# Article structure

An article is a piece of text, which may optionally contain code starting with #code# and ending with #end#.

In each code segment between #code# and #end#, the following must appear sequentially:

* Any number of imports
* Any number of template, class or variable declarations
* Zero or one expression (if the expression is omitted, the program returns null)

Everything outside code segment is directly turned into PlainText, where the result of a code segment is converted to Text by implicitly calling text in the standard library. If text cannot be called on the result, an error occurs. The pieces are join together into a TextSequence for processing by the page generator.

## Imports

Only one symbol can be imported at a time. The syntax is

* @import some\_library.function
* @import "another article".variable new\_name

If new\_name is omitted, the same name is used in the current scope. The results of evaluating other libraries / articles are cached, i.e. importing multiple symbols from a single external article does not cause huge performance penalty.

## Declarations

### Variable scoping

All names have their associated scope. The outermost scope is the article. All program segments between #code# and #end# share the same scope, allowing the user to declare something and use it later in the same article, with passed-through text between.

Function parameters are of its own scope. That means function parameters can hide variables with the same name outside, and the function body can hide function parameters.

The only language constructs with their own scoping is variable declarations and function literals, where declarations are allowed inside.

Function parameters cannot have identical names, for example, the following is an error:

@function (x : @Integer, x : @Integer) {}

However, the following is not an error:

@function (x : @Integer) { @function (x : @Integer) {} }

In this example, x in the inner function hides x in the outer function.

### Class declarations

A class must be declared before use.

A class declaration must contain its name and the list of members, including the member’s name, type and optionally a default value. It may also include a superclass. If template parameters are specified, the declaration becomes a class template declaration.

The members are declared similar to ordinary variables, but the initialiser may be omitted, in this case, a value must be supplied at the point of construction.

The class name must be unique in its scope. The member names must be unique in the class and all its ancestors. No overloading is allowed for class members.

The types of members must be specified explicitly.

### Variable declarations

A variable must be declared before use.

A variable declaration must contain its name and its value, and may optionally contain its type. If the variable is a function, the parameters may also be specified, or alternately use a function literal for that. If the type is not specified, it is automatically deduced from its value, otherwise, its value must be compatible to the declared type.

#### Function overloading

This language supports very simple function overloading, which the rules are listed below:

* Only proper functions (i.e. variables with function types) can be overloaded. Classes, templates (including function templates), primitives, and arrays cannot be overloaded.
* Overload resolution is only based on the first parameter. A function can only be overloaded if and only if the type of its first parameter is not compatible with any other functions with the same name, and vice versa.
* Only functions within the same scope are overloaded. Functions of the same name within enclosing scopes are hidden.

#### Recursion

The return type of a recursive function must be specified, otherwise, the function itself cannot be resolved in its body due to the lack of type information.

Mutual recursion is not yet supported.

### Template declarations

If a template parameter list is given on a class or variable declaration, it becomes a template. A template is not evaluated until the point of template application, where the type parameters are substituted by their arguments and evaluated to a proper class or variable.

## Expressions

An expression has its type and value. There are different kinds of expression, listed below from highest precedence to the lowest:

### Parenthesized expressions

A pair of parentheses () can be used around any expression to override its precedence.

### Literals

There are three kinds of primitive literals and four kinds of non-primitive literals:

#### Integer literals

The valid range of integers are [-2^31, 2^31 - 1]. Denary and hexadecimal integers are supported.

#### Floating-point literals

For a number to be considered a floating point literal, it must either contain a dot or the exponent.

#### Character literals

A single character or escape sequence representing a single character, enclosed in a pair of single quote, is accepted as a character literal.

The list of escape sequences are follow:

* \0 : null character
* \b : backspace
* \f : form feed
* \n : new line
* \r : carriage return
* \t : tab
* \v : vertical tab
* \\ : a literal backslash
* \' : a single quote
* \" : a double quote
* \123 (three octal digits) : Unicode code point
* \xEE (x followed by two hexadecimal digits) : Unicode code point
* \uFEFF (u followed by four hexadecimal digits) : Unicode code point
* \u{10ffff} (u followed by brackets containing any number of hexadecimal digits) : Unicode code point

#### String literals

A string is a sequence of characters in double quotes. The type of a string literal is [@Character] (a character array).

#### Array literals

An array literal is of the following form:

[1, 2, 3, 4, 5]

Trailing comma is accepted but no leading comma or two consecutive commas are accepted.

All array elements must be of compatible types. An empty array literal is compatible with array types of any element type.

#### Function literals

A function literal is of the following form:

@function (param1 : Type, param2 : Type) : ReturnType { @import blah.xxx; @var x = 3; blah x }

where the only mandatory item is the keyword and the pair of braces.

A function literal must contain a function body, which may, like the program itself, contain imports and declarations local to the function, and must contain zero or one expression. It may also include the list of parameters and return type. If the return type is specified, the body must be compatible to it, otherwise, the return type is deduced automatically.

A function body is a scope. Anything declared inside the function body does not exist outside it.

A function literal is a closure. The symbols in scope where the function literal resides are packaged inside the function value, and can be safely be assigned to variables, passed as arguments and returned from functions.

#### Object literals

An object literal is of the following form:

Type { member = 3; another = 4; }

An object literal is a class followed by the list of member assignments. All members without a default value must be given a user-supplied value.

### Identifiers

A valid identifier is specified by the regular expression [\_a-zA-Z\xA0-\uFFFF][\_a-zA-Z0-9\xA0-\uFFFF]\*, and also operators prefixed by keyword @operator (to specify unary + and -, use @operator++ and @operator-- respectively)

### Native call

@native(name : type) specifies a native call. The name refers to the key in the natives module object, while the type specifies the return type of the native function.

To add a native function into the installation, add it into object natives in natives.js

**Warning: little type checking is done on the native function. It is the library’s responsibility to correctly specify the type. Misusing it can corrupt the interpreter.**

The native function receives two arguments: the symbol table and the location.

The symbol table is an object of key-array pairs. The key is the identifier name, and each element of the array represents an overload.

For each element representing a value (object of type Value), there are two members, type and value. The type has a member compatibleWith which can see if the object is compatible with some type.

The native function only needs to return the value, with the type automatically added by the interpreter.

The relationship between the language types and the Javascript type of the member value is follows:

|  |  |
| --- | --- |
| @Integer | number |
| @Float | number |
| @Character | string (containing a single Unicode code-point) |
| @Boolean | boolean |
| @Null | null |
| array | Array |
| class | object (containing key-value pairs of its members) |
| function | An object with the following members:  parameter : an object containing its name and type  body : the AST tree of the function body  symbols : the closure symbols |

The location is an object representing where the interpreter is at, containing elements “start” and “end”, each containing name (the article where it comes from), line, column (1-based) and offset (0-based).

### Error

@error("message") causes an interpreter error when evaluated. It can be used anywhere where an expression is expected. Currently, only a string literal is supported as the message.

### Instance check

@instance(object, type) returns boolean true if the dynamic type of the object is compatible with type, false otherwise. It can be used prior to downcasting.

### Down casting

@cast(object, type, default) returns object if the dynamic type of the object is compatible with type, default otherwise. Putting an @error as the default will result in an error only when object is not compatible with type.

The declared type of object can be any class type, but the declared type of default must be compatible with type.

### If

@if(condition, true\_branch, false\_branch) is the primary mean of branching (the only other mean of branching is by down casting). Condition must be a boolean and the types of branches must be compatible.

Known issue: if the branches are subclasses of the same parent, type checking fails because they are not mutually compatible to each other. This may be worked around by wrapping one branch in a function literal declaring to return the parent class.

### Template application

template<type> applies type to template to get the actual class or variable. The same syntax can also be used to select among overloaded functions by specifying the argument type without calling it.

### Member access

object.member access member in object. member must exists in the declared type (rather than actual type) of object.

### Member update

object { member = 3; another = 4; } creates a new instance of object’s class by changing the specified members.

### Function call

function argument calls function with argument. Internally this language only supports functions with a single argument. The presence of multiple arguments in function declarations is only syntactic sugar, they are converted into curried single-argument functions internally.

#### Operators

The language supports the following operators, from highest to lowest precedence, with the functionally from the standard library specified below:

|  |  |  |  |
| --- | --- | --- | --- |
| Level | Operators | Description | Default supported primitives |
| 1 | +  - | Unary plus  Unary minus (Negation) | Integer, Float |
|  | ~ | Not | Integer, Boolean |
| 2 | \* | Multiplication | Integer, Float |
|  | / | Division | Float |
|  | % | Modulus | Integer, Float |
| 3 | +  - | Addition  Subtraction | Integer, Float |
| 4 | <<  >>  >>> | Shift left  Shift right with sign  Shift right without sign | Integer |
| 5 | <  <=  >  >= | Comparisons | Integer, Float, Character |
| 6 | ==  != | Equal  Not equal | Integer, Float, Boolean, Character, Null |
| 7 | & | And | Integer, Boolean |
| 8 | ^ | Xor | Integer, Boolean |
| 9 | | | Or | Integer, Boolean |

All operators are actually functions, and can be specified by name by @operator& (where & is the operator). However, to specify unary + and -, @operator++ and @operator-- is needed to prevent confusion with its binary counterparts.

# Type system

The types in the language can be divided to:

## Native types

These are the primitives, and can be specified by keywords @Integer, @Character, @Float, @Boolean and @Null.

### Integer

A signed 32-bit integer.

### Character

A single Unicode code point.

### Float

An IEEE 754 64-bit floating point number

### Boolean

true or false. The values are specified by using standard library functions true or false, which accept 0 arguments and returns the boolean value.

### Null

null is the only possible value of @Null. null is a standard library function, accepting no arguments and return the only value of @Null.

## Array

An array type is specified by [T], where T is the type of the elements.

## Function

A function type is specified by X Y, where X is the argument type and Y is the return type.

This language only supports functions of a single argument. Multiple arguments are converted to curried functions internally.

## Class

An object of a class type is a collection of its superclass and members, and can be specified by the name of class.

## Type compatibility

Type compatibility is an important concept in the language. A value can only be assigned to variables, passed as parameters or returned from function if its static type is compatible with the declaration.

Types of different kinds are not compatible.

Native types are compatible only if they are exactly the same.

A value of array type [A] can be assigned to array type [B] if and only if A can be assigned to B.

The compatibility of classes is determined by their name normally. However, if any of the classes compared are the result of a class template, the structure is compared instead. A class A can be assigned to declared class type B if and only if:

* If A and B are both named normal class, A is the same as B; AND
* If any of A and B are classes resulting from template, the superclass of A is compatible with the superclass of B AND all members of A is compatible with the respective member of the same name in B. A can have extra members or superclass not exist in B, but not any missing members or superclass from B, OR
* A has a superclass and the superclass of A is compatible with B itself (i.e. upcasting)

A function value of type P Q can be assigned to function type R S if and only if:

* Q can be assigned to S (return type), AND
* R can be assigned to P (argument)

## Full syntax (PEG)

head = ('#code#' program '#end#' / passthrough)\*

passthrough = $(!'#code#' .)

program = \_ import\* declaration\* expression?

import = "@import" \_ source '.' \_ identifier identifier? ';' \_

source = identifier / string\_literal

declaration = class\_declaration / variable\_declaration

class\_declaration = "@class" \_ ( "<" \_ template\_parameter\_list ">" \_ )?

identifier (":" \_ type)? "{" \_ member\_declaration\* "}" \_

variable\_declaration = "@var" \_ ( "<" \_ template\_parameter\_list ">" \_ )?

identifier ("(" \_ parameter\_list ")" \_ )? (":" \_ type)? initialiser

member\_declaration = "@var" \_ identifier ("(" \_ parameter\_list ")" \_ )? ":" \_ type (initialiser / (";" \_ ))

initialiser = '=' \_ expression ';' \_

/ '{' \_ program '}' \_

template\_parameter\_list = identifier ("," \_ identifier)\*

template\_arguments\_list = type ("," \_ type)\*

parameter\_list = parameter ("," \_ parameter)\* / ""

parameter = identifier ":" \_ type

expression = bor\_expression

bor\_expression = xor\_expression ('|' \_ xor\_expression)\*

xor\_expression = band\_expression ('^' \_ band\_expression)\*

band\_expression = eq\_expression ('&' \_ eq\_expression)\*

eq\_expression = comp\_expression (("==" / "!=") \_ comp\_expression)\*

comp\_expression = shift\_expression (("<" / "<=" / ">=" / ">") \_ shift\_expression)\*

shift\_expression = add\_expression (("<<" / ">>>" / ">>") \_ add\_expression)\*

add\_expression = mult\_expression (("+" / "-") \_ mult\_expression)\*

mult\_expression = unary\_expression (("\*" / "/" / "%") \_ unary\_expression)\*

unary\_expression = postfix\_expression / ("+" / "-" / "~") \_ unary\_expression

postfix\_expression = member\_expression member\_expression\*

member\_update = '{' \_ '}' \_ / '{' \_ member\_assignment ( (','/';') \_ member\_assignment )\* ((','/';') \_)? '}' \_

member\_assignment = identifier '=' \_ expression \_

member\_expression = atom ( '.' \_ identifier / member\_update)\*

template\_application = identifier '<' \_ template\_arguments\_list '>' \_

atom = literal

/ template\_application

/ identifier

/ '(' \_ expression ')' \_

/ native\_expression

/ if\_expression

/ instance\_expression

/ cast\_expression

/ error\_expression

if\_expression = "@if" \_ '(' \_ expression ',' \_ expression ',' \_ expression ')' \_

instance\_expression = "@instance" \_ '(' \_ expression ',' \_ type ')' \_

cast\_expression = "@cast" \_ '(' \_ expression ',' \_ type ',' \_ expression')' \_

error\_expression = "@error" \_ '(' \_ string\_literal ')' \_

built\_in\_type = ("@Character" / "@Integer" / "@Float" / "@Boolean" / "@Null") \_

native\_expression = '@native' \_ '(' \_ identifier ':' \_ type ')' \_

identifier = $ ([\_a-zA-Z\xA0-\uFFFF][\_a-zA-Z0-9\xA0-\uFFFF]\*) \_ / '@operator' \_ ( '++' / '--' / '+' / '-' / '\*' / '/' / '%' / '&' / '^' / '|' / '~' / '<<' / '>>>' / '>>' / '==' / '!=' / '<=' / '<' / '>=' / '>') \_

type = ( built\_in\_type / '[' \_ type ']' \_ / '(' \_ type ')' \_ / identifier )+

literal = string\_literal / char\_literal / float\_literal / int\_literal / array\_literal / function\_literal

char\_literal = "'" (char / '"') "'" \_

string\_literal = "\"" (char / "'")\* "\"" \_

float\_literal = ( ([0-9]+ "." [0-9]\* / "." [0-9]+) ([eE] [+-]? [0-9]+)? / [0-9]+\_ [eE] [+-]? [0-9]+) \_

int\_literal = ("0x"[0-9a-fA-F]+ / [1-9][0-9]\* / '0' ) \_

array\_literal = '[' \_ (expression ',' \_ )\* ']' \_ / '[' \_ (expression ',' \_ )\* expression ']' \_

function\_literal = '@function' \_ ('(' \_ parameter\_list ')'\_ )? (':' \_ type)? '{' \_ program '}' \_

char = $ ((! ('\\' / '"' / "'")) .) / "\\" ( [bfnrtv\\'"] / ([0-7] / [0-7][0-7] / [0-3][0-7][0-7]) / 'x' ([0-9a-fA-F][0-9a-fA-F]) / 'u' ([0-9a-fA-F][0-9a-fA-F][0-9a-fA-F][0-9a-fA-F]) / 'u{' [0-9a-fA-F]\* '}' )

\_ = (comment / whitespace)\*

whitespace = [ \n\t\r\v]

comment = '/\*' comment\_tail / '//' [^\n]\* ('\n' / EOF)

comment\_tail = '\*/' / . comment\_tail

EOF = !.