10/25/2025

The Muni Language Specifications

Edition 1

BLONDON Azure

The Muni Language

SUMMARY

SUMMARY

SUMMARY

SUMMARY

Introduction

Muni is a small, statically typed programming language designed for simplicity, safety, and portability. Its goal is to provide a predictable and explicit environment where every operation has well-defined behavior and no hidden runtime costs.

Design goals

Muni is a small predictable language that aims to compile to *WebAssembly* with a minimal and well-specified runtime. The following goals guide all design choices in Edition 1.

- ***WebAssembly*-first**: straight mapping to WASM, traps map to WASM traps.
- **Small core**: few primitives, with a powerful structure entity.
- **Deterministic & portable**: left-to-right eval, no host quirks.
- **Explicit interop**: file/foreign imports, explicit exports.

Notation

The syntax used is Wirth syntax notation (abbreviated `Wsn`):

```Wsn
Syntax     = { Production } .
Production = production_name "=" Expression "." .
Expression = Term { "|" Term } .
Term       = Factor { Factor } .
Factor     = identifier | literal | Group | Option | Repetition .
Group      = "(" Expression ")" .
Option     = "[" Expression "]" .
Repetition = "{" Expression "}" .
```

`literals` are enclosed in double quotes `""`.

`identifiers` are in snake_case, with non-terminals having their first character uppercased.

Source code representation

Unicode and source encoding

Source files must be encoded in UTF-8 without BOM. Implementations reject files that are not valid UTF-8.

A Unicode scalar value (USV) is any code point in `U+0000..U+D7FF` or `U+E000..U+10FFFF` (surrogates excluded).

Lexical elements

Comments

`comments` are a common form of program documentation. They can be expressed in two ways:

1. `line comments` start with the character `#` and end at the end of the line.
2. `multi-line comments` start with the character sequence `/*` and end with the next nearest `*/` sequence. They can be nested.

Comments can't start in a literal.

Semicolons

The semicolon `;` acts like a separator in many different statements.

They can be omitted after the last statement of a `block`.

New line insertion is not supported.

Identifiers

`identifiers` serve to designate entities such as variables, functions and types.

They are defined thusly

```
Wsn

identifier = id_start { id_continue } .
```

Where :

1. `id_start` is any Unicode code point with the ID_Start property (per UAX#31), or `_`.
2. `id_continue` is any code point with the ID_Continue property, or `_`.

Some `identifiers` are predefined.

Keywords

These `keywords` are reserved and cannot be used as `identifiers`.

alias	as	break	continue
do	else	export	for
if	import	null	nullable
return	static	structure	until
while			

Operators & Punctuation

These sequences of characters are operators and punctuation.

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

Literals

Constants vs. literals

A `literal` is a syntactic form that denotes a value. Some literals are constant expressions (evaluable at compile time and usable where a constant is required), others are ordinary value-producing expressions (evaluated at run time).

In muni:

1. Integer, floating point, and char literals are constant expressions.
2. String and array literals construct values (not constants).

Integer

An integer literal is a sequence of digits that represents an integer constant.

Muni supports two notations:
1. **Decimal** (base 10): sequences of digits `0-9`
2. **Hexadecimal** (base 16): prefix `0x` or `0X` followed by digits `0-9, a-f, A-F`

An underscore `_` can be inserted between any two digits for readability. Underscores are ignored by the compiler and do not affect the literal's value.

```
Wsn

int_literal = decimal_literal | hex_literal .

decimal_literal = decimal_digits .
decimal_digits  = decimal_digit { [ "_" ] decimal_digit } .
decimal_digit   = "0" … "9" .

hex_literal = "0" ( "x" | "X" ) hex_digits .
hex_digits  = hex_digit { [ "_" ] hex_digit } .
hex_digit   = "0" … "9" | "a" … "f" | "A" … "F" .
```

Decimal literals

Standard base-10 integer notation.

Examples:

```muni
0
42
1000
1_000_000         # one million (underscores for readability)
2147483647        # INT_MAX (2^31 - 1)
```

Hexadecimal literals

Hexadecimal literals start with `0x` or `0X` followed by one or more hex digits.

- Case insensitive: `0xFF`, `0XFF`, `0xff`, `0Xff` are all equivalent
- Hex digits `a-f` can be uppercase or lowercase
- Must have at least one digit after `0x`

Examples:

```
0x0             # zero
0x2A            # 42 in decimal
0xFF            # 255
0x00FF00        # 65280 (green in RGB)
0x7FFF_FFFF     # INT_MAX with underscores
```

Value range and overflow

Integer literals denote signed 32-bit integers in two's complement representation.

- Valid range: `[-2^31, 2^31 - 1]` or `[-2147483648, 2147483647]`
- Decimal range: `0` to `2147483647`
- Hexadecimal range: `0x00000000` to `0x80000000`

Hexadecimal literals are interpreted like decimal literals:
- `0x00000000` to `0x7FFFFFFF` → non-negative values `0` to `2147483647`
- from `0x80000000` → out-of-range error

If a literal's magnitude exceeds the range, it is a **compile-time error**.

Examples:

```muni
int x = 2147483647;      # OK: INT_MAX
int y = 0x7FFFFFFF;      # OK: INT_MAX in hex

int z = 0x80000000;      # ERROR: exceeds INT_MAX

int bad = 2147483648;    # ERROR: exceeds INT_MAX
int bad2 = 0x100000000;  # ERROR: exceeds 32-bit range
```

Using hex for characters

Since `char` is an alias of `int`, hexadecimal literals are particularly useful for Unicode code points:

muni

```
char newline = 0x0A;         # '\n'
char space = 0x20;           # ' '
char euro = 0x20AC;          # '€'
char emoji = 0x1F600;        # '😀'

# With explicit cast for clarity:
char letter_A = as<char>(0x41);
```

Note: When using hex literals for characters, ensure the value is a valid Unicode scalar value (`0x0000..0x10FFFF`, excluding `0xD800..0xDFFF`). The validation occurs at the point of conversion to `char` type, not in the literal itself.

Negative literals

The minus sign `-` is not part of the literal, it's a unary operator.

muni

```
int x = -42;             # unary minus applied to literal 42
int y = -0xFF;           # unary minus applied to 255 → -255
```

Underscore placement rules

Underscores can appear between any two digits but not:

- At the start of the literal
- Immediately after `0x` prefix
- At the end of the literal
- Adjacent to another underscore (no `__`)

Valid:

muni

```
1_000
0xFF_FF
0x00_11_22_33
1_2_3_4_5
```

Invalid:

muni

```
_100          # ERROR: starts with underscore
0x_FF         # ERROR: after prefix
100_          # ERROR: trailing underscore
1__000        # ERROR: consecutive underscores
```

Summary

Notation	Example	Decimal value
Decimal	42	42
Decimal with _	1_000_000	1000000

Notation	Example	Decimal value
Hexadecimal	0×2A	42
Hexadecimal	0xFF	255
Hexadecimal	0×7FFFFFFF	2147483647
Hexadecimal with _	0xAC_ED	44269

Floating point

A floating point literal represents a floating point constant in decimal base.

An underscore can be added between two digits for readability. It is also removed by the compiler and doesn't change the value.

```
Wsn

float_literal = decimal_digits "." [ decimal_digits ] [ exponent ]
              | decimal_digits exponent
              | "." decimal_digits [ exponent ] .
exponent      = ( "e" | "E" ) [ "+" | "-" ] decimal_digits .
```

Char

A `char` denotes a Unicode scalar value (USV). It is represented as a 32-bit signed integer whose valid range is `0x0000..0x10FFFF`, excluding `0xD800..0xDFFF`.

```
Wsn

char_literal = "'" character "'" .
character    = unicode_scalar | "\" escape_seq .
escape_seq   = simple_escape
             | hex_byte_escape
             | unicode_short_escape
             | unicode_long_escape .
simple_escape        = "a" | "b" | "f" | "n" | "r" | "t" | "v" | """ | "'" | "\" .
hex_byte_escape      = "x" hex_digit hex_digit .
unicode_short_escape = "u" hex_digit hex_digit hex_digit hex_digit .
unicode_long_escape  = "U" hex_digit hex_digit hex_digit hex_digit
                           hex_digit hex_digit hex_digit hex_digit .
hex_digit = "0"… "9" | "a"… "f" | "A"… "F" .
line_terminator = "\n" | "\r" | "\r\n"
```

Where `unicode_scalar` is any USV except `"'"`, `"\"`, or the `line_terminator`

`\xHH` encodes a single byte (must decode to a valid USV `0x00..0xFF`).

`\uHHHH` and `\UHHHHHHHH` encode a USV, values in the surrogate range are illegal.

String

A string literal constructs a string value: a finite sequence of characters obtained after escape processing. String literals are not constants.

```Wsn
string_literal = DQUOTE { character } DQUOTE .
DQUOTE = """ .
```

1. After escape processing, the resulting sequence becomes the content of the constructed string value.
2. Adjacent string literals are not implicitly concatenated.
3. muni has no raw string literal form for the moment.

Array

An array literal constructs a new array value. It is a value-producing expression (not a constant). The element length is inferred from the literal.

```Wsn
array_literal  = "[" [ element_list [ "," ] ] "]" .
element_list   = element { "," element } .
element        = Expression .
```

Typing and inference:

1. If all elements have a common type `T` (after usual literal typing rules and numeric unification), the literal has type `array<T>` with length equal to the number of elements.
2. If the element types are not assignable by transitivity to the type of the first element, it is a compile-time error.
3. If the context provides an expected array type `array<T>`, each element must be assignable to `T`. The element type of [] must be known from context.

Evaluation:

1. Each evaluation of an array literal constructs a fresh array value.
2. Elements are evaluated left-to-right and stored into the newly constructed array.

Variables

A `variable` is a location in memory where it is possible to store a value. For a single variable, the set of all possible values is determined by its `type`.

Types

A `type` determines a set of values along with a set of operations and methods specific to those values. A type is denoted by its name and, if it is generic, by its type arguments.

Wsn

```
Type         = Simple_type | Generic_type .
Simple_type  = Type_name .
Generic_type = Type_name "<" Type_list ">" .
Type_name    = identifier .
Type_list    = Type { "," Type } .
```

The language predefines certain types, which are: `int, float, array, vec`.

muni distinguishes value types and reference types:

- Value types: int, float and any of their alias.

  Value operations act on the value itself, assignment and parameter passing copy the value.
- Reference types: array and any structure type or their alias.

  Assignment and parameter passing copy a reference to a value, operations that observe or mutate the value require a non-null reference.

The zero value of a value type is its conventional zero (e.g., `0` or `0.0`).

The zero value of a reference type is `null` (absence of a value).

Two types are identical if they have the same name and pairwise identical type, otherwise they are distinct (even if structurally similar unless explicitly aliased in the Declarations section).

Numeric types

An integer or floating point type is a numeric type.

- `int`: signed integer value represented in two's complement in 32-bit.
- `float`: IEEE-754 binary32 floating point value.

Both `int` and `float` have a fixed width of 32 bit.

Boolean

A `boolean` is a type alias for `int`. It is used to represent truth values.

The identifiers `true` and `false` are predefined constants of type `boolean`.

1. `true` has the value 1.
2. `false` has the value 0.

Any non-zero value is considered `true` in conditional contexts, and `0` is considered `false`.

Because `boolean` is an alias of `int`, `boolean` arithmetic follows that of `int`.

Character

A `char` is an alias of `int` whose valid values are Unicode scalar values (USV).

Conversions between `char` and `int` are lossless for values in `0x0000..0x10FFFF` excluding surrogates, converting an out-of-range `int` to `char` is a runtime error.

Structures

A structure is a sequence of named elements, called fields, and functions that act on the structure, called methods.

1. Each field has a name and a type, explicitly designated in the structure definition.
2. Fields and methods may be static or instance members. Static members are associated with the structure type, instance members with each value.
3. Equality, ordering, and hashing of structures are not defined by default, they may be provided by methods.

Array type

An `array<T>` denotes finite, indexable sequences of elements of type `T`. It is a reference type.

- Length is a property of each value and may differ between array values.
- Elements are addressable and mutable.
- The empty array [] is a valid non-null value with length 0.
- The zero value of type `array<T>` is `null`.
- Indexing requires a non-null receiver and a valid index; otherwise a runtime error occurs.
- `array<T>` is invariant in `T`: `array<S>` is not assignable to `array<T>` unless `S` and `T` are identical.

It is predefined with two operations, used for indexing and element assignment operations.

Vec type

`vec<T>` is a predefined structure provided by the standard library. It denotes growable, mutable sequences of elements of type `T`.

Being a structure, `vec` is a reference type.

- `vec<T>` values support dynamic resizing and element access through library-defined operations.
- Assignment and parameter passing copy the reference.
- The zero value of `vec<T>` is null.

- Indexing and method calls require a non-null receiver, otherwise a runtime error occurs.

Its specific methods and behaviors (e.g., how elements are added or removed, `_equals` hook) are defined in the standard library specification.

String type

`string` is a type alias for `vec<char>` with additional operations specified in the standard library.

It stores sequences of Unicode scalar values. Indexing returns a `char`.

Methods for encoding/decoding are library-provided.

Program structure

Top level constructs

A top-level construct is a syntactic form that may appear directly in a module (source file). Top-level constructs introduce names in the module scope. The order of top-level constructs is not semantically significant: name binding is performed for all top-level constructs before type-checking their bodies (this allows mutual recursion of functions). Unless noted otherwise, it is a compile-time error to define the same top-level name more than once in a module.

All top-level names (aliases, structures, foreign imports, functions) are bound first. After that, alias/structure types and function bodies are type-checked and validated.

The binding order is as follows:

1. first pass: bind all top-level names.
2. Second pass: type-check aliases/structures/functions. This enables mutual recursion.

Forward references are allowed to any bound top-level name.

A top-level identifier must be unique across the module.

Top-level names are private to the module unless explicitly exported.

Diagnostics should identify the construct, the name, and (when applicable) the conflicting earlier definition.

```Wsn
Program = { Top_level_construct } .
Top_level_construct = Alias_definition
                    | Import_declaration
                    | Export_declaration
                    | Structure_definition
                    | Function_definition .
```

Alias definition

Aliases can be used to define a new type name (optionally generic) equal to an existing type expression.

```Wsn
Alias_definition  = "alias" Alias_head "=" Type ";" .
Alias_head        = identifier [ "<" Type_param_list ">" ] .
Type_param_list   = identifier { "," identifier } .
```

1. Binding. Alias_head defines a new type constructor in module scope. Its arity is the number of parameters in Type_param_list.

2. Uniqueness. It is an error to reuse an existing top-level type/alias/structure name or to redeclare the same alias head.
3. Generic parameters. Type parameters are names bound only within the right-hand side Type. Shadowing is allowed, capture is not.
4. Cycles. An alias must not depend on itself through aliases without passing through a nominal type boundary (i.e., no pure alias cycles).

   Example error: `alias A = B; alias B = A;`
5. Instantiation. When applied, an alias behaves exactly as its target type. Aliases do not create distinct nominal types.

Examples:

```muni
structure couple<A, B> { A a; B b; }
alias string = vec<char>;
alias matrix<T> = array<array<T>>;
alias index<T> = couple<int, T>;
```

Instantiating an alias with the wrong number of type arguments is a compile-time error.

Import declaration

Imports enable the use of types and functions defined in other modules or provided by the host environement.

```Wsn
Import_declaration = "import" ( File_import | Foreign_import ) ";" .

File_import        = Import_path .
Import_path        = "<" file_path ">" .
; file_path: any characters except unescaped ">"

Foreign_import     = module_id "." func_id
                     "(" [ Type_list ] ")" [ "->" Type ]
                     [ "as" identifier ] .
module_id          = identifier .
func_id            = identifier .
Type_list          = Type { "," Type } .
```

Where `file_path` is any sequence of character that doesn't contain the non escaped character `>`.

There are two forms:

1. File import loads and binds a module (file) prior to analyzing the current one. It may implicitly bind that module's public names into the current module's import namespace (details in the module system spec—can stay "implementation defined" for now).
2. Foreign import declares a host-provided function with a given signature. It binds the function name in the module scope as if it were a function definition without a body (callable at runtime, no inline body).

Name binding:

- File_import binds the imported module into the import environment (re-export behavior is controlled by export, see below).
- Foreign_import binds the right-hand identifier as the callable name in this module (e.g., env.print_i32(int) → void binds print_i32), if an `"as" identifier` is provided, the callable name is the specified identifier instead of the right-hand one.

Foreign functions are type-checked against their declared signature; bodies are not available.

Re-importing the same file path is allowed and idempotent. Binding the same foreign name twice with different types is an error.

The import is implicit in every module.

Export declaration

Exports marks a previously bound top-level name as publicly visible to importers of this module.

```
Wsn
Export_declaration = "export" identifier ";".
```

The identifier must refer to a top-level alias, structure name, or function name defined (or foreign-imported) in the same module.

May appear before or after the corresponding definition (export collection happens after binding).

Exporting the same name multiple times is allowed and idempotent.

Exporting an unknown name is a compile-time error.

Structure definition / extension

Structures introduce a new nominal reference type with the given name and arity.

```
Wsn
Structure_definition = [ "extend" ] "structure" Type_name
                       [ "<" Type_param_list ">" ]
                       Structure_block .
Structure_block      = "{" { Field_declaration | Method_definition } "}" .
```

Non-static fields are forbidden in extensions (compile-time error).

The structure type constructor is bound in module scope. Method names are bound in the structure's member namespace, static members are also namespaced under the type.

Type parameters are visible in fields and methods. Arity must match at use sites, specialization/monomorphization is implementation-defined.

Multiple extensions may appear across the module, member names must not conflict with existing ones.

Zero value. The zero value of a structure type is null.

Equality & hashing are not defined by default, user methods may provide them.

Structure fields and methods

Field declaration

```
Wsn
```

```
Field_declaration            = Non_static_field_declaration
                             | Static_field_definition .
Non_static_field_declaration = Type identifier ";" .
Static_field_definition      = "static" Type identifier "=" Expression ";" .
```

Fields may use the structure type (including with type args) in positions allowed by your reference semantics.

Method definition

```
Wsn
```

```
Method_definition = [ Method_modifiers ] Type identifier "(" [ Parameter_list ] ")" Block
                  | Type "(" Parameter_list ")" Block .

Parameter_list    = Type identifier { "," Type identifier } .

Method_modifiers  = { "static" | "nullable" } .
```

Every non-static method has an implicit parameter `this` of the enclosing structure type.

If the method is `nullable`, calls with `this = null` are allowed, inside the body this may be null and may be reassigned to any value of the structure type (including `null`).

If the method is not `nullable`, passing `null` is a runtime error, inside the body `this` is non-null. Assigning `null` to `this` is a compile-time error assigning another non-null instance is allowed.

Field access through `null` traps at runtime.

Call desugaring: expr.m(args) evaluates expr, then arguments left-to-right, and calls m with this = expr.

Extending structures and aliases

A structure may be extended using the structure extension syntax. The extension applies to the underlying type, making added methods visible on all aliases of that type.

Semantics:

1. Extending an alias adds methods to its underlying type.
2. Methods are visible on all values of the alias and the underlying type.
3. Duplicate method definitions across all extensions (whether via alias or base type) are compile-time errors.

```muni
alias string = vec<char>;

extend structure string {
    int length() {
        # implementation
        return 0;
    }

    boolean is_empty() {
         return this.length() == 0;
     }
 }

 # Both string and vec<char> can use these methods
 string s = "hello";
 s.length();              # OK

 vec<char> v = "world";
 v.is_empty();            # OK: methods visible on vec<char> too
```

Warning: Extending Aliases

The compiler issues a warning when extending an alias to make the shared namespace explicit:

```
warning: extending alias 'string'
  --> stdlib.mun:15:1
   |
15 | extend structure string {
   | ^^^^^^^^^^^^^^^^^^^^^^^ extension applies to underlying type 'vec<char>'
   |
note: methods added here will be visible on both 'string' and 'vec<char>' values
```

Rationale: This warning alerts developers that their extension affects all aliases of the underlying type. This is particularly important when:

- Multiple aliases exist for the same type
- Standard library types are being extended
- Methods might have unexpected visibility

The warning can be suppressed with a pragma if the behavior is intentional:

```muni
#![allow(extend_alias)]

extend structure string {
    # ... methods
}
```

Multiple Aliases Example

When multiple aliases refer to the same type, extensions through any alias affect all of them:

```muni
alias string = vec<char>;
alias buffer = vec<char>;
alias text = string;

extend structure string {
    boolean starts_with(char c) { /* ... */ return false; }
}

# All three aliases and vec<char> gain the method
 string s = "hello";
 s.starts_with('h');      # OK

 buffer b = "world";
 b.starts_with('w');      # OK

 text t = "test";
 t.starts_with('t');      # OK

 vec<char> v = "raw";
 v.starts_with('r');      # OK
```

Design Rationale

This design keeps aliases transparent (they truly are the same type) while maintaining practical usability:

1. Aliases don't create new nominal types
2. Standard library can extend `string` without creating a distinct type
3. Warnings prevent surprises about method visibility
4. Same implementation as extending the base type

Alternative designs (creating distinct nominal types for aliases) would complicate the type system and require runtime type distinctions that contradict the "zero-cost alias" principle.

Function definition

A function definition binds an identifier, the function name, to a function.

```Wsn
Function_definition = Type identifier "(" [ Parameter_list ] ")" Block .
```

The definition binds `identifier` in the module's value namespace. Each function name appears at most once per module (no overloading).

Parameters are locals in the function scope, their names must be unique.

Every path must return a value matching the declared return type exactly.

Evaluation order. Function arguments are evaluated left-to-right before the call.

- Value types (int, float) are passed by value (copied).

- Reference types (array, structures, vec, string) pass a reference (copy of the reference).

Side effects are unspecified by the language, foreign functions may have effects.

About genericity

In a function or method signature, any `Type_name` that is not bound as an existing `type` denotes a new `type parameter` of that definition.

Because every import is treated before any structure/function definition, imported names will never be treated as a generic type.

These `type parameters` are in scope in the whole definition (params, return, body).

Structure headers keep explicit type params (e.g., `structure Box<T> { ... }`), functions/methods use implicit ones.

If a type param remains undetermined after constraints, it's a compile-time error.

Example:

```muni
T identity(T x) { return x; }

int main() {
    identity(null);  # Error: null is undetermined
}
```

Implicit_type_params:

- Collected by first appearance while scanning parameters left-to-right, then the return type.
- In scope throughout the definition (parameters, return type, body).
- Name must not collide with an existing type in scope.

For each call:

1. create fresh type vars for the callee's implicit type params,
2. generate constraints from actuals vs formals and any expected result type,
3. solve by unification,
4. substitute into the callee's signature.

If a type param remains undetermined (after any literal defaults), report a compile-time error.

Prelude

Predefined identifiers

Types

Primitive types: `int`, `float`, `array<T>`, `void`

`int` aliases: `char`, `boolean`

Structure: `vec<T>`

`vec<char>` alias: `string`

The type `void` has no values. It may only appear as the return type of a function. It is illegal to declare a variable, field, or parameter of type `void`.

Values

These identifers are reserved and cannot be redefined.

Boolean values

```
true  = 1
false = 0
```

Built-in functions

Expressions

```
                                                                    Wsn
Expression      = Assignment .

Assignment      = Conditional
                | Lhs Assign_op Assignment .

Lhs             = identifier
                | Postfix "." identifier
                | Postfix "[" Expression "]" .

Assign_op       = "=" | "+=" | "-=" | "*=" | "/=" | "%="
                | "<<=" | ">>=" | "&=" | "^=" | "|=" .

Conditional     = Logical_or .   ; (no ?: operator in this edition)

Logical_or      = Logical_and { "||" Logical_and } .
Logical_and     = Bit_or      { "&&" Bit_or } .
Bit_or          = Bit_xor     { "|"  Bit_xor } .
Bit_xor         = Bit_and     { "^"  Bit_and } .
Bit_and         = Equality    { "&"  Equality } .
Equality        = Relational { ( "==" | "!=" ) Relational } .
Relational      = Shift       { ( "<" | "<=" | ">" | ">=" ) Shift } .
Shift           = Additive    { ( "<<" | ">>" ) Additive } .
Additive        = Multiplicative { ( "+" | "-" ) Multiplicative } .
Multiplicative  = Unary       { ( "*" | "/" | "%" ) Unary } .

Unary           = Postfix
                | ( "+" | "-" | "!" | "~" ) Unary .

Postfix         = Primary { Post_op } .
Post_op         = "(" [ Arg_list ] ")"
                | "[" Expression "]"
                | "." identifier
                | "++"
                | "--" .

Arg_list        = Expression { "," Expression } .

Primary         = identifier
                | literal
                | "(" Expression ")"
                | "null" .
```

Operators

Operator precedence

Precedence	Type	Operators	direction
1	Postfix	x.f() x() x++ x-- x[i]	left-to-right

Precedence	Type	Operators	direction
2	Unary	+x -x !x ~x	right-to-left
3	Multiplicative	* / %	left-to-right
4	Additive	+ -	left-to-right
5	Arithmetic Shift	← >>	left-to-right
6	Bitwise AND	&	left-to-right
7	Bitwise XOR	^	left-to-right
8	Bitwise OR	|	left-to-right
9	Relational	< <= > >=	left-to-right
10	Equality	== !=	left-to-right
11	Logical AND	&&	left-to-right
12	Logical OR	||	left-to-right
13	Assignment	= += -= *= /= %= ←= >>= &= ^= |=	right-to-left

Short-circuiting applies to && and || (left to right).

`identifier`, `Postfix "." identifier`, `Postfix "[" Expression "]"` are assignable. `Postfix ++/--` require an lvalue of numeric type.

`<<` and `>>` require `0 <= amount < 32`, otherwise trap.

`<<` on int wraps on overflow.

`x op= y` is equivalent to `x = (x op y)` with one evaluation of the `x` receiver and one of `y`.

`&&/||` evaluate left then conditionally right, result is `boolean`.

`==/!=` on reference types compare identity, relational ops on references are compile-time errors.

Assignments yield values.

`x = y` evaluates to the assigned value (the new value), and `op=` evaluates as if `x = x op y` but with one evaluation of each side.

Bitwise/shift operators are only defined on int.

Using them on float (or structures, arrays, etc.) is a type error.

Expression evaluation model

Order of evaluation

Subexpression order: Left-to-right, depth-first

Argument evaluation: Left-to-right before function call

Receiver evaluation: Always before member access or method call

Example:

```muni
f(g(), h())    # g() evaluated, then h(), then f() called
x.y.z          # x evaluated, then .y, then .z
arr[i++]       # arr evaluated, then i++, then index operation
```

Side effects and sequencing

A side effect is any change to the program state:

1. The modification of a variable.
2. Any I/O operation (via foreign imports).
3. The modification of an array element or structure field.

Sequencing ensures that side effects occur at well defined sequence points. All side effects in a subexpression are handled before the next subexpression (in left-to-right order).

All side effects of a statement are handled before the next statement.

Sequence points can occur after evaluating
1. A statement.
2. The left part of `&&` or `||`.
3. Each function argument.
4. The receiver of a member access or method call.

They can also ocurr before and after a function call.

In a single sequence point, the order of side effects that affect different memory locations is unspecified.

Example:

```muni
# Well-defined: sequence point after each statement
x = 1;
y = x;              # y = 1

# Well-defined: left-to-right evaluation
arr[i++] = i;       # arr evaluated, i++ evaluated, i read, assignment performed

# Well-defined: short-circuit guarantees order
func1() && func2();  # func1 called first; func2 only if func1 returns true
```

Value categories

The language has two value categories:

1. lvalue (locator value): An expression that designate a memory location. They can appear on the left side of assignments.
   - Variables
   - Array elements
   - Structure fields
2. rvalue (readable value): An expression that produces a value but does not designate a memory location. They cannot appear on the left side of assignments.

- Literals
- Arithmetic expressions
- Function calls

An lvalue can be used as an rvalue. But a rvalue cannot be used as an lvalue.

Primary expressions

Primary expressions are the building blocks of all expressions.

Identifiers

An identifier used as an expression refered to a declared entity.

1. Variable: evaluates to the current value stored
2. Function: a callable entity (used in function calls)

Type: The type an identifier expression is the declared type of the entity.

Value category: Identifiers referring to variables are lvalues, others as rvalues.

An identifier is resolved to the nearest declaration in the enclosing scopes:

1. Current block scope
2. Enclosing block scopes (innermost to outermost)
3. Function parameter scope (if in a function)
4. Structure member scope (if in a method)
5. Module scope
6. Imported scope

An inner declaration shadows (hides) an outer declaration of the same name.

```muni
int x = 1;
{
    int x = 2;      # Shadows outer x
    # x evaluates to 2 here
}
# x evaluates to 1 here
```

Literals

Literal are constant values directly in source code.

Integer literals

See Integer literals

Type: `int`

Floating point literals

See Floating point literals

Type: `float`

Char literals

See Char literals

Type: `char`

String literals

See String literals

Type: `string`

Value: Each evaluation produces a new string value.

Array literals

See Array literals

Type inference:

- If all elements share a common type `T`, then the literal has type `array<T>`.
- The element type of `[]` has to be known from context.

Value: Each evaluation produces a new array value.

Null literal

The keyword literal `null` represents the absence of value for reference types.

Type: Assignable to any reference type. **Value**: The null reference.

Assigning `null` to value types is a compiler error.

Parenthesised expressions

Parentheses group subexpressions to bypass default precedence.

Type: Same as subexpression

Value: Same as subexpression

Postfix expressions

Postfix expressions apply operations to a primary expression or a postfix expression.

Function calls

A function call invokes a function with zero or more arguments.

The callable expression must be a function name or an expression with function type (member access).

The number of arguments must match the function's parameter count

Each argument must be assignable to the corresponding parameter type.

Evaluation order:

1. Evaluate the function expression (if applicable)
2. Evaluate arguments left-to-right
3. Call the function with the evaluated arguments

Type: The return type declared in the function signature

Value category: rvalue.

```muni
int add(int a, int b) { return a + b }

int result = add(2, 3);                   # result = 5
int x = add(f(), g());                    # f() called, then g(), then add

# Generic function call
T identity(T x) { return x }
int y = identity(42);                     # T inferred as int
```

Array indexing

Array indexing accesses the element of an array.

`arr[index]`

arr must be of type assignable to `array<T>`. index must be of type assignable to `int`.

Type: Element type `T`

Value category: lvalue.

Evaluation order:

1. Evaluate arr (the array expression)
2. Evaluate index (the index)
3. Check: array is non-null
4. Check: 0 <= index < array.length
5. Access the element

```muni
array<int> arr = [10, 20, 30];
int x = arr[0];                           # x = 10
arr[1] = 25;                             # arr is now [10, 25, 30]

int i = 0;
arr[i++] = 100;                          # arr[0] = 100, i = 1

array<int> null_arr = null;
int y = null_arr[0];                     # TRAP: null dereference

 int z = arr[10];                         # TRAP: index 10 >= length 3
 int w = arr[-1];                         # TRAP: negative index
```

Member access

Member access reads or writes a field of a structure

`struct.field`

`struct` must be of a structure type. `field` must name a field of that structure.

Type: The declared type of the field **Value category**: lvalue.

```muni
structure Point { int x; int y; }

Point p = Point(3, 4);
int x_val = p.x;                        # x_val = 3
p.y = 5;                                # p is now Point(3, 5)

Point null_p = null;
int bad = null_p.x;                     # TRAP: null dereference
```

Static fields are accessed through the type name, not through instances. They are rvalues.

```muni
structure Counter {
    static int count = 0;
    int value;
}

int c = Counter.count;                  # Access static field
Counter.count++;                        # ERROR: Counter.count is an rvalue
```

Method calls

A method call invokes a method on a structure instance.

`struct.method(arg_list)`

`struct` must be of structure type `method` must name a method of that structure. arguments must match the methods's parameters.

Evaluation order:
1. Evaluate Postfix (the receiver)
2. Evaluate arguments left-to-right
3. Check: If method is not nullable, receiver must be non-null
4. Call the method with this set to the receiver

```muni
structure Optional {
    int value;
    boolean has_value;

    # Non-nullable method: this cannot be null
    int get() {
        return this.value;
    }

     # Nullable method: this can be null
     nullable boolean is_present() {
         if (this == null) {
             return false;
         }
         return this.has_value;
     }
 }

 Optional opt = null;
 boolean present = opt.is_present();      # OK: nullable method
 int val = opt.get();                     # TRAP: non-nullable method, null receiver
```

Static methods are called through the type name.

```muni
structure Math {
    static int abs(int x) {
        if (x < 0) { return -x; }
        return x;
    }
}

int a = Math.abs(-5);                   # a = 5
```

Post-increment and post-decrement

Post-increment `x++` and post-decrement `x--` modify a variable and return its previous value.

They require an lvalue of type assignable to `int`.

Type: Same type as `x`. **Value**: The previous value of `x`.

Evaluation order:

1. Evaluate Postfix to obtain an lvalue
2. Read the current value (this is the result)
3. Compute value + 1 or value - 1 (wrapping)
4. Store the new value back to the lvalue

Example:

```muni
int x = 5;
int y = x++;                        # y = 5, x = 6

int arr[] = [1, 2, 3];
int i = 0;
int elem = arr[i++];                # elem = arr[0] = 1, i = 1

# Overflow wraps
int max = 2147483647;               # INT_MAX
 int overflow = max++;               # overflow = 2147483647, max = -2147483648
```

Using post-increment in complex expressions can be confusing due to order of evaluation.

```muni
int i = 0;
int x = i++ + i++;                  # x = 0 + 1 = 1, i = 2
# First i++: returns 0, i becomes 1
# Second i++: returns 1, i becomes 2
```

Unary operators

Unary operators apply to a single operand.

Unary plus

Unary minus

Logical not

Bitwise not

Binary operators

Arithmetic operators

All `int` arithmetic wraps modulo 2^32: `+ - *` and unary `-` wrap.

`/` truncates toward zero.

`a % b` satisfies `a == (a / b) * b + (a % b)` and has the sign of `a`.

Division or modulo by zero traps.

Bitwise operators

`x << n` and `x >> n` require `0 ≤ n < 32`, else trap. `<<` wraps; `>>` is arithmetic (sign-extending).

Relational operators

Relational operators are undefined on reference types.

Equality and inequality

For value types (`int`, `float`, and their aliases), `==`/`!=` compare values.

For reference types, `==`/`!=` compare identity (same reference), unless the referenced type defines an equality hook:

If a structure `S` defines a static method `static boolean _equals(S a, S b)`, then `a == b` is defined as `S._equals(a, b)` and `a != b` as `!S._equals(a, b)`. Otherwise:

- value types compare by value;
- reference types compare identity.

The standard library provides `boolean _equals(vec<T>, vec<T>)` so `==` on `vec<T>` compares elements (by transitivity, `string` compares codepoints).

Logical operators

Assignment operators

The left operand of any assignment must be an lvalue.

Simple assignment

Compound assignment

Special topics

Type inference in expressions

Constant expressions

Implicit conversions in expressions

Reference semantics

Overflow and undefined behaviour

Examples

Comprehensive expression examples

Common pitfalls

Statements

```
Wsn
```

```
Statement_list = { Statement ";" }.
Statement      = Empty_statement
                | Expression_statement
                | Variable_definition
                | Block
                | If_statement
                | While_statement
                | Do_statement
                | For_statement
                | Return_statement
                | Break_statement
                | Continue_statement .
```

Terminating statements

Empty statements

The empty statement does nothing.

```
Wsn
```

```
Empty_statement = .
```

Expression statements

Expressions are statements by themselves, but their value are discarded.

```
Wsn
```

```
Expression_statement = Expression .
```

Variable definitions

```
Wsn
```

```
Variable_definition = Type identifier [ "=" Expression] .
```

The lifetime of a local begins at its declaration (after its initializer, if any, has been evaluated) and ends at the closing brace `}` of its defining block.

If a local is declared without an initializer, it is zero-initialized:

- value types → their conventional zero (`0`, `0.0`).
- reference types → `null`.

Reads of a local are well-defined because of zero-initialization, using an uninitialized local cannot occur.

Variable assignments are expressions

Blocks

```Wsn
Block = "{" Statement_list [ Statement ] "}" .
```

In a `Block`, the trailing semicolon after the last statement may be omitted.

Each `Block` introduces a new lexical scope. Names declared in a block are visible from their declaration point to the end of that block.

A name may be redeclared in an inner scope (shadowing the outer one). Redeclaring the same name in the same scope is a compile-time error.

Declarations in a for initializer belong to a scope associated with that loop and are visible in the loop condition, post-expression, and body.

Execution maintains a stack of activation records (frames). Calling a function pushes a frame; returning pops it. Entering a block introduces a new scope and conceptually extends the current frame with storage for its locals; leaving the block ends the lifetime of those locals. Implementations may allocate all locals of a function in a single frame and reuse slots for disjoint lifetimes, provided that the observable behavior matches the abstract model (no use after end-of-lifetime, distinct concurrently-alive locals have distinct storage).

If statements

```Wsn
If_statement = "if" "(" Expression ")" Block [ "else" ( If_statement | Block ) ] .
```

Conditions in `if`, `while`, and `do ... until` must be of a type assignable to boolean.

While statements

```Wsn
While_statement = "while" "(" Expression ")" Block .
```

Do statements

Do statements are special in muni.

```Wsn
Do_statement = "do" [ "(" Expression ")" ] Block [ "until" "(" Expression ")" ";" ] .
```

`do (N) S until (C);`

Evaluate N once at loop entry; require N : int. If N <= 0, the loop does not execute.

Execute S exactly N times unless an iteration ends with C true or a break.

After each iteration of S, evaluate C : boolean; if C != 0, exit the loop (success).

continue skips to the end-of-iteration check of C.

`do S until (C);`

Equivalent to do (`INT_MAX`) S until ©; (i.e., “repeat until C” with no fixed cap).

`do (N) S;`

Equivalent to a counted loop with no until (just repeat N times).

The condition is checked after the body (classic do-until).

If both the count is exhausted and C is still false, the loop terminates without success.

For statements

```Wsn
For_statement = "for" "(" Statement ";" Expression ";" Statement ")" Block [ "else" Block ] .
```

If an `else` branch is specified, the loop will jump to it when it ends without breaking. A `continue` statement is not a `break` statement, this means that `continue` does not prevent the `else` branch from executing if the loop completes normally.

Return statements

```Wsn
Return_statement = "return" [ Expression ] .
```

In `void` functions, only return; is allowed.

In non-void functions, return expr is required and expr must match the declared return type exactly.

All control paths must return.

Break statements

```
Wsn
Break_statement = "break" .
```

Continue statements

```
Wsn
Continue_statement = "continue" .
```

Properties of types and values

Type Conversions and Assignability

Muni distinguishes between implicit conversions (assignability) and explicit conversions (casts).

- **Assignability**: Type `S` is assignable to type `T` if a value of type `S` can be used where `T` is expected without an explicit conversion.
- **Explicit conversion**: A cast expression transforms a value from one type to another, potentially with runtime checks or data transformation.

Assignability Rules

Identity

A type is always assignable to itself.

`T` is assignable to `T` for any type `T`.

Examples:

```muni
int x = 5;
int y = x;            # OK: int assignable to int

array<int> arr = [1, 2, 3];
array<int> arr2 = arr;  # OK: array<int> assignable to array<int>
```

Numeric types

Numeric types (`int` and `float`) are **not** assignable to each other.

- `int` is **not** assignable to `float`
- `float` is **not** assignable to `int`

Rationale: Prevents accidental precision loss and makes numeric conversions explicit.

Examples:

```muni
int x = 42;
float f = x;        # ERROR: int not assignable to float

float g = 3.14;
int y = g;          # ERROR: float not assignable to int
```

Aliases

Type aliases denote two assignable types.

If `alias A = T`, then:

- `A` is assignable to `T`
- `T` is assignable to `A`

Examples:

```muni
alias string = vec<char>;
alias boolean = int;

string s = "hello";
vec<char> v = s;     # OK: string is vec<char>

boolean b = true;
int i = b;           # OK: boolean is int
i = 1;
b = i;               # OK: int assignable to boolean
```

This property is transitive, if `alias A = B;` and `Alias C = A;`, then:

- `B` is assignable to `C`
- `C` is assignable to `B`

Null assignability

The literal `null` is assignable to any reference type.

Reference types include:

- `array<T>` for any `T`
- Any structure type

`null` is **not** assignable to value types (`int`, `float`, or their aliases).

Examples:

```muni
array<int> arr = null;      # OK
vec<float> v = null;        # OK
string s = null;            # OK

structure Point { int x; int y; }
Point p = null;             # OK

int x = null;               # ERROR: null not assignable to int
float f = null;             # ERROR: null not assignable to float
boolean b = null;           # ERROR: boolean is int (value type)
```

Array types

Array types are **invariant** in their element type.

`array<S>` is assignable to `array<T>` **if and only if** `S` and `T` are assignable.

Rationale: Arrays are mutable. Covariance would allow type-unsafe mutations.

Examples:

```muni
alias number = int;

array<int> arr_int = [1, 2, 3];
array<number> arr_num = arr_int;   # OK: number is assignable to int (from alias)

array<int> arr_int2 = [1, 2, 3];
array<float> arr_float = arr_int2; # ERROR: array<int> not assignable to array<float>
```

Structure types

Structure types have no subtyping relationship.

`struct S` is assignable to `struct T` **if and only if** `S` and `T` are the same structure or are aliases of each other.

Structures are nominal types, not structural: even if two structures have identical fields, they are distinct types.

Examples:

```muni
structure Point { int x; int y; }
structure Vector { int x; int y; }

alias Complex = Point;

Point p = Point(1, 2);
Vector v = p;              # ERROR: Point not assignable to Vector
Complex c = p;             # OK: Point is assignable to Complex

 string s = "hello";
 vec<char> v = s;            # OK: string is assignable to vec<char>
 s = v;                      # OK: vec<char> is assignable to string
```

Generic type assignability

For generic types, assignability follows from the rules above.

`Container<S>` is assignable to `Container<T>` **if and only if** `S` is assignable to `T` under the variance rules of `Container`.

Built-in generic types (`array`, `vec`) are invariant. User structures are also invariant by default.

Examples:

```muni
alias matrix<T> = array<array<T>>;

matrix<int> m1 = [[1, 2], [3, 4]];
matrix<int> m2 = m1;        # OK: identical types

matrix<float> m3 = m1;      # ERROR: array is invariant
```

Explicit Conversions (Casts)

When implicit conversion is not allowed, use an explicit cast expression.

Cast syntax

```
Wsn

Cast_expression = "as" "<" Type ">" "(" Expression ")" .
```

A cast has type `Type` and evaluates `Expression`, then converts it.

Examples:

```
muni

float f = as<float>(42);            # int to float
int i = as<int>(3.14);              # float to int
char c = as<char>(65);              # int to char
```

Cast expressions can appear anywhere an expression is allowed:

```
muni

float result = as<float>(x) + as<float>(y);
array<int> arr = [as<int>(1.5), as<int>(2.7)];
```

Numeric conversions

int to float

`as<float>(expr)` where `expr : int` converts the integer to the nearest representable float.

- Exact conversion for integers in range `[-2^24, 2^24]`
- May lose precision for larger magnitudes
- Never fails (no runtime trap)

Examples:

```
muni

int x = 42;
float f = as<float>(x);     # f = 42.0

int big = 16777217;         # 2^24 + 1
float f2 = as<float>(big); # May round to 16777216.0
```

float to int

`as<int>(expr)` where `expr : float` converts by truncating toward zero.

- Rounds toward zero: `as<int>(3.7) == 3`, `as<int>(-3.7) == -3`
- **Runtime trap** if the value is NaN, infinity, or outside `[INT_MIN, INT_MAX]`
- `INT_MIN = -2^31`, `INT_MAX = 2^31 - 1`

Examples:

muni

```muni
float f = 3.14;
int i = as<int>(f);         # i = 3

float g = -2.9;
int j = as<int>(g);         # j = -2

float inf = as<float>(1) / as<float>(0);
int k = as<int>(inf);       # TRAP: infinity not representable as int

float nan = as<float>(0) / as<float>(0);
int m = as<int>(nan);       # TRAP: NaN not representable as int
```

Character conversions

int to char

`as<char>(expr)` where `expr : int` validates the integer is a valid Unicode scalar value.

- Valid range: `[0x0000..0x10FFFF]` excluding `[0xD800..0xDFFF]` (surrogates)
- **Runtime trap** if the value is outside the valid range
- Since `char` is an alias of `int`, the bit pattern is unchanged, but semantic validation occurs

Examples:

muni

```muni
int x = 65;
char c = as<char>(x);       # c = 'A' (U+0041)

int y = 0x1F600;
char emoji = as<char>(y);   # emoji = '😀' (U+1F600)

int bad = 0xD800;
char c2 = as<char>(bad);    # TRAP: surrogate code point

int bad2 = 0x110000;
char c3 = as<char>(bad2);   # TRAP: outside Unicode range
```

char to int

`as<int>(expr)` where `expr : char` is always safe.

- Since `char` is an alias of `int`, this is a no-op cast (documentation only)
- Produces the Unicode scalar value as an integer
- It is stricly equivalent to an implicit assignment

Examples:

```muni
char c = 'A';
int x = as<int>(c);        # x = 65

char emoji = '😀';
int y = emoji;    # y = 0x1F600
```

Boolean conversions

Since `boolean` is an alias of `int`, conversions follow int rules and can be left implicit.

int to boolean

Direct assignment (no cast needed, since `boolean` is an alias of `int`):

```muni
int x = 1;
boolean b = x;             # OK: boolean is int

boolean b2 = 0;            # false
boolean b3 = 42;           # any non-zero is "true"
```

Reference type conversions

Structure casts

There are **no conversions** between different structure types, even if their fields match.

```muni
structure Point { int x; int y; }
structure Vector { int x; int y; }

Point p = Point(1, 2);
Vector v = as<Vector>(p);  # ERROR: no conversion defined
```

To convert, write an explicit conversion function:

```muni
Vector point_to_vector(Point p) {
    Vector v = Vector(p.x, p.y);
    return v;
}
```

Null casts

Casting `null` to any reference type is allowed and results in `null`:

```muni
array<int> arr = as<array<int>>(null);  # arr = null
```

Casting a non-null reference to a different reference type is a **compile-time error**:

```muni
array<int> arr = [1, 2, 3];
vec<int> v = as<vec<int>>(arr);  # ERROR: no such conversion
```

Invalid conversions

The following casts are **compile-time errors**:

1. Value type ↔ reference type (except null literal to reference)

```muni
int x = 42;
array<int> arr = as<array<int>>(x);  # ERROR
```

2. Reference type → value type

```muni
array<int> arr = [1, 2, 3];
int x = as<int>(arr);                # ERROR
```

3. Between unrelated structure types

```muni
Point p = ...;
Vector v = as<Vector>(p);            # ERROR
```

Type checking algorithm

Assignability check

To determine if expression `e : S` is assignable to context requiring type `T`:

1. If `S` and `T` are identical → **accept**
2. If `S` is an alias, resolve to its underlying type and retry
3. If `T` is an alias, resolve to its underlying type and retry
4. If `e` is the literal `null` and `T` is a reference type → **accept**
5. For generic types, check element type identity (invariance)
6. Otherwise → **reject** (compile-time error)

Cast validation

To validate cast expression `as<T>(e)` where `e : S`:

1. If `S` assignable to `T` → **accept** (cast is redundant but legal)
2. If `S = int` and `T = float` → **accept** (numeric widening)
3. If `S = float` and `T = int` → **accept** (numeric narrowing, runtime checked)
4. If `S = int` and `T = char` → **accept** (runtime checked)
6. If `e` is literal `null` and `T` is reference type → **accept**
7. Otherwise → **reject** (compile-time error: no such conversion)

Summary table

`S` and `T` are types that are not related.

From	To	Implicit	Explicit
T	T	✓	✓
int	float	×	✓ (as<float>)
float	int	×	✓ (as<int>, checked)
int	char	×	✓ (as<char>, checked)
char	int	✓	✓ (as<int>, no-op)
null	ref type	✓	✓
array<T>	array<T>	✓	✓
array<S>	array<T>	×	×
struct S	struct S	✓	✓
struct S	struct T	×	×
value	ref	×	×
ref	value	×	×

Examples

Valid implicit conversions

muni

```
# Identity
int x = 5;
int y = x;

# Through aliases
alias number = int;
number n = 42;
int i = n;

 alias string = vec<char>;
 string s = "hello";
 vec<char> v = s;

 # Null to reference
 array<int> arr = null;
 vec<float> v = null;
```

Valid explicit conversions

```muni
# Numeric
int x = 42;
float f = as<float>(x);      # 42.0

float g = 3.14;
int i = as<int>(g);          # 3

# Character
int code = 65;
char c = as<char>(code);     # 'A'

char letter = 'Z';
int val = as<int>(letter);   # 90
```

Invalid conversions

```muni
# No implicit numeric conversion
int x = 42;
float f = x;                 # ERROR

# No implicit narrowing
float g = 3.14;
int i = g;                   # ERROR

# No structure conversion
Point p = Point(1, 2);
Vector v = as<Vector>(p);    # ERROR

# No array element conversion
array<int> arr_i = [1, 2];
array<float> arr_f = arr_i;  # ERROR

# No value-to-reference
int x = 42;
array<int> arr = as<array<int>>(x);  # ERROR
```

Errors

Errors are divided into compile-time errors and runtime traps.

Compile-time Errors

Compile-time errors must be diagnosed before code generation. The compiler should report all errors it can detect in a single pass, but may stop after encountering errors that prevent further meaningful analysis.

Lexical Errors

1. Invalid tokens or malformed literals:
   - Invalid UTF-8 encoding in source file
   - Unterminated string literal (missing closing `"`)
   - Unterminated multi-line comment (missing closing `*/`)
   - Invalid escape sequence in char or string literal
   - Surrogate code point in `\u` or `\U` escape (`U+D800..U+DFFF`)
   - Unicode escape with too few hex digits
   - Hex literal with no digits after `0x`
   - Integer or float literal with invalid underscore placement
   - Character literal with no content or multiple characters (e.g., `''` or `'ab'`)
   - Character literal with unescaped line terminator

Syntax Errors

2. Malformed program structure:
   - Missing semicolons where required
   - Mismatched parentheses, brackets, or braces
   - Unexpected token in expression or statement
   - Invalid operator combinations
   - Incomplete or malformed declarations

Name Resolution Errors

3. Use of undefined names:
   - Reference to undefined variable, function, or type
   - Reference to undefined field or method
   - Import of non-existent file or foreign function

4. Duplicate definitions:
   - Multiple top-level constructs with the same name in a module
   - Multiple fields with the same name in a structure
   - Multiple parameters with the same name in a function
   - Redeclaration of the same variable name in the same scope

5. Circular dependencies:
   - Alias cycles (alias A depends on B, B depends on A without nominal type boundary)
   - Circular module imports that cannot be resolved

Type Errors

6. Type mismatches:
   - Expression type not assignable to target type
   - Function argument type not assignable to parameter type
   - Return expression type not matching function return type
   - Array element types not admitting a common supertype
   - Operand types incompatible with operator (e.g., `"string" + 5`)
7. Generic type errors:
   - Wrong number of type arguments (e.g., `array<int, float>` when `array` takes one argument)
   - Failed type inference (cannot determine type parameters)
   - Conflicting type constraints in generic function call
   - Type parameter used where concrete type expected
8. Invalid operations for type:
   - Indexing a non-array type (e.g., `5[0]`)
   - Calling a non-function value
   - Field access on non-structure type or primitive type
   - Method call on non-structure type
   - Arithmetic operations on non-numeric types
   - Relational operators on reference types (e.g., `array1 < array2`)
   - Modulo operator on floating point types

Control Flow Errors

9. Return statement violations:
   - `return` with expression in `void` function
   - `return` without expression in non-void function
   - Not all control paths return a value in non-void function
   - Unreachable code after unconditional `return`, `break`, or `continue`
10. Loop and branch errors:
    - `break` or `continue` outside of a loop
    - Multiple `else` clauses on the same `if` or `for` statement

Declaration Errors

11. Variable declaration errors:
    - Variable declared with type `void`
    - Static field without initializer
    - Array literal `[]` without type context
12. Structure definition errors:
    - Non-static field in structure extension
    - Structure inheriting from or extending non-existent structure
    - Duplicate method names in structure (no overloading)
    - Method parameter shadowing `this` (if `this` were explicit)
13. Function definition errors:
    - Foreign import with conflicting signatures for same function name
    - Function parameter or field declared with type `void`
    - Recursive type dependency without indirection (if applicable)

Literal and Constant Errors

14. Out-of-range literals:
    - Integer literal exceeds `[-2³¹, 2³¹-1]` range
    - Hexadecimal literal exceeds `0xFFFFFFFF`
    - Float literal exceeds representable range (becomes infinity)
15. Invalid literal conversions:
    - Cast between incompatible types (e.g., `as<array<int>>(5)`)
    - Cast between unrelated structure types
    - Cast from non-null reference to unrelated reference type

Export/Import Errors

16. Module system errors:
    - Export of undefined name
    - Import of file that cannot be read or parsed
    - Duplicate imports of conflicting foreign function signatures
    - Circular module dependencies (if not allowed)

Reserved Word Violations

17. Use of keywords as identifiers:
    - Attempt to use `if`, `while`, `return`, etc. as variable, function, or type names
    - Use of predefined identifiers (`true`, `false`, `null`) as user-defined names

Runtime Traps

Runtime traps occur during program execution and immediately abort the program unless caught by host integration. There is no language-level exception handling in this edition.

Null Reference Errors

1. Null dereference:
    - Array indexing with null receiver (e.g., `null[0]`)
    - Field access with null receiver (e.g., `null.field`)
    - Method call with null receiver on non-nullable method
    - Any operation requiring a non-null reference receiving `null`

Bounds Checking Errors

2. Array index out of bounds:
    - Index is negative
    - Index is greater than or equal to array length
    - Accessing element of zero-length array

Arithmetic Errors

3. Division by zero:
    - Integer division by zero: `x / 0` where both are `int`
    - Integer modulo by zero: `x % 0` where both are `int`
    - Note: Float division by zero produces infinity, not a trap

4. Invalid shift amounts (trap):
   - Shift amount is negative
   - Shift amount is greater than or equal to 32

Type Conversion Errors

5. Invalid cast to `char`:
   - Converting `int` value outside `[0x0000..0x10FFFF]` range
   - Converting `int` value in surrogate range `[0xD800..0xDFFF]`
6. Invalid cast to `int` from `float`:
   - Converting NaN to `int`
   - Converting positive or negative infinity to `int`
   - Converting float value outside `[INT_MIN, INT_MAX]` range

Resource Exhaustion

7. Stack overflow:
   - Excessive recursion depth
   - Function call nesting exceeds implementation limit
8. Out of memory:
   - Array allocation fails
   - String or `vec` allocation fails
   - Structure instantiation fails
   - Note: Behavior is implementation-defined but should trap rather than returning invalid values

Foreign Function Errors

9. Foreign function failure:
   - Host function signals an error condition
   - Foreign function implementation not found at runtime
   - Foreign function receives invalid arguments (host-side validation)
   - Mapping from host errors to traps is implementation-defined

Undefined Behavior (if any)

10. Implementation-specific traps:
    - Any behavior marked as "implementation-defined trap" in this specification
    - Violation of implementation-specific resource limits beyond those listed above

Error Reporting Requirements

Compile-time Error Messages

Compilers should provide:
- **Source location**: File name, line number, and column number
- **Error category**: Lexical, syntax, type, etc.
- **Description**: Clear explanation of what went wrong
- **Context**: Relevant source code snippet
- **Suggestion** (optional): Possible fix or similar names if applicable

Example:

```
error: type mismatch in assignment
  --> example.mun:5:12
   |
 5 |     int x = "hello";
   |             ^^^^^^^ expected `int`, found `string`
   |
help: use `as<int>()` to convert, or change the variable type to `string`
```

Runtime Trap Messages

Implementations should provide:

- **Trap type**: Null dereference, bounds check, division by zero, etc.
- **Stack trace**: Function call chain leading to the trap
- **Location**: Source location if debug info available

Example:

```
runtime trap: array index out of bounds
  at example.mun:12:8
  index: 5, array length: 3
  in function: process_array
  called from: main
```

Multiple Errors

Compilers may report multiple errors in a single compilation, but should:

- Stop after fatal errors that prevent further analysis
- Limit cascading errors (don't report 100 errors from one root cause)
- Prioritize errors by severity and source location

Rationale

No Undefined Behavior

Unlike C/C++, Muni avoids undefined behavior where possible:

- Arithmetic overflow should wrap (not be undefined)
- All variables are zero-initialized (no uninitialized reads)
- All pointer-like operations (indexing, field access) are checked

This makes the language safer and more predictable at a small runtime cost.

Traps vs. Exceptions

Runtime traps immediately abort rather than being catchable because:

1. Simplicity: No exception handling mechanism needed in edition 1
2. WASM compatibility: Traps map naturally to WASM traps
3. Correctness: Forces programmers to prevent errors rather than handle them

Future editions may add exception handling or `Result<T, E>` types.

Compile-time Error Recovery

The specification allows but does not require error recovery:

- Simple compilers may stop at the first error
- Advanced compilers may continue to find more errors
- Both approaches are conformant

Memory Management Model

Implementations manage lifetimes automatically (GC or reference counting). There is no user-visible `free` in Edition 1

The Muni Language Specifications

Edition 1

10/25/2025

BLONDON Azure