Electron Programming Language Specification

Version 0.9.0
Madd Games
October 2016

Table of Contents

1. Introduction

The *Electron* programming language is designed, along with its standard library, to be an advanced, cross-platform, object-oriented language with emphasis on parallel computing, It also consists of an ABI which guarantees that precompiled libraries may continue to be linked into programs generated by newer versions of a compiler, or other compilers, unlike for example C++ which tends to require libraries to be static and made for a certain compiler version. Electron creates a fully object-oriented native environment through polymorphism implemented using standard data structures and help from a runtime library, mixed with link-time dynamic binding. This interface is fully specified in the *Electron Generic Binary Interface (EGBI) Specification*, along with supplementary specifications for specific architectures and operating systems.

It is possible, in theory, to use the EGBI with other programming languages, as long as an EGBI-compatible compiler exists for them. Electron, however, is designed from the ground up to be compatible with the EGBI. Therefore, this document will specify the language as well as how a compiler should map it to the EGBI.

The programming language provides strong typing, polymorphism, generic typing, explicit atomicity, exception handling, automatic memory management, error detection and other features designed to make the language perfect for large applications.

1.1. Grammar

The grammar of Electron is specified below. Further sections shall refer to the symbols.

```
<int-literal> ::= <dec-int value> | <hex-int value> | <oct-int value>
<float-literal> ::= <float value>
<str-literal> ::= <string value>
<true> ::= 'true'
<false> ::= 'false'
<bool-literal> ::= <true value> | <false value>
<this> ::= 'this'
<null> ::= 'null'
<path> ::= <identifier id> '.' <path next> | <identifier id>
<brackets> ::= '(' <expression expr> ')'
<primary-expr> ::= <brackets expr> | <int-literal expr> | <str-literal expr> |
<bool-literal expr> | <this expr> | <null expr> | <path expr> | <float expr>

<pre-inc> ::= '++' <primary-expr sub>
<post-inc> ::= <primary-expr sub> '++'
<pre-dec> ::= '--' <primary-expr sub>
<post-dec> ::= <primary-expr sub> '--'
<incdec-expr> ::= <pre-inc expr> | <post-inc expr> | <pre-dec expr> | <post-dec
expr> | <primary-expr expr>

<generic-arg-expr> ::= <type-name type> ',' <generic-arg-expr next> | <type-name
type>
<generic-arg-list> ::= '<' <generic-arg-expr head> '>'
```

<type-name> ::= <primitive *prim*> | <path *path*> <generic-arg-list *generic-args*> |
<path *path*>

<argument-list> ::= <expression *value*> ',' <argument-list *next*> | <expression
value>
<new-expr> ::= 'new' <type-name *class-name*> '(' <argument-list *args*> ')' | 'new'
<type-name *class-name*> '(' ')'
<func-expr> ::= <incdec-expr *sub*> '(' <argument-list *args*> ')' | <incdev-expr *sub*>
'(' ')'
<call-expr> ::= <new-expr *expr*> | <func-expr *expr*> | <incdec-expr *expr*>

<logical-not> ::= '!' <call-expr *sub*>
<bitwise-not> ::= '~' <call-expr *sub*>
<unary-expr> ::= <logical-not *expr*> | <bitwise-not *expr*> | <call-expr *expr*>

<cast-expr> ::= '(' <type-name *dest-type*> ')' <unary-expr *sub*>
<secondary-expr> ::= <cast-expr *expr*> | <unary-expr *expr*>

<mul-op> ::= '*'
<div-op> ::= '/'
<mod-op> ::= '%'
<muldiv-op> ::= <mul-op *op*> | <div-op *op*> | <mod-op *op*>
<muldiv-expr> ::= <secondary-expr *left*> <muldiv-op *op*> <muldiv-expr *right*> |
<secondary-expr *expr*>

<add-op> ::= '+'
<sub-op> ::= '-'
<addsub-op> ::= <add-op *op*> | <sub-op *op*>
<addsub-expr> ::= <muldiv-expr *left*> <addsub-op *op*> <addsub-expr *right*> | <muldiv-
expr *expr*>

<shl-op> ::= '<<'
<shr-op> ::= '>>'
<shift-op> ::= <shl-op *op*> | <shr-op *op*>
<shift-expr> ::= <addsub-expr *value*> <shift-op *op*> <shift-expr *count*> | <addsub-
expr *expr*>

<instanceof-expr> ::= <shift-expr *sub*> 'instanceof' <type-name *class-name*> |
<shift-expr *expr*>

<smaller-expr> ::= <instanceof-expr *left*> '<' <instanceof-expr *right*>
<larger-expr> ::= <instanceof-expr *left*> '>' <instanceof-expr *right*>
<smaller-equal-expr> ::= <instanceof-expr *left*> '<=' <instanceof-expr *right*>
<larger-equal-expr> ::= <instanceof-expr *left*> '>=' <instanceof-expr *right*>
<equal-expr> ::= <instanceof-expr *left*> '==' <instanceof-expr *right*>
<not-equal-expr> ::= <instanceof-expr *left*> '!=' <instanceof-expr *right*>
<compare-expr> ::= <smaller-expr *expr*> | <larger-expr *expr*> | <smaller-equal-expr
expr> | <larger-equal-expr *expr*> | <equal-expr *expr*> | <not-equal-expr *expr*> |
<instanceof-expr *expr*>

<bitwise-or-op> ::= '|'
<bitwise-and-op> ::= '&'
<bitwise-xor-op> ::= '^'
<bitwise-op> ::= <bitwise-or-op *op*> | <bitwise-and-op *op*> | <bitwise-xor-op *op*>
<bitwise-expr> ::= <compare-expr *left*> <bitwise-op *op*> <bitwise-expr *right*> |
<compare-expr *expr*>

```
<logical-or-op> ::= '||'
<logical-and-op> ::= '&&'
<logical-op> ::= <logical-or-op op> | <logical-and-op op>
<logical-expr> ::= <bitwise-expr left> <logical-op op> <logical-expr right> |
<bitwise-expr expr>

<simple-assign-op> ::= '='
<add-assign-op> ::= '+='
<sub-assign-op> ::= '-='
<mul-assign-op> ::= '*='
<div-assign-op> ::= '/='
<mod-assign-op> ::= '%='
<bitwise-and-assign-op> ::= '&='
<bitwise-or-assign-op> ::= '|='
<bitwise-xor-assign-op> ::= '^='
<logical-and-assign-op> ::= '&&='
<logical-or-assign-op> ::= '||='
<shl-assign-op> ::= '<<='
<shr-assign-op> ::= '>>='
<assign-op> ::= <simple-assign-op op> | <add-assign-op op> | <sub-assign-op op> |
<mul-assign-op op> | <div-assign-op op> | <mod-assign-op op> | <bitwise-and-assign-
op op> | <bitwise-or-assign-op op> | <bitwise-xor-assign-op op> | <logical-and-
assign-op op> | <logical-or-assign-op op> | <shl-assign-op op> | <shr-assign-op op>
<assign-expr> ::= <logical-expr left> <assign-op op> <assign-expr right> |
<logical-expr expr>

<expression> ::= <assign-expr expr>
```

2. Lexical Parsing

A plain-text file is first passed through a *lexer* to divide it into a list of *tokens* which are then interpreted according to structures defined in the remaining sections of this document in BNF. The lexer first initializes its pointer to the beginning of the source file, and starts with an empty token list. It then tries to match what it sees with one of the regular expressions specified below. It decides the *type* of the token based on which expression was matched, and its *value* is made form the first characters of the source file, which match the expression. The pointer is then incremented past those characters, the token is added to the list (if it is not an *irrelevant* token), and the process is repeated until the end of the file. After this process, the token list is scanned again, and all tokens of type `<identifier>` are converted to one of other types of tokens, if their value matches one on one of the lists defined below in *Special Identifiers*. This is used for reserved keywords, etc.

There are 2 token type groups: *relevant* tokens and *irrelevant* tokens. Irrelevant tokens are deleted from the token list, and hence ignored by the compiler; those are white spaces and comments. They still need to be parsed by the lexer because, for example, white spaces may separate tokens.

At the end of the process, a special token called `<eof>` is added to the end of the list.

2.1. Relevant Tokens

The following tokens are *relevant*.

Type	**Regular Expression**
`<identifier>`	`[_a-zA-Z][_0-9a-zA-Z]*`
`<char>`	`(\'[^']\'|\'\\.\')`
`<float>`	`[0-9]*\.[0-9]+(e[\+\-][0-9]+)?[fF]?`
`<dec-int>`	`[1-9][0-9]*`
`<hex-int>`	`0x[0-9a-fA-F]+`
`<oct-int>`	`0[0-7]*`
`<string>`	`\"(\\.|\\\"|[^"])*\"`
`<operator>`	*See below*

A regular expression to match an operator is extremely complex so instead we define it using NBF as follows:

```
<operator> ::= '<<=' | '>>=' | '<<' | '>>' | '!=' | '==' | '[]' | '<=' | '>=' |
'&&' | '||' | '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '^=' | '|=' | '++' | '--' |
',' | '.' | '+' | '-' | '*' | '/' | '%' | '!' | '=' | '(' | ')' | '[' | ']' | '{' |
'}' | ';' | '<' | '>' | '|' | '&' | '^' | '~' | ':'
```

2.2. Irrelevant Tokens

The following tokens are *irrelevant* and will be removed from the token list upon being matched.

Type	Regular Expression
<whitespace>	$+
<line-comment>	\/\/(%n)!*%n
<block-comment>	\/*(*\/)!**\/

2.3. Special Identifiers

When an <identifier> token has one of the value specified below, its type must be changed.

2.3.1. Keywords <keyword>

The following identifiers must have their type changed to <keyword>.

extern	class	interface	extends
implements	package	import	void
as	instanceof	return	if
for	while	break	continue
new	do	switch	case
default	else	try	catch
atomically	true	false	this
null			

2.3.2. Access Qualifier <access-qualifier>

The following identifiers must have their type changed to <access-qualifier>.

private	protected	public	

2.3.3. Storage Qualifier <storage-qualifier>

The following identifiers must have their type changed to <storage-qualifier>.

abstract	static	final	synchronized

2.3.4. Primitive <primitive>

The following identifiers must have their type changed to <primitive>.

int	uint	float	bool
int8	int16	int32	int64

uint8	uint16	uint32	uint64

3. Data Types and Expressions

In Electron, an expression is a set of tokens describing a mathematical or logical operation, which may be evaluated to produce a result of a specific type. An expression may be a *compound expression* which means it is made of one or more sub-expressions connected by operators, or it may be a *simple expression*, which does not contain sub-expressions.

3.1. Data Types

All expressions evaluate to values of a specific data type. There are 3 *kinds* of data types: *void*, which indicates the lack of a result, *primitive types* which are numeric values and *managed types* which refer to objects and are passed by reference. Primitive types are predefined by the language itself, which managed types can be defined by the programmer. The *void* type is not assignable to anything.

Furthermore, all expressions result in a *value*, which can be read. Some expressions result in an *lvalue* which refers to a variable, field, or some other form of writable storage; those can also be written to. Most notably, they are allowed to be the left side of an assignment operator (hence the name *lvalue*). An expression evaluates only to a value, unless explicitly stated that it is an *lvalue*.

3.1.1. Primitive Types

The following primitive types are currently defined.

Type Name	**Description**
`bool`	A boolean value; either `true` or `false`.
`int`	2's complement signed integer of the target architecture's natural size; 32 bits on 32-bit architectures, and 64 bits on 64-bit architectures (including 1 sign bit).
`uint`	Unsigned integer of the target architecture's natural size.
`float`	Floating-point value.
`int8`	8-bit signed integer.
`int16`	16-bit signed integer.
`int32`	32-bit signed integer.
`int64`	64-bit signed integer.
`uint8`	8-bit unsigned integer.
`uint16`	16-bit unsigned integer.
`uint32`	32-bit unsigned integer.
`uint64`	64-bit unsigned integer.

A primitive value can only be assigned to an *lvalue* of its own type. Electron does not perform implicit casts on primitives.

3.1.2. Managed Types

Managed types are interfaces and classes. A managed value can be assigned to an *lvalue* of its own type, one of its ancestor's types (hence always to *lvalues* of type `elec.rt.Object`), or of the type of one of the interfaces that it, or one of its parents, implements.

3.2. Evaluating Expressions

The *evaluator* generates lower-level code from a parse tree built from an expression according to the expression grammar. Each node has zero or more named branches, shown in italics in the grammar definition above. The evaluator is initially passed a node of type `<expression>`. It will then call itself to evaluate different types of branches starting from there, each being evaluated in a different way; the sub-sections below explain how each type is to be evaluated.

An *evaluation* is made up of the lower-level code that evaluates the expression (its *fetch code*), the type of the data returned by the evaluation (its *type*), and, if known, its *compile-time value*. If it is an *lvalue*, then it also contains the code for storing a value of its *type* at the location it indicates (its *store code*).

Evaluating some expressions requires an *operation* to be applied. Operations are defined when multiple types of expressions need to do a similar job. All operations are described in *Section 3.4. Operations*. Operations also return evaluations.

3.2.1. Expression `<expression>`

The result is an evaluation of the *expr* branch.

3.2.2. Assignment `<assign-expr>`

If the *expr* branch is present, the result is simply the evaluation of that branch.

Otherwise, there are 3 branches: *left*, *op* and *right*. The *left* branch must evaluate to an *lvalue*, otherwise a compile-time error shall be generated.

If *left* evaluates to a primitive type, then *right* must evaluate to the same type, otherwise a compile-time error is generated. Otherwise (when *left* has a managed type), *right* must evaluate to a primitive type that is assignable to the type of *left*; otherwise, a compile-time error is generated.

Depending on the node type of *op*, a different value must be used for the assignment. The evaluation that will be assigned will henceforth be referred to as *value*.

- If *op* is `<simple-assign-op>` then *value* is the evaluation of *right*.
- If *op* is `<add-assign-op>` then *value* is the result of applying the `ADD` operation on *left* and *right*.

- If *op* is `<sub-assign-op>` then *value* is the result of applying the `SUB` operation on *left* and *right*.
- If *op* is `<mul-assign-op>` then *value* is the result of applying the `MUL` operation on *left* and *right*.
- If *op* is `<div-assign-op>` then *value* is the result of applying the `DIV` operation on *left* and *right*.
- If *op* is `<mod-assign-op>` then *value* is the result of applying the `MOD` operation on *left* and *right*.
- If *op* is `<bitwise-and-assign-op>` then *value* is the result of applying the `BITWISE_AND` operation on *left* and *right*.
- If *op* is `<bitwise-or-assign-op>` then *value* is the result of applying the `BITWISE_OR` operation on *left* and *right*.
- If *op* is `<bitwise-xor-assign-op>` then *value* is the result of applying the `BITWISE_XOR` operation on *left* and *right*.
- If *op* is `<logical-and-assign-op>` then *value* is the result of applying the `LOGICAL_AND` operation on *left* and *right*.
- If *op* is `<logical-or-assign-op>` then *value* is the result of applying the `LOGICAL_OR` operation on *left* and *right*.
- If *op* is `<shl-assign-op>` then *value* is the result of applying the `SHL` operation on *left* and *right*.
- If *op* is `<shr-assign-op>` then *value* is the result of applying the `SHR` operation on *left* and *right*.

If *value* is managed, then its reference count is incremented because it will be placed in the new variable and also returned as a result of the evaluation. The value is swapped into the location indicated by *left*; and if the old value is managed, then its reference count is decremented.

This expression evaluates to *value*, which is also the new value of *left*.

3.2.3. Logical Expression `<logical-expr>`

If the *expr* branch is present, the result is simply the evaluation of that branch.

Otherwise, there are 3 branches: *left*, *op* and *right*. The *left* and *right* branches are evaluated, and the type of the *op* branch determines the exact operation:

- If *op* is `<logical-or-op>`, the result is taken from the `LOGICAL_OR` operation performed on *left* and *right*.
- If *op* is `<logical-and-op>`, the result is taken from the `LOGICAL_AND` operation performed on *left* and *right*.

3.2.4. Bitwise Expression `<bitwise-expr>`

If the *expr* branch is present, the result is simply the evaluation of that branch.

Otherwise, there are 3 branches: *left*, *op* and *right*. The *left* and *right* branches are evaluated, and the type of the *op* branch determines the exact operation:

- If *op* is `<bitwise-or-op>`, the result is taken from the `BITWISE_OR` operation performed on *left* and *right*.
- If *op* is `<bitwise-and-op>`, the result is taken from the `BITWISE_AND` operation performed on *left* and *right*.
- If *op* is `<bitwise-xor-op>`, the result is taken from the `BITWISE_XOR` operation performed on *left* and *right*.

3.2.5. Comparison Expression `<compare-expr>`

There is always an *expr* branch; the result is its evaluation.

3.2.6. Smaller-than Expression `<smaller-expr>`

There are 2 branches: *left* and *right*, which are to be evaluated. Depending on the type of their evaluations, the exact operation is determined:

- If *left* has a managed type that includes a public method named `opSmallerThan` that can take *right* as an argument, then this expression evaluates to `left.opSmallerThan(right)`. The resulting type in this case is whatever type that method returns.
- If *right* has a managed type that includes a public method named `opLargerThan` that can take *left* as an argument, then this expression evaluates to `b.opLargerThan(a)`. The resulting type in this case is whatever type that method returns.
- If *left* has a numeric type (`int`, `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` or `uint64`) then *right* must have the same type, and the result is of type `bool`. It is `true` if the value of *left* is strictly smaller than the value of *right*; otherwise `false`.
- In any other case, a compile-time error is generated.

3.2.7. Larger-than Expression <larger-expr>

There are 2 branches: *left* and *right*, which are to be evaluated. Depending on the type of their evaluations, the exact operation is determined:

- If *left* has a managed type that includes a public method named `opLargerThan` that can take *right* as an argument, then this expression evaluates to `left.opLargerThan(right)`. The resulting type in this case is whatever type that method returns.
- If *right* has a managed type that includes a public method named `opSmallerThan` that can take *left* as an argument, then this expression evaluates to `b.opSmallerThan(a)`. The resulting type in this case is whatever type that method returns.
- If *left* has a numeric type (`int`, `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` or `uint64`) then *right* must have the same type, and the result is of type `bool`. It is `true` if the value of *left* is strictly larger than the value of *right*; otherwise `false`.
- In any other case, a compile-time error is generated.

3.2.8. Smaller-then-or-equal Expression <smaller-equal-expr>

There are 2 branches: *left* and *right*, which are to be evaluated. Depending on the type of their evaluations, the exact operation is determined:

- If *left* has a managed type that includes a public method named `opSmallerEqual` that can take *right* as an argument, then this expression evaluates to `left.opSmallerEqual(right)`. The resulting type in this case is whatever type that method returns.
- If *right* has a managed type that includes a public method named `opLargerEqual` that can take *left* as an argument, then this expression evaluates to `b.opLargerEqual(a)`. The resulting type in this case is whatever type that method returns.
- If *left* has a numeric type (`int`, `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` or `uint64`) then *right* must have the same type, and the result is of type `bool`. It is `true` if the value of *left* is smaller than or equal to the value of *right*; otherwise `false`.
- In any other case, a compile-time error is generated.

3.2.9. Larger-than-or-equal Expression <larger-equal-expr>

There are 2 branches: *left* and *right*, which are to be evaluated. Depending on the type of their evaluations, the exact operation is determined:

- If *left* has a managed type that includes a public method named `opLargerEqual` that can take *right* as an argument, then this expression evaluates to `left.opLargerEqual(right)`. The resulting type in this case is whatever type that method returns.
- If *right* has a managed type that includes a public method named `opSmallerEqual` that can take *left* as an argument, then this expression evaluates to `b.opSmallerEqual(a)`. The resulting type in this case is whatever type that method returns.
- If *left* has a numeric type (`int`, `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` or `uint64`) then *right* must have the same type, and the result is of type `bool`. It is `true` if the value of *left* is larger than or equal to the value of *right*; otherwise `false`.
- In any other case, a compile-time error is generated.

3.2.10. Equal Expression `<equal-expr>`

There are 2 branches: *left* and *right*, which are to be evaluated. Depending on the type of their evaluations, the exact operation is determined:

- If *left* has a managed type that includes a public method named `opEqual` that can take *right* as an argument, then this expression evaluates to `left.opEqual(right)`. The resulting type in this case is whatever type that method returns.
- If *right* has a managed type that includes a public method named `opEqual` that can take *left* as an argument, then this expression evaluates to `b.opEqual(a)`. The resulting type in this case is whatever type that method returns.
- If *left* has a numeric type (`int`, `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` or `uint64`) then *right* must have the same type, and the result is of type `bool`. It is `true` if the value of *left* is the same as the value of *right*; otherwise `false`.
- In any other case, a compile-time error is generated.

3.2.11. Not Equal Expression `<not-equal-expr>`

There are 2 branches: *left* and *right*, which are to be evaluated. It shall operate in the same way as `<equal-expr>`, except the result is reversed such that `true` becomes `false` and vice versa.

3.2.12. `instanceof` Expression `<instanceof-expr>`

If the *expr* branch is present, the result is its evaluation.

Otherwise, there are 2 branches: *sub* and *class-name*. The *sub* branch is evaluated, and the result must have a managed type, otherwise a compile-time error is generated. The *class-name* branch must name a managed data type. This expression evaluates to a `bool` value, which is the result of testing, at run-time, whether the object returned by *sub* contains a sub-object of type *class-name*. If *sub* evaluates to `null`, the result is `false`.

The evaluation is implemented by calling the `_Elec_InstanceOf` function as described in the *EGBI Specification, Section 6.4*.

3.2.13. Bit Shift Expression <shift-expr>

If the *expr* branch is present, the result is its evaluation.

Otherwise, there are 3 branches: *left*, *op*, and *right*. The *left* and *right* branches are expressions to be evaluated. The result depends on the type of *op*:

- If *op* is <shl-op>, the result is derived by performing the SHL operation on *left* and *right*.
- If *op* is <shr-op>, the result is derived by performing the SHR operation on *left* and *right*.

3.2.14. Summation Expression <addsub-expr>

If the *expr* branch is present, the result is its evaluation.

Otherwise, there are 3 branches: *left*, *op*, and *right*. The *left* and *right* branches are expressions to be evaluated. The result depends on the type of *op*:

- If *op* is <add-op>, the result is derived by performing the ADD operation on *left* and *right*.
- If *op* is <sub-op>, the result is derived by performing the SUB operation on *left* and *right*.

3.2.15. Product Expression <muldiv-expr>

If the *expr* branch is present, the result is its evaluation.

Otherwise, there are 3 branches: *left*, *op*, and *right*. The *left* and *right* branches are expressions to be evaluated. The result depends on the type of *op*:

- If *op* is <mul-op>, the result is derived by performing the MUL operation on *left* and *right*.
- If *op* is <div-op>, the result is derived by performing the DIV operation on *left* and *right*.
- If *op* is <mod-op>, the result is derived by performing the MOD operation on *left* and *right*.

3.2.16. Secondary Expression <secondary-expr>

The result is the evaluation of the *expr* branch.

3.2.17. Cast Expression <cast-expr>

There are 2 branches: *dest-type* and *sub*. The *sub* branch is an expression, and is to be evaluated. The *dest-type* branch names the destination type. If *sub* has a type that is assignable to *dest-type*, then the result is simply the evaluation of sub, and the compiler may generate a warning of an unnecessary cast.

If *sub* has the void type, or a compile-time value of `null`, a compile-time error is generated.

If *sub* has a managed type, then *dest-type* must name a managed type, and the evaluation shall have the value of *sub*, but of type *dest-type*, and the fetch code shall make a call to `_Elec_Cast` as described in the *EGBI Specification, Section 6.5*.

If *sub* has an unsigned integer type (`uint`, `uint8`, `uint16`, `uint32` or `uint64`), and *dest-type* is another unsigned integer type, then the result has the same numeric value as *sub*, as long as the value fits in the destination type; otherwise, `elec.rt.OverflowError` is thrown. If the destination type is wider, the value is zero-extended.

If *sub* has a signed integer type (`int`, `int8`, `int16`, `int32` or `int64`), and *dest-type* is another signed integer type, then the result has the same numeric value as *sub*, as long as the value fits in the destination type; otherwise, `elec.rt.OverflowError` is thrown. If the destination type is wider, the value is sign-extended.

If *sub* has a signed integer type, and *dest-type* is an unsigned integer type, then the value must be positive, otherwise `elec.rt.UnderflowError` is thrown, and must fit within the destination type (and hence it is zero-extended if the destination is wider), otherwise `elec.rt.OverflowError` is thrown.

If *sub* has an unsigned integer type, and *dest-type* is a signed integer type, then the value must fit within the destination type, such that the sign bit is clear; otherwise, `elec.rt.OverflowError` is thrown.

If *sub* is any integer type, and *dest-type* is `float`, then the resulting value is the integer value converted to a floating-point format (for example, 5 becomes 5.0); the result is undefined if the value does not fit.

If *sub* is the `float` type, and *dest-type* is a signed integer type, the result is the integer part of the floating-point value; the result is undefined if the value does not fit.

If *sub* is the `float` type, and *dest-type* is an unsigned integer type, the rules are the same as above, but if the floating-point value is negative, then `elec.rt.UnderflowError` is thrown.

In any other case, a compile-time error is generated.

3.2.18. Unary Expression <unary-expr>

The result is the evaluation of the *expr* branch.

3.2.19. Logical NOT Expression `<logical-not>`

There is one branch, named *sub*. It is an expression, and must evaluated to a value of type `bool`, otherwise a compile-time error is generated. The result is the inverse of the value of *sub*.

3.2.20. Bitwise NOT Expression `<bitwise-not>`

There is one branch, named *sub*. It is an expression to be evaluated. If it evaluates to a managed type, then it must have a method named `opBitwiseNot`, otherwise a compile-time error is generated; if the method exists, the result is a call to that method. Otherwise, *sub* must be of an integer type, and the result is that value with all bits flipped. In any other case, a compile-time error is generated.

3.2.21. Call Expression `<call-expr>`

The result is the evaluation of the *expr* branch.

3.2.22. `new` Expresion `<new-expr>`

There are 2 branches: *class-name* and optionally *args*. The *class-name* branch names a type to instantiate. If it does not name a class type, a compile-time error is generated. The compiler shall emit a call to `_Elec_New`, as described in the *EGBI Specification*, *Section 6.1*, to perform the instantiation. This ends up calling the default constructor in every case.

If the *args* branch is present, then it forms a linked list, where each entry has a *value* branch, and all entries except the last one have a *next* branch, which points to the next argument. The *value* branch of each argument node is an expression that shall be evaluated to produce a list of arguments. The compiler shall then emit a call to a constructor that accepts the types of arguments given. If no such constructor exists, a compile-time error is generated.

If the *args* branch is not present, and the default constructor has an access level of private, a compile-time error shall be generated to prevent solely the default constructor from being executed.

The result is a new instance of the specified class, with a reference count of 1, and the type is *class-name*.

3.2.23. Function Call Expression `<func-expr>`

There are 2 branches: *sub* and optionally *args*.

3.3. Operations

3.3.1. `ADD`

The `ADD` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opAdd` that can take *b* as an argument, then this operation evaluates to `a.opAdd(b)`. The resulting type in this case is whatever type that method returns.
- If *b* has a managed type that includes a public method named `opAdded` that can take *a* as an argument, then this operation evaluates to `b.opAdded(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is the sum of their numeric values, of type `int`. If they both have compile-time values, then the result likewise has their sum as a compile-time value.
- The same logic as above is applied to the types `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.2. SUB

The `SUB` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opSub` that can take *b* as an argument, then this operation evaluates to `a.opSub(b)`. The resulting type in this case is whatever type that method returns.
- If *b* has a managed type that includes a public method named `opSubbed` that can take *a* as an argument, then this operation evaluates to `b.opSubbed(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* minus *b*, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value.
- The same logic as above is applied to the types `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.3. MUL

The `MUL` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opMul` that can take *b* as an argument, then this operation evaluates to `a.opMul(b)`. The resulting type in this case is whatever type that method returns.

- If *b* has a managed type that includes a public method named `opMulled` that can take *a* as an argument, then this operation evaluates to `b.opMulled(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* multiplied by *b*, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value.
- The same logic as above is applied to the types `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.4. `DIV`

The `DIV` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opDiv` that can take *b* as an argument, then this operation evaluates to `a.opDiv(b)`. The resulting type in this case is whatever type that method returns.
- If *b* has a managed type that includes a public method named `opDived` that can take *a* as an argument, then this operation evaluates to `b.opDived(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* divided by *b*, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value; if *b* has a compile-time value of 0, then a compile-time error is generated. If, at runtime, the value of *b* is zero, then `elec.rt.ArithmeticError` is thrown.
- The same logic as above is applied to the types `uint`, `float`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.5. `MOD`

The `MOD` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opMod` that can take *b* as an argument, then this operation evaluates to `a.opMod(b)`. The resulting type in this case is whatever type that method returns.

- If *b* has a managed type that includes a public method named `opDived` that can take *a* as an argument, then this operation evaluates to `b.opDived(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* modulo *b*, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value; if *b* has a compile-time value of 0, then a compile-time error is generated. If, at runtime, the value of *b* is zero, then `elec.rt.ArithmeticError` is thrown.
- The same logic as above is applied to the types `uint`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.6. `BITWISE_AND`

The `BITWISE_AND` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opBitwiseAnd` that can take *b* as an argument, then this operation evaluates to `a.opBitwiseAnd(b)`. The resulting type in this case is whatever type that method returns.
- If *b* has a managed type that includes a public method named `opBitwiseAnded` that can take *a* as an argument, then this operation evaluates to `b.opBitwiseAnded(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* bitwise-AND *b*, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value.
- The same logic as above is applied to the types `uint`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.7. `BITWISE_OR`

The `BITWISE_OR` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opBitwiseOr` that can take *b* as an argument, then this operation evaluates to `a.opBitwiseOr(b)`. The resulting type in this case is whatever type that method returns.

- If *b* has a managed type that includes a public method named `opBitwiseOred` that can take *a* as an argument, then this operation evaluates to `b.opBitwseOred(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* bitwise-OR *b*, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value.
- The same logic as above is applied to the types `uint`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.8. `BITWISE_XOR`

The `BITWISE_XOR` operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opBitwiseXor` that can take *b* as an argument, then this operation evaluates to `a.opBitwiseXor(b)`. The resulting type in this case is whatever type that method returns.
- If *b* has a managed type that includes a public method named `opBitwiseXored` that can take *a* as an argument, then this operation evaluates to `b.opBitwseXored(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* bitwise-XOR *b*, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value.
- The same logic as above is applied to the types `uint`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.9. `LOGICAL_AND`

The `LOGICAL_AND` operation takes 2 arguments, *a* and *b*. Both arguments must have type `bool`. This operation evaluates to `true` if both *a* and *b* are `true`; otherwise to `false`.

3.3.10. `LOGICAL_OR`

The `LOGICAL_OR` operation takes 2 arguments, *a* and *b*. Both arguments must have type `bool`. This operation evaluates to `true` if either *a* or *b* or both are `true`; otherwise to `false`.

3.3.11. SHL

The SHL operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opShl` that can take *b* as an argument, then this operation evaluates to `a.opShl(b)`. The resulting type in this case is whatever type that method returns.
- If *b* has a managed type that includes a public method named `opShled` that can take *a* as an argument, then this operation evaluates to `b.opShled(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* shifted left by *b* bits, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value.
- The same logic as above is applied to the types `uint`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.

3.3.12. SHR

The SHR operation takes 2 arguments, *a* and *b*. The exact operation depends on their type and some other conditions:

- If *a* has a managed type that includes a public method named `opShr` that can take *b* as an argument, then this operation evaluates to `a.opShr(b)`. The resulting type in this case is whatever type that method returns.
- If *b* has a managed type that includes a public method named `opShred` that can take *a* as an argument, then this operation evaluates to `b.opShred(a)`. The resulting type in this case is whatever type that method returns.
- If *a* and *b* have type `int`, the result is *a* shifted left by *b* bits, of type `int`. If they both have compile-time values, then the result likewise has a compile-time value.
- The same logic as above is applied to the types `uint`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` and `uint64`; but the types must match.
- In all other cases, a compile-time error is generated.