(R)`	Bitwise left shift	
	`x >>> y`	`binRRot(R)`	Bitwise right circular shift	
	`x <<< y`	`binLRot(R)`	Bitwise left circular shift	
7	`x is y`	`binIs(R)`	Is (similar to equal to)	Cannot be chained
	`x !is y`	`binIsNot(R)`	Is not	
	`x in y`	`binIn(R)`	In	
	`x !in y`	`binNotIn(R)`	Not in	
	`x != y`	`binNeq(R)`	Not equal to	
	`x < y`	`binLess(R)`	Less than	Left to right, specific chaining rules
	`x <= y`	`binLeq(R)`	Less than or equal to	

	`x == y`	`binEq(R)`	Equal to	
	`x >= y`	`binGeq(R)`	Greater than or equal to	
	`x > y`	`binGrt(R)`	Less than	
8	`x && y`	`binLogAnd(R)`	Logical AND	Left to right,
	`x || y`	`binLogOr(R)`	Logical OR	only the same
9	`x .. y`	(not overloadable)	Range	Cannot be chained
10	`x = y`	`assign(R)`	Assignment	Right to left,
	`x := y`	`refAssign(R)`	Reference assignment	only the same
	`x += y`	`plusAssign(R)`	Various assignments	Cannot be chained
	`x -= y`	`minusAssign(R)`		
	`x *= y`	`multAssign(R)`		
	`x /= y`	`divAssign(R)`		
	`x %= y`	`modAssign(R)`		
	`x &= y`	`andAssign(R)`		
	`x |= y`	`orAssign(R)`		
	`x ^= y`	`xorAssign(R)`		
	`x <<= y`	`lSftAssign(R)`		
	`x >>= y`	`rSftAssign(R)`		
	`x <<<= y`	`lRotAssign(R)`		
	`x >>>= y`	`rRotAssign(R)`		
	`x => y`	(not overloadable)	Association	Cannot be chained
11	`x, y`		Comma	Left to right

(4.3)

Associativity – explanation

- **Left to right** Operations from the same group are processed from left to right, meaning `x + y + z` is processed as `(x + y) + z`.
- **Right to left** Operations from the same group are processed from right to left, meaning `x = y = z` is processed as `x = (y = z)`.
- **Cannot be chained** 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.
- **Only the same** Only the same operators can be chained, meaning `x & y & z` and `x | y | z` is correct, but `x & y | z` is not.

Operator precedence specialities In order to improve code readability, it is prohibited to use priority 3 and 4 operations as priority 5 operands – you must wrap them in the parentheses.

```
x + y >> z; // Error
(x + y) >> z; // Ok
```

(4.4)

Also, you cannot use P9 operations as operands in the `[args]` operator.

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;
```

```
7 // Syntax error
8 a > b < c;
```

(4.5)

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.

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

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  }
```

(4.6)

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 sees 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 Full identifier resolution

When there's an expression that starts with an identifier (`AtomExpr`, the *`Identifier`* alternative), a full resolution is performed for it. This is done by searching symbols with the desired identifier in various scopes. As soon as a match is found, the resolution ends, resulting an overloadset with all matching symbols in the currently searched set of scopes.

The rules that decide which scopes will be looked for are following:

Function scopes

1. Current scope
2. Parent scope (recursive); if the current scope is a nested function (lambda) root, all parent function scopes are skipped except when the lambda does capture all outer scope variables

Object (class, enum, ...) scopes Please note that object's dynamic and static fields are in separate scopes (depends if you're accessing an instance or an object). You can somewhat mix them using `@dynamicAccess`.

1. Current object scope and scopes of all ancestors of the current object (whether the member is static or not is not taken into consideration in this phase). Parent scopes are not searched recursively.
2. Scope imports of the current object and all its ancestors (scope imports are not implemented as a language feature in the current version, however some language-defined classes use them internally) and all ancestors of all the scope imports
3. Parent scope (recursive)

Module level scopes (modules, templates)

1. Current scope
2. Parent scope (recursive)

If no results are found, root scopes of all imported modules are searched. If nothing is found even then, identifier resolution ends with an empty result (or error, depending on use case).

4.2.2 Scoped 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 only be used 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 object the parameter is expected to be type of**.

```
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 }
```

(4.7)

This of course does not work with '**`auto`**' parameters (they are ignored in the overload resolution).

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 accept arguments (error is shown otherwise).

Note that not only functions accept arguments - for example parametric classes (similar to template classes) also accept arguments (they can however be considered to be compile-time functions returning a class as they are indistinguishable from them).

Overload resolution consists of these steps:

1. Overloads that do not match given arguments are ignored (also considering constness and staticness)
2. 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
3. Find minimal scope import nesting level in the overloadset; remove all overloads with higher nesting levels (overloads that are not from scope imports have higher priority)
4. If there are any non-static overloads, ignore all static overloads

5. If there are any non-compiler-generated overloads, ignore all compiler-generated overloads (implicit constructors etc)
6. If there are any non-const overloads, ignore all const overloads

If there's more than one overload remaining after application of the process mentioned above, the "ambiguous overloads"error occurs.

4.2.5 Resolution application on expressions

Application of the resolutions will be described in the next chapter for each operator.

4.3 Operator overloading

```
enum Operator {
  funcCall,
  brackets,
  dispatch,
  preNot,
  // etc
  ;

@public:
  enum : Set {
    unary = preNot | suffNot | preMinus /* etc */,
    unaryPre = preNot | preMinus | preTilde /* etc */,
    unarySuff = suffNot | suffMult | suffAnd /* etc */,

    binary = binMult | binDiv | binMod /* etc */,
    binaryRight = binMultR | binDivR | binModR /* etc */,
    anyAssign = assign | refAssign | plusAssign /* etc */
  }
}
```

(4.8)

All operators can be overloaded. This is done by overloading/implementing function:

```
auto #operator( @ctime Operator op, auto args ... );
```

(4.9)

So for example, you can overload all unary operators at once using:

```
@final class UnaryProxy( Type T ) {

@public:
  Void #ctor( T? value ) {
    referencedValue = value;
  }

@public:
  auto #operator( @ctime Operator op, auto args ... )
    if( op is :unary )
  {
    return referencedValue.#operator( op, args ... );
  }

@private:
  T! referencedValue;

}
```

(4.10)

or you can overload one exact operator:

```
Boolean #operator( Operator.preNot ) {
  return !referencedValue;
}
Int16 #operator( Operator("x++") ) {
  return referencedValue ++;
}
```

(4.11)

You can also make operator overload implementations virtual (if they don't contain any ctime variables):

```
class A {
  @public @abstract Void #operator( Operator.plusAssign, A@ other );
}

@final class B : @public A {
  @public @override Void #operator( Operator.plusAssign, A@ other ) {
    // code here
  }
}
```

(4.12)

4.3.1 The '`x.ident`' (dispatch) operator

Consider expression

```
x.ident
```

(4.13)

Here, `x` is the left operand and the `ident` is the right operand.

Behavior

1. Check if the left operand overloadset contains only one record (or error).
2. Perform a scoped identifier resolution for the left operand scope, looking for the `ident`; return the results if any.
3. If the resolution resulted an empty overloadset, try calling (left operand scope identifier resolution + overload resolution + call)
   `instance.#operator( Operator.dispatch, "ident" )`
4. If it fails, show an error

Consequences The `x.ident` operator can result with multiple overloads. You can implement a dispatch function.

```
class Vector( UInt16 dimensions ) {

@public:
  Float32! data[ dimensions ];

  (...)

  auto #operator( Operator.dispatch, @ctime String str )
    if( str.allMatch( x => "xyz".contains( x ) ) )
```

```
10    {
11      @ctime Index[ Char ] translationArray = [
12        'x' => 0,
13        'y' => 1,
14        'z' => 2
15      ];
16
17      Vector( str.length )  result;
18      foreach( Index i, Char ch; str )
19        result.data[ i ] = data[ translationArray[ ch ] ];
20
21      return result;
22    }
23
24
25  }
26
27  Void main() {
28    Vector( 3 ) v1( 1, 2, 3 );
29    auto v2 = v1.xxx; // v2 = ( 1, 1, 1 )
30    auto v3 = v1.zyx; // v3 = ( 3, 2, 1 )
31  }
```

(4.14)

TODO Exceptions used for overload filtering

4.3.2 The 'x(args)' (funcCall) operator

Consider expression

```
1  a.f( arg1, arg2 )
```

(4.15)

Here, `a.f` is the left operand and `arg1` and `arg2` are the parameter operands.

Behavior Behavior of this operator varies on the properties of the overloadset.

If the overload set contains only functions:

1. Perform an overload resolution with parameters `( arg1, arg2 )`.
2. Call the only resulting overload (or error).

If not:

1. Check if the left operand overloadset contains only one record (or error).
2. Resolve and call `a.f.#operator( Operator.funcCall, arg1, arg2 )`.

Consequences Behavior described above makes the function overloading work.

Inline variable declarations You can use variable declaration as a parameter. The declared variable is valid from the point declaration to end of current scope.

```
1  {
2    JsonDocument doc = JSON.parse( stringData, Bool isOkay );
3    if( !isOkay )
4      throw;
5  }
```

(4.16)

The 'x ...' operator is used for 'expading' a value across multiple parameter slots. It does so using standard iterating process performed at ctime.

```
1 Void f( Int a, Int b, Int c ) { /* ... */ }
2 Void g( String x, Int a ) { /* ... */ }
3
4 Void main() {
5   f( [ 1, 2, 3 ] ... ); // Valid, the array is iterated at compile time and each
      value is passed as a separate parameter
6   g( tuple( "5", 12 ) ... ); // Valid
7 }
```

(4.17)

4.3.3 The 'x[args]' (brackets) operator

Consider expression

```
1   x[ arg1, arg2 ]
```

(4.18)

Here, the x is the left operand and `arg1` and `arg2` are the parameters.

Behavior

1. Check if the left operand overloadset contains only one record (or error).
2. Resolve and call `x.#operator( Operator.brackets, arg1, arg2 )` (or error).

4.3.4 The '$' (dollar) operator

In the bracket parameters, you can use the $ operator.

It doesn't take any operands and translates to `x.#operator( Operator.dollar, Index paramId )` where `paramId` is the index of the parameter the dollar operator was used in (starting from 0).

4.3.5 Unary operators

Consider expression

```
1   ++x
```

(4.19)

Here, x is the operand and ++ is the operator.

Behavior

1. Check if the operand overloadset contains only one record (or error).
2. Resolve and call `x.#operator( op )` (or error).

4.3.6 Binary (and assign) operators

Consider expression

```
1   x + y
```

(4.20)

Here, the x is the left operand, y is the right operand and + is the operator.

Behavior

1. Check if the operand overloadset contains only one record (or error).
2. Resolve and call `x.#operator( op, y )`.
3. If previous step did not found any matching overload, resolve and call `y.#operator( opR, x )` (or error).

Every binary operator has two records in the `Operator` enum – one that ends with the `R` and one that does not. The `R` here means right. For the expression shown above, the compiler would call `x.#operator( Operator.binPlus, y )`, eventually `y.#operator( Operator.binPlusR, x )`.

4.3.7 The 'x .. y' (range) operator

This operator is almost equivalent to `Range( x.#type )( x, y )`, except it infers the Range type from expected parameter type if possible.

4.3.8 The 'x => y' (association) operator

This operator is almost equivalent to `Association( x.#type, y.#type )( x, y )`, except it infers the Association types from expected parameter type if possible.

4.3.9 The 'x, y' (comma) operator

Operands separated by the comma are evaluated from left to right. Value of the rightmost operand is returned.

4.4 Variable declarations

```
VarDeclCmd   ::=  TypeExpr Identifier [ VarDeclCtor ]
                       { ',' Identifier [ VarDeclCtor ] } ';'

TypeExpr     ::=  { Decoration } ( UnaryExpr | AutoType )
AutoType     ::=  'auto' [ '?' ] [ '!' ]
VarDeclCtor  ::=  ParentCommaExpr | ( '=' | ':=' ) Expression
```

(4.21)

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 three syntactical alternatives of how to declare a variable:

- `Type var;` results in calling the implicit constructor `var.#ctor();` if presents, yield an error otherwise.
- `Type var( arg1, arg2 );` obviously calls `var.#ctor( arg1, arg2 );`
- `Type var = val;` calls `var.#ctor( val );`
- `Type var := val;` calls `var.#ctor( Operator.refAssign, val );`

Auto declarations When using **`auto`** instead of type, the type is inferred from the first argument of `ParentCommaExpr` (which must be the only one) or from the value type of the `Expression` after the '='. Multiple declarations per one auto keyword are acceptable, each inferring its own type.

In-expression variable declarations In NATI, you can declare variables inside expressions.

4.4.1 Variable lifetime

When variable's lifetime ends, it's implicit destructor `var.#dtor();` is called.

Local variables Local variables are destroyed as soon as their scope ends. Although this is straightforward in most cases, there are some that require further explanations:

- Variables declared inside **if** conditions are destroyed after the end of the if statement, including following else statements.

```
1 if( Int i( getNumber() ) == 5 ) {
2
3 } else if( i == 4 ) {
4
5 } // i is destroyed here
```

(4.22)

- Variables declared inside **where** conditions are destroyed at the end of every iteration as they are declared at the beginning of every iteration.

```
1 while( ( Int! x = 0 )++ < 32 ) {
2   // This is WRONG! x is created every iteration, so its value for each
       iteration is 1. This loop would not end.
3 }
4
5 foreach( x; 0 .. 32 ) {
6   // This loop is correct and much nicer
7 }
```

(4.23)

- For variables declared inside **for**:
  - Variables in the initialization part are destroyed after the loop end
  - Variables in the condition part are created before each iteration and destroyed after each iteration
  - Variables in the post-loop part are destroyed just after the post-loop part is executed
- For variables inside **foreach**:
  - Variables in the variable declaration part are created before each iteration and destroyed after each iteration
  - Variables in the iterated expression part are destroyed when the foreach ends.
- Variables inside the **switch** expression part are destroyed when the switch ends.

Static variables are destroyed when the program ends in the reverse order they were constructed.

Static `@ctime` variables are never destroyed (they were created before the program started).

5 | Functions

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

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

(5.1)

5.1 Variadic functions

5.2 The '`auto`' keyword

5.3 Parameter namespace accessor (`:ident`)

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.nati.main` has to be in file `project/src/straw/nati/main.nati` or `project/include/straw/nati/main.nati`.

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
```

```
Void aFunc() {
  // Do something
}

/* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
module b;

import a; /* Equivalent with @local import a; */

Void bFunc() {
  aFunc(); // Ok
}

/* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
module c;

import b;

Void cFunc() {
  import a;
  aFunc(); // Ok
}
Void cFunc2() {
  aFunc(); // Error
}
```

(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).

```
module a;

Void aFunc() {
  // Do something
}

/* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
module b;

@global import a;

Void bFunc() {
  aFunc(); // Ok
}

/* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
module c;

@global import b;

Void cFunc() {
  bFunc(); // Ok
  aFunc(); // Ok
}

/* - - - - - - - - DIFFERENT MODULE - - - - - - - - */
module d;

import c;

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

TODO: noWarning, final, abstract, base, virtual, override, static

```
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

In 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 NATI, 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
  functionModifier, // system only
  classModifier, // system only
  enumModifier, // system only
  variableModifier, // system only
  accessModifier,
  fieldWrapper,
  typeWrapper,
  controlStatementModifier, // system only
  codeBlockModifier // system only
}
```

(8.3)

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 always considered 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 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. In some cases, parameters don't have to be decorated with `@ctime` – class, enum and template parameters are always compile-time, so the `@ctime` is implicit and decorating them with it results in warning.

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
        statement
5
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
```

```
}

@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 convenient.

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. For example, container types should be decorated with `@autoCtime` so they can store compile-time types.

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, Type Value ) {

@public:
  // 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)
  T!?! #operator!( Operator("x[args]"), @autoCtime Key key ) {
    Index hash = key.#hash() % tableSize_;

    // Locating the record here
  }

}

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

  table[ stdin.read() ] = 5;

  table[ "key" ] = 10;
  // 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)
}
```

(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.

9 | Ctime

Ctime is a powerful concept introduced in NATI. 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 NATI.

Consequences Consequences of this one simple rule are following:

10 | NATI 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`).

10.1 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.

11 | 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

11.1 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

11.2 Random thoughts scrapbook

```
auto max( auto x, x.#type y ) = ( x > y ) ? x : y;

T!? #new( @ctime Type T ) {
  return malloc( T.#size ).to( T!@ );
}

auto Int = Int32;
auto Float = Float32;

auto I1 = Int8;
auto I2 = Int16;
auto I4 = Int32;
auto I8 = Int64;
auto I16 = Int128;

Int8.#identifier == "Int8"
I8.#identifier == "I8"
I8.#name = "Int8"

decorator.ctime or decorator( ctime )
module.straw.nati.test or module( straw.nati.test )

class null {
  @public @static T #implicitCast( Type T )
    if( T is Pointer )
  {
    module.this
    module.straw.nati.functionInAnotherModule();
    module( "asd.test" ).functionInAnotherModule();
    return T(0);
```

```
31|   }
32| }
33|
34| class Void {}
35|
36|
37| // X class
38| class X( Int32 i ) {
39| }
40|
41| X(5).#type == Class
42| X(5).#template == X
43| X.#type == ClassTemplate
44| X.#identifier == "X"
45| X(5).#name == "X"
46| X(5).#identifier == null
47|
48| // Overloadsets
49| I8
```

(11.1)

11.2.1 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	∞	*	*		*	*	*	*

(11.2)

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