4 Language Design

Based on the traits described in Section 3 a language design for Ecma Enhanced has been created. The language design describes the syntax, behaviour and compilation behaviour. A few defintions are defined below to compress the length of the language design specification. **Terms**

- **item**: collection term for a variable / function / enum defined in a scope;
- **Either**: This term is used when decision has to made in the future because it requires more research or would have drastic impact on the language.
- **expr**: A executable statement
- **const-expr**: A at compile time executable statement
- **ident**: An identifier
- **Ident**: An identifier (starting with capital letter)
- **type**: A typestatement.
- **string**: a string literal
- **number**: a numerical literal

The syntax defintions use a standardised notation. In short, the syntax defintion is parsed using the same rules as the tokeniser. The resulting identiefiers have to be matched explicitly, an identifier between and connected to angle brackets has to be match according to specified terms. The vertical pipe character is used to indicate that the token left or right has to be matched. The question mark makes the token to its left optional to match. The square brackets define a group of tokens as one. Any punctuation between backticks is matched literally. A token followed by any punctuation followed by a plus sign indicates a list of tokens seperated by said punctuation and may end with said punctuation. If instead of a plus sign, an asteriks is used, then the list may not end with a punctuation. A pair of curly braces containing only two dot characters indicates a curly braced body with tokens inside.

4.1 Declarations

4.1.1 Import declaration

4.1.1.1 Syntax

```
import {[<ident> | [<ident> as <ident>]],+} from <string> | <Ident>

import * from <string>

import * as <ident> from <string>

import <string>
```

4.1.1.2 Behaviour

- includes a file globally relative to the module
- import all exports from the imorted module to the importing module's global scope, or as module object with a defined name, or import objects from a module or enum type

4.1.1.3 Compilation

This statement is omited in compilation

4.1.2 Export declaration

4.1.2.1 Syntax

```
export <statement>

export {[<ident>,+]}

export * from <string>

export * as <ident> from <string>
```

4.1.2.2 Behaviour

- Adds a raises the export flag for a statement
- Marks a list identifiers as exported
- Export all specified module's exports
- Export all specified module's exports as an module object with the specified name

4.1.2.3 Compilation

This statement is omited in compilation

4.1.3 Let declaration

```
let <ident> = <expr>
let <ident> : <type> = <expr>
```

4.1.3.1 Behaviour

- Let declarations have an optional type.
- The defined type can be used as an type hint for the **expr**.
- Assert that the value-type of the **expr** does match the defined type.
- In the case no type was defined, the value-type of the expression will be used as the type of the variable.
- Defines a variable with type accessible to the current scope and child scopes.
- **Either**
  - A variable cannot be defined if the current scope already has a item of the same name.

    ```
    let x = 0;
    let x = 4; //error, variable x already exists
    ```
  - A variable will be overwritten after the let declarations in the current scope that already has an item of the same name.

    ```
    let x = 4;//x is an int
    let x = 0;//x is an int
    let x = "str";//x is an string
    ```
- The variable is reasignable;

4.1.3.2 Compilation

- Should compile to a JavaScript Let-statement. See **Operators.Assign** for more details

4.1.4 Function declaration

```
function <ident> [< [<type>],+ >]? ([<ident> : <type>],+) : <type> {..}
```

4.1.4.1 Behaviour

- Optional generics.
- Can only be defined at the global scope
- May be marked as exported
- May be marked as internal
- May be marked as having vardic args
- Only internal function can be vardic

4.1.4.2 Compilation

- Should compile to a JavaScript Function-statement.
- In case of generics, a version of this function for every possible used combination of generic functions should be compiled.
- Internal functions will not be compiled only type checked

4.1.5 Enum declaration

```
enum <Ident> {
   [<Ident> | [<Ident> = <int>] | [<Ident>([<type>],+)] | [<Ident>([<type>],+) = <int>]
| [<Ident>{[<ident> : <type>],+}] | [<Ident>{[<ident> : <type>],+} = <int>]],+
}
```

4.1.5.1 Behaviour

- Enum cases have an optional discriminator.
- The discriminator must be an int or a **const-expr with value-type int**
- An enum case may hold either a tuple type definition, a structured type definition or no type definition
- An enum without cases becomes an algebraic **never** and must be treated by the compiler as such
- When a case doesn't have a discriminator explicitly assigned the lowest possible not already taken integer will be its discriminator
- Enum values are pattern matchable
- Enums are not variables, as such `let a = MyEnumType` will fail.
- Enums-values are pattern matchable
- Enum-cases are a type of their own:

  ```
  enum Color {
      Red,
      Green,
      Blue
  }
  import {Red,Green,Blue} from Color
  //Color.Red == Red
  ```

4.1.5.2 Compilation

- A plain enum cases will be compiled to its int-discriminator form

  ```
   enum Color {
      Red,
      Green,
      Blue=0
  }
  console.log(Color.Red,Color.Green,Color.Blue)//1 2 0
  ```
- A data-holding enum will use the following compilation strategy:
  - If 1 case holds a tuple => don't include discriminator in tuple
  - If 2 or more cases hold a tuple => include discriminator in tuple
  - If 1 case holds a structure => don't include discriminator in structure
  - If 2 or more cases hold a structure => include discriminator in structure
  - If only 1 case doesn't hold data and there are others that do => let that case's compiled value be `null`

4.1.6 Struct declaration

4.1.6.1 Syntax

```
struct <Ident> [< [<type>],+ >] {
    [<ident> : <Ident>],+
}
```

4.1.6.2 Behaviour

- Can only be defined at the top level
- Optional Generics

4.1.6.3 Compilation
This statement does not compile

4.1.7 Implementation

4.1.7.1 Syntax
```
implement <type> {..}
```

4.1.7.2 Behaviour
- Can only be defined at the top level
- The body must only include function declarations
- all functions inside the body are marked as associative
- all functions in the body must have a first parameter named self with the type being the same as the type that is being implemented.
- Implementations are globally defined and do not have to be imported
- Only the file that defines a type can implement it

4.1.7.3 Compilation
- For types that are not genric compile all functions according to Section 4.1.4.2
- for types that do have generics publicise all functions as having generics and compile them according to Section 4.1.4.2

4.1.8 Extension

4.1.8.1 Syntax
```
extension <type> {..}
```

4.1.8.2 Behaviour
- Can only be defined at the top level
- The body must only include function declarations
- all functions inside the body are marked as associative
- all functions in the body must have a first parameter named self with the type being the same as the type that is being implemented.
- Extensions have to be imported and are only in affect if they are in the modules scope
- All files can declare extensions

4.1.8.3 Compilation
see Section 4.1.7.3

4.2 Statements

4.2.1 Match

4.2.1.1 Syntax
```
match (<expr>) {
  [<literal> | <ident> | [<Ident>.* ([literal> | <ident>],+)] => {..}], +
}
```

4.2.1.2 Behaviour
- performs pattern matching on a value.

4.2.1.3 Compilation
- compiles to a series of if-else statements to properly match a type

- enums values using null pointer optimalisation will be checked using a null check

4.2.2 If

4.2.2.1 Syntax

```
if (<expr>) {..} [[else if (<expr>) {..}]+]? [else {..}]?
```

4.2.2.2 Behaviour

- identical to javascript implementation

4.2.2.3 Compilation

- compiles to a javascipt if else if else chain

4.2.3 For loop

4.2.3.1 Syntax

```
for (<ident> in <expr>) {..}
```

4.2.3.2 Behaviour

- expr must be an iterator
- loops over all elements in the iterator

4.2.3.3 Compilation

- compiles to a javascript for in loop
- in case of iteration over a range => compile to a javascript `for(let ident = range_start;ident < range_end;++ident);`

4.2.4 While loop

4.2.4.1 Syntax

```
while(<expr>) {..}
```

4.2.4.2 Behaviour

- same as a javascript while loop

4.2.4.3 Compilation

- compiles to a javascript while loop

4.2.5 Loop

4.2.5.1 Syntax

```
loop {..}
```

4.2.5.2 Behaviour

- same as a javascript `while(true)` loop

4.2.5.3 Compilation

- compiles to a javascript `while(true)` loop

4.2.6 Macro

4.2.6.1 Syntax

```
//! <expr> <expr>
```

4.2.6.2 Behaviour

- adds a flag defined by the first expression to the seccond expression

4.2.6.3 Compilation

- only the second expression gets compiled

4.3 Expressions

4.3.1 Function call

4.3.1.1 Syntax

`<expr> (<expr>,+)`

4.3.1.2 Behaviour

- left hand expression must be a callable

4.3.1.3 Compilation

- if the callable is an associative function then compile to javascript as `<expr:function_name>(<expr:root_value>,<expr>,+)`
- else compile as a JavaScript function call

4.3.2 Math

4.3.2.1 Syntax

``<expr> `+`|-|`*`|/ <expr>``

4.3.2.2 Behaviour

- order of opperations is preserved
- performs the mathmatical operation as it would in javascript
- the left and right hand side must be of the same type

4.3.2.3 Compilation

- same as in javascript

4.3.3 Assign

4.3.3.1 Syntax

`<expr> = <ident>`

4.3.3.2 Behaviour

- left and right must be of the same type

4.3.3.3 Compilation

- same as in javascript.

4.3.4 Dot

4.3.4.1 Syntax

`<expr> . <ident>`

4.3.4.2 Behaviour

- get a property of of a value
- the propery must exists on the object

4.3.4.3 Compilation

- same as in javascript unless the property is an associative function in which a direct refrence to the associative function is returned

4.3.5 Range

4.3.5.1 Syntax

```
<number> .. <number>
```

4.3.5.2 Behaviour

- The first number must be smaller than the first number

4.3.5.3 Compilation

- returns an iterator ranging from the first to second number

4.3.6 Math Assign

4.3.6.1 Syntax

```
<expr> `+`|-|`*`|/ =  <expr>
```

4.3.6.2 Behaviour

- left and right must be of the same type

4.3.6.3 Compilation

- same as in javascript.

4.3.7 Binary Comparison

4.3.7.1 Syntax

```
<expr> ==|>=|<=|>|<|!=  <expr>
```

4.3.7.2 Behaviour

- left and right must be of the same type
- always returns a boolean

4.3.7.3 Compilation

- same as in javascript.