Edo's extended Jass and Edo's extended Jass Compiler

Standard document

Eduard Láhl

Table of Contents

Language Specification

1. Basics

1.1 eJass Construct

eJass Construct is everything that can be specified in eJass syntax.

Example:

```
function a takes nothing returns nothing
      ...
endfunction

struct A
      ...
endstruct
```

All names are case sensitive.

1.2 Keyword

Keyword is every sequence of characters that is reserved for implementation. Some keywords can be context-sensitive, other can be reserved for future use, rather than being currently used.

1.2.1 Keyword List

The following is the list of keywords reserved to the compiler:

#else	encrypted	false	private
#elseif	endallocator	final*	public
#endif	endblock	for	readonly
#if	endconstructor	function	requires
after*	enddestructor	globals	return
alias	endextendor	hook	returns
allocator	endexternal	if	runtextmacro
and	endexternalblock	implement	scope
array	endfor	import	set
auto	endfunction	initializer	sizeof
before*	endglobals	inline*	static
break	endif	interface	static_assert
call	endinterface	library	struct
cast	endlibrary	local	stub
catch	endloop	loop	takes
compiletime	endmethod	method	template
constant	endmodule	module	temporary
construct	endnamespace	namespace	textmacro
constructor	endscope	native	then
continue	endstruct	needs	this
debug	endtextmacro	not	thistype

```
defaults          endwhile          null              true
deprecated*       exitwhen          operator          type
destruct          extendor          optional          uses
destructor        extends           or                using
else              external          override          while
elseif            externalblock     priority
```

* Context sensitive keywords.

1.3 Scope

Scope is everything that directly splits code to sections.

Example:

```
library L
	//Scope L
	scope Q
		//Scope L.Q
		function f takes nothing returns nothing
			//Scope L.Q.f
			if true then
				//Scope L.Q.f.if_1
			endif
		endfunction
		//Scope L.Q
	endscope
	//Scope L
endlibrary
//Scope Global
function a takes nothing returns nothing
	//Scope a(same as Scope Global.a)
endfunction
```

Special Scope is the Global Scope, which contains everything in given Map Script.

1 Defining Scope

Defining Scope is the Scope that defines given eJass construct. If, for instance, function defines local variable, then the defining Scope is the function defining the local variable.

2 External Scope

External Scope is every Scope other than Scope of given context(for instance, Defining Scope).

1.3.1 Global Scope

Global Scope is the outer-most Scope of the map script, which is not inside any other Scope and all other scopes are inside that Scope. All name-referable eJass constructs are reachable from Global Scope.

1.3.1.1 Modifier Global

Special modifier Global can be used as part of name of name-referable eJass construct to specify that the Scope resolution(See 1.3.2 - Scope resolution) happens from the Global Scope.

1.3.2 Scope resolution

Scope resolution in eJass is performed by using the dot syntax for Scope names.

1.3.2.1 Scope resolution rules

When looking for names on name-referable eJass constructs, the compiler will take these rules into consideration:

1. Global Scope is visible and accessible from any other Scope
2. For any given Scope all its parent Scopes are implicitly declared and considered in Scope resolution and their parts of full path for given eJass construct can be omitted
3. All other Scopes are External for given point of map script, and to be usable, they must be accessed by their full path

Example:

```
library A
    function f takes nothing returns nothing
    endfunction

    scope S
        function q takes nothing returns nothing
            call f()            //same as call A.f() or Global.A.f()
        endfunction
      endscope

      function w takes nothing returns nothing
        call q()                //compilation error: q is undeclared from
                                //this scope
        call S.q()              //fine, will call A.S.q
    endfunction
endlibrary
```

1.3.2.2 Keyword using

To explicitly declare External Scope as implicit, the keyword using can be used inside any scope to declare that scope as implicit for Scope resolution. Keyword using can only be used on Scopes that are created by scope or library keywords. If using is used and the Scope using is tried to make implicit is not found from that point in map script, a error is issued.

Example:

```
library A
    using S        //compilation error: No scope S found

    scope S
```

```
            function f takes nothing returns nothing
            endfunction

            scope Q
                function w takes nothing returns nothing
                endfunction
            endscope
        endscope

        using S
        using S.Q

        function q takes nothing returns nothing
            call f()
            call w()
        endfunction
    endlibrary
```

1.3.2.2.1 White spaces

```
using scope_name
```

1. keyword using and scope_name are bound tightly into each other

1.3.3 Shadowing rules

If there exists name-referable eJass construct called g inside Scope S, and this Scope contains another Scope W which also contains name-referable eJass construct called g, the Scope W will shadow name-referable eJass construct g from beginning of g's definition until the end of scope W. (Case 1)
If there exists name-referable eJass construct called g inside Scope S, and this Scope contains another Scope W which also contains name-referable eJass construct called g, and there exists Scope Q which has both Scope S and Scope W in its list of implicit Scopes, Scope Q will have ambiguous reference to g and error will be issued. (Case 2)

Example:

```
    scope S
        function f takes nothing returns nothing
        endfunction

        scope Q
            function f takes nothing returns nothing
                call f()        //will call S.Q.f
                                //Case 1
            endfunction

            function w takes nothing returns nothing
                call f()        //will call S.Q.f
                                //Case 1
            endfunction
        endscope
```

```
    function w takes nothing returns nothing
        call f()            //will call S.f
                            //Case 1
    endfunction
endscope

scope W
    using S
    using Q        //same as using S.Q at this point

    //Error: function f() defined multiple times


    function r takes nothing returns nothing
        call f()            //Error: Could not resolve call to f()
                            //Could be: S.f, S.Q.f
                            //Case 2
    endfunction

    function f takes nothing returns nothing
                            //Error: function f() defined multiple times
    endfunction
endscope
```

1.3.4 Bracket-scoping

Brackets can also create pseudo scopes for certain Syntactic Units(see 1.12 - Syntactic Unit). The lowest bracket-scope is the lowest, and the more opening brackets are encountered, the higher bracket-scope is.

Example:
`A(B((C(D)E)F)G)Q`

A is in bracket-scope 0
B is in bracket-scope 1
C is in bracket-scope 3
D is in bracket-scope 4
E is in bracket-scope 3
F is in bracket-scope 2
G is in bracket-scope 1
Q is in bracket-scope 0

1.4 Qualifiers

Qualifiers are eJass Constructs that can be placed before any other eJass Constructs.

1.4.1 Visibility-Qualifier

Visibility-Qualifier distinguishes whether given eJass Construct is visible to the whole script, or if only the local scope can access and use it. Abbreviation: Vis-Qualifier.

Valid Vis-Qualifiers:

```
public
private
--none--
```

In case Visibility-Qualifier is required or optional, and none is provided, the eJass Construct is automatically marked with Public Visibility-Qualifier.

1.4.2 Staticity-Qualifier

Staticity-Qualifier determinates whether given eJass Construct is usable only for one instance of struct/interface, or if it is usable by all instances. Abbreviation: Static-Qualifier.

Valid Static-Qualifiers:

```
static
--none--
```

In case Staticity-Qualifier is required or optional, and none is provided, the eJass Construct is automatically marked with Nonstatic Staticity-Qualifier.

1.4.3 Readonly-Qualifier

Readonly-Qualifier determinates whether given eJass Construct is only readable from scopes other than defining scope, or if it is possible to also write to such eJass Construct from external scopes.

Valid Readonly-Qualifiers:

```
readonly
--none--
```

In case Readonly-Qualifier is required or optional, and none is provided, the eJass Construct is automatically marked with nonreadonly Readonly-Qualifier.

1.4.4 Const-Qualifier

Const-Qualifier determinates whether given eJass Construct is constant(unmodifiable), or if it can be modified after definition or declaration. Const-Qualifier before function definition have special meaning.

Valid Const-Qualifiers:

```
constant
--none--
```

In case Const-Qualifier is required or optional, and none is provided, the eJass Construct is automatically marked with nonconstant Const-Qualifier.

1.4.5 Comp-Qualifier

Comp-Qualifier determinates whether given eJass construct can be marked as compiletime, meaning that it can be executed by interpreter during compilation of the map script.

Valid Comp-Qualifiers:

```
compiletime
--none--
```

In case Comp-Qualifier is required or optional, but none is provided, the eJass Construct is automatically marked with non-compiletime Comp-Qualifier.

1.5 Literals

Literals are raw values for basic in-built types(See 2.2 - Basic Types).

1.5.1 Integer literal

1.5.1.1 Decimal literals

Decimal Integer literal describes given integer in decimal format. Decimal literals must never start with 0. Valid characters, that can occur in decimal representation are 0123456789.

Exmple:

```
54677
```

1.5.1.2 Octal literals

Octal Integer literal describes given integer in octal format. Octal Integer literals must always start with 0. Valid characters, that can occur in octal representation are 01234567.

Example:

```
055
071
```

1.5.1.3 Hexadecimal literals

Hexadecimal Integer literal describes given integer in hexadecimal format. Hexadecimal Integer literals must always start with 0x, or 0X. Valid characters, that can occur in hexadecimal representation are 0123456789abcdefABCDEF.

Example:

```
0x77
0X7A
```

```
0x7a
0X7A
```

1.5.1.4 256-base literals

256-base literal describes given integer in special, base 256 format. 256-base integer literals can be represented by either 1, or 4 characters, enclosed in pair of ampersands(').

1.5.1.4.1 25-base One character literals

One character literals can be represented by the following characters:
0123456789`~!@#$%^&*()_+-=[]{};:"|,\./<>?abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
Additionally, One character literals can also be represented by '\n' or '\t' as well as physical tab or space.(\t == "tab"), as well as '\\'.

Example:

```
'    '
'5'
```

1.5.1.4.2 Four character literals

Four character literals can be represented by the following characters:
0123456789`~!@#$%^&*()_+-=[]{};:"|,./<>?abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
Additionally, Four character literals can also be represented by physical tab or space.

Example:

```
'ABCD'
'5`0]'
'    5w2'
```

1.5.2 Real literals

Example:

```
1.
1.1
0.1
.1
3.14159
```

1.5.3 Boolean literals

Boolean literal represents logical values. Only available literals are `true` and `false`.

1.5.4 String literals

String literal is such sequence of characters, that is enclosed in between quotation marks("). Maximal length of String literal is 1023 characters long. String literal can also contain tab, space and newline characters.
Special characters that must be pre-placed with backslash(\) to be valid:

`\n` - new line
`\t` - tab
`\"` - quotation mark
`\\` - backslash

Example:

```
nonStringLiteral"String Literal"
"Also
String literal"
```

1.5.5 Code literals

Code literal represents a user-defined function, that is addressed by it. Code literal can address any function that takes nothing. Return type does not play. To form Code literal, the name of function must be prefixed with function keyword.

Example:

```
function f
function myComplicatedFunction
```

To address static method, user must prefix the name with the struct's name and must use keyword method instead of function.

Example:

```
method MyStructType.myMethod
method thistype.myMethod
```

1.5.5.1 Advanced Code literals

Advanced Code literal allows user to address function that has arguments. To address such literal, user must specify which function to address.

Example:

```
function myF(integer)
function incredibleFunc(real, string, unit)
function MyStructType.myMemberFunc(real, real)
```

Additionally, user can address nonstatic member function, by writing `this` as first argument.

Example:

```
method myNonstaticMemberFunction(this, integer)
```

1.5.5.1.1 Advanced code literals as function argument

For functions, see 4. - Function.
Functions can take advanced code literals as their arguments. Type of any advanced code literal is:
`opt:(Const-Qualifier) function(opt:(type_list)) opt:(return_type)`

Example:

```
constant function(integer, integer, real, string) real
function(integer)
```

If no return_type is provided, function can have any return type. If Const-Qualifier is not provided, function passed can be constant or normal. If Const-Qualifier is provided, function passed as argument must be constant.

1.6 Expression

Expression is everything that can be stored within callable eJass construct(see 4. - Function, see 8.3 – Method). Expressions include(but are not limited to) assignments, function calls, control flow blocks and return statements. Expressions are always evaluated from left to right, meaning that following code:

```
call A(B(), C(D()), E())
```

will proceed in following order:

1. `call B()`
2. `call D()`
3. `call C()`
4. `call E()`
5. `call A()`

1.6.1 Expression levels

Every expression can be assigned value representing the depth. This value is called level. Top-level expressions are such expressions that are not encapsulated in another expressions. Every expression that is encapsulated in N expressions is of Nth-level expression. Expressions can be any level deep.

1.7 Comments

Comments serve to split code, and make it more readable, and potentially add notes to code.

1.7.1 Single line comments

Single line comments are comments that only allow commenting out text in one line,
from the beginning of the comment up to new line character.

1.7.1.1 Single line comments Syntax

`//` - Marks the beginning of Single line comment.

Example:

```
noncomment//comment
```

Beginning of Single line comments found within String literals are never treated as beginnings of comment lines.

1.7.2 Multiple line comments

Multiple line comments are comments that allow user to split their code
intelligently.

1.7.2.1 Multiple line comments syntax

`/*` - marks the beginning of Multiple line comment
`*/` - marks the end of Multiple line comment

Example:

```
nonComment/*Comment/*StillComment*/StillComment
Still Comment*/ not comment anymore
```

Multiple line comments can be nested indefinitely. Beginning and endings of Multiple line comments found within String literals are never treated as beginnings and endings(respectively) of Multiple comment lines.

1.8 Constness

1.8.1 Constant

Constant objects are such objects that can not be modified and can not modify the environment.

1.8.1.1 Constant Function behaviour

Constant functions(see 4. - Function) are such functions, that can not modify the arguments passed to them and can also not modify any global variable. Some native functions are not marked as constant but do not modify state of taken variables(Geters, Isers), and these functions are treated as constant.

1.8.1.2 Constant Function argument

Constant function argument(see 4. - Function) is such argument, that can not be modified when passed into function. Nonconstant variables may be passed into function expecting constant arguments.

1.8.1.3 Constant variable

Constant variable is such variable that can only be assigned to when defined. Constant variable can be passed into function's non-constant argument, if the variable is of copyable type(see 2.2.1 - Copyable Types).

1.8.1.4 Constant value

Constant value is any literal to Basic Type, Constant expression(see 1.9.4 – Compiletime Expression evaluation) or Constant variable.

1.8.1.5 Function returning constant variable

Function(see 4. - Function) can return constant variable, meaning it can not be modified and can only be stored in another constant variable; or can be passed as nonconstant argument, or stored in nonconstant variable if the return type is copyable(see 2.2.1 - Copyable Type).

1.8.2 Compiletime

1.8.2.1 Compiletime function behaviour

Compiletime function(See 4. - Function) is such function, that can be called in compiletime environment, such as inside static if block, or from other compiletime function. Additionally, all arguments of such function must also be compiletime values and the return value is also compiletime.

Compiletime function must have all paths in its body evaluable in compiletime. This means that there may only be if(see 5.2 - Compile time conditional block) or a loop of any sort with compiletime deducible number of iterations, or compiletime deducible exitwhen condition, or static if. Additionally, the function must return the same value at all times in one branch and therefore the return type may only be dependable on compiletime conditions.

Function marked as compiletime can be executed in both compiletime and runtime environment. If compiletime function is never used in runtime environment, the function is not generated in the resulting output.

Function is also considered compiletime if all of its arguments are compiletime and its return type is also marked as compiletime. Static method inside struct can also be declared as compiletime. It may, however, not access any nonstatic variables of any struct.

1.8.2.2 Compiletime function arguments

Function's arguments can also be specified as compiletime explicitly, and don't require whole function to be compiletime. Such variable can only be initialized to value deducible in compiletime, so return value of compiletime function, or function with compiletime return value, or literal.

1.8.2.3 Compiletime variable

Compiletime variable is such variable that can have its content read while script is compiling. Both global and local variables can be compiletime. Global compiletime variables can be initialized to literals of Basic Types, compiletime expression(see 1.9.4 - Compiletime Expression evaluation) or to compiletime value returned from Compiler method, or from user-defined function. Local compiletime variables can be initialized to literals of Basic Types, compiletime expression(see 1.9.4 - Compiletime Expression evaluation) or to compiletime value returned from Compiler method, or from user-defined function.

1.8.2.4 Compiletime value

Compiletime value is any literal to Basic Type, compietime expression(see 1.9.4 – Compiletime Expression evaluation) or compiletime variable.

1.8.2.5 Function returning compiletime variable

Function can mark its return value as compiletime. Such function must return a compiletime deducible value on all paths of execution.

Function with compiletime return value can only be used in in compiletime environment if all of its arguments are marked as compiletime as well, or if the whole function is marked as compiletime, or the function has no arguments.

1.8.2.6 Compiletime struct

Any user-defined object(struct) can be declared as compiletime, and can then be used in compiletime environment. All members of given struct(methods and variables) are implicitly declared as compiletime as well.

1.9 Optimizations

Various optimizations are done on the resulting map script after compilation.

1.9.1 Name Shortening

Every variable and function name is shortened to the shortest length possible, saving both file size and speed(thanks to nature of Jass Virtual Machine). The names start go in following pattern:
a - z, A - Z, a0 - a9, b0 - b9, ..., z0 - z9, A0 - A9...
Special name sequences may be reserved for special uses.

1.9.2 Function Inlining

Functions that are marked with **`inline`** are requested to be inlined, and will be inlined under certain conditions(see 4.1.1 - Inlining Rules). Short functions that are not marked as **`inline`** will also be inlined if the above conditions are met.

1.9.3 Constant/Compiletime variable inlining

Every compiletime and constant variable is inlined, if Optimization for speed is required.
Otherwise all compiletime and constant variables are only optimized if their respective values take less space then the variables themselves.

1.9.4 Compiletime Expressions evaluation

If there are expressions have compile-time deducible value(`Sin(3.0)`, `Pow(2.0, 5.)`, `(3 * (2 + 5))`), then the expressions are evaluated to the corresponding numbers(`0.05233596`, `32.` and `21` respectively).

1.9.5 Dead code removal

Every callable eJass construct or Expression that is not called, or referenced(taken address of) anywhere in code is automatically removed. Every unnecessary white space, as well as all delimited and normal comments are automatically removed. Every expression after return statement until end of nearest Scope is automatically removed. Every unreferenced variable is automatically removed.

1.9.6 As-If rule

Compiler can do anything to the code, as long as the resulting behaviour of the code does not change from the original intended behaviour of script. This means that the Compiler can for instance remove functions, even when called or referenced from trigger, timer, boolexpr or other Build-in type, if the function does not modify any visible state(For instance only defines local variable and sets it to some literal). If by this action timer is removed, and the function has no side effects but destroying the timer, the function can still be omitted. Declaring and initializing local variable with either literal or return value of constant function, or calling constant function has no side effects.

1.10 Deprecated

Keyword deprecated marks that given callable eJass construct(such as function[see 4. - Function] or method[see 8.3 – Method]) is old and should no longer be used. If callable eJass construct is marked with deprecated, the compilation will stop with error in Debug mode when such function is called, suggesting 'text', or pops warning in Release mode.

1.10.1 Syntax

```
deprecated opt:[(string message opt:(, replacement_function_name))]
```

If no message is provided, or empty string("") is passed, a "deprecated function call attempt" error or warning is popped. Otherwise, the format is always: 'called_func_name message'.
User can also pass a function name as second argument into deprecated, hinting which function should be called instead. The function name must be valid function name visible to the deprecating function as well as the caller in the map script, or name of native function, or name of Blizzard

defined function.
Function referenced in deprecated keyword does not need to have the same signature or return type as the deprecated function. If the function is of different signature, its signature is printed in the error/warning message.

Example:

```
deprecated("no longer supported") function myF takes nothing
                                                  returns integer
    return 5
endfunction

call myF()    //outputs: Error: function myF no longer supported

deprecated function still_Legal_Function takes nothing returns nothing
endfunction

function foo takes integer i returns integer
    return i + 1
endfunction

deprecated("outdated", foo) function bar takes nothing returns boolean
    return false
endfunction


call bar()      //outputs: Error:
                //function bar outdated.
                //Use function foo takes integer returns integer instead

scope s
    function f takes nothing returns integer
        return 5
    endfunction
endscope

deprecated("", s.f) function q takes nothing returns nothing
endfunction

scope Q
    struct MyStruct
        method a takes nothing returns nothing
        endmethod

        static method w takes nothing returns nothing
        endmethod


        deprecated("", thistype.w) method t takes nothing returns nothing
            //Error: deprecated static method MyStruct.t call attempt
            //       Use static method MyStruct.w instead
        endmethod
    endstruct
endscope
```

```
deprecated("don't use anymore!", Q.MyStruct.a) function QQ takes nothing
returns nothing
    //Error: function QQ don't use anymore!
    //       Use method Q.MyStruct.a instead
endfunction

deprecated("", Q.MyStruct.a) function QW takes nothing returns nothing
    //Error: deprecated function QW call attempt
    //       Use method Q.MyStruct.a instead
endfunction

deprecated("", Q.MyStruct.w) function qqw takes nothing returns nothing
    //Error: deprecated function qqw call attempt
    //       Use static method Q.MyStruct.w instead
endfunction
```

1.11 Operators

1.11.1 Arithmetic operators

Arithmetic operators perform arithmetic operations on operand or operands. List of arithmetic operators:

1. +
2. -
3. /
4. *
5. +=
6. -=
7. /=
8. *=
9. ++
10. --
11. %
12. %=
13. <<
14. >>
15. <<=
16. >>=

None of operators accept non-implicitly convertible types as their operands. Operators 9, 10, 11, 12, 13, 14, 15 and 16 only accept integer variables.

1.11.2 Logical operators

Logical operators perform logical operations on operand or operands. List of logical operators:

1. ==
2. <=
3. >=
4. !=
5. !

6. not
7. and
8. or

Operators ==, <=, != and >= can accept any implicitly convertible types, and can also accept any mix of integers and reals. Operators 5, 6, 7 and 8 only accept boolean values(unless overloaded). Additional operations on operators can be found in 8.8.1 - List of overloadable operators.

1.11.3 Operator precedence

The following is list of precedences of operators in descending order(highest precedence to lowest):

Name	Symbolic representation
1. Parenthesis	`()`
2. unary increment/decrement	`X++, ++X, X--, --X`
3. bracket operator	`X[Y]`
4. dot operator	`X.Y`
5. not operator	`not A, !A`
6. multiplication, division operator	`X * Y, X / Y`
7. addition, subtraction operator	`X + Y, X - Y`
8. bit shift operator	`X << Y, X >> Y`
9. boolean operators	`X < Y, X <= Y, X > Y, X >= Y, X == Y, X != Y`
10. and, or operators	`X and Y, X or Y`
11. Assignment operators	`X = Y, X += Y, X -= Y, X *= Y, X /= Y, X %= Y, X <<= Y, X >>= Y`
12. Comma operator	`X, Y`

1.12 Syntactic Unit

Syntactic unit is every non-white space character inside map script, be it name, keyword or parenthesis.

1.13 White spaces - General

Every Syntactic Unit(see 1.12 - Syntactic Unit) must be separated by at least one white space. For example, there must always be space between function and its name, and its name and takes. If two Syntactical Units can not be separated by new line, they are bound tightly. If two Syntactical Units can be separated by new line, they are bound loosely.

1.14 Initialization rules

Various functions can be marked as initializers which makes them run when the map is loaded. Initialization goes in this specified order:

1. Library(see 7. - Library) initializers in following order:
    1. Library's initializer
    2. Initializer of every scope inside such library, which also follow 2.
    3. Initializers of modules implemented inside every struct inside such library
    4. Initializer of every struct inside such library

5. Free-standing initializer

2. Scope(see 6. - Scope) initializers in following order:
   1. Scope's initializers
   2. Initializer of every scope inside such scope, which also follow 2.
   3. Initializers of modules implemented inside every struct inside such scope
   4. Initializer of every struct inside such library
   5. Free-standing initializer

3. Struct(see 8. - Struct) initializers in following order:
   1. Initializers of modules implemented by the struct
   2. Struct's initializer

4. Free-standing Initializer(See 4.5 - Initializer function)

1.15 Debug

Debug mode can be represented by global variable `DEBUG_MODE`(set to `true` when compiling in debug mode) or with keyword debug.

1.15.1 DEBUG_MODE and RELEASE_MODE

The Compiler defines 2 compile-time global boolean variables usable anywhere in the script called DEBUG_MODE and RELEASE_MODE. When compiling in debug mode, DEBUG_MODE is set to true and RELEASE_MODE is set to false. When compiling in release mode, RELEASE_MODE is set to true and DEBUG_MODE is set to false.

1.15.2 debug keyword

When any Syntactic Unit is pre-placed with debug keyword, given Syntactic Unit will only be generated when The Compiler is compiling the map script in Debug mode.
Example:

```
debug function f takes nothing returns string
    debug local string s = "Some String literal"
    debug return s
debug endfunction


function w takes nothing returns nothing
    local string toPrint = "TPr"
    debug if SomeFunction() then
        debug call BJDebugMsg(Compiler.nameOf(SomeFunction) +
                              " returned true! RUN AWAY!")
    debug else
        call BJDebugMsg(toPrint)
    debug endif
endfunction
```

Function f as well as its contents will only be generated when eComp compiles in Debug mode.

When calling function w, it will create local string and initialize it to string literal "Tpr". Then, if The Compiler is compiling in debug mode, then SomeFunction will be called, and if it returns true then it will print that the function returns true. If it returns false, it will output contents of the string. If eComp is compiling in Release mode, it will only output the contents of the string, even though it is defined inside else branch of the if, because the whole if block is removed from the resulting script.

1.16 Priorities

Priority defines when should given eJass Construct be executed/evaluated or in other means ran.

1.16.1 Syntax

```
priority=[integer literal]
```

Example:

```
priority=54
priority=-200
```

If eJass construct has optional priority, and none is provided, implicit priority of 0 is assumed. Priorities can have range from -2,147,483,648 to 2,147,483,647. Special Priorities are "Lowest, Highest".
priority=Highest requires given eJass Construct to be ran before any other same eJass Construct.
priority=Lowest requires given eJass Construct to be ran after any other same eJass Construct.
If 2 same eJass Constructs are defined with the same priority, they are ran in order of appearance of these eJass Constructs to the parser.

1.16.1.1 White spaces

```
priority=[integer literal]
```

1. priority keyword is bound tightly into following =.
2. = is bound tightly into [integer literal].

1.17 Static_assert

While compiling, when eComp encounters static_assert, it will stop compilation with error message if certain condition evaluates to true.

1.17.1 Syntax

```
static_assert(condition opt:[, error_string])
```

Example:

```
static_assert(integer == real, "Internal eComp error")
```

If condition, which must be compile-time condition, inside static_assert evaluates to true, a compilation is immediately stopped with error. If error_string is provided, it is output with the

error message.

1.17.1.1 White spaces

See 4.2.2 - Function Calling

1.18 preprocessor

Preprocessor is part of the compiler that runs over the map script before the script is parsed and validated, optimized and converted into Jass. Preprocessor directives are: #if, textmacro, extendor, import, external.
Preprocessor does 2 passes over the map script. In first pass, the preprocessor will import all code that import preprocessor directive requested and remove these directives as well as parse external blocks. In second pass, the preprocessor handles the rest(static ifs, textmacros, extendors).

The preprocessor starts at the top of the map script, and parses the map script. If preprocessor finds library definition, it will jump into the libraries that this library requires/uses/needs before parsing that library. If preprocessor hits external with before as ExecMode, it will comment out this external block and after passing through the map script, it will attempt running the external tools with given input. After that, the map script starts parsing again. If preprocessor hits external with after as ExecMode, it will comment it out, continue compiling the map, and after eComp compiled the map it executes the external tools.

If there is static if and it contains any other preprocessor directive other than import and external, these preprocessor directives will only be parsed if the static if condition evaluates into true. If there is import or external inside static if, it is parsed regardless of static if.

Preprocessor holds state of currently defined eJass constructs for use in static if in combination with Compiler struct.

1.19 Function(method) weight on resolution

When function is called, every version of given function is weighted and function with best weight is picked to be called. The lowest weight function is always picked.
The weighting system works in following way:

1. Function with the same signature requiring no casts has weight 0. If multiple functions with weight 0 for single call exist, a compiler error is issued.
2. Function with the same number of arguments but requiring one or more implicit conversions has weight 1. If multiple functions with weight 1 are found and none with weight 0, the function that requires the least number of implicit conversions is picked. If there are multiple functions with the same number of required implicit conversions and one of them is picked to be the called function, a error is issued.
3. Specialized template of function has weight of 2. If multiple specialized templates with the same number of arguments are found, they are decided with rules 1. and 2.
4. Template function has weight 3. If after type deduction multiple functions fit the function call, the correct function to call is decided with rules 1. and 2.

1.20 namespace

Namespace is a special eJass construct that encapsulates any other code(including Libraries) and gives them special name. Access to members of namespace is done via scope resolution operator. Namespace with any given name can be defined multiple times, and this will not create collisions, only if there are 2 members with colliding names inside given namespace will collision be created. Access to contents of namespace is done by scope resolution operator. Code inside namespace A that wants to use code from the same namespace does not need to use scope resolution operator, because inside namespace, the Scope of given namespace is always implicitly defined.

1.20.1 Syntax

```
namespace name
    namespace-contents
endnamespace
```

Namespace can only be found within Global Scope or another namespace.

1.20.1.1 White spaces

```
namespace name
    namespace-contents
endnamespace
```

1. keyword namespace is bound tightly into name
2. pair namespace name as well as endnamespace must be placed on separate line from other code

1.21 Naming rules

1.21.1 Directly inside Scope A

Directly inside Scope A means that the eJass Construct resides inside Scope A, not in any Scope inside Scope A.

Example:

```
scope A
    function f takes nothing returns nothing
    endfunction
    scope B
        function w takes nothing returns nothing
        endfunction
    endscope
endscope
```

In this code, f is directly inside Scope A, and w is not directly inside Scope A, but is directly inside Scope B.

1.21.2 Rules for Functions, Methods, variables

If there exists a variable directly inside Scope A, then there can not be any other variable directly inside Scope A with the same name, regardless of variable type but can exist in any subScope of scope. If there exists a function or method directly inside Scope A, then there can not be any other function or method with the same signature and same name directly inside Scope A but can exist if it is of different signature.
2 Functions/Methods can share names if one of them is template and the other one is not, or if one is template and the other one is specialized template, or if they are both templates. Disambiguation happens via function weighting.

1.21.3 Rules for Scope introducing Constructs

Scope introducers are: namespace, library, scope, struct.
If there is scope introducer directly inside Scope A, then there can not be another scope introducer directly inside Scope A with the same name.

1.21.4 Rules for other eJass constructs

For any given eJass Construct there directly inside Scope A there can not exist another eJass Construct of same type with the same name directly inside Scope A.

1.22 Nesting rules

Not all eJass constructs can be nested into other eJass constructs.
This table defines who can be nested into who:

Name	Can be found inside
namespace	Global Scope, namespace
library	Global Scope, namespace
scope	Global Scope, namespace, library
function	Global Scope, namespace, library, scope
struct	Global Scope, namespace, library, scope, struct
module	Global Scope, namespace, library, scope
method	struct, module
globals block	Global Scope, namespace, library, scope
interface	Global Scope, namespace, library, scope
concept	Global Scope, namespace, library, scope
import	Global Scope, namespace, library, scope
free standing method	Global Scope, namespace, library, scope
allocator	Global Scope, namespace, library, scope

hook	Global Scope, namespace, library, scope
textmacro	Global Scope, namespace, library, scope
extendor	Global Scope, namespace, library, scope
loop	function, method, free standing method
while	function, method, free standing method
for	function, method, free standing method
if block	function, method, free standing method
variable definition	globals block, function, method, struct, allocator, module, if block, for, loop, while
external	any
externalblock	any
textmacro execute	any
extendor execute	any
static if	any excluding external and externalblock

Notes: Struct inside another struct has no special enclosing rules, and the outer struct functions like scope for the inner struct.

1.23 Constructs allowing Visibility-Qualifiers

Following eJass constructs allow Visibility-Qualifiers to be used inside their Scope:

- namespace
- library
- scope
- struct
- module

2. Type

Type is a named eJass Construct that differentiates different variables by their meaning.

2.1 Syntax

```
type type_name opt:(extends type_name)
```

Users may not declare new types, and all types shall only be declared from within common.j or common.ai.

2.2 Basic types(Built-in types)

Basic types are types, that are elementary, and are not directly tight to the game logic.

2.2.1 Integer

Integer is representation of a whole number, with range from -2,147,483,648 to 2,147,483,647. Default operations on integers are possible, including +, -, *, /, <, >, <=, >=, ==, !=. Extra operations on integers are %, <<, >>, ++, --, +=, -=, *=, /=, %=, <<=, >>=.

(v << n) multiplies the integer value by 2^n.
(v >> n) divides the integer value by 2^n.
(v % n) returns reminder of (v / n). This value is calculated as (v - (v / n) * n).
v++ increments the stored value and returns it.
v-- decrements the stored value and returns it.
v++ and v-- can only be used with set keyword on one given variable, possibly array.
Meaning that

```
set v = v++ + v++
set v = v++
```

is malofrmed on both lines and

```
set v++
```

is allowed.

2.2.2 Real

Real literals represent numbers with decimal point.
Range, as well as special values of Real literals is the same as IEEE 754 binary32. Default operations on reals are possible, including +, -, *, /, <, <=, >, >=, ==, !=. Extra operations on reals are +=, -=, *=, /=.

2.2.3 Boolean

Boolean represents either true, or false. Default operation on boolean is ==, !=, !, not, and, or.

2.2.4 String

Strings represent text in character-by-character form. Default operations on strings are +, ==, !=.

2.2.5 Code

Code represents address of user-defined function. Basic code variable can only address function that takes nothing and returns anything.

2.2.5.1 Advanced Code

Advanced code is extension to Code. It allows user to address functions that take any number of any parameters and also pass in immediate values of variables or literals into the functions.

2.2.6 Handle

Handle is special type that is base for every other non basic type. Handle and built-in types derived from handle are not copyable and are a mere pointers to physical object lying in memory.

2.2.6.1 Null

null is special value, which represents nothing being stored in variable and can be used with any variable which derives from handle, handle or for string. It is implicitly convertible to any of these types.

2.2.7 Nothing

Nothing is a special type, that can not be instantiated, and is used to mark when function does not have any parameters, or when it does not return any value.

2.3 Built-in Type

Non Basic Type is such type, that is provided by Blizzard in common.j file. The number of Non Basic Types is dependant on version of common.j file.
Examples of Non Basic Types are: unit, texttag, location, group, force, lightning and many more.
Non basic types are not copyable, meaning that if function attempts to modify state of its argument, it will also modify state of passed value.

2.4 User-Defined Type

Users can't define types using type keyword.
For User-defined Types definition, see 8. - Struct.

2.5 Conversions and Comparisons

2.5.1 Conversions in operators

1. Integer and real can be compared with any comparison operators
2. Any combination of mathematical operations on Integer and real result in real

2.5.2 Implicit conversion

1. Every derived object is implicitly convertible to any of its parent types.
2. Real is always implicitly convertible to integer, if the decimal point and decimal part is not explicitly typed.
3. Integer is always implicitly convertible into real.
4. Null is implicitly convertible to any non-user defined type deriving from handle, including handle and string. Null is not implicitly convertible to any other type if it is explicitly cast to specific type.
5. Conversion of type from non-constant to constant is always considered as implicit conversion.

2.5.3 Explicit conversion

1. Using casting operator(see 8.9 - Conversion operator), any user-defined type may be cast to any other user-defined type or to basic or non basic type.
2. Basic and Non Basic types can be cast into user-defined types, but can only be cast to other Basic or Non Basic types if they can be implicitly converted to that type.

2.5.4 Comparisons

1. A comparison of types in compile-time environment exists with syntax: TypeA == TypeB, TypeA != TypeB. This comparison is the same as Compiler.isSame(TypeA, TypeB) and ! Compiler.isSame(TypeA, TypeB) respectivelly.

3. Variable

Variable stores instances of objects of particular type, which are addressable by the variable. Variables can be either global, visible to the whole map script, or local to function or method, visible only inside the function or method respectively. Variables can also be declared for structs(see 8. - Struct), or inside modules(see 11. - Module). Every variable's name must start with [a-z] or [A-Z] and must end with [a-z][A-Z][0-9]. Variable's name can additionally contain '_' anywhere but first or last character.

3.1 Global Variable

Every global variable in eJass must be defined inside globals/endglobals block.

3.1.1 Syntax

```
globals
    opt:Visibility-Qualifier opt:Comp-Qualifier opt:Const-Qualifier type_name
        variable_name opt:(=Initial_value)
    ...
endglobals
```

Example:

```
globals
    private integer i
    private constant integer q
    public real r = 4.22
    real w = 7.99
    private compiletime real qR = 7.66
    private compiletime constant real constqR = qR
endglobals
```

Visibility-Qualifier is only usable, if globals block is declared within non-global Scope(see 1.3 - Scope).

3.1.1.1 White spaces

```
globals
    opt:Visibility-Qualifier opt:Comp-Qualifier opt:Const-Qualifier type_name
        variable_name opt:(=Initial_value)
    ...
endglobals
```

1. globals and endglobals keywords must be placed on separate lines
2. Visibility-Qualifier, Comp-Qualifier and Const-Qualifier are always bound tightly into each other if provided.
3. type_name and variable_name are bound loosely.
4. If Initial_value is provided, it is bound loosely into preceding = and = is bound tightly with variable_name Additionally, Initial_value must also follow its respective rules.
5. Every other Syntactic Unit is bound tightly.

3.2 Local Variable

Every local variable in eJass must be defined within method or function. Local variables can be placed anywhere in function(method), and they are only usable from end of their definition.
Additionally, local variables are only visible to the Scope they are declared in(local variable declared inside if block is only visible inside if block) and after the end of given Scope, the variable is no longer existing.

Variables inside Scopes can, as per rule of shadowing shadow other local variables, such that variable defined inside if block can have the same name as variable defined before the if block, and this will not result in ambiguity, and only the inner variable is modified inside given if block.

Following code:

```
local integer i = 0

if true then
    local integer i = 5
    set i++
endif

call BJDebugMsg(i)        //prints 0
```

as well as

```
if true then
    local integer i = 5
endif

local integer i = 6
```

is valid code.

3.2.1 Syntax

```
opt:Comp-Qualifier opt:Const-Qualifier local type_name variable_name
opt:(=Initial_value)
```

Example:

```
local integer i
local unit u = CreateUnit(...)
constant local real r = 4.44
```

Arguments of callable eJass constructs(see 4. - Function, see 8.3 - Method) also count as local variables. Constant locals have to be initialized before end of their definition.

3.2.1.1 White spaces

```
opt:Comp-Qualifier opt:Const-Qualifier local type_name variable_name
opt:(=Initial_value)
```

1. Comp-Qualifier, Const-Qualifier and keyword local are always bound tightly together if provided.
2. type_name and variable_name are bound loosely.
3. If Initial_value is provided, it is bound loosely into preceding '=' and '=' is bound tightly with variable_name. Additionally, Initial_value must also follow its respective rules.
4. Every other Syntactic Unit is bound tightly.

3.3 Temporary Variable

Every temporary variable in eJass must be defined within method or function. It functions the same way as local variable, but before every assignment, other than in definition, object stored in the variable is destroyed/removed. The object is also destroyed before the function returns or loses control in any standard way.

Only location, group, force, lightning, effect, trigger, region, rect or boolexpr can be declared as temporary.

3.3.1 Syntax

```
opt:Comp-Qualifier temporary valid_type_name variable_name opt:(=Initial_value)
```

Example:

```
temporary group g = CreateGroup()
call GroupAddUnit(g, CreateUnit(...))
set g = CreateGroup()                    //first calls DestroyGroup(g),
                                         //and then executes this command
```

Initial_value can be any local or temporary variable defined in the same scope before the temporary, global variable visible to the Defining Scope, or a literal implicitly convertible into valid_type_name.

3.3.1.1 White spaces

```
opt:Comp-Qualifier temporary valid_type_name variable_name opt:(=Initial_value)
```

1. valid_type_name and variable_name are bound loosely.
2. If Initial_value is provided, it is bound loosely into preceding '=' and '=' is bound tightly with variable_name Additionally, Initial_value must also follow its respective rules.
3. Every other Syntactic Unit is bound tightly.

3.4 Array Variable

Array Variable is such variable that has one name and can store up to 8192 different objects accessed by non-negative integers. The index range goes from 0 to 8191. If user tries to access element out of the bounds, nothing happens. Both global and local variables can be declared as arrays.

3.4.1 Syntax

```
opt:(local) type_name array variable_name opt:(= {initial values})
```

Example:

```
globals
    unit array myUnitArray = { null, null, null, CreateUnit(...), null }
    constant unit array Arr = { null, CreateUnit(...) }
```

Local keyword is only provided if declaring local array variable. Temporary variables can not be marked as arrays. Arrays can be initialized with {} block in following way:

> comma separated values implicitly convertible into type of variable, which will be stored in order of their appearance starting at index 0. 0 elements may also be specified between { and }.

Example:

```
local unit array u = {}
//the same as local unit array u
```

All indices of array, with or without braces-enclosed initializer list are filled with default value of given type and specified indexes are filled with values inside braces-enclosed initializer list afterwards.

3.4.1.1 White spaces

```
opt:(local) type_name array variable_name opt:(= {initial values})
```

1. local keyword, if provided, is bound tightly into type_name.
2. type_name is bound tightly into 'array' keyword.
3. array keyword is bound loosely with variable_name.
4. Every other Syntactic Unit is bound tightly to each other.

`(= {Initial values})` follow these rules:

1. = Syntactic Unit and the initial value block is bound loosely to the previous syntactic unit.
2. = Syntactic unit is bound loosely to initial value block.
3. Initial value's curly brackets are bound loosely into values placed in between them.
4. If no initial values are used, but curly brackets are used, they are bound tightly to each other.
5. Every value inside 'Initial Value' block is bound loosely to every other value, but is bound tightly to following comma(if any)
6. Closing curly bracket is bound loosely with last value.
7. Every other Syntactic Unit is bound tightly into any other Syntactic Unit.

3.5 Struct variable

Structs(see 8. - Struct) can contain variables, which are global to the struct, accessible from any point of the struct. Struct variables are also possibly accessible from outside of struct.

3.5.1 Array struct variable

3.5.1.1 Syntax

```
opt:Visibility-Qualifier opt:Readonly-Qualifier
opt:Staticity-Qualifier opt:Comp-Qualifier opt:Const-Qualifier
type_name array variable_name [compiletime integer] opt:(= {Initial values})
```

Example:

```
readonly unit array myArr[10]
private static real array myRealArray = {1., 2., 3., 4.}
private static constant real array arr = {2., 3/4., Sin(5)}
```

Readonly-Qualifier can only be used, if Visibility-Qualifier is not specified, or is specified as public. Initial values follow the same rules as 3.4 - Array variables.
[compiletime integer] means that there must be pair of brackets([]) with compiletime integer value inside to specify number of elements reserved per instance, if the array is non-static. If array is static, no size must be provided, and if one is provided, it has no effect and given array can be modified past the specified size. If array is initialized, it is not required to provide size, and if it does not provide size, the size of initialization is used instead. Arrays can be initialized with {} block in following way:

comma separated values implicitly convertible into type of variable, which will be stored in order of their appearance starting at index 0. 0 elements may also be specified between { and }.

Example:

```
unit array u = {}
```

If struct is marked as using array, then the maximum number of instances given struct can have allocated at the same time is equal to floor(8190/biggest_nonstatic_array_size). So for instance, if there are 3 non-static arrays inside struct, with sizes 10, 10 and 15, the maximum number of

allocated instances is 546(floor(8190/15)).

3.5.1.1.1 White spaces

```
opt:Visibility-Qualifier opt:Readonly-Qualifier
opt:Staticity-Qualifier opt:Comp-Qualifier opt:Const-Qualifier
type_name array variable_name [compiletime integer]
opt:(= {Initial values})
```

1. Name of type and array keyword are bound tightly.
2. Name of variable and keyword array are bound loosely.
3. Any and all Qualifiers and name of type are bound tightly.
4. = Syntactic Unit and the initial value block is bound loosely to the previous syntactic unit.
5. = Syntactic unit is bound loosely to initial value block.
6. Initial value's curly brackets are bound loosely into values placed in between them.
7. If no initial values are used, but curly brackets are used, they are bound tightly to each other.
8. Every value inside 'Initial Value' block is bound loosely to every other value, but is bound tightly to following comma(if any)
9. Every Other Syntactic Unit in Array struct variable definition is bound tightly to each other.

3.5.2 Nonarray struct variable

3.5.2.1 Syntax

```
opt:Visibility-Qualifier opt:Readonly-Qualifier
opt:Staticity-Qualifier opt:Comp-Qualifier opt:Const-Qualifier
type_name variable_name opt:(= Initial value)
```

Example:

```
public static constant integer myVar = 6
private static real myReal
private unit u
compiletime constant integer i = 5
```

Readonly-Qualifier can only be used, if Visibility-Qualifier is not specified, or is specified as public. Readonly-Qualifier can not be specified, if Const-Qualifier is specified. Static Constants must be initialized before end of their definition.

3.5.2.1.1 White spaces

```
opt:Visibility-Qualifier opt:Readonly-Qualifier
opt:Staticity-Qualifier opt:Comp-Qualifier opt:Const-Qualifier
type_name variable_name opt:(= Initial value)
```

1. Name of type and name of variable are bound loosely.
2. Every Other Syntactic Unit in Nonarray struct definition is bound tightly to each other.

4. Function

Function is a callable, addressable eJass construct, which can have arguments passed into it and it can also return value.

4.1 Function's Signature

Function's signature consists of function's name and its number and type of arguments. Return type of function is not part of function's signature.

4.2 Syntax

4.2.1 Definition

```
opt:deprecated opt:inline opt:Visibility-Qualifier
opt:Comp-Qualifier opt:Const-Qualifier
function function_name takes opt:arguments returns
opt:Comp-Qualifier opt:Const-Qualifier return_type_name
    ...
endfunction
```

Arguments are of following form:

```
opt:Comp-Qualifier opt:Const-Qualifier type_name1 variable_name1
opt:(, opt:Comp-Qualifier opt:Const-Qualifier type_name2 variable_name2 ...)
```

Every argument may declare required type as constant and/or compiletime, and arguments must be comma separated.

Example:

```
deprecated function myF takes nothing returns nothing
    ...
endfunction

inline function myF takes integer a, integer b returns compiletime string
    return "5"
endfunction

deprecated inline function myF takes integer a returns constant integer
    return a
endfunction

constant function q takes nothing returns compiletime string
    compiletime local string s = "gg"
    return s
endfunction

constant function qW takes nothing returns compiletime string
    return "qq"
endfunction

function qWW takes nothing returns compiletime string
    return qW()
```

```
endfunction

function www takes compiletime constant integer i returns nothing
endfunction
```

Visibility-Qualifier is only usable when used inside non-global Scope(See 1.3 – Scope). If no Visibility-Qualifier is used, function is treated as public.
If function takes no arguments, then nothing shall be wrote between takes and returns. If function takes arguments, then they must be fully defined and they must be separated by commas.
If function does not return anything, nothing must be used as return type. If function does return some type, then all returned values of given function must be implicitly convertible into specified return type, but they don't need to be of same type. Return type may me marked with constant, in which case the caller may only store it inside constant variable, or pass into other function as constant argument, or the variable must be of copyable type.

4.2.1.1 Function argument's implicit values

Function arguments may have implicit values specified, meaning that when the function is called, some arguments may end up unspecified from callers point of view, and have implicit value defined inside the function definition.

```
opt:Comp-Qualifier opt:Const-Qualifier type_name1 variable_name1 opt:(= Initial
Value) opt:(, ...)
```

Example:

```
... takes compiletime constant string s, constant integer i,
          unit u = null returns ...
```

Implicit value of variable N may only be specified, if all function's arguments to the right from N already have implicit value specified, meaning that takes integer i = 5, unit u returns is malformed. Implicit value can also be return value of function, in which case, return value of function must be implicitly convertible to the type of variable setting implicit value for, or must be cast explicitly to given type.

4.2.1.2 White spaces

```
opt:deprecated opt:inline opt:Visibility-Qualifier
opt:Comp-Qualifier opt:Const-Qualifier
function function_name takes opt:arguments returns
opt:Comp-Qualifier opt:Const-Qualifier return_type_name
    ...
endfunction
```

1. If both deprecated and inline keywords are specified, they are bound tightly.
2. If Visibility-Qualifier is specified, and inline keyword is specified, they are bound tightly.
3. If Comp-Qualifier is specified, it is bound loosely to preceding Syntactic Unit.
4. If Const-Qualifier keyword is specified, it is bound tightly to both preceding Syntactic Unit and 'function' keyword.

5. 'function' keyword is bound tightly to the name of function, which is bound tightly to the 'takes' keyword.
6. all arguments are bound loosely to each other with special rules(see 4.2.1.3 - Argument's White spaces).
7. returns keyword is bound tightly to the last argument specified in argument's list or to nothing keyword otherwise.
8. Return type with its Qualifiers is bound tightly to 'returns' keyword.
9. If Const-Qualifier and/or Comp-Qualifier are specified for return type, they are bound tightly with the return type and each other.
10. Every other Syntactic Unit is bound tightly to each other.

4.2.1.3 Argument's White spaces

```
opt:Const-Qualifier type_name1 variable_name1
opt:(, opt: opt:Const-Qualifier type_name2 variable_name2 ...)
```

1. If Const-Qualifier or compiletime keyword are specified, they are bound tightly to name of argument's type.
2. Argument's name is bound tightly to name of argument's type.
3. Comma following argument N, if specified is bound tightly into argument's name.
4. Every independent argument is bound loosely to each other.

4.2.2 Calling

```
call function_name(opt:arguments)
```

Arguments are of following form:

```
opt:value1 opt:(, value2 opt:(, value3 ...))
```

Example:

```
call myF(1)
call myW(1, 2, null)
call myQ()
call myZ(
         1,
         2,
         3,
         null
        )
```

Every argument can be enclosed in any number of parenthesis. However, every comma must be inside the top bracket-scope.

Example:

```
call myF(((1)), (2))    //valid
call myF(((1), 2))      //invalid, comma before second argument is
                        //not in top bracket-scope
```

4.2.2.1 White spaces

```
call function_name(opt:arguments)
```

1. Keyword call and function's name are bound tightly.
2. Opening parenthesis is bound tightly with function's name.
3. If no arguments are provided, closing parenthesis is bound tightly with opening parenthesis.
4. If any number of arguments are provided, then the first argument is bound loosely to the opening parenthesis, the last argument is loosely bound to the last parenthesis and arguments are bound loosely between each other.
5. Every other Syntactic Unit is bound tightly to each other.

Argument's White spaces

```
opt:value1 opt:(, value2 opt:(, value3 ...))
```

1. If multiple arguments are provided, then the value is bound tightly to following comma, and following comma is bound loosely to the next value.

4.2.4 Inlining Rules

Function marked with inline keyword is requested to be inlined. Functions may be inlined only if none of the following rules are violated:

- Optimization for size is specified and the length of expanded code is smaller than the size of original function
- function is not referenced inside trigger callback, timer timeout or used as boolexpr, or types derived from boolexpr
- Optimization for speed is specified and size function is less than 100 bytes(after other optimizations)

4.3 Function overloading

Functions may have the same name, and only be differentiated by their signature(see 4.1 - Function's Signature). If two functions only differentiate in signature by types that are implicitly convertible, these functions are conflicting.

Example:

```
constant function myF takes integer i returns boolean
    return i > 5
endfunction

function myF takes unit u returns integer
    return GetUnitLifePercent(u)
endfunction

call myF(GetUnitLifePercent(null))    //calls myF(integer)
call myF(null)                        //calls myF(unit)

function myF2 takes unit u returns nothing
function myF2 takes location l returns nothing
```

```
call myF2(Location(0., 0.))        //fine, calls myF2(Location)
call myF2(null)                    //compiler error: ambiguous myF2(null),
                                   //can be myF2(unit) or myF2(location)

call myF2(cast<unit>(null))        //fine, calls myF2(unit)
```

4.4 Anonymous Functions

Also called lambda functions, are functions that are nameless. Lambda function can be passed into function expecting Extended Code Literals, or can be invoked after its definition.

4.4.1 Syntax

```
opt:Comp-Qualifier opt:Const-Qualifier function(opt:arguments) opt:return_type
opt:catch{
    //body
}
```

Example:

```
constant function(integer i) boolean catch{
    return i > externalVariable
}
```

Constness as well as arguments must follow normal function rules. If function does not return anything, no return type is required. Catch keyword allows anonymous function to use variables local to Defining Scope. If anonymous function is not constant and catches Defining Scope's variables, it may modify them.

Manual invoking Example:

```
function(integer i) boolean{
    return i > 5
}(5)
```

4.4.1.1 White spaces

```
opt:Comp-Qualifier opt:Const-Qualifier function(opt:arguments) opt:return_type
                                                        opt:catch{
    //body
}
```

1. Comp-Qualifier and Const-Qualifier are bound tightly into each other as well as with function keyword, when provided.
2. function keyword is bound tightly to the opening parenthesis.
3. In case of no arguments, closing parenthesis is bound tightly to opening parenthesis.
4. For every argument of type T and name N, T and N are bound tightly, together with following comma, if another argument exists.
5. Return type, if specified, is bound loosely to the closing parenthesis of arguments.

6. catch keyword, if specified is bound tightly with return type, if specified.
7. Opening curly brackets is bound loosely with its preceding Syntactic Unit, be it catch, return type or closing parenthesis.
8. Closing curly bracket must be placed on separate line in regards to the body of anonymous function.
9. If explicit call is specified after closing curly bracket, the same rules apply as to normal function's call(see 4.2.2 – Calling).
10. Every other Syntactic Unit is bound tightly to each other.

4.5 Initialization functions

Initialization functions are such functions that perform some initial initialization of code relevant to the code in its Trigger file. Initialization functions must have specific name, otherwise they act as normal functions.
Name format:
InitTrig_XXXX.

XXXX is the name of Trigger file, for which the initialization trigger will run.

5. Control flow

Control flow allows user to control how the code runs in various ways.
The basic control flow structures are loops and conditional blocks.

5.1 Conditional block

Conditional block allows user run code only if certain condition is met.

5.1.1 Syntax

```
if condition opt:(bool_operand condition2 opt:(...)) then
    ...
elseif boolean_evaluated_expression then
    ...
else
    ...
endif
```

Example:

```
if myBooleanVar and not GetBoolean(null, 5.42 * 1.22) then
    call MyFunction1()
elseif not myBooleanVar then
    call MyOtherFunction()
else
    call SomeYetAnotherFunction()
endif

if A or (not B and C) and D then
    ...
```

```
endif
```

Condition is such Syntactic Unit, that is implicitly evaluable into boolean. Condition can be variable, return value of function or validly enclosed bracket-scope. not keyword is part of condition.
Examples:

```
if not A or B then
    not A     = condition 1
    B         = condition 2
```

If-Block is block of code from if keyword up until then keyword.
Then-Block is block of code after then keyword up until matching else, endif or elseif keyword.
ElIf-Block is block of code after then keyword preceded by elseif keyword up until matching else, endif or elseif keyword.
Else-Block is block of code after else keyword up until matching endif keyword.

Keyword if must always be followed by valid condition, that is implicitly convertible into boolean. Additionally, the condition may be preceded by not boolean operator, which will negate the resulting boolean condition. If there is condition A, then the next boolean operator after that must be in the same bracket-scope(see 1.3.1 – Bracket-scope), or in lower bracket-scope then A itself, or A must not be in 0th bracket-scope.
Not boolean operator is always tight up into the following condition. And and or boolean operators must be separated by exactly one condition.
If one of the operands of and boolean operator is evaluated to false, then the whole and operator is evaluated to false. If one of the operands of or boolean operator is evaluated to true, then the whole or operator is evaluated to true.
Evaluation of conditions always goes from right to left.

If the result of initial if-block is true, the body of then-block is executed. Otherwise, the next if-block begins evaluation, if there is any, or the else-block starts execution, if it exists.
Any number of if blocks can be nested into any block.

5.1.1.1 White spaces

```
if condition opt:(bool_operand condition2 opt:(...)) then
    ...
elseif boolean_evaluated_expression then
    ...
else
    ...
endif
```

1. if keyword is bound tightly to the following condition.
2. then keyword is bound tightly to the preceding condition.
3. elseif, else and endif keywords must be placed on separate lines.
4. Everything else is bound loosely.

5.2 Compile time conditional block

Compile time conditional block controls code generation during compilation.
Special values are usable, and reserved, when using Compile time conditional block:

- `DEBUG_MODE`
- `RELEASE_MODE = not DEBUG_MODE`

DEBUG_MODE evaluated to true, when script is being compiled in debug mode, and to false otherwise.

5.2.1 Syntax

```
#if condition opt:bool_operand opt:(opt:not_operand value...) then
    //SThen-Block
#elseif ... then
    //ElIf-Block
#else
    //Else-Block
#endif
```

Every other eJass construct can be found within Compile time conditional block, including other Compile time conditional blocks.
Compile time conditional block can be found within any other eJass construct, excluding external block.
Any variable declared within Compile time conditional block is not existing before the condition of given conditional block is evaluated. Meaning that:

```
globals
    #if SomeExternalGlobalDeclaredBelow then
        //evaluates to false
        integer a = 4
    #endif

    #if Compiler.exists(a) then
        //since the above static if evaluated to false,
        //the integer never got created so this evaluates
        //to false just as much
        integer b
    #endif
endglobals
```

will never create integer b inside this globals block.
The whole condition must be compiletime.
If any sub-expression of the whole condition is not evaluable in compile time, the whole condition is evaluated to false. Example is trying to read non compiletime variables, which will instantly yield false for the whole condition.

5.2.1.1 White spaces

```
#if condition opt:bool_operand opt:(opt:not_operand value...) then
    //SThen-Block
```

```
#elseif ... then
    //ElIf-Block
#else
    //Else-Block
#endif
```

1. #if keyword and the first condition are bound tightly.
2. then keyword and the last condition are bound tightly.
3. Values are bound loosely into each other.
4. Every and and or boolean operator is bound tightly to the following value.
5. not keyword that is part of value is bound tightly into that value.
6. #if, #elseif, #else and #endif keywords must be placed on separate lines.
7. Every other Syntactic Unit is bound tightly to each other.

5.3 Control flow modifiers

5.3.1 exitwhen

Exitwhen is special keyword followed by expression, that is evaluable into boolean result.
If the boolean result evaluates into true, the control flow structure is immediately stopped and the code execution continues after the end of given control flow structure. Exitwhen can only appear inside loop(See 5.4 – Loop)

5.3.1.1 White spaces

```
exitwhen boolean_expression
```

1. Exitwhen and boolean_expression are bound tightly.
2. Boolean_expression must follow its respective rules for white spacing.
3. The whole expression must be placed on separate line from any other code.

5.3.2 break

When break is hit inside control flow structure, the control flow structure immediately ends its execution and lets the code after it run. Break is equal to exitwhen true. Break can only appear inside Loop, While and For.

5.3.2.1 White spaces

```
break
```

1. The Break expression must be placed on separate line from any other code.

5.3.3 continue

When continue is hit inside control flow structure, the code immediately jumps back into the beginning of given control flow structure. Continue can only appear inside Loop, While and For.

5.3.3.1 White spaces

continue

1. The Continue expression must be placed on separate line from any other code.

5.4 Loop

Loop is control flow structure, that allows certain code to run multiple times, according to some condition.

5.4.1 Syntax

```
loop
    opt:(expressions)
endloop
```

Example:

```
loop
    call A(myInteger)
    set i = i + 1

    if (myInteger == 5) then
        call B()
        set myInteger++
        continue
    endif

    exitwhen myInteger > 10
    set myInteger += 1
endloop
```

Loop-Body is block of code within loop and endloop keywords.
Exit-Cond is condition after exitwhen keyword.
Break-Expr is expression of type break.
Continue-Expr is expression of type continue.
Loop-Body can be filled with any number of valid expressions, including 0, as well as with 3 special expressions: exitwhen, break keywords and continue.
The Loop-Body will be executed until any of the encountered Exit-Cond evaluate to true, or until Break-Expr is hit, or until the OP limit is reached.
Exit-Cond, Break-Expr and Continue-Expr are bound into the closest higher loop block.
One loop block can have any number of Exit-Cond, Break-Expr and Continue-Expr, and they can be nested into other eJass constructs, such as if block.

5.4.1.1 White spaces

```
loop
    opt:(expressions)
endloop
```

1. loop keyword, as well as endloop keyword must be placed on separate lines from any other code.
2. Every expression must follow rules of corresponding white spacing rules, so for expression of type function call, the rules listed in 4.2.2.1 - White spaces.
3. Exit-Cond as well as Break-Expr expressions must be placed on separate lines.
4. Condition after exitwhen keyword must follow the same rules as conditions inside If-Block(5.1.1 - Syntax) and additionally the first expression of the condition is bound tightly into exitwhen keyword. Special case is, when the first expression is parenthesis pair, which must begin on the same line as the preceding exitwhen.

5.5 While loop

While loop is similar control flow structure as Loop, but has only one condition, which gets evaluated after each iteration of the body.

5.5.1 Syntax

```
while (condition)
    opt:(expressions)
endwhile
```

Example:

```
    while (myInteger <= 10)
        call A(myInteger)
        set i = i + 1
        set myInteger += 1
    endwhile
```

While-Body is block of code between while and endwhile keywords.
While-Cond is condition following while keyword.
While-Body gets executed as long as While-Cond evaluates to true.
While-Cond must evaluate, or be implicitly convertible into boolean.
Result of While-Cond is checked on all iterations, including the first, before entering While-Body and gets checked after every iteration before entering While-Body on next iteration.

5.5.1.1 White spaces

```
while (condition)
    opt:(expressions)
endwhile
```

1. while as well as endwhile keywords must be placed on separate lines from any other code.
2. All expressions inside While-Body must follow corresponding white spacing rules, so for expression of type function call, the rules listed in 4.2.2.1 - White spaces.
3. Condition after while keyword must follow the same rules as conditions inside If-Block(5.1.1 - Syntax) and additionally the first expression of the condition is bound tightly into opening parenthesis.
4. while keyword is bound tightly into opening parenthesis.
5. Every other Syntactic Unit is bound tightly into each other.

5.6 For loop

For loop is a control flow structure that allows certain code to run multiple times, depending on some condition.

5.6.1 Syntax

```
for (opt:[initializer_list]; opt:[condition]; opt:[stepper_list])
    opt:(expressions)
endfor
```

Example:

```
for(integer i = 0, integer j = someMaxValue; i < j; ++i)
    call uberFunction(myEvenMoreUberArray[i])
endfor
```

For-Init is code block from opening parenthesis until first semicolon.
For-Cond is code block from first semicolon until second semicolon.
For-Step is code block from second semicolon until the closing parenthesis.
For-Body is code between the closing parenthesis and endfor keyword.

For-Init, For-Cond as well as For-Step are optional, and can be left empty for no code in that part specified.

For-Init runs only once, before the first execution of For-Body.
For-Cond runs before every iteration inside For-Body, including after For-Init.
For-Step runs after every iteration inside For-Body.

Iteration over For-Body stops only when For-Cond evaluates to true.

Initializer_list serves to initialize variables used inside for body, and therefore variables defined inside Initializer_list cannot be referenced by name from outside the For-Body, For-Cond, For-Step or For-Init.
Initializer_list can not have free standing function calls or set operations.
Variables defined inside Initializer_list are shadowing any other variables with the same name.
Variables defined inside Initializer_lsit must be separated by comma from each other.
All Variables defined inside Initializer_list must have their type specified, omitting local or temporary keyword. All Variables defined inside Initializer_list are local, and can not be declared as temporary.

Condition must be evaluable, or implicitly convertible into boolean value.

Stepper_list servers to advance state of variables. Stepper_list can perform any number of calls of any function or set operations, but can not define variables.

All three For blocks(For-Init, For-Cond and For-Step) are optional, and they can be provided in

any combination. The semicolons, are however not optional, and must be provided even with any, or all empty blocks.

5.6.1.1 White spaces

```
for (opt:[initializer_list]; opt:[condition]; opt:[stepper_list])
    opt:(expressions)
endfor
```

1. for keyword is bound tightly into opening parenthesis.
2. opening parenthesis is bound to 'local' keyword, if any variables are provided, otherwise the semicolon following initializer_list is bound tightly to opening parenthesis.
3. All variables declared inside initializer_list must be separated by comma, and are bound loosely to each other. However, comma after every variable defined, but the last one, is bound tightly to the variable defined before it.
4. Variable definition must follow the same rules as normal variable definition(3.2 Local Variable)
5. Condition is bound loosely to the previous semicolon.
6. Condition is bound tightly to the following semicolon.
7. If no condition is provided, semicolons are bound loosely to each other.
8. First expression of Stepper_list is bound tightly to previous semicolon.
9. If no Stepper_list is provided, the closing parenthesis is bound loosely into second semicolon.
10. Stepper_list expressions must be separated with commas, and are bound loosely to each other.
11. Expressions inside Stepper_list must follow their respective rules, including white spacing.
12. Every other Syntactic Unit is bound tightly into each other.

6. Scope

Scope is eJass construct that allows encapsulating other eJass constructs. Visibility-Qualifiers can be applied where applicable on eJass constructs inside scope.
Scope can contain any other eJass construct, except library and every eJass construct inside must follow respective rules defined for given eJass construct.

6.1 Syntax

```
opt:Visibility-Qualifier scope scope_name opt:(initializer init_function_name)
    Scope-Body
endscope
```

Example:

```
scope MyScope
    function a takes nothing returns nothing
    endfunction
endscope

scope MyOtherScope initializer i
```

```
        private function i takes nothing returns nothing
            call MyScope.a()          //See 1.3.2 - Scope resolution
        endfunction
    endscope
```

Visibility-Qualifier is only usable if given scope is inside another scope, or library.
Scope can declare exactly one function from inside the Scope's body as its initializer. This function is ran exactly once(unless explicitly called) according to 1.14 - Initialization rules.
Code from within scope can be accessed using scope resolution operator.

6.1.1 White spaces

```
opt:(Visibility-Qualifier) scope scope_name opt:(initializer
                                                  init_function_name)
    Scope-Body
endscope
```

1. Visibility-Qualifier, if specified, is bound tightly into scope keyword.
2. scope keyword is bound tightly into scope_name.
3. Scope_name is bound tightly into initializer keyword, if provided.
4. initializer keyword is bound tightly into init_function_name, if provided.
5. endscope keyword must be placed on separate line from other code.

7. Library

Library is eJass construct, similar to scope, which allows encapsulation of other eJass constructs.
Library can contain every other eJass construct, excluding another library and every eJass construct inside library must follow its respective rules.
Libraries can only be placed in Global Scope.

7.1 Syntax

```
library lib_name opt:(initializer init_func_name) opt:(requires/uses/needs
                                                    opt:[optional] required_lib_name)
    Library-Body
endlibrary
```

Example:

```
    library MyLibrary initializer i uses A, optional B, C
        //Code Code Code
    endlibrary
```

Library can declare exactly one function or static method from inside the Library's body as its initializer. This function is ran exactly once(unless explicitly called) according to 1.14 - Initialization rules.
Library can also declare that it wants to use eJass constructs from another library with keyword uses, needs or requires. All required libraries must have names of existing library anywhere in the Global Scope, and their names can be pre-placed with optional keyword, meaning that if such

library does not exist inside Global Scope, no compiler error will be issued. If more than one required library is specified, the libraries must be separated with commas. All required libraries, however, must not depend in any way on the library that depends on them, therefore creating circular dependencies.
Code from library can be accessed using scope resolution operator.
All code from library is moved to the top of the map script, after all required libraries so that any other library requiring this one and every other code(in or outside of scope) can use code from given library.

7.1.1 White spaces

```
library lib_name opt:(initializer init_func_name) opt:(requires/uses/needs
                                                    opt:[optional] required_lib_name)
    Library-Body
endlibrary
```

required library entry:
`opt:(optional) lib_name`

1. library keyword is bound tightly into lib_name.
2. Lib_name is bound tightly into initializer keyword, if provided.
3. initializer keyword is bound tightly into init_func_name, if provided.
4. If needs, uses or requires keyword is provided, it is bound tightly into the previous Syntactic Unit.
5. If needs, uses or requires is provided, it is bound tightly into the first required library entry.
6. optional keyword is bound tightly into the following lib_name.
7. Required_library_name is bound tightly with the following comma, if any more required libraries are provided.
8. Required library entries are bound loosely into each other.
9. Every other Syntactic Unit is bound tightly into each other.

8. Struct

Struct is a used-defined type, which can contain other types as well as methods describing its behaviour and operators for easier access to members, or overloading existing operators for user-friendly syntax.
Structs automatically generate several methods, see 8.6.5 - Automatically generated methods.
All members as well as methods defined inside struct can be accessed using scope resolution operator.

8.1 Syntax

8.1.1 Definition

```
opt:Comp-Qualifier opt:Visibility-Qualifier struct struct_name
opt:(uses array/hashtable) opt:(final) opt:(extends parent_list)
    Struct-Body
```

```
endstruct
```

Example:

```
struct A
    ...
endstruct

struct B uses array
    ...
endstruct

struct C extends A
endstruct

struct Q final extends C
endstruct
```

If Comp-Qualifier is used, the uses array/hashtable is only considered if given struct is used in runtime environment and has no effect in compiletime environment.
Visibility-Qualifier is only usable, if given struct is defined inside Scope other than Global Scope.
If struct is marked as final, it can no longer be inherited(see 8.10 - Inheritance and Polymorphism).
Structs can extend from any number of parent structs(see 8.10 - Inheritance and Polymorphism).
Structs automatically generate several methods when use is requested(see 8.6 - Automatically generated methods).
Uses keyword marks what will the struct use to store its indices in. If none is provided, implicit value of array is assumed.
Struct that uses array will only allow at 8190 instances concurrently allocated. Struct that uses hashtable allows any number of instances at the same time. If non-static array member variable is defined with size N then maximum number of simultaneously allocated instances is floor(8190 / N) or infinite if struct uses hashtable.
If struct uses hashtable or array, all structs that inherit from it will also use hashtable or array, regardless of type provided in uses.

8.1.1.1 White spaces

```
opt:Comp-Qualifier opt:Visibility-Qualifier struct struct_name
opt:(uses array/hashtable) opt:(final) opt:(extends parent_list)
    Struct-Body
endstruct
```

1. If provided, Comp-Qualifier is bound tightly into following Syntactic Unit.
2. If provided, Visibility-Qualifier is bound tightly into both preceding and following Syntactic Unit.
3. struct keyword is bound tightly into struct_name.
4. Struct_name is bound tightly with uses keyword, if provided.
5. uses keyword is bound into array/hashtable modifier tightly.
6. final keyword is bound tightly into preceding Syntactic Unit and tightly to following(if any).

7. If extend_list is provided, the keyword extends is bound loosely into the first entry from parent_list.
8. Every entry in parent_list is bound loosely to the preceding Syntactic Unit, tightly into the following comma(if any) and loosely to the following entry from parent_list(if any).
9. Every other Syntactic Unit is bound tightly into each other.

8.1.2 Member access

```
opt:(this/thistype) opt:( . ) member_name
name.member_name
```

Both this variable, respectively thistype keyword and dot are optional. If access to members of different struct is required, either name of the struct must be provided, in case of access to static members, or name of object of given type must be specified, in case of access to either static or non-static members.
Member_name must be valid member from inside the referenced struct.
Name, as well as this/thistype can be enclosed in any number of brackets, meaning that `((Soemthing.some).else).e = 5` is valid code.

8.1.2.1 White spaces

```
opt:(this/thistype) opt:( . ) member_name
name.member_name
```

1. If provided, this variable/thistype keyword is bound tightly into the following dot.
2. If provided, the dot is bound tightly into member_name.
3. If second version is used, name is bound tightly into dot, and dot is bound tightly into member_name.
4. If parenthesis enclose 'name' or this, or thistype, the bracket-scope is bound tightly into following dot.

8.2 Variable definition

See 3.5 - Struct variable.
All variables with initial values are initialized when instance is allocated.

8.2.1 this variable

While inside methods of struct, the currently running instance of the object of the struct can be referenced using this variable.

this variable is only implicitly available from inside non-static methods, and inside static methods, it must be declared explicitly, or it cannot be used. **[Please note that this behaviour will be removed in later verisons.]**

No variable defined inside body of struct can be called this.
Every member variable must have such unique name, that there is no other member variable or

method defined with the same name inside the same struct.

Example:

```
struct A
    integer q

    method myMethod takes nothing returns nothing
        set this.q = 5
    endmethod

    static method myStaticMethod takes nothing returns nothing
        local thistype this = someArrayOfA[56]
        set this.q = 5
    endmethod
endstruct
```

8.3 Methods

Method is a function inside struct, which defines specific behaviour of struct. Methods can either be static, or non-static. Static methods define specified, shared behaviour for all instances, whereas non-static methods can work on instances independently. Methods also obey rules specified in 4.3 - Function overloading.

8.3.1 Syntax

8.3.1.1 Definition

```
opt:(deprecated) opt:(inline)
opt:(Visibility-Qualifier) opt:(Comp-Qualifier) opt:(Const-Qualifier)
opt:(Staticity-Qualifier) opt:(override) opt:(stub)
method opt:(operator) takes opt:(arguments) returns
opt:(Comp-Qualifier) opt:(Const-Qualifier) return_type
    Method-Body
endmethod
```

Arguments are of following form:

```
opt:Comp-Qualifier opt:Const-Qualifier type_name1 variable_name1
opt:(= Initial_value) opt:(, opt:Comp-Qualifier opt:Const-Qualifier type_name2
variable_name2 opt:[= Initial_value] ...)
```

Example:

```
private static method m takes nothing returns nothing
    //code
endmethod

stub method q takes nothing returns integer
    return 5
endmethod

method operator += takes integer i returns thistype
```

```
        set this.storage = this.storage + 1
        return this
    endmethod
```

For override keyword, as well as stub methods, check 8.10 – Inheritance and Polymorphism.
Deprecated and inline keywords follow the same rules as in function definition(see 4.2 - Definition Syntax).
Methods and method operators defined as static can not use implicit variable this, unless defined explicitly inside the body of the method.**[Note: Explicit declaration of this, therefore shortened or normal syntax involving this operator will be removed in future version]**

If method takes no arguments, then nothing shall be wrote between takes and returns. If method takes arguments, then they must be fully defined and they must be separated by commas.
If method does not return anything, nothing must be used as return type. Otherwise all returned values of given method must be implicitly convertible into specified return type, but they don't need to be of same type.
Methods marked as constant can be called from both constant and non-constant instance, and may not modify any global state, or state of the object that the method is called on. The object inside the method is marked as constant.
Static members of struct can not be modified from constant methods. Normal methods can not be called from constant instances of object. Methods can call other methods only if they are defined below the called methods.

8.3.1.1.1 Method argument's implicit values

Method arguments may have implicit values specified, meaning that when the method is called, some arguments may end up unspecified from callers point of view, and have implicit value defined inside the method definition.

```
opt:Comp-Qualifier opt:Const-Qualifier type_name1 variable_name1 opt:(= Initial
Value) opt:(, ...)
```

Example:

```
    ... takes compiletime integer q, constant integer i, unit u = null
    returns ...
```

Implicit value of variable N may only be specified, if all method's arguments to the right from N already have implicit value specified, meaning that takes integer i = 5, unit u returns is malformed.
Implicit value can also be return value of method, in which case, return value of method must be implicitly convertible to the type of variable setting implicit value for, or must be cast explicitly to given type.

8.3.1.1.2 White spaces

```
opt:(deprecated) opt:(inline)
```

```
opt:(Visibility-Qualifier) opt:(Comp-Qualifier) opt:(Const-Qualifier)
opt:(Staticity-Qualifier) opt:(override) opt:(stub)
method opt:(operator) takes opt:(arguments) returns
opt:(Comp-Qualifier) opt:(Const-Qualifier) return_type
    Method-Body
endmethod
```

1. If both deprecated and inline keywords are specified, they are bound loosely.
2. If Visibility-Qualifier is specified, and inline keyword is specified, they are bound tightly.
3. If Comp-Qualifier is specified, it is bound loosely to preceding Syntactic Unit.
4. If Const-Qualifier is specified, it is bound tightly into preceding Syntactic Unit and loosely to following.
5. If Staticity-Qualifier is provided, it is bound loosely into preceding and tightly to following Syntactic Unit.
6. override and stub keywords are bound together, if either provided. If not, either of provided keyword is bound tightly into preceding Syntactic Unit and is bound loosely into following Syntactic Unit.
7. operator keyword, if provided is bound tightly into method keyword.
8. Preceding keyword is bound tightly to the name of method, which is bound tightly to the takes keyword.
9. all arguments are bound loosely to each other with special rules(see 4.2.1.3 - Argument's White spaces).
10. returns keyword is bound loosely to the last argument specified in argument's list or to type nothing otherwise.
11. Return type with its Qualifiers is bound loosely to returns keyword.
12. If Const-Qualifier and/or Comp-Qualifier is specified for return type, it is bound tightly with the return type and to each other.
13. Every other Syntactic Unit is bound tightly to each other.

Argument's White spaces

```
opt:Comp-Qualifier opt:Const-Qualifier type_name1 variable_name1
opt:(= Initial_value) opt:(, opt: opt:Const-Qualifier type_name2 variable_name2
...)
```

1. Comp-Qualifier and Const-Qualifier are bound tightly together, and if any or both provided, they are bound tightly to name of argument's type.
2. Argument's name is bound tightly to name of argument's type.
3. Initial_value, as well as = is bound tightly into variable_name, if provided.
4. Comma following argument N, if specified is bound tightly into argument's name.
5. Every independent argument is bound loosely to each other.

8.3.1.2 Calling

```
call opt:(this) opt:( . ) method_name(opt:arguments)
call variable.method_name(opt:arguments)
```

Arguments are of following form:

```
opt:value1 opt:(, value2 opt:(, value3 ...))
```

Example:

```
call myF(1)
call .myW(1, 2, null)
call this.myQ()
call variable.myZ(
        1,
        2,
        3,
        null
       )
```

If the first calling method is used, and this keyword is provided, the following dot is necessary.
Static methods can be called on this/thistype/struct's name and this invocation will ignore the instance.
Every argument can be enclosed in any number of parenthesis. However, every comma must be inside the top bracket-scope.

Example:

```
call myF(((1)), (2))        //valid
call this.myF(((1), 2))     //invalid, comma before second argument
                            //is not in top bracket-scope
```

8.3.1.2.1 White spaces

```
call opt:(this) opt:( . ) method_name(opt:arguments)
call variable.method_name(opt:arguments)
```

1. Keyword call and following Syntactic Unit are bound tightly into each other.
2. Opening parenthesis is bound loosely with method's name.
3. If no arguments are provided, closing parenthesis is bound tightly with opening parenthesis.
4. If any number of arguments are provided, then the first argument is bound loosely to the opening parenthesis, the last argument is loosely bound to the last parenthesis and arguments are bound loosely between each other.
5. Every other Syntactic Unit is bound tightly to each other.

Argument's White spaces

```
opt:value1 opt:(, value2 opt:(, value3 ...))
```

1. If multiple arguments are provided, then the value is bound tightly to following comma, and following comma is bound loosely to the next value.

8.4 thistype keyword

While inside struct, thistype keyword can be used, which gets transformed into the name of struct currently being parsed. thistype keyword will not be replaced inside strings.

Example:

```
struct QQ
    static method a takes nothing returns thistype
        return thistype.allocate()
    endmethod
endstruct
```

will get converted into

```
struct QQ
    static method a takes nothing returns QQ
        return QQ.allocate()
    endmethod
endstruct
```

8.4.1 Thistype cast

Thistype keyword(as well as user_type, see 8.9.3 - Predefined conversion operations) can also be used to convert unique identifier of type integer of that type into instance.

Example:

```
call functionExpectingInstance(cast<thistype>(5))
            //calls functionExpectingInstance with instance with number 5
```

If instance with given unique identifier is not allocated the values are still affected, and when the instance is allocated it can have malformed values, unless overridden after creation.

8.5 Keyword redundancy

Because both this variable and thistype keyword operate on members and methods of currently parsed struct, they can be omitted. The dot preceding the members inside currently being parsed struct can also be omitted. Both omissions are optional.

Example:

```
struct AA
    integer w = 5
    static integer q = 0
    method myA takes nothing returns nothing
        set w = w + 1                       //1
        set .w = .w + 1                     //equivalent to 1
        set this.w = this.w + 1             //equivalent to 1

        set q = q + 1                        //2
        set .q = .q + 1                      //equivalent to 2
        set AA.q = AA.q + 1                  //equivalent to 2
        set thistype.q = thistype.q + 1      //equivalent to 2
    endmethod
endstruct
```

8.6 Automatically generated methods

All automatically generated methods are only actually generated when code requires usage of given methods.

8.6.1 private static method allocate takes nothing returns thistype

Allocate is static method, that takes nothing and returns new, unique instance of given type. This method is private, so it cannot be called from outside the struct. This method is un-overloadable. This method is not generated unless used.

8.6.2 private method deallocate takes nothing returns nothing

Deallocate is static method which pairs up with allocate. Deallocate takes the running instance and puts it back into bucket of free instances, effectively destructing the instance. This method is private, so it cannot be called from outside the struct. This method is un-overloadable. This method is not generated unless used.

8.6.3 private static method onInit takes nothing returns nothing

onInit is a static method, which is executed with script's initial initialization. If no onInit is provided, there is no execution for given struct. This method can be overloaded and any attempt in doing so will result in the automatically generated to not be generated. Overloaded onInit method may return any value, must not have any parameters and must be static. Overloaded onInit method can be declared as private and will still be executed with initial execution.

8.6.4 legacy create and destroy

For backwards compatibility reasons, a default `static method create takes nothing returns thistype` and `method destroy takes nothing returns nothing` are generated when not overloaded and use of them is requested, which forward call allocate and deallocate respectively.**[These methods will be removed from list of automatically generated methods in future version]**

8.7 Constructor and Destructor

Constructor and destructor are special methods which are called when instance of struct is created and destroyed respectively. When constructor is defined, allocate is also generated, and when destructor is defined, the deallocate method is generated.

8.7.1 Syntax

8.7.1.1 Declaration

```
opt:(Const-Qualifier) constructor opt:(takes arguments)
    Constructor-Body
```

```
endconstructor

opt:(Const-Qualifier) destructor
    Destructor-Body
enddestructor
```

Example:

```
struct A
    private unit u
    private boolean b
    constructor
        set u = CreateUnit(Player(0), 'hfoo', 0, 0, 0)
        set b = true
    endconstructor

    constructor takes unit u
        set this.u = u
        set b = false
    endconstructor

    destructor
        if b then
            call KillUnit(u)
        endif
    enddestructor
endstruct
```

Constructor can be overloaded like normal functions and may take any number of any arguments. Destructor can be overloaded exactly once and can not take any arguments. If they take no arguments, a takes nothing is optional.
Constructor and destructor never specify return types.
Constructor can additionally have arguments with implicit values. These follow the same rules as normal method invocation.
Constructor implicitly calls underlying .allocate(), which can be provided in allocators(See 12. - Allocator) and after executing it returns allocated instance into caller. Created instance can be pointed at with this variable.
Destructor always calls .deallocate(), which can be provided in allocators(See 12. - Allocator) after executing the body of destructor.
When destructor of any struct is executed, it calls destructors of all parent types(See 8.10 – Inheritance and Polymorphism) as well as all member variables.
If no user-defined constructor is provided and is requested, a default constructor with no arguments is generated with empty body. If no user-defined destructor is provided and is requested, a default destructor with empty body is generated. Additionally, if no user-defined constructor which takes exactly 1 integer is provided, a implicit one is generated which returns instance with id equal to argument it is called with.
If constructor is marked as private, the constructor can not be run outside of the struct. Same rule applies for destructor.

Example:

```
local A a = thistype(5)
local integer i = a.someInteger
local A b = thistype(i).someInteger
```

Mixing crate call and destructor, or constructor and destroy is only safe if:

1. First line of create contains local thistype this = allocate() and on every possible path of execution returns given value.
2. Destroy's last operation before return statement or falling off of scope is call deallocate().

8.7.1.1.1 White spaces

```
constructor opt:(takes arguments)
    Constructor-Body
endconstructor

destructor
    Destructor-Body
enddestructor
```

1. constructor, destructor, endconstructor and enddestructor must be placed on separate lines.
2. If arguments are provided for constructor, then constructor is bound tightly into takes keyword and separate arguments follow the same rules as in arguments for normal method(See 8.3.1.1 - Definition).

8.7.1.2 Invocation

Construction: `call MyType(opt:[args]), call thistype(opt:[args])`
Destruction: `destruct expression, destruct this`

Example:

```
struct S
    constructor
    endconstructor

    constructor takes integer a
    endconstructor

    destructor
    enddestructor
endstruct

local S s = S()
local S q = S(5)
//do work with s
destruct s
```

The de-facto function called should always be the same as the name of the construction object.
To destruct instance of object, user must use keyword destruct followed by expression that evaluates into some user-defined type.

8.7.1.2.1 White spaces

```
call MyType(opt:[args])
destruct expression
```

1. Normal function call rules must apply.
2. Keyword destruct is bound tightly into following expression.

8.7.1.3 Construction of parent structs

Structs can explicitly declare which constructor to use when constructing parent structs.

8.7.1.3.1 Syntax

```
construct ParentA(parent1args), ParentB(parent2args)...
```

Example:

```
struct D extends A, C, B
    constructor
        construct A(5), B("string"), C()
    endconstructor
endstruct
```

Construct is only valid if it is placed before any other expression inside constructor of substruct.
If Construct is not provided, a default constructor is called for each parent. If list of parents after Construct does not include any parent structs, all missing parent structs are constructed using default constructor.
Struct names can be in any order and will not change the order of construction of individual structs(See 8.10.3 - Order of creation and destruction of objects). this variable can not be used inside construct expression, because the type is not yet fully constructed.

8.7.1.3.1.1 White spaces

```
construct ParentA(parent1args), ParentB(parent2args)...
```

1. construct is bound tightly into the first parent struct's name.
2. Calls to parent structs must following rules of constructor call(See 8.7.1.2.1 - White spaces).

8.8 Method operator

8.8.1 List of overloadable operators

	Operator declaration	Example of usage
1.	`method operator +`	`set lhs = instance + rhs`
2.	`method operator -`	`set lhs = instance - rhs`
3.	`method operator *`	`set lhs = instance * rhs`
4.	`method operator /`	`set lhs = instance / rhs`
5.	`method operator %`	`set lhs = instance % rhs`
6.	`method operator ++`	`set instance++`

```
7.  method operator --          set instance--
8.  method operator +=          set instance += rhs
9.  method operator -=          set instance -= rhs
10. method operator *=          set instance *= rhs
11. method operator /=          set instance /= rhs
12. method operator %=          set instance %= rhs
13. method operator <<          set lhs = instance << rhs
14. method operator >>          set lhs = instance >> rhs
15. method operator <<=         set instance <<= rhs
16. method operator >>=         set instance >>= rhs
17. method operator ==          instance == rhs   lhs == instance
18. method operator !=          instance != rhs   lhs != instance
19. method operator <=          instance <= rhs
20. method operator >=          instance >= rhs
21. method operator <           instance < rhs
22. method operator >           instance > rhs
23. method operator !           !instance
24. method operator []          call SomeFunction(instance[argument])
25. method operator []=         set instance[argument] = argument2
26. static method operator []   call SomeFunction(StructName[argument])
27. static method operator []=  set StructName[argument] = argument2
```

Note: instance is always of user-defined type, while rhs and lhs are of undefined type(depending on signature of operator).
Arguments of operators:

1. Operators 1-5, 8-22, 24 and 26 require exactly one argument.
2. Operators 6, 7 as well as 23 require no parameters.
3. Operators 25 and 27 require exactly two arguments.

Return values of operators:

1. Operators 1-5, 8-12, 15, 16, 25 and 27 may not return any value.
2. Operators 6-7, 13, 14, 17-24 and 26 must return a value.

Boolean operators(17-22) can be performed on implicitly convertible types without need of such method operator to exist.
If operator 23 is not provided, implicit version is generated, returning whether given instance of user-defined type was allocated successfully.
Assignment operator is not overloadable, but is always implicitly declared(See 8.9.3 - Predefined Conversion operators) for every type to accept the same type or implicitly convertible types.

For example:

```
local A a = A()
local A b = a

local integer i = 5
local real r = i
```

is valid code.

8.9 Conversion operator

Conversion operators allow conversion from one user-defined type to another, or to native type(either basic or non-basic).

8.9.1 Declaration Syntax

```
method operator ValidType
    operator-body
endmethod
```

Conversion operators never take any arguments, and they must always return value that is implicitly convertible into ValidType on all paths of execution. Only one conversion operator to given type can exist inside one struct.
ValidType must be fully defined at point of method operator definition.
Every instance of user-defined object can be implicitly converted to boolean, checking if given instance was allocated successfully.

8.9.1.1 White spaces

```
method operator ValidType
    operator-body
endmethod
```

1. method and operator keywords are bound tightly.
2. operator keyword is bound tightly with ValidType.
3. There must be any number of white spaces ending with new line character after ValidType.
4. endmethod keyword must be placed on separate line.

8.9.2 Invocation Syntax

```
cast<ValidType>(expression)
```

Conversion operator is always explicit, and therefore the type to convert to must be fully mentioned at all times.
Attempt to convert resulting type of expression into ValidType happens at compile-time, and if expression can not be directly converted into ValidType, or into type that is implicitly convertible into ValidType, then an error shall be issued.

8.9.2.1 White spaces

```
cast<ValidType>(expression)
```

1. Every Syntactic Unit is bound tightly into each other.

8.9.3 Predefined Conversion operators

For ease of use, few conversion operators are generated implicitly:

```
1. method operator string(integer)            //equal to I2S
```

```
2. method operator string(real)                //equal to R2S
3. method operator integer(string)             //equal to S2I
4. method operator real(string)                //equal to S2R
5. implicit method operator integer(real)      //equal to R2I
6. method operator real(integer)               //equal to I2R
7. method operator thistype(integer)           //thistype(5)
8. method operator user_type(integer)          //YourStruct(5)
9. implicit method operator boolean(thistype) //if (this) then
```

Implicit operator means that it is not required explicitly to cast one type to another, but the conversion happens implicitly.

Special Conversion Operators:

```
7. Alternative: thistype(integer_value)
8. Alternative: user_type(integer_value)
```

This cast operators are deprecated and outdated, and are kept for backwards compatibility.**[These operators will be removed from the standard in future version]**
Proper operators are `cast<thistype>(integer_value)` and `cast<user_type>(integer_value).`
If user-defined conversion operator is provided for integer, both 7 alternative and 8 alternative will call that operator instead of doing direct conversion from integer to instance.

8.9.4 Predefined constructors

For ease of use, few predefined constructors are generated implicitly:

```
1. integer.constructor(thistype)
2. real.constructor(integer)
3. integer.constructor(real)
4. string.constructor(integer)
5. string.constructor(real)
6. boolean.constructor(thistype)
```

Example:

```
local MyType m = MyType()
local integer q = integer(m)                    //calls 1
local real w = real(q)                          //calls 2

local string o = string(w)                      //calls 5
local string p = string(q)                      //calls 4

local boolean isAllocated = boolean(m)          //calls 6
```

When any of these constructors is used, the compiler will not cast the value into given type.
For instance, the second line of example above will not try to call conversion operator even if one is defined for given struct.

8.10 Inheritance and Polymorphism

Polymorphism allows exact code implementation to be hidden under certain signature.
To make struct polymorphic in eJass, it must be inheritable(see 8.9.2 - Rules of Inheriting). Only stub methods(see 8.11 - Stub Method) can have polymorphic behaviour. Inheriting from a base struct allows the struct to derive methods and member variables from its parents.

8.10.1 Syntax for Inheriting

```
struct A extends Parent1, Parent2, ...
```

One struct can have any number of direct parent structs.

8.10.2 Rules of Inheriting

1. Struct that is marked as final can not be inherited
2. Only public members and variables are derived from parent struct
3. None of create and destroy methods are inherited
4. Native types and non-basic types may not be derived

8.10.3 Order of creation and destruction of objects

All structs are constructed in top-down order, meaning that the top-most parent struct(struct that does not inherit anything) will be constructed first, followed by its children followed by their children until all parent structs are constructed.
If there exists a diamond-shape(B and C inherit A and D inherits B and C), only one instance of A will be created when object D is created, and the creation depends on the order of declaration in extend list.
Destruction happens in down-top order, meaning that the destructing struct is destructed first, and after that its parents destruct themselves and parents of parents, until all top-most structs are destructed.
If there exists a diamond-shape(B and C inherit A and D inherits B and C), A-part of object D will only be destroyed once.
Construction always goes in the same order as in extend list and destruction in the opposite order.

Example:

```
struct A extends B, C
```

When constructing instance of A, B will run its consructor and after that C will run its constructor. When destructing instance of A, C will run its destructor and after that B will run its destructor.

8.11 Stub Method

Stub method is such method, that can be overridden(don't confuse with overloaded) in children of implementing struct. Stub methods follow rules of normal methods for both white spacing and declaration, as well as invoking.

When calling stub method in derived struct that does not override the stub method is invoked, the nearest parent with stub method with given name and signature will be executed.
Templated methods can not be declared as stubs.

8.11.1 Rules of overriding

1. Overridden method must have the exact same arguments as well as return type, and must have the same Const-Qualifier as the overriding method, otherwise the method is merely a overload.
2. Overridden method does not need to have the same Visibility-Qualifier.
3. Overridden method does not need to be stub.
4. static methods can not be overridden, therefore can not be declared as stub methods.
5. destroy and create methods can not be overridden, therefore can not be declared as stub methods.

8.12 override method

Override method is such method, that will try to override parent's stub method. If no overridable stub method with given signature and return type exists in any of parents, an error is issued.
Override method does not need, but can, be stub method.

Example:

```
struct A
    stub method a takes integer i returns nothing
    endmethod
endstruct

struct B extends A
    override method a takes real r returns nothing
        //OOPS! wrong type, compilation error
    endmethod
endstruct
```

8.13 super variable

Special, reserved variable super is used when manual invocation of parent's method that is overridden in struct declaring running method is requested.

Example:

```
struct A
    stub method m takes integer a returns nothing
        call BJDebugMsg(cast<string>(a))
    endmethod
endstruct

struct B extends A
    method m takes integer a returns nothing
        call BJDebugMsg("Son running parent")
```

```
            call super.m(a)
        endmethod
    endstruct
```

If more than one parent have stub method with the same signature then conversion cast must be used on super variable.

Example:

```
    struct A
        stub method m takes integer a returns nothing
            call BJDebugMsg(cast<string>(a))
        endmethod
    endstruct

    struct B
        stub method m takes integer a returns nothing
            call BJDebugMsg("B: " + cast<string>(a))
        endmethod
    endstruct

    struct C extends A, B
        method m takes integer a returns nothing
            call (cast<A>(super)).m(a)        //calls A.m, outputs "a"
            call (cast<B>(super)).m(a)        //calls B.m, outputs B: "a"
        endmethod
    endstruct
```

8.14 Visibility-Block

Visibility-Block allows different variables as well as methods to be grouped under single Visibility-Qualifier without need to specify it.

8.14.1 Syntax

```
private:
public:
...
endblock
```

Example:

```
    struct S
        private:
        static method onInit takes nothing returns nothing
        endmethod
        //onInit is private

        public method q takes nothing returns integer
            return 5
        endmethod
        //q is public
```

```
        endblock
    endstruct
```

Starting and ending of Visibility-Block must be placed outside of any method's body.
If method or variable declares different Visibility-Qualifier than enclosing Visibility-Body does, visibility is set to the Visibility-Qualifier set for each method or variable individually. Alternation in Visibility-Body can happen before ending of Visibility-Body.

Example:

```
    private:

    static method operator [] takes integer i returns integer
        return i
    endmethod

    public:

    method q takes nothing returns nothing
    endmethod

    endblock
```

8.14.1.1 White spaces

```
private:
public:
...
endblock
```

1. ':' is bound tightly into prceeding keyword.
2. endblock must be placed on separate line from other code.

9. Interface

9.1 Syntax

9.1.1 Definition

```
interface name
    Interface-Body
endinterface
```

Interface, unlike struct, can not extend from any other interface or struct. Every variable declared inside interface must follow rules of declaring variables inside struct(See 3.5 - Struct Variable). Every method declared inside interface must follow rules of method declaration, excluding endmethod keyword(See 8.3 - Methods).
Children of interfaces are non-instancable, unless they provide implementation of all methods that do not default and they cannot use thistype casts as well. If direct children of interface does not

provide implementation of all methods that are not default, but any of its children do provide body for the methods, then the children can be instantiated.

9.1.1.1 White spaces

```
interface name
    Interface-Body
endinterface
```

1. interface keyword is bould tightly into name of the interface
2. Interface-Body must be separated by at least one new line character at both beginning and end
3. endinterface keyword must be separated from anything else.

9.1.2 Variables

See 8.2 - Variable Definition

9.1.3 Methods

```
opt:(Visibility-Qualifier) opt:(Comp-Qualifier) opt:(Const-Qualifier)
opt:(Staticity-Qualifier) method opt:(operator) name
takes opt:(arguments) returns opt:(Comp-Qualifier) opt:(Const-Qualifier)
                                    return_type opt:(defaults (impl_value))
```

Example:

```
      method m takes integer i, integer a returns real
      method q takes integer i, integer c returns integer defaults (i * c + 5)
```

Arguments are of following form:

```
opt:(Comp-Qualifier) opt:(Const-Qualifier) type_name variable_name
opt:(= Initial_value) opt:(, arg2...)
```

Visibility-Qualifier is ignored if provided.

Deprecated, nor inline keywords can be specified for methods declared inside interface.
Method declared inside interface follows the same rules as method declared inside struct(see 8.3 – Methods). These methods do not need to provide stub keyword, and it will be rejected with syntax error if one is provided, because they are implicitly overridable. Overridable methods in children of interface can be declared as stub but are not needed to be.
Methods declared inside interface can also default to some calculable value or returned value from function call that is implicitly convertible into return_type. This expression must be bracket-enclosed. If they do so, children of given interface do not need to provide implementation of given methods, and if none is provided and call for object of any children is attempted, the defaulted value is returned.
Methods declared inside interface can also provide implicit values for arguments, which mark default values for arguments. Implicit values must follow the same rules as implicit values for functions(See 4.2.1.1 - Function argument's implicit values).

Methods declared inside interface may not be templated.
Children types that derive given interface can not declare such methods with different implicit values as well as different number of implicit arguments.

9.1.3.1 White spaces

```
opt:(Visibility-Qualifier) opt:(Comp-Qualifier) opt:(Const-Qualifier)
opt:(Staticity-Qualifier) method opt:(operator) name
takes opt:(arguments) returns opt:(Comp-Qualifier) opt:(Const-Qualifier)
                                    return_type opt:(defaults (impl_value))
```

1. Comp-Qualifier and Const-Qualifier are bound tightly into each other if provided, and bound loosely to preceding and following Syntactic Unit.
2. Staticity-Qualifier is bound loosely to both preceding and following Syntactic Unit.
3. operator keyword, if provided is bound tightly into method keyword.
4. Preceding keyword is bound tightly to the name of method, which is bound tightly to the takes keyword.
5. all arguments are bound loosely to each other with special rules(see 4.2.1.3 - Argument's White spaces).
6. returns keyword is bound loosely to the last argument specified in argument's list or to nothing keyword otherwise.
7. Return type with its Qualifiers is bound loosely to returns keyword.
8. If Const-Qualifier is specified for return type, it is bound tightly with the return type.
9. Keyword defaults, if provided, is bound tightly into preceeding Syntactic Unit.
10. impl_value is bound tightly into defaults and every sub-expression inside impl_value follows its own rules for white spaceing.
11. Every other Syntactic Unit is bound tightly to each other.

9.1.3.2 Argument's White spaces

```
opt:(Comp-Qualifier) opt:(Const-Qualifier) type_name variable_name
opt:(= Initial_value) opt:(, arg2...)
```

1. If Const-Qualifier is specified, it is bound tightly to name of argument's type.
2. Argument's name is bound tightly to name of argument's type.
3. Initial_value, as well as = is bound tightly into variable_name, if provided.
4. Comma following argument N, if specified is bound tightly into argument's name.
5. Every independent argument is bound loosely to each other.

10. Free standing methods

Free standing methods declare struct-like syntax for Basic and Non-Basic types.

10.1 Syntax

```
opt:(deprecated) opt:(inline) opt:(Visibility-Qualifier) opt:(Comp-Qualifier)
opt:(Const-Qualifier) method Type.method_name takes opt:(arguments)
```

```
returns opt:(Comp-Qualifier) opt:(Const-Qualifier) return_type
    Free-Standing_Method-Body
endmethod
```

Example:

```
private method unit.kill takes string toPrint returns boolean
    local boolean b = KillUnit(this)
    if b then
        call BJDebugMsg(toPrint)
    endif
    return b
endmethod
```

this variable always stores the object for running instance(for instance inside free standing method for unit, this will refer to the unit that the method was called on).
Free standing methods may not be declared inside structs.
Rules other than mentioned above as well as white spacing of Free standing methods are the same as member methods(See 8.3 – Method).
If Free standing method is declared as private, it will only be usable inside that Scope.
Free standing method can not be declared as static, cannot be declared as stub or override and cannot be operator.

10.1.1 White spaces

```
opt:(deprecated) opt:(inline) opt:(Visibility-Qualifier) opt:(Comp-Qualifier)
opt:(Const-Qualifier) method Type.method_name takes opt:(arguments)
returns opt:(Comp-Qualifier) opt:(Const-Qualifier) return_type
    Free-Standing_Method-Body
endmethod
```

1. Free-standing methods follow the exact same rules of white spacing as normal methods do, but Type, "." and method_name are all bound to each other tightly.

11. Module

Module is a piece of code that can be implemented inside struct or another module. Module can contain any lines of code that are also valid inside struct and any number of them.

11.1 Syntax

11.1.1 Definition

```
opt:(Visibility-Qualifier) module module_name
    Module-Body
endmodule
```

Visibility-Qualifier is only applicable if module is defined inside Library(see 7. - Library) or Scope(see 6. - Scope).
Modules can also contain parts of methods and other methods below them, effectively chopping

parts of methods away.
Deprecated feature: Modules can also contain allocate and deallocate methods, which will override generated methods.[This feature will be removed in future version of standard.]
onInit method declared within module will not override onInit method from inside struct that implements given module, but will instead run according to 1.14 - Initializer rules.

11.1.1.1 White spaces

```
opt:(Visibility-Qualifier) module module_name
    Module-Body
endmodule
```

1. Visibility-Qualifier, if provided, is bound tightly into 'module' keyword.
2. module keyword is bound tightly into module_name.
3. endmodule keyword must be separated from anything else.

11.1.2 Usage

```
implement opt:(optional) myModule
```

Example:

```
module M
    static method fromM takes nothing returns nothing
        call BJDebugMsg("5")
    endmethod
endmodule

struct Q
    implement M
endstruct

struct W
    implement M
endstruct
```

Only structs can implement module.
When module is implemented, its code is unfolded into the struct, allowing sets and calls, or even ending blocks to be inside module. If module being implemented has method with same signature as implementing struct, an error is issued.
If optional keyword is provided and module doesn't exist, no compiler error is issued.

11.1.2.1 White spaces

```
implement opt:(optional) myModule
```

1. The whole expression must be placed on a separate line.
2. Every Syntactic Unit is bound tightly into every other Syntactic Unit.

12. Allocator

Allocator is special module that contains allocate and deallocate methods.

12.1 Syntax

12.1.1 Definition

```
opt:(Const-Qualifier) allocator alloc_name
    Allocator-Body
endallocator
```

Example:

```
allocator MyAllocator
    static method allocate takes nothing returns thistype
        return 1
    endmethod
endallocator
```

Allocators can only contain 2 methods:

- `static method allocate takes opt:(arguments) returns thistype`
- `method deallocate takes nothing returns nothing`

Allocator can additionally only contain declaration of variables used inside either allocate or deallocate methods. Both methods must be provided inside allocator, otherwise allocator is malformed and cannot be declared as used(See 11.1.2 - Usage). If such allocator is used, compiler error is issued.

12.1.1.1 White spaces

```
opt:(Const-Qualifier) allocator alloc_name
    Allocator-Body
endallocator
```

1. Every Syntactic Unit is bound tightly to every other Syntactic Unit.
2. endallocator keyword must be placed on separate line, separated from any code.

12.1.2 Usage

```
using opt:(optional) alloc_name
```

Example:

```
allocator A
    ...
endallocator

allocator B
    ...
endallocator
```

```
struct Q
    using B
    using optional A
endstruct
```

Struct must declare usage of allocator before first method, otherwise it is ignored. If allocator is used, but is not present in the script and 'optional' keyword is not provided, an error is issued.
One struct can declare usage of multiple allocators, and the last existing optional or existing non-optional allocator declared as used will be used.

Example:

```
struct S
    using allocator1
    using optional allocator2
endstruct
```

Allocator1 will be used if allocator2 is not present on map. Otherwise Allocator2 will be used.

12.1.2.1 White spaces

```
using opt:(optional) alloc_name
```

1. Every Syntactic Unit is bound tightly to every other Syntactic Unit.
2. The whole line must be placed on separate line, separated from any other code.

13. Alias

Alias masks type name or variable name so that it can be used as another type name or variable name, respectively.

13.1 Syntax

```
alias existing_type_variable new_type_variable
```

Example:

```
alias integer int
alias MyLibrary.AwesomeType T

globals
    integer DDS_Damage_Event
endglobals

alias DDS_Damage_Event dEvent

set dEvent++
```

Alias can not be used in Global Scope. Alias never aliases type or variable globally, only inside Scope it is declared in.

13.1.1 White spaces

```
alias existing_type_variable new_type_variable
```

1. Every Syntactic Unit is bound tightly to every other Syntactic Unit.
2. The whole line must be placed on separate line, separated from any other code.

14. Hook

Hook allows to run user-defined functions when other user-defined or Blizzard-defined functions are called.

14.1 Syntax

```
hook opt:(optional) opt:(hookmode) hooked_func to_call opt:(priority)
```

Example:

```
hook optional after RemoveUnit onRemove
hook before RemoveUnit onRemoveB priority=54
hook RemoveUnit onRemoveC
```

hooked_func represents function that should be hooked, e.g. when it is called, the other one should be executed according to hookmode. hooked_func must not be private to its scope.
to_call represents function that is called when invocation of hooked_func is attempted in any way.
to_call must have the same arguments as the hooked_func, and can return any value.
to_call can be function or static method with any Visibility-Qualifier and with any Const-Qualifier.
hooked_func must be visible at the point of hooking, can be of any Const-Qualifier, and must be function or static method.
If optional keyword is provided and hooked_func does not exist, no error is issued, otherwise compilation error is issued.
hookmode is special variable representing when to call the caller. If none is provided, before is implicitly assumed.
Available hookmodes: before, after.
before hookmode will result in call to the caller before the actual call to hooked function.
after hookmode will result in call to the caller after the actual call to hooked function.
With before hookmode, the caller has no way to prevent call to the hooked function, other than forcefully stopping currently executing virtual thread.
Optionally a priority can be provided, for priority explanation, see 1.17 – Priorities.

13.1.1 White spaces

```
hook opt:(optional) opt:(hookmode) hooked_func to_call opt:(priority)
```

1. Every Syntactic Unit is bound tightly to every other Syntactic Unit.
2. The whole line must be placed on separate line, separated from any other code.
3. Priority must follow its separate rules, but priority is bound tightly into to_call Callable

object.

15. Textmacro

Textmacros are compile-time text expanders, which take code and expand it with some optional changes to it at places in map script.

15.1 Syntax

15.1.1 Definition

```
opt:(Visibility-Qualifier) textmacro opt:(takes argument_list)
    Textmacro-Body
endtextmacro
```

Example:

```
textmacro MyMacro takes A, B, C, D, E
    $A$ function $B$ takes $C$ returns $D$
        return $E$
    endfunction
endtextmacro
```

Arguments are of following form:

```
ARG1, ARG2, ARG3, ARG4...
```

Visibility-Qualifier is only applicable when textmacro is declared inside any non-Global Scope.
If no arguments are passed, takes keyword must be omitted.
All arguments, when used, must be enclosed in $ signs, marking the actual usage of given argument. Arguments of textmacro are typeless.

15.1.1.1 White spaces

```
opt:(Visibility-Qualifier) textmacro opt:(takes argument_list)
    Textmacro-Body
endtextmacro
```

1. If Visibility-Qualifier is provided, it is bound tightly into 'textmacro' keyword.
2. textmacro keyword is bound tightly into name.
3. if arguments are provided, they are bound tightly into each other as well as takes keyword.
4. takes keyword is bound tightly into the name.

15.1.2 Usage

```
runtextmacro opt:(optional) name( opt:(args) )
```

Example:

```
runtextmacro optional MyMacro("First", "Second")
```

If optional keyword is provided and given textmacro does not exist, nothing happens, otherwise compilation error is issued. Parenthesis are required even if textmacro has no arguments. All textmacros arguments must be compile-time string values.

15.2.1 White spaces

```
runtextmacro opt:(optional) name( opt:(args) )
```

1. Every Syntactic Unit is bound tightly into each other.
2. The whole line must be placed separated from any other code.

16. Extendor

Extendor is, similarly to textmacro, compile-time text expander, but rather than expanding one fixed body into multiple places, extendor can have multiple bodies expanded at multiple places.

16.1 Syntax

16.1.1 Defining

```
extendor Name opt:(priority) opt:(RunMode Mode_Extendor_Name)
                             opt:(takes Argument_List)
    Extendor-Body
endextendor
```

Arguments are of following form:

```
ARG1, ARG2, ARG3...
```

Example:

```
extendor X priority=50 takes A, B, C
    //CODE A
endextendor

extendor Y priority=-1 After X takes A, B
    //CODE B
endextendor

extendor Z After X takes A
    //CODE C
endextendor
```

Every extendor can have any number of bodies, and every body can take different, and any number of arguments, and can have its own priority.

If one extendor's definition specifies RunMode, all definitions of the same extendor must also provide the same RunMode, or Compilation error is issued.
Extendor can contain any number of any eJass Constructs other than another Extendor.
RunMode is special mode, specifying when should the extendor be evaluated.

Possible values for RunMode: AfterEarly, After, AfterLate, BeforeEarly, Before, BeforeLate,.
RunMode is case sensitive.

RunModes explained:

- AfterEarly - Runs after Mode_Extendor_Name has been instructed to run, and before any After extendors for given extendor.
- After - Runs after Mode_Extendor_Name has been instructed to run, before AfterLate, and after AfterEarly extendors for given extendor.
- AfterLate - Runs after Mode_Extendor_Name has been instructed to run and after all AfterEarly and After extendors for given extendor.

BeforeX - The same rules as for After, but they are executed before the Mode_Extendor_Name, instead of after.

If no RunMode is provided, the extendor is not attached to any other's extendor execution.
For Priority, See 1.17 - Priorities.
If two definitions of the same extendor have same priority and same RunMode, they will be evaluated in order of appearance to the compiler.
Any and all strings that are same as any of the arguments which are surrounded by at least one non alphanumeric character with the passed argument will get replaced with the value of given argument.

16.1.1.1 White spaces

```
extendor Name opt:(priority) opt:(RunMode Mode_Extendor_Name)
                             opt:(takes Argument_List)
    Extendor-Body
endextendor
```

1. Every Syntactic Unit before Extendor-Body is bound tightly into each other.
2. Extendor-Body must start at separate line from extendor's definition.
3. endextendor must be placed on separate line.

16.1.2 Usage

```
runextendor Name opt:[(args)]
```

Example:

```
runextendor X("A", "B", "C", "D")
runextendor X("A", "B")
```

One extendor can be ran any number of times. If user attempts to run extendor that does not exist, nothing happens, and compilation continues.

Every ran extendor can provide any number of arguments, and if any body of ran extendor require more arguments, they are implicitly generated as empty, and if extendor is ran with more arguments, than the rest of arguments are ignored.
When extendor is run, it will be evaluated regardless if it has RunMode specified or not, and all

extendors that have RunMode of this extendor will also be evaluated.
All extendor's arguments can be treated as compile-time and therefore be used in compile-time environment and expanded argument can only be used in compile-time environment if it evaluates into compile-time value.

16.1.2.1 White spaces

```
runextendor Name opt:[(args)]
```

1. runextendor is bound tightly into name.
2. If arguments are provided, the Name is bound tightly into the starting (, and all arguments are bound loosely to each other, first argument is bound tightly into the opening parenthesis, last argument is bound tightly into closing parenthesis and each argument is bound tightly to following comma, if any.

17. External, externalblock

External and externalblock let eComp run external programs according to the path with name of the program with given input.

17.1 Syntax

17.1.1 external

```
external opt:(runmode) program_name program_command_line_input
```

Example:

```
external ObjectMerger "" w3a Amls A000 anam "Fire"
external before ObjectMerger "" w3a Amls A000 anam "Fire"
```

program_name must be executable program and must accept command line options program_command_line_input. If given program does not exist, a compiler error will be issued.
program_name's console output will be print to the console and compilation output file.
Runmode specifies when should the program run. Available modes are: 'before', 'after'.
before runmode will execute given external program with its input before actual map script compilation begins.
after runmode will execute given external program after eComp finished compiling the map script.
The only implicitly passed argument into the external program is the map name and every other argument must be specified in the script.
A special variable is provided which can be passed into external program as map name, in which case the map name is not passed as implicit. This variable is $MAPNAME$.

17.1.1.1 White spaces

```
external opt:(runmode) program_name program_command_line_input
```

1. Every Syntactic Unit is bound tightly into each other.

2. The whole expression must be placed on separate line.

17.1.2 externalblock

```
externalblock opt:(runmode) opt:(extension=EXT) program_name
                                                program_command_line_input
    ExternalBlock-Body
endexternalblock
```

Example:

```
externalblock extension=lua ObjectMerger "" $FILENAME$
    //lua script for ObjectMerger
endexternalblock
```

program_name must be executable program and must accept command line options program_command_line_input. If given program does not exist, a compiler error will be issued.
program_name's console output will be print to the console and compilation output file.
Runmode specifies when should the program run. Available modes are: 'before', 'after'.
before runmode will execute given external program with its input before actual map script compilation begins.
after runmode will execute given external program after eComp finished compiling the map script.
Extension defines what extension should be the file saved with. For instance, if extension is set to txt, the file will be saved as Temp_File.txt.
The only implicitly passed argument into the external program is the map name and every other argument must be specified in the script.
If a temporary file made by externalblock is to be used, a $FILENAME$ shall be used as part of program_command_line_input.
A special variable is provided which can be passed into external program as map name, in which case the map name is not passed as implicit. This variable is $MAPNAME$.
All Jass comments are removed from ExternalBlock-Body before it is passed into the external program.

17.1.2.1 White spaces

```
externalblock opt:(runmode) opt:(extension=EXT) program_name
                                                program_command_line_input
    ExternalBlock-Body
endexternalblock
```

1. externalblock keyword is bound tightly into the following Syntactic Unit
2. runmode as well as extension, if provided, are bound tightly into following and preceding Syntactic Unit
3. program_name is bound tightly into preceding Syntactic Unit and bound loosely into program_command_line_input.
4. endexternalblock keyword must be placed on separate line from any other code.

18. Template

Templates provide compile-time type deduction for specific eJass constructs.

18.1 Syntax

18.1.1 Definition

```
template <opt:(Type type_name opt:(, Type type_name...))>
                                        opt:(requires Constraints_List)
permitted_construct
```

Example:

```
template<type A> requires A == integer
function MyFunc takes A a returns integer
      return cast<integer>(a)
endfunction

template<type T>
method operator T
      return cast<T>(this).myUnit
endmethod

template <type T>
    requires !Compiler.exists(method operator T<integer>(T<real>))
function f takes T<integer> t_inst returns T<real>
      return cast<T<real>>(t_inst)
endfunction

template <type T, integer Q>
function f takes T t returns integer
      return t.method(Q)
endfunction
```

Requires list must be provided before the start of templated construct's definition.
For Constraints, See 18.5 - Constraints.
Only functions, methods, structs, aliases and concepts can be templated.
Template can have any number of arguments, and all arguments must be either of type or some specified type. If type is mentioned, the parameter is of undefined type and the template will accept any valid type in given slot, if some given type is provided, the template expects compiletime value of given type as argument at that place.
Arguments can have implicit values(See 18.4 - Template argument's implicit value), but only in such way that if Nth argument is implicit than all arguments to the right of Nth argument must also have implicit values.
Templates are always instantiated from inside out, meaning that A<B<C<D>>> will first instantiate C<D>, then it will instantiate B<C<D>> and finally A<B<C<D>>>.
Templates are only instantiated when explicitly or implicitly required.
Templates can have variadic number of arguments(See 18.3 - Variadic Templates) or can provide partial specialization(See 18.2 - Partial Specialization).

One eJass Construct can also be templated with multiple templates with different combination or number of arguments.
Templated eJass Constructs can be used without specializing explicitly the template arguments if they are deducible. Only template arguments used in function's argument list are deducible. Return types for functions, in-function variables using templated type or structs must have explicitly stated template arguments.
If template argument is deduced from null, the argument is of type handle.
Operators inside struct can be templated, and they are accessed using template arguments after expression evaluating into instance of type for which exists templated operator, and for static operators, the expression must evaluate into type name.

Example:

```
struct A
    template <type T> method operator [] takes integer x returns T
        return T(x)
    endmethod
endstruct

function f takes nothing returns nothing
    local A a = A()
    call someFunction(a<integer>[5])    //calls A.operator[]<integer>(5)
    call someFunction(a<handle>[78])    //calls A.operator[]<handle>(78)
    call someFunction(a<A>[5]<integer>[7])
                        //calls A.operator[]<A>(5).operator[]<integer>(7)
endfunction
```

Operators that can not have template type deduced:

- !
- []
- static []
- ++
- – –

18.1.1.1 White spaces

```
template <opt:(Type type_name opt:(, Type type_name...))>
                                        opt:(requires Constraints_List)
permitted_construct
```

Arguments are of following type:

```
Arg_type Arg_name, Arg2_type Arg2_name, ...
```

1. template keyword is bound tightly into following <.
2. < is bound tightly into following Syntactic Unit.
3. If any template arguments are specified, the first one is bound tightly into preceding Syntactic Unit, the last one is bound tightly into the following Syntactic Unit, the argument's type is bound tightly into the argument's name, the argument's name is bound tightly into following comma and each argument is bound loosely to each other.

4. requires, if provided, is bound loosely to preceding and following Syntactic Unit.
5. The Constraints_List after requires as well as templated eJass Construct must follow its own rules and white spacing rules.

18.1.2 Usage

```
TypeName < template_args > ...
```

Example:

```
local List<integer> l = List<integer>()
call l.something()
call l.templated_something<integer, real, real>()

call List<integer>.static_something()
call List<integer>.static_templated_something<integer, real, real>()
```

TypeName must be a valid type name and must be templatable.
templated_args must be of same types as in definition.
After template instantiation, either nothing in regards to the instantiation can appear, or ".", or (can appear.
If nothing appears, the resulting expression must evaluate into type, otherwise it can be either function, method or struct.

18.1.2.1 White spaces

```
TypeName < template_args > ...
```

1. Every Syntactic Unit is bound tightly into each other.

18.2 Specialized Templates

Specialized templates are such templates that have no arguments and provide specialized body for given eJass Construct with given signature. Specialized template can only be provided for eJass Construct that is already templetized.

18.2.1 Syntax

```
template <> opt:(requires Constraint_List)
Permitted_construct_name<specified_args>
    Permitted_construct_Body
Permitted_construct_end
```

Example:

```
template <type A, type B>
functon f takes A a, B b returns nothing
    call a.something(b)
endfunction

template<>
function f<integer, real> takes integer i, real ff returns nothing
```

```
    call someFunc(i, ff)
endfunction

template <>
function f<Matrix, real> takes Matrix m, real ff returns nothing
    call a.somethingDifferent(ff)
endfunction

template <type T>
struct S
    T t
    static method myM takes T q returns thistype
        local thistype this = .allocate()
        set t = q
        return this
    endmethod
endstruct

template <>
struct S<integer>
    integer t
    static method myM takes integer q returns thistype
        local thistype this = allocate()
        set t = q
        call print("set t to " + cast<integer>(q))
        return this
    endmethod
endstruct

template <integer I>
struct Q
    static integer i = I * Q<I-1>.i
endstruct

template <>
struct Q<1>
    static integer i = 1
endstruct

call BJDebugMsg(cast<string>(Q<10>.i))        //prints 3628800
```

Resulting eJass Construct in Specialized template must not have a difference in signature other than type mismatching.
When eJass Construct is templated using specialized template, the template arguments must be provided explicitly in <> block after the name of eJass Construct. This template specialization must have unique arguments compared to other specialized templates of given eJass Construct.
Specialized template can only be used if the original template has at least one argument of unspecified type, and can only override unspesified types. All specified types must stay the same.
Constructor or destructor of struct cannot be templated. If struct's constructor or destructor are templated, a compiler error is issued.

18.2.1.1 White spaces

```
template <> opt:(requires Constraint_List)
Permitted_construct_name<specified_args>
    Permitted_construct_Body
Permitted_construct_end
```

Arguments are of following type:

```
Arg_type Arg_name, Arg2_type Arg2_name, ...
```

1. template keyword is bound tightly into following <.
2. < is bound tightly into >.
3. requires Constraint_List, if provided is bound tightly into preceding Syntactic Unit and loosely into the following Syntactic Unit.
4. < after Permitted_construct_name is bound tightly into it as well as first argument of specialized template.
5. > is bound tightly into last argument of specialized template and requires that Permitted_construct_Body is placed on different line.
6. Argument's type is bound tightly into the Argument's name and following comma, if any, and is bound loosely to the next argument of specialized template.
7. Permitted_construct must also follow its respective rules.

18.3 Variadic Templates

Variadic template is such template that has undefined number of arguments.

18.3.1 sizeof operator

The sizeof operator returns the number of arguments inside variadic template argument as compiletime integer.

Example:

```
template <type... C>
function number takes C c returns compiletime integer
    return sizeof(c)
endfunction
```

Implementation in eJass:

```
template <type C>
struct templateDetailImpl
    compiletime integer size = 1
endstruct

template <type A, type... B>
struct templateDetailImpl
    compiletime integer size = 1 + templateDetailImpl<B>.size
endstruct

template <type... C>
```

```
compiletime function sizeof takes C c returns integer
    return templateDetailImpl<C>.size
endfunction
```

18.3.2 Syntax

```
template <type A, type B, type... C>
perimtted_construct
```

Example:

```
template <type A, type... B>
function f takes A a, B b returns HashStorage
    compiletime local integer BSize = sizeof(B)
    local HashStorage h = HashStorage()
    compiletime local integer iter = 0
    loop
        call h.add(b[iter])
        //optionally:
        call h.add<B[iter]>(b[iter])
        exitwhen iter == BSize - 1
        set iter++
    endloop
    return h
endfunction
```

f(5, 3, 2.00, "my string") -> h now stores [5, 3, 2.00, "my string"]
One template can only have one deducible variadic template or any number of explicitly deducible variadic templates.

Example:

```
template <type... A, type... B>
function f takes A a, B b returns nothing
endfunction
```

is not valid because compiler does not know where does A end and where does B start, whereas

```
template<type C, type... A, type... B>
function f takes C<A> q, B b returns nothing
endfunction
```

is valid, since C is the outer-most type passed into the function as first argument, and its separate template type is deduced to be A, and B is deduced as the rest of arguments passed into the function.
Variadic template can also have 0 arguments.
Basic array subscript operator can be used to get type at given position out of variadic template pack but only if the argument to array subscript operator is integer literal, or the templated eJass construct is compiletime. The same applies for variables in functions, if variable is of variadic template pack type.

No variable other than function argument can be of variadic template pack type.
Variadic templates can be specialized as any other template.

18.3.2.1 White spaces

```
template <type A, type B, type... C>
perimtted_construct
```

1. ... is bound tightly into type keyword as well as following name of variadic template argument.
2. the same rules as for normal template apply.

18.4 Template argument's implicit value

Normal template arguments can default into some value or type but only in such way that if certain template argument defaults to some value, than all template arguments to the right of it also must provide implicit value.

18.4.1 Syntax

Example:

```
template <type A, type B, type C = B<A>>
function q takes C c, A a, B b returns nothing
endfunction

template <type A, integer B = 5, type C = integer>
struct S
endstruct
```

18.4.1.1 White spaces

1. The same rules for default values apply as for function's argument
2. default values(See 4.2.1.1 - Function argument's implicit values).
3. For the rest, same rules apply as for normal template.

18.5 Constraints

Templates depending on constraints are only generated when all Constraints evaluate to true. One template can have any number of comma separated constraints. Constraints can be either Concept(See 18.5.1 - Concept), or a compile-time condition, or a special operator call(See 2.6.4 - Comparisons).

Example:

```
template <type T> requires Arithmetic<T>, T != Table,
                           T extends MyAwesomeType and
                           !(T extends MyAnotherAwesomeType),
                           Compiler.isParentOf(T, SomeType) and
                           Compiler.isSame(T, T)
function f takes T t returns T
```

```
        return t
    endfunction
```

18.5.1 Concept

Concepts are a constraintment feature for templates, providing compiletime information about types. Concepts can only be used as Constraints for templates or other concepts. Concepts evaluate into true if and only if all methods or functions declared inside given concept exist for given type. Otherwise, they evaluate into false.

18.5.1.1 Syntax

```
concept Name
    Concept-Body
endconcept
```

Example:

```
    template <type T>
    concept Arithmetic
        method T.operator+ takes T rhs returns T
        method T.operator- takes T rhs returns T
        method T.operator* takes T rhs returns T
        method T.operator/ takes T rhs returns T
        //constructor takes T
        //destructor takes T
        //static method T.met_hod takes T something returns nothing
        //stub method T.m ...
        //function add takes T a, T b returns T
    endconcept

    concept Another requires integer != real,
                             Compiler.isChildOf(unit, handle)
        method MyStruct.myMethod takes nothing returns nothing
    endconcept


    concept General requires Another
        function SomeFunction takes integer something returns integer
    endconcept
```

Concepts can be templated, and can put following requirements on any type visible to the concept:

1. existence of method with given name, arguments and return type
2. existence of function with given name, arguments and return type

Concept can only become Constraint after it has been fully defined. Concept itself can have Constraints. If function or method with given name, signature and return type does not exist, the concept returns false.

18.5.1.1.1 White spaces

```
concept Name
    Concept-Body
endconcept
```

1. concept and Name are bound tightly.
2. Concept-Body must start at separate line.
3. endconcept must end at separate line.

19. Auto

Auto keyword allows compiler to deduce type of variable according to the initial assignment.

19.1 Syntax

```
auto var_name = var_initial_value
```

Example:

```
local auto v = Location(5., 5.)

globals
    private auto var = StringLength("")
endglobals

temporary auto q = CreateGroup()
```

Global, Local and temporary variables may be declared as auto. Every variable declared with automatic type must have initial value, or the type is ineducable. The type is always deduced at compilation time to the type of assignment expression.

19.1.1 White spaces

```
auto var_name = var_initial_value
```

1. Keyword auto is bound tightly into var_name.
2. Other Syntactic Units must follow same rules as with assignment to normal variables(See 3. - Variable).

20. Import

Import allows the Compiler to load and parse Jass files from external sources. External sources can be local hard-drive, or file placed on internet.

20.1 Syntax

```
import "file_to_import"
```

Example:

```
import "http://raw.github.com/some_repository/some_jass_file.j"
import "../../scripts/My_Awesome_Jass_Script.j"
```

If file_to_import is invalid file, a compiler error is issued, and compilation stops. If file_to_import starts with "http://", "//" or "www.", Compiler attempts to download given file from internet. If such file has already been downloaded from the internet before, and it has not been changed since, the cached version is used instead. Otherwise, a open file request is sent to Operating System with provided path.
Path can be relative or absolute.
Imported file must be fully valid Jass or eJass file with any of above mentioned features included. Otherwise, compiler error is issued.

20.1.1 White spaces

```
import "file_to_import"
```

1. All Syntactic Units are bound to each other tightly.

21. Encrypted

Keyword encrypted allows the Compiler compiler to encrypt marked strings when using raw string data with certain algorithms.
Note: while encryption has no performance impact, the decryption requires time linearly scaling with size of string, and may eat a lot of OPs, resulting in Op-heavy code crashing the running virtual thread because of decryption of strings. Use this wisely.

21.1 Syntax

```
local encrypted opt:[(Enc_Mode)] string opt:(array) string_name
opt:(= Initial_value)
```

Example:

```
local encrypted string s = "My awesome" + " brutally encrypted string!"
local encrypted(ME) string s = "Even better encryption!"
```

String encryption only happens on raw strings, and happens at compile time, resulting in no performance impact of the code. Decryption must be done during runtime, resulting in possibly huge impact on Ops as a result of decryption.

If encrypted does not provide Enc_Mode, a default Enc_mode is used from eComp.
Possible Enc_Mode-s:

Nan, No, None, BAE, BSE, ME.

Nan, None, No - no encryption shall be used, even if user required encrypted strings.
BAE - Basic addition encryption

All required strings are encrypted in such way that their letters get changed ASCII values, and are decrypted when use is requested.

BSE - Basic shuffle encryption

All required strings are encrypted in such way that their letters get changed positions, and are decrypted when use is requested.

ME - Mixed Encryption

All required strings are encrypted in such way that their letters get changed ASCII values as well as changed position, and are decrypted when use is requested.

21.1.1 White spaces

```
local encrypted opt:[(Enc_Mode)] string opt:(array) string_name
opt:(= Initial_value)
```

1. encrypted is bound to local, as well as following Syntactic Unit tightly.
2. (Enc_Mode), if provided, is bound to both preceding and following Syntactic Unit tightly.
3. The remaining Syntactic Units follow 3.2.1.1 - White spaces

Standard Library Specification

1. namespace Std

Is a namespace that hosts all Standard Library contents excluding struct Compiler.

2. Compiletime Standard Library

This section contains compiletime standard library features. Compiletime part of Standard Library does not need to be wrapped inside library/endlibrary, because the Compiler knows all about these structs.

2.1 struct Compiler

2.1.1 Synopsis

```
compiletime struct Compiler
    private construcor
    private destructor

    //2.1.1.1 options
    static method hasNetworking takes nothing returns boolean
    static method getOptimizationLevel takes nothing returns integer

    //2.1.1.2 versioning
    static method getVersion takes nothing returns string
    static method mainVersion takes nothing returns integer
    static method majorVersion takes nothing returns integer
    static method minorVersion takes nothing returns integer

    //2.1.1.3 assertion
    static method assert takes string message returns nothing

    //2.1.1.4 existence
    static method exists takes _rconstruct construct returns boolean
    static method exists takes _pconstruct construct returns boolean

    //2.1.1.5 naming
    static method nameOf takes _rconstruct construct returns boolean
    static method nameOf takes _pconstruct construct returns boolean
    static method fullNameOf takes _rconstruct construct returns boolean
    static method fullNameOf takes _pconstruct construct returns boolean

    //2.1.1.6 stack tracing
    static method initTrace takes nothing returns nothing
    static method getTrace takes nothing returns string
endstruct
```

2.1.1.1 options

static method hasNetworking **takes** **nothing** **returns** **boolean**

Returns: Whether the Compiler has allowed networking access.

static method getOptimizationLevel **takes** **nothing** **returns** **integer**

Returns: Which optimization level the script is compiled with.

Possible values:

0 - no optimization

1 - full optimization

2.1.1.2 versioning

static method getVersion **takes** **nothing** **returns** **string**

Returns: The version of the Compiler as string.

static method mainVersion **takes** **nothing** **returns** **integer**

Returns: The main version of the Compiler as integer.

static method majorVersion **takes** **nothing** **returns** **integer**

Returns: The major version of the Compiler as integer.

static method minorVersion **takes** **nothing** **returns** **integer**

Returns: The minor version of the Compiler as integer. If the compiler has non-integer representation of minor version, such as .1a, the number will still be unique from any other version. These numbers increase sequentially.

2.1.1.3 assertion

static method assert **takes** **string** message **returns** **nothing**

Effect: Stops compilation immediately and issues an error with error message "message".

2.1.1.4 existence

static method exists **takes** _rconstruct construct **returns** **boolean**

Returns: Whether given construct exists in map script or not.

static method exists **takes** _pconstruct construct **returns** **boolean**

Returns: Whether given construct exists in map script or not.

2.1.1.5 naming

static method nameOf **takes** _rconstruct construct **returns** **boolean**

Returns: Name of given construct as is exposed in eJass.

static method nameOf **takes** _pconstruct construct **returns** **boolean**

Returns: Name of given construct as is exposed in eJass.

static method fullNameOf **takes** _rconstruct construct **returns** **boolean**

Returns: Name of given construct after applying modifications to the name because of scope resolution and overloading.

static method fullNameOf **takes** _pconstruct construct **returns** **boolean**

Returns: Name of given construct after applying modifications to the name because of scope resolution and overloading.

2.1.1.6 stack tracing

static method initTrace **takes** **nothing** **returns** **nothing**

Effect: Initializes given function, and all its callers, and all functions it calls for tracing.

static method getTrace **takes** **nothing** **returns** **string**

Returns: String representing current call stack in format function > function > function.

2.2 struct File

2.2.1 Synopsis

```
namespace Std
    compiletime struct File
        //2.2.1.1 position struct
        compiletime struct Position
            private constructor
        endstruct

        //2.2.1.2 constants
        readonly constant Position beg
        readonly constant Position cur
        readonly constant Position end

        //2.2.1.3 constructors
        constructor
        constructor takes string name
        static method open takes string name returns boolean

        //2.2.1.4 destructors
        destructor
        method close takes nothing returns boolean

        //2.2.1.5 reads
        method readLine takes nothing returns string
        method readBytes takes integer howMany returns string
        method readWord takes nothing returns string

        //2.2.1.6 writes
        method write takes string what returns boolean

        //2.2.1.7 positioning
        method getPosition takes nothing returns integer
        method setPosition takes integer newPosition, Position pos
                                                    returns boolean
```

```
        //2.2.1.8 sizes
        method size takes nothing returns integer
        method remaining takes nothing returns integer

        //2.2.1.9 miscellaneous
        method flush takes nothing returns boolean
        method remove takes nothing returns boolean
        method rename takes string newName, boolean keep = false
                                                    returns boolean
        method forceInMemory takes boolean doForce = true returns boolean
        method getLimit takes nothing returns integer
    endstruct
endnamespace
```

2.2.1.1 position struct

A dummy struct to be passed into setPosition.

2.2.1.2 constants

```
readonly constant Position beg
readonly constant Position cur
readonly constant Position end
```

Mark the position to set the position at.

beg - offset from beginning of the file

cur - offset from current position

end - offset from end of the file

2.2.1.3 constructors

```
constructor
```

Effect: Creates new file instance with unassigned file buffer.

```
constructor takes string name
```

Effect: Creates new file instance and attempts to open file buffer with name of the file "name". Same as calling constructor and then calling open(name)

```
static method open takes string name returns boolean
```

Effect: Attempts to open file with name "name".

Returns: whether the open request was successful or not.

2.2.1.4 destructors

```
destructor
```

Effect: Closes assigned file buffer, if any, and then destructs instance. Also performs flush operation.

```
method close takes nothing returns boolean
```

Effect: Attempts to close assigned file buffer. Also performs flush operation.
Returns: whether file buffer could be closed or not.

2.2.1.5 reads

method readLine **takes** **nothing** **returns** **string**
Effect: Moves the pointer position after the read line.
Returns: Read line, or empty string if at end of file, or if read failed.

method readBytes **takes** **integer** howMany **returns** **string**
Effect: Moves the pointer position to new position.
Returns: String representing 1 character per byte in file in range [position, position + howMany). The string can be cut if the end of file appears before "howMany" bytes are read from file. Empty string if read failed, or if already at end of file.

method readWord **takes** **nothing** **returns** **string**
Effect: Moves the pointer to position of closest next white space.
Returns: String representing the read word. Empty string if already at end of file, or if reading failed.

2.2.1.6 writes

method write **takes** **string** what **returns** **boolean**
Effect: Inserts contents of string what into current position inside file.
Returns: Whether write operation could be performed.

2.2.1.7 positioning

method getPosition **takes** **nothing** **returns** **integer**
Returns: Current pointer position.

method setPosition **takes** **integer** newPosition, Position pos **returns** **boolean**
Effect: Moves pointer position into newPosition according to pos. If outside of range, does nothing.
Returns: true if position is in range [0, size()), false otherwise.

2.2.1.8 sizes

method size **takes** **nothing** **returns** **integer**
Returns: Size of the underlying file buffer.

method remaining **takes** **nothing** **returns** **integer**
Returns: How many more bytes can be stored inside given file buffer.

2.2.1.9 miscellaneous

method flush **takes** **nothing** **returns** **boolean**
Effect: Flushes the write buffer, performing all writes to file immediately. If file is forced into memory, performs write into disk.

Returns: true if flush was successful, false if flush could not be performed, or the file buffer was not opened.

method remove **takes** **nothing** **returns** **boolean**

Effect: Removes the file from disk.

Returns: Whether the file could be removed.

method rename **takes** **string** newName, **boolean** keep = **false** **returns** **boolean**

Effect: Renames the file on disk. If keep is true, the file with old name will be preserved, otherwise it will be removed.

Returns: Whether renaming the file was possible.

method forceInMemory **takes** **boolean** doForce = **true** **returns** **boolean**

Effect: Attempts to store the whole file's contents inside memory.

Returns: Whether storing the file in memory was possible.

method getLimit **takes** **nothing** **returns** **integer**

Returns: The number of bytes file opened with struct File can contain.

Currently returns 8388608(8 MiB).

2.3 struct External

2.3.1 Synopsis

```
namespace Std
    compiletime struct External
        //2.3.1.1 construction
        constructor
        method create takes string oneLiner returns External
        method load takes string filePath returns External
        method load takes File file returns External

        //2.3.1.2 destruction
        destructor

        //2.3.1.3 execution
        method execute takes nothing returns boolean

        //2.3.1.4 errors
        method error takes nothing returns string

        //2.3.1.5 extension
        method setExtension takes string extension returns External
        method getExtension takes nothing returns string

        //2.3.1.6 command line arguments
        method setArgument takes string argument returns External
    endstruct
endnamespace
```

2.3.1.1 construction

constructor

Effect: constructs empty External instance

method create takes string oneLiner returns External

Effect: Creates a External command from first line inside oneLiner string. Same as external ...

Returns: this

method load takes string filePath returns External

Effect: Loads and constructs External command from file "filePath".

Returns: this

method load takes File file returns External

Effect: Loads and constructs External command from file.

Same as

```
local Std.External ext = /* */
call file.setPosition(0, file.beg)
call ext.load(file.readBytes(file.size()))
```

Returns: this

2.3.1.2 destruction

destructor

Effect: Destructs this

2.3.1.3 execution

method execute takes nothing returns boolean

Effect: Attempts to execute given External command. Note: If external command modifies the map script, the new map script is ignored.

Returns: true on success, false if the External command could not be executed.

2.3.1.4 errors

method error takes nothing returns string

Returns: The error message if method execute could not be run in string.

2.3.1.5 extension

method setExtension takes string extension returns External

Effect: Sets the extension type of file loaded via load, if any, to extension.

method getExtension takes nothing returns string

Effect: Returns extension set for this as string.

2.3.1.6 command line arguments

method setArgument **takes** **string** argument **returns** External

Effect: Creates a command line argument and assigns it to value of argument.

All arguments are upon execution send to the External tool in order of their registration, separated by spaces.

Note: Single instance of External can store up to 256 arguments.

2.4 struct MPQReader

2.4.1 Synopsis

```
namespace Std
    compiletime struct MPQReader
        //construction
        constructor

        //destruction
        destructor
    endstruct
endnamespace
```

2.4.1.1 construction

constructor

Effect: Constructs empty MPQReader.

2.4.1.2 destruction

destructor

Effect: Destructs this.

2.5 struct PluginManager

2.5.1 Synopsis

```
namespace Std
    //2.5.1.1 Plugin Settings
    compiletime struct PluginSettings
        //fields
        readonly string name
        readonly string authorName
        readonly string version
        readonly string description
        readonly string atPhase
        readonly boolean before
        readonly boolean after          /* = !before */
        readonly string priority

        private constructor
```

```
    endstruct

    //2.5.1.2 Plugins
    compiletime struct Plugins
        private constructor

        //2.5.1.2.1 fetching
        method get takes integer number returns string
        method getNext takes nothing returns string
        method size takes nothing returns integer
    endstruct

    completime struct PluginManager
        //2.5.1.3 construction
        constructor

        //2.5.1.4 destruction
        destructor

        //2.5.1.5 notification
        method notifyPlugin takes string whatNotif, string notifMessage
                                                            returns integer

        //2.5.1.6 execution
        method runPlugin takes nothing returns PluginManager
        method runPlugin takes string pluginName returns PluginManager
        method parsePlugin takes nothing returns PluginManager
        method parsePlugin takes string pluginName returns PluginManager
        method setPlugin takes string pluginName returns PluginManager

        //2.5.1.7 settings
        method getPluginSettings takes nothing returns PluginSettings

        //2.5.1.8 plugins
        static method getPlugins takes nothing returns Plugins
    endstruct
endnamespace
```

2.5.1.1 Plugin Settings

PluginSettings is a struct that stores all possible settings for plugins under named, readonly field. Its constructor is private so that users cant create instances of given struct.

2.5.1.2 Plugins

Plugins is a struct that represents all plugins currently available on compilation.

2.5.1.2.1 fetching

method get takes integer number returns string

Returns: Number-th plugin's name in the internal list as string.

method `getNext` **takes** `nothing` **returns** `string`

Returns: Next plugin's name in the internal list as string.

method `size` **takes** `nothing` **returns** `integer`

Returns: Number of plugins inside internal list.

2.5.1.3 construction

constructor

Effect: Constructs empty PluginManager.

2.5.1.4 destruction

destructor

Effect: Destructs this.

2.5.1.5 notification

method `notifyPlugin` **takes** `string whatNotif, string notifMessage` **returns** `integer`

Effect: Sends a notification into plugin stored in given instance of PluginManager.

Returns: Integer returned by plugin, or 0 if plugin could not process notification.

2.5.1.6 execution

method `runPlugin` **takes** `nothing` **returns** `PluginManager`

Effect: executes plugin stored in given instance of PluginManager regardless of its settings. If plugin wasn't parsed beforehand, also parses the plugin.

Returns: this

method `runPlugin` **takes** `string pluginName` **returns** `PluginManager`

Effect: call setPlugin(pluginName).parsePlugin().runPlugin()

Returns: this

method `parsePlugin` **takes** `nothing` **returns** `PluginManager`

Effect: Parses the plugin script, if required.

Returns: this

method `parsePlugin` **takes** `string pluginName` **returns** `PluginManager`

Effect: call setPlugin(pluginName).parsePlugin()

Returns: this

method `setPlugin` **takes** `string pluginName` **returns** `PluginManager`

Effect: Loads and sets plugin under name pluginName to given instance of PluginManager.

2.5.1.7 settings

method `getPluginSettings` **takes** `nothing` **returns** `PluginSettings`

Returns: New instance of PluginSettings representing the plugin.

2.5.1.8 plugins

static method getPlugins **takes** **nothing** **returns** Plugins

Returns: New instance of Plugins holding information about all plugins that are visible to the Compiler.

2.6 struct TypeInfo

2.6.1 Synopsis

```
namespace Std
    //2.6.1.1 Argument List
    compiletime struct ArgumentList
        private constructor
        destructor

        //2.6.1.1.1 getters
        method getArgumentName takes integer which returns string
        method getArgumentType takes integer which returns string
        method size takes nothing returns integer
    endstruct

    //2.6.1.2 Func List
    compiletime struct FuncList
        //2.6.1.2.1 Info
        compiletime struct Info
            private constructor
            destructor

            readonly string name
            readonly boolean isPrivate
            readonly boolean isPublic        /* = !isPrivate */
            readonly boolean isConstant
            readonly boolean isStub

            readonly boolean isDeprecated
            readonly string deprecatedMessage
            readonly string deprecatedFunc

            readonly boolean isInline
            readonly ArgumentList arguments

            readonly string returnType
            readonly boolean isReturnConstant
        endstruct

        private constructor

        //2.6.1.2.2 getters
        method get takes integer which returns Info
        method size takes nothing returns integer
    endstruct
```

```
    compiletime struct TypeInfo
        //2.6.1.3 constructor and destructor
        private constructor
        private destructor

        //2.6.1.4 string
        static method toString takes type t returns string

        //2.6.1.5 inheritance
        static method isParentOf takes type a, type b returns boolean
        static method isDirectParentOf takes type a, type b returns boolean
        static method isChildOf takes type a, type b returns boolean
        static method isDirectChildOf takes type a, type b returns boolean

        //2.6.1.6 polymorphism
        static method isPolymorphic takes string typeName returns boolean

        //2.6.1.7 methods
        static method getMethodList takes string structName returns FuncList
        static method getMethodArguments takes string methodName
                                                    returns ArgumentList

        //2.6.1.8 functions
        static method getFunctionArguments takes string funcName
                                                    returns ArgumentList
        static method getFunction takes string funcName returns FuncList.Info
    endstruct
endnamespace
```

2.6.1.1 Argument List

This struct stores data about separate arguments for methods and functions.

2.6.1.1.1 getters

method getArgumentName **takes** **integer** which **returns** **string**

Returns: Which-th argument's name as string.

method getArgumentType **takes** **integer** which **returns** **string**

Returns: Which-th argument's type as string.

method size **takes** **nothing** **returns** **integer**

Returns: Number of arguments in this.

2.6.1.2 Func List

Func List stores data about functions and methods.

2.6.1.2.1 Info

Stores all the named values for given function or method.

2.6.1.2.2 getters

```
method get takes integer which returns Info
```
Returns: Which-th method inside this.

```
method size takes nothing returns integer
```
Returns: Size of list stored inside this.

2.6.1.3 constructors and destructors

```
private constructor
private destructor
```
Explicitly marked as private so that TypeInfo is unconstructable struct.

2.6.1.4 string

```
static method toString takes type t returns string
```
Returns: Name of type t as string. Returns with scope parts(return can be "A.B.C").

2.6.1.5 inheritance

```
static method isParentOf takes type a, type b returns boolean
```
Returns: Whether a is parent type of b.

```
static method isDirectParentOf takes type a, type b returns boolean
```
Returns: Whether a is direct parent of b.

```
static method isChildOf takes type a, type b returns boolean
```
Returns: Whether a is child of b.

```
static method isDirectChildOf takes type a, type b returns boolean
```
Returns: Whether a is direct child of b.

2.6.1.6 polymorphism

```
static method isPolymorphic takes string typeName returns boolean
```
Returns: Whether typeName is polymorphic or not.
Always returns false for native types, and true only when struct typeName has at least one stub method.

2.6.1.7 methods

```
static method getMethodList takes string structName returns FuncList
```
Returns: New instance of FuncList representing all methods inside struct structName.

```
static method getMethodArguments takes string methodName returns ArgumentList
```
Returns: New instance of ArgumentList representing arguments of given method.

2.6.1.8 functions

static method getFunctionArguments **takes** **string** funcName **returns** ArgumentList

Returns: New instance of ArgumentList representing arguments of given function.

static method getFunction **takes** **string** funcName **returns** FuncList.Info

Returns: New instance of Info representing information about function funcName.

3. Runtime standard library

All Runtime standard libraries must be placed inside a folder compiler folder/std/runtime, and all jass files inside that folder are automatically imported into the map script if their contents are used. Contents of these files are imported into as closest to top of map script as possible, so code inside these files does not need to reside inside library.

3.1 UnitIndexer

3.1.1 Synopsis

```
namespace Std
    library UnitIndexer
        //3.1.1.1 getters
        function GetUnitId takes unit u returns integer
        function GetUnitById takes integer id returns unit

        //3.1.1.2 checks
        function IsUnitIndexed takes unit u returns boolean

        //3.1.1.3 allowance
        function DisableIndexing takes nothing returns nothing
        function EnableIndexing takes nothing returns nothing

        //3.1.1.4 event callback
        function RegisterOnIndex takes code c returns nothing
        function TriggerRegisterOnIndex takes trigger t returns nothing
        function RegisterOnDeindex takes code c returns nothing
        function TriggerRegisterOnDeindex takes trigger t returns nothing
        function GetIndexedUnit takes nothing returns unit
        function GetIndexedUnitId takes nothing returns integer
    endlibrary
endnamespace
```

3.1.1.1 getters

function GetUnitId **takes** **unit** u **returns** **integer**

Returns: Index of given unit u that UnitIndexer registered for that unit.

function GetUnitById **takes** **integer** id **returns** **unit**

Returns: Unit that has assigned index id from UnitIndexer.

3.1.1.2 checks

function IsUnitIndexed **takes** **unit** u **returns** **boolean**

Returns: Whether unit u has been indexed by UnitIndexer or not.

3.1.1.3 allowance

function DisableIndexing **takes** **nothing** **returns** **nothing**

Effect: Disallows UnitIndexer to index units after this function call.

function EnableIndexing **takes** **nothing** **returns** **nothing**

Effect: Allows UnitIndexer to index units after this function call.

3.1.1.4 event callback

function RegisterOnIndex **takes** **code** c **returns** **nothing**

Effect: Adds function stored inside code c to internal buffer and UnitIndexer executes it every time unit is indexed.

function TriggerRegisterOnIndex **takes** **trigger** t **returns** **nothing**

Effect: Adds trigger t into internal buffer and UnitIndexer evaluates given trigger every time unit is indexed.

function RegisterOnDeindex **takes** **code** c **returns** **nothing**

Effect: Adds function stored inside code c to internal buffer and UnitIndexer executes it every time unit is deindexed.

function TriggerRegisterOnDeindex **takes** **trigger** t **returns** **nothing**

Effect: Adds trigger t into internal buffer and UnitIndexer evaluates given trigger every time unit is deindexed.

function GetIndexedUnit **takes** **nothing** **returns** **unit**

Returns: Unit that is being indexed or deindexed.

function GetIndexedUnitId **takes** **nothing** **returns** **integer**

Returns: Index UnitIndexer assigned to indexed unit.

3.2 Damage Detection System

3.2.1 Synopsis

```
namespace Std
    //3.2.1.1 DDS
    struct DDS
        private constructor
        private destructor
```

```
        //3.2.1.1.1 values
        readonly static unit target
        readonly static unit source
        static          real amount
        readonly static integer damageType

        //3.2.1.1.2 constants
        constant integer PHYSICAL = /* implementation defined */
        constant integer SPELL    = /* implementation defined */
        constant integer CODE     = /* implementation defined */

        //3.2.1.1.3 events
        static method onDamage takes code c returns nothing

        //3.2.1.1.4 damage
        static method damageTarget takes unit source, unit target, real amount,
                                         boolean attack, boolean ranged,
                                         attacktype at, damagetype dt,
                                         weapontype wp returns boolean

        //3.2.1.1.5 getters
        static constant method getLife takes unit u returns real
        static constant method getMaxLife takes unit u returns real

        //3.2.1.1.6 setters
        static method setLife takes unit u, real value returns nothing
    endstruct

    //3.2.1.2 getters
    constant function GetLife takes unit u returns real
    constant function GetMaxLife takes unit u returns real

    //3.2.1.3 setters
    function SetLife takes unit u, real value returns nothing
endnamespace
```

3.2.1.1 DDS

DDS is a struct that provides main functionality of the Standard Damage Detection System.

3.2.1.1.1 values

```
readonly static unit target
readonly static unit source
static          real amount
readonly static integer damageType
```

Provide values for onDamage callbacks.

target - target unit that is being hit.

source - source unit that is dealing the damage.

amount - the damage amount

damageType - type of damage dealt. Check against constants.

3.2.1.1.2 constants

```
constant integer PHYSICAL = /* implementation defined */
constant integer SPELL    = /* implementation defined */
constant integer CODE     = /* implementation defined */
```

Provide values for damageType.

PHYSICAL - marks that damage is physical(auto attack, either melee or range damage).

SPELL - marks that damage is done via ability.

CODE - marks that damage is dealt with code.

3.2.1.1.3 events

```
static method onDamage takes code c returns nothing
```

Effect: Adds function stored inside code c into internal list of callbacks and given function gets executed every time any unit takes damage.

3.2.1.1.4 damage

```
static method damageTarget takes unit source, unit target, real amount,
                                 boolean attack, boolean ranged, attacktype at,
                                 damagetype dt, weapontype wp returns boolean
```

Effect: Deals damage from source to target using all parameters. This method is meant to be used inside onDamage callbacks, and shall be recursive safe. Must provide proper behaviour at all times.

3.2.1.1.5 getters

```
static constant method getLife takes unit u returns real
```

Returns: Current life of unit. Meant to be used inside onDamage event callbacks, but must provide proper behaviour at all times.

```
static constant method getMaxLife takes unit u returns real
```

Returns: Maximum life of unit. Meant to be used inside onDamage event callbacks, but must provide proper behaviour at all times.

3.2.1.1.6 setters

```
static method setLife takes unit u, real value returns nothing
```

Effect: Sets life of unit u to value. Meant to be used inside onDamage event callbacks, but must provide proper behaviour at all times.

3.2.1.2 getters

```
constant function GetLife takes unit u returns real
```

Effect: return DDS.getLife(u)

```
constant function GetMaxLife takes unit u returns real
```

Effect: return DDS.getMaxLife(u)

3.2.1.3 setters

```
function SetLife takes unit u, real value returns nothing
```

Effect: call DDS.setLife(u, value)

3.3 Table and TableArray

3.3.1 Synopsis

```
namespace Std
    //3.3.1.1 Table
    struct Table
        //3.3.1.1.1 construction and destruction
        constructor
        destructor

        //3.3.1.1.2 modifier
        method flush takes nothing returns nothing
        method remove takes integer where returns nothing

        //3.3.1.1.3 access
        method operator [] takes integer key returns integer
        template <type T> method operator [] takes integer key returns T
        template <type T> method operator []= takes integer key, T t
                                                          returns nothing

        //3.3.1.1.4 checking
        method has takes nothing returns nothing
    endstruct

    //3.3.1.2 Table Array
    struct TableArray
        //3.3.1.2.1 construction
        constructor takes integer size

        //3.3.1.2.2 destruction
        destructor

        //3.3.1.2.3 flush
        method flush takes nothing returns nothing

        //3.3.1.2.4 size
        method size takes nothing returns integer

        //3.3.1.2.5 fetch
        method operator [] takes integer key returns Table
    endstruct
endnamespace
```

3.3.1.1 Table

3.3.1.1.1 construction and destruction

constructor

Effect: Constructs new Table instance

destructor

Effect: calls flush and destructs this.

3.3.1.1.2 modifier

method flush **takes** **nothing** **returns** **nothing**

Effect: Flushes all stored variables inside this.

method remove **takes** **integer** where **returns** **nothing**

Effect: Removes value stored inside this at position where.

3.3.1.1.3 access

method **operator** [] **takes** **integer** key **returns** **integer**

Returns: integer stored inside this at position key.

template <**type** T> **method** **operator** [] **takes** **integer** key **returns** T

Returns: T stored inside this at position key.

template <**type** T> **method** **operator** []= **takes** **integer** key, T t **returns** **nothing**

Effect: Saves t of type T into internal buffer at index key.

3.3.1.1.4 checking

method has **takes** **nothing** **returns** **nothing**

Returns: Whether there is something assigned at index key.

3.3.1.2 Table Array

3.3.1.2.1 construction

constructor **takes** **integer** size

Effect: Constructs new instance of TableArray with internal buffer of size "size".

3.3.1.2.2 destruction

destructor

Effect: Calls flush and destructs this.

3.3.1.2.3 flush

method flush **takes** **nothing** **returns** **nothing**

Effect: Removes all values stored inside this.

3.3.1.2.4 size

method `size` **takes** **nothing** **returns** **integer**

Returns: Size of internal buffer.

3.3.1.2.5 fetch

method **operator** `[]` **takes** **integer** `key` **returns** `Table`

Returns: Returns Table stored inside internal buffer at index key.

Compiler Specification

1. Command line arguments

1.1 commonj=path

Specifies path to common.j file. If no path is provided, a folder where eComp is placed is used.

1.2 blizzj=path

Specifies path to blizzard.j file. If no path is provided, a folder where eComp is placed is used.

1.3 commonai=path

Specifies path to commonai.j file. If no path is provided, a folder where eComp is placed is used.

1.4 map=path

Path to map archive which should be compiled. If no such map archive exists, compiler issues error.

1.5 script=path

Path to script which should be parsed. If no such script exists, compiler issues error.
If map flag is provided, script flag is omitted.

1.6 mapout=path

A path to map archive that the compiled script should be outputted to. If no such map archive exists, a one will be created. If map flag is provided, mapout flag is omitted and the script will be outputted to the same map archive.

1.7 scriptout=path

A path to Jass script that the compiled script should be outputted to. If no such script exists, a one will be created. Possible path option is also 'script', which will output the compiled script into the same script file.

1.8 /OX

Specifies optimization level. Possible values:

0 - no optimizations should be run
1 - full optimizations

If flag is not provided, /O1 is assumed.

1.9 debug=on, debug=off

Specifies whether map script should be compiled with debug mode or not. If flag is not provided, debug=off is assumed.

1.10 compdebug=on, compdebug=off

Specifies whether the Compiler should run in debug mode or not.
When the Compiler runs in debug mode, it will forcefully create backups of map archive or script it is compiling before compilation starts, and will output debug messages when certain conditions are met. If flag is not provided, jhdebug=off is assumed.

1.11 -help

prints help message.

1.12 forcebackup=X

Specifies whether a forced backup of compiling map/script and resulting map/script(if already exists) should be created. Possible values:

false - Do not forcefully back up

true - Do forcefully back up

If flag is not provided, forcebackup=true is assumed.

1.13 networking=X

Specifies whether network access should be granted to the Compiler. Possible values:

false - Do not allow eComp to connect to internet

true - Do allow eComp to connect to internet

If flag is not provided, networking=true is assumed.

1.14 maxinlinesize=X

Specifies at how many bytes should function no longer be inlined.
Possible values: If number lower than 0, or number bigger than 2,147,483,647 is used, maxinlinesize is ignored. Numbers should be passed without ",", otherwise they are considered invalid. If flag is not provided, maxinlinsize=50 is assumed.

1.15 configfile=path

Specifies where should configuration file be looked for at. If no such configuration file exists, a configuration file from folder where eComp is placed will be loaded. If that configuration file does not exist either, this flag is omitted. If flag is not provided, a automatic path will be assumed which is equal to path eComp sits in.

1.16 templatemaxdepth=X

Specifies what is the maximum depth template can try to instantiate certain eJass

Construct(instantiation within instantiation). Possible values: If number lower than 1, or number bigger than 1000 is used, templatemaxdepth is ignored. If flag is not provided, templatemaxdepth=300 is assumed.

1.17 -version

Prints current version of the Compiler as well as creator's name.

1.18 -credits

Prints list of contributors to eComp.

1.19 -contact

Prints contact information.

1.20 -about

Alias for -version -credits -contact.

1.21 -encryptmethod=X

Specifies what encryption method for encrypted local strings should be used.
Possible values: Nan, None, No, BAE, BSE, ME.
Values explanation:

Nan, None, No - no encryption shall be used, even if user required encrypted strings.

BAE - Basic addition encryption

All required strings are encrypted in such way that their letters get changed ASCII values, and are decrypted when use is requested.

BSE - Basic shuffle encryption

All required strings are encrypted in such way that their letters get changed positions, and are decrypted when use is requested.

ME - Mixed Encryption

All required strings are encrypted in such way that their letters get changed ASCII values as well as changed position, and are decrypted when use is requested.

If flag is not provided, -encryptmethod=BAE is assumed.

2. Configuration File

Configuration file can contain additional options for compiler, easing command line options.
Configuration file should always start with [JH CONFIG FILE] and then options should follow. Configuration file can additionally contain single line contents with the same syntax as the language.

2.1 LOOKUPPATHS="VAL1", "VAL2", ...

Specifies lookup paths for relative paths passed in command line

Plugin Specification

To come.