# Appendix B - Operators and Symbols

This appendix contains a glossary of Cairo's syntax, including operators and other symbols that appear by themselves or in the context of paths, generics, macros, attributes, comments, tuples, and brackets.

## Operators

Table B-1 contains the operators in Cairo, an example of how the operator would appear in context, a short explanation, and whether that operator is overloadable. If an operator is overloadable, the relevant trait to use to overload that operator is listed.

Operator	Example	Explanation	Overloadable?
`!`	`!expr`	Logical complement	`Not`
`~`	`~expr`	Bitwise NOT	`BitNot`
`!=`	`expr != expr`	Non-equality comparison	`PartialEq`
`%`	`expr % expr`	Arithmetic remainder	`Rem`
`%=`	`var %= expr`	Arithmetic remainder and assignment	`RemEq`
`&`	`expr & expr`	Bitwise AND	`BitAnd`
`&&`	`expr && expr`	Short-circuiting logical AND	
`*`	`expr * expr`	Arithmetic multiplication	`Mul`
`*=`	`var *= expr`	Arithmetic multiplication and assignment	`MulEq`
`@`	`@var`	Snapshot	
`*`	`*var`	Desnap	
`+`	`expr + expr`	Arithmetic addition	`Add`
`+=`	`var += expr`	Arithmetic addition and assignment	`AddEq`
`,`	`expr, expr`	Argument and element separator	
`-`	`-expr`	Arithmetic negation	`Neg`
`-`	`expr - expr`	Arithmetic subtraction	`Sub`
`-=`	`var -= expr`	Arithmetic subtraction and assignment	`SubEq`

`->`	`fn(...) -> type`, <code>&vert;...&vert; -> type</code>	Function and closure return type	
`.`	`expr.ident`	Member access	
`/`	`expr / expr`	Arithmetic division	`Div`
`/=`	`var /= expr`	Arithmetic division and assignment	`DivEq`
`:`	`pat: type`, `ident: type`	Constraints	
`:`	`ident: expr`	Struct field initializer	
`;`	`expr;`	Statement and item terminator	
`<`	`expr < expr`	Less than comparison	`PartialOrd`
`<=`	`expr <= expr`	Less than or equal to comparison	`PartialOrd`
`=`	`var = expr`	Assignment	
`==`	`expr == expr`	Equality comparison	`PartialEq`
`=>`	`pat => expr`	Part of match arm syntax	
`>`	`expr > expr`	Greater than comparison	`PartialOrd`
`>=`	`expr >= expr`	Greater than or equal to comparison	`PartialOrd`
`^`	`expr ^ expr`	Bitwise exclusive OR	`BitXor`
<code>&vert;</code>	<code>expr &vert; expr</code>	Bitwise OR	`BitOr`
<code>&vert;&vert;</code>	<code>expr &vert;&vert; expr</code>	Short-circuiting logical OR	
`?`	expr?	Error propagation	

<span class="caption">Table B-1: Operators</span>

## Non Operator Symbols

The following list contains all symbols that are not used as operators; that is, they do not have the same behavior as a function or method call.

Table B-2 shows symbols that appear on their own and are valid in a variety of locations.

Symbol	Explanation
`..._u8`, `..._usize`, `..._bool`, etc.	Numeric literal of specific type
`"..."`	String literal
`'...'`	Short string, 31 ASCII characters maximum

`_`	“Ignored” pattern binding

<span class="caption">Table B-2: Stand-Alone Syntax</span>

Table B-3 shows symbols that are used within the context of a module hierarchy path to access an item.

Symbol	Explanation
`ident::ident`	Namespace path
`super::path`	Path relative to the parent of the current module
`trait::method(...)`	Disambiguating a method call by naming the trait that defines it

<span class="caption">Table B-3: Path-Related Syntax</span>

Table B-4 shows symbols that appear in the context of using generic type parameters.

Symbol	Explanation
`path<...>`	Specifies parameters to generic type in a type (e.g., `Array<u8>`)
`path::<...>`, `method::<...>`	Specifies parameters to a generic type, function, or method in an expression; often referred to as turbofish
`fn ident<...> ...`	Define generic function
`struct ident<...> ...`	Define generic structure
`enum ident<...> ...`	Define generic enumeration
`impl<...> ...`	Define generic implementation

<span class="caption">Table B-4: Generics</span>

Table B-5 shows symbols that appear in the context of specifying attributes on an item.

Symbol	Explanation
`#[derive(...)]`	Automatically implements a trait for a type
`#[inline]`	Hint to the compiler to allow inlining of annotated function
`#[inline(always)]`	Hint to the compiler to systematically inline annotated function
`#[inline(never)]`	Hint to the compiler to never inline annotated function
`#[must_use]`	Hint to the compiler that the return value of a function or a specific returned type must be used
`#[generate_trait]`	Automatically generates a trait for an impl

`#[available_gas(...)]`	Set the maximum amount of gas available to execute a function
`#[panic_with('...', wrapper_name)]`	Creates a wrapper for the annotated function which will panic if the function returns `None` or `Err`, with the given data as the panic error
`#[test]`	Describe a function as a test function
`#[cfg(...)]`	Configuration attribute, especially used to configure a `tests` module with `#[cfg(test)]`
`#[should_panic]`	Specifies that a test function should necessarily panic
`#[starknet::contract]`	Defines a Starknet smart contract
`#[starknet::interface]`	Defines a Starknet interface
`#[starknet::component]`	Defines a Starknet component
`#[starknet::embeddable]`	Defines an isolated embeddable implementation that can be injected in any smart contract
`#[embeddable_as(...)]`	Defines an embeddable implementation inside a component
`#[storage]`	Defines the storage of a smart contract
`#[event]`	Defines an event in a smart contract
`#[constructor]`	Defines the constructor in a smart contract
`#[abi(embed_v0)]`	Defines an implementation of a trait, exposing the functions of the impl as entrypoints of a contract
`#[abi(per_item)]`	Allows individual definition of the entrypoint type of functions inside an impl
`#[external(v0)]`	Defines an external function when `#[abi(per_item)]` is used
`#[flat]`	Defines a enum variant of the `Event` enum that is not nested, ignoring the variant name in the serialization process, very useful for composability when using Starknet components
`#[key]`	Defines an indexed `Event` enum field, allowing for more efficient queries and filtering of events

<span class="caption">Table B-5: Attributes</span>

Table B-6 shows symbols that appear in the context of calling or defining macros.

Symbol	Explanation

`print!`	Inline printing
`println!`	Print on a new line
`consteval_int!`	Declare a constant that is the result of a computation of integers
`array!`	Instantiate and fill arrays
`panic!`	Calls `panic` function and allows to provide a message error longer than 31 characters
`assert!`	Evaluates a Boolean and panics if `false`
`assert_eq!`	Evaluates an equality, and panics if not equal
`assert_ne!`	Evaluates an equality, and panics if equal
`assert_lt!`	Evaluates a comparison, and panics if greater or equal
`assert_le!`	Evaluates a comparison, and panics if greater
`assert_gt!`	Evaluates a comparison, and panics if lower or equal
`assert_ge!`	Evaluates a comparison, and panics if lower
`format!`	Format a string and returns a `ByteArray` with the contents
`write!`	Write formatted strings in a formatter
`writeln!`	Write formatted strings in a formatter on a new line
`get_dep_component!`	Returns the requested component state from a snapshot of the state inside a component
`get_dep_component_mut!`	Returns the requested component state from a reference of the state inside a component
`component!`	Macro used in Starknet contracts to embed a component inside a contract

<span class="caption">Table B-6: Macros</span>

Table B-7 shows symbols that create comments.

Symbol	Explanation
`//`	Line comment

<span class="caption">Table B-7: Comments</span>

Table B-8 shows symbols that appear in the context of using tuples.

Symbol	Explanation
`()`	Empty tuple (aka unit), both literal and type
`(expr)`	Parenthesized expression
`(expr,)`	Single-element tuple expression
`(type,)`	Single-element tuple type

`(expr, ...)`	Tuple expression
`(type, ...)`	Tuple type
`expr(expr, ...)`	Function call expression; also used to initialize tuple `struct`s and tuple `enum` variants

<span class="caption">Table B-8: Tuples</span>

Table B-9 shows the contexts in which curly braces are used.

Context	Explanation
`{...}`	Block expression
`Type {...}`	`struct` literal

<span class="caption">Table B-9: Curly Braces</span>