Abstract

This document specifies the semantics and behavior of the Hare programming language and serves to inform the development of compilers and programs for its use.

The scope of this document covers the language itself (its grammar and semantics), as well as standard library types & functions which are necessary for all Hare environments to provide. The document does not specify any additional details of the environment or the programming libraries available in that environment.

This specification is a **DRAFT**, and is not considered authoritative. Revisions to this draft are developed under the direction of the Hare project on SourceHut at *https://sr.ht/~sircmpwn/hare*, and the final specification shall be published there as well. You may also contact the editor via email to Drew DeVault <sir@cmpwn.com>.

Contents

1 Introduction

1.1 The purpose of this document is to promote the portability of Hare programming systems and to serve as a reference for implementors and users of Hare programming environments.

1.2 Numbered text in this document is authoritative unless otherwise noted.

1.3 Sentences displayed in italics are non-authoritative (they are informative).

This is an example of informative text.

1.4 The abstract and appendices are informative.

1.1 Copyright

2 Scope

2.1 This document establishes the form and semantics of the Hare programming language. It specifies:

2.1.1 The representation of Hare programs.

2.1.2 The syntax and constraints of the Hare language.

2.1.3 The semantic rules for the correct interpretation of a Hare program.

2.2 This standard does not specify:

2.2.1 The means by which program source code is processed by an interpreter or compiler.

*However, «Appendix **??**: Hare compiler conventions» provides an informative reference of common conventions for compiler and interpreter programs.*

2.2.2 The means by which the environment interprets Hare programs.

2.2.3 The minimum requirements or maximum capabilities of a system capable of interpreting Hare programs.

3 Terms and definitions

3.1 **abort**: a process in which the «5.4: Execution environment» immediately proceeds to the program teardown step «5.4.3: Execution environment».

3.2 **alignment**: a specific multiple of an octet-aligned storage address at which some data is required to be stored. The size of an object must be a multiple of its alignment.

Example *An object with an alignment of 8 may be stored at addresses 8, 16, 32, and so on; but not at address 4.*

3.3 **character**: a single Unicode code-point encoded in the UTF-8 format.

3.4 **expression**: a description of a computation which may be executed to obtain a **result** with a specific **result type**.

3.5 **expression class**: a grouping of **expression**s with similar properties or for grammatical disambiguation.

3.6 **implementation-defined**: a detail which is not specified by this document, but which the implementation is required to define.

3.7 **operand**: an input into an **operator**, together they form an **operation**. An **operand** is an expression, and the input to the **operator** shall be its result.

3.8 **padding**: unused octets added to the storage of some data in order to meet a required alignment. The value of these octets is undefined.

3.9 **size**: the number of octets required to represent some data, including padding.

3.10 **undefined**: a detail for which no definition is provided, neither by this specification nor by the implementation. Programs which rely on these details are non-conforming.

4 Conformance

4.1 "Shall" is interpreted as a requirement imposed on the implementation or program; and "shall not" is interpreted as a prohibition.

4.2 "May" is used to clarify that a particular interpretation of a requirement of this specification is considered within the acceptable bounds for conformance. Conversely, "may not" is used to denote an interpretation which is not considered conformant.

4.3 A **strictly conforming implementation** shall meet the following requirements:

4.3.1 It shall implement all of the behavior defined in the authoritative text of this specification.

4.3.2 It shall not implement any behavior which is **included** by «2.1: Scope» but is not defined by this specification.

This is to say that vendor extensions are prohibited of conformant implementations.

4.3.3 It may implement behavior which is **excluded** by «2.2: Scope» and which is not defined by this specification.

4.4 A **conforming freestanding implementation** shall implement all requirements of this specification *except* for those defined in «**??**: Runtime Library». A **conforming hosted implementation** shall implement all requirements of this specification, including those defined in «**??**: Runtime Library».

*Some language features require the implementation to provide the features defined by «**??**: Runtime Library». In order to use a conformant freestanding implementation, the program may be required to provide its own implementation of the features defined by «**??**: Runtime Library»; or refrain from using language features which require these implementations.*

5 Program environment

5.1 The implementation translates source files and executes programs in two phases, respectively referred to as the *translation phase* and the *execution phase*. The context in which these phases occur is referred to as the *translation environment* and the *execution environment*.

5.2 Translation environment

5.2.1 A Hare program consists of one or more *source files* which are provided to the translation phase. A source file shall be represented as UTF-8 text.

5.2.2 Each *source file* is a member of exactly one *module*, and the collective source files for a module form a *translation unit*. Each module may define its own private types, data, procedures, and so on, for the purpose of accomplishing its tasks. It may also *export* these types for other modules to use.

5.2.3 A Hare program may be translated incrementally, rather than all at one time. However, the composite source files of a single translation unit must be compiled together. Once compiled, an opaque representation of their *exported interface* (the identifiers and types of their exported procedures, data, and types) provides sufficient information to translate other units which require the module. The translated modules may be composed into the final program at the last step.

5.2.4 If the source files for a translation unit are not changed, the translated module may be used repeatedly without repeating the translation step.

5.3 Translation steps

5.3.1 The list of source files constituting the translation unit are identified. Steps «5.3.2: Translation steps» and «5.3.3: Translation steps» are repeated for each source file.

5.3.2 Lexical analysis is conducted on the source file, translating it into a stream of *tokens*.

Forward references: «**??**: Lexical analysis»

5.3.3 Syntax analysis is conducted on the token stream, mapping the tokens to an abstract syntax tree (AST). The Hare grammar shall define the relationships between tokens necessary to produce a valid AST.

Forward references: «**??**: Grammar»

5.3.4 Logical analysis is conducted on the ASTs. In this step, the implementation verifies the constraints imposed on the program. The result of this step is a *verified program module*.

In this step, colloquially referred to as the "check" step, a module composed of several source files is consolidated into a single verified program module.

5.3.5 Once the verified program module is obtained in the translation phase, the remainder of

the translation phase shall be completed with no further diagnostic messages, except in the case where external factors from the execution environment prevent successful completion.

Example *Memory exhaustion or lack of disk space are situations which may cause a failure in the remainder of the translation process.*

5.3.6 The verified program module is combined with any applicable external modules and translated into a single program image which is suitable for interpretation by the execution environment.

5.4 Execution environment

5.4.1 Two execution environments are defined: *hosted* and *freestanding*. The implementation must support at least one environment.

5.4.2 During *program startup*, the execution environment shall initialize all global declarations to their initial values, call all initialization functions in an unspecified order, then transfer control to the program *entry point*. The manner of this initialization is implementation-defined.

5.4.3 During *program teardown*, the execution environment shall call all finalization functions in an unspecified order, then terminate.

Forward references: «**??**: Initialization functions», «**??**: Finalization functions»

5.4.4 The freestanding environment

5.4.4.1 The name and signature of the program entry point function is undefined in the freestanding environment.

5.4.5 The hosted environment

5.4.5.1 In the hosted environment, the program entry point shall be an exported function named `main` in the global namespace. It shall have no parameters and no result type.

The signature of a conformant entry point follows:

```
export fn main() void;
```

The program shall provide this function in the global namespace.

5.4.6 Program execution

5.4.6.1 The evaluation of an expression may have *side-effects* in addition to computing a value. Calling a function or modifying an object is considered a side-effect.

5.4.6.2 If the implementation is able to determine that the evaluation of part of an expression is not necessary to compute the correct value and cause the same side-effects to

occur in the same order, it may rewrite or re-order the expressions or sub-expressions to produce the same results more optimally.

The interpretation of this constraint shall be conservative. Implementations should prefer to be predictable over being fast. Programs which require greater performance shall prefer to hand-optimize their source code for this purpose.

Forward references: «**??**: Expressions»

5.5 Diagnostics

5.5.1 If the constraints are found to be invalid during the translation phase, the implementation shall display an error indicating which constraint was invalidated, and indicate that the translation has failed in whatever manner is semantically appropriate.

On a Unix system, the semantically appropriate indication of failure is to exit with a non-zero status code.

5.5.2 In the translation environment, if the implementation is able to determine that multiple constraints are invalid, it may display several diagnostic messages.

5.5.3 If the constraints are found to be invalid during the execution phase, a hosted implementation shall abort the execution phase, display a diagnostic message, and indicate that the execution has failed in whatever manner is semantically appropriate.

6 Language

6.1 Notation

*A summary of the language syntax is given in «Appendix **??**: Language syntax summary».*

6.1.1 The notation used in this specification indicates non-terminals with *italic type*, terminals with **`bold type`**, and optional symbols use "opt" in subscript. Non-terminals referenced in the text use the expression notation. The following example denotes an optional expression enclosed in literal braces:

`{` *expression*$_{opt}$ **`}`**

6.1.2 When there are multiple options for a single non-terminal, they will either be printed on successive lines, or the preceeding authoritative text shall use the key phrase "one of".

6.1.3 Most grammatical constructs are tolerant of white-space characters inserted between their terminals. However, some are not — these shall use the key phrase "exactly" in their grammar description.

6.1.4 A non-terminal is defined with its name, a colon (':'), and the options; indented and shown with one option per line. For example, switch-cases is defined like so:

switch-cases:
    switch-case **`,`**$_{opt}$
    switch-case **`,`** *switch-cases*

6.1.5 Additionally, text may appear in the notation without italics or bold font; it appears in the same style as the authoritative text. Such examples are used to describe how a particular terminal sequence is matched when enumerating all of the possibilities is not practical.

string-char:
    Any character other than **** or **"**.

6.2 Lexical analysis

token:
    keyword
    name

6.2.1 A token is the smallest unit of meaning in the Hare grammar. The lexical analysis phase processes a UTF-8 source file to produce a stream of tokens by matching the terminals with the input text.

6.2.2 Tokens may be separated by *white-space* characters, which are defined as the Unicode code-points `U+0009` (horizontal tabulation), `U+000A` (line feed), and `U+0020` (space). Any number of whitespace characters may be inserted between tokens, either to disambiguate

from subsequent tokens, or for aesthetic purposes. This whitespace is discarded during the lexical analysis phase.

Within a single token, white-space is meaningful. For example, the string-literal token is defined by two quotation marks `"` *enclosing any number of literal characters. The enclosed characters are considered part of the string-literal token and any whitespace therein is not discarded.*

6.2.3 The lexical analysis process consumes Unicode characters from the source file input until it is exhausted, performing the following steps in order. At each step, it shall consume and discard white-space characters until a non-white-space characters is found, then consume the longest sequence of characters which constitutes a token and emit it to the token stream.

6.2.4 The terminal sequence `//` is used to mark a comment. When the lexical analyzer encounters this terminal sequence, it shall discard it and all subsequent characters until a line feed `U+000A` is encountered, then resume normal processing.

6.3 Keywords

keyword: one of:

```
abort alloc append as assert bool break char const continue def
defer delete else enum export f32 f64 false fn for free i16 i32
i64 i8 if int is len let match null nullable offset return rune
size static str struct switch true type u16 u32 u64 u8 uint
uintptr union use void _
```

6.3.1 Keywords (or *reserved words*) are terminals with special meaning. These names are case-sensitive. Keywords are reserved for elements of the syntax and shall not appear as user-defined names, in particular in «6.4: Identifiers».

6.4 Identifiers

identifier: exactly:
name
name **::** *identifier*

name: exactly:
nondigit
name *alnum*

nondigit: one of:
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
_

digit: one of:
0 1 2 3 4 5 6 7 8 9

alnum:
digit
nondigit

6.4.1 An *identifier* is a user-defined name which denotes a module, object, function, type alias, struct or union member, or enumeration member.

6.4.2 An identifier is only meaningful within a specific *scope* of the program. The scope is defined by the region of the AST in which the identifier is applicable; it may be the program, a translation unit, a sub-unit, a function, or an expression-list. The identifier is considered *visible* within the region that defines its scope.

6.4.3 A translation unit is assigned a unique *namespace* within the program. These namespaces may be nested recursively; that is to say that a translation unit may have a *parent* which is another translation unit. One translation unit may be assigned to the *root namespace*, which has no name.

6.4.4 Identifiers declared within a translation unit scope are assigned the namespace of the translation unit. The double-colon terminal **::** is used to denote the namespace of an identifier, ordered from least to most specific.

6.4.5 An identifier is either *fully-qualified* or *unqualified*. Unqualified identifiers require the context of their enclosing scope to be interpreted unambiguously. Fully-qualified identifiers are used for *exported* identifiers, and include the namespace in which they reside.

Example *The fully-qualified identifier* `sys::start::start_ha` *qualifies the un-qualified identifier* `start_ha` *with the start namespace, which is itself a member of the sys namespace.*

6.4.6 An identifier without the namespace qualification may be fully-qualified regardless, if it exists in the root namespace.

6.4.7 The implementation may define the maximum length of an identifier or name.

6.5 Types

type:
    const$_{opt}$ **!**$_{opt}$ *storage-class*

storage-class:
    scalar-type
    struct-union-type
    tuple-type
    tagged-union-type
    slice-array-type
    function-type
    alias-type
    unwrapped-alias
    string-type

scalar-type:
    integer-type
    floating-type
    enum-type
    pointer-type
    rune
    bool
    void

6.5.1 A type defines the storage and semantics of a value. The attributes common to all types are its *size*, in octets; *alignment*; its *constant* or *mutable* nature; its *error flag*, or lack thereof; and its *default value*, which may be undefined.

6.5.2 The implementation shall assign a globally unique ID to every type, in a deterministic manner, such that several subsequent translation environments, perhaps with different inputs, will obtain the same unique ID; and such that distinct types shall have distinct IDs.

This specification details under what circumstances two types are equivalent to one another, and thus shall have the same ID. For all types, any two types are distinct if their type class, their constant or mutable nature, and their error flag or lack thereof, are distinct. Each type class may impose additional distinguishing characteristics on their types, which are specified in their respective sections.

6.5.3 Some types have an undefined size. This includes function-type, and some cases of slice-array-type.

6.5.4 Some types have an undefined default value. If the default value of such a type would be used, the implementation shall instead print a diagnostic message and abort the translation phase.

6.5.5 The **const** terminal, when used in a type specifier, enables the constant flag and prohibits

write operations on any value of that type. Types without this attribute are considered mutable by default.

6.5.6 The `!` terminal, when used in a type specifier, sets the error flag for this type.

6.5.7 A scalar type, also called a *built-in* or *primitive* type, stores one value at a specific, predefined precision. Scalar types are the most basic unit in the Hare type system. Other types are referred to as *aggregate types*, with the exception of alias types, which may be either scalar or aggregate.

6.5.8 The *type class* of a type is defined for scalar types as the terminal which represents it, for example **`i32`**.

6.5.9 Integer types

integer-type: one of:

`i8 i16 i32 i64 u8 u16 u32 u64 int uint size uintptr char`

6.5.9.1 Integer types represent an integer value at a specific precision. These values are either *signed* or *unsigned*; which respectively **are** and **are not** able to represent negative integers. Zero is not negative. Integer types are considered *numeric types*.

6.5.9.2 Signed integer types shall be represented in two's complement form.

6.5.9.3 The endianness (byte order) of integer types shall be implementation-defined.

6.5.9.4 The precision of i8, i16, i32, i64, u8, u16, u32, and u64 are specified by the numeric suffix, which represents their precision in bits. Of those, types prefixed with **`u`** are unsigned, and those prefixed with **`i`** are signed.

6.5.9.5 The precision of int and uint are implementation-defined. int shall be signed, and uint shall be unsigned. Both types shall be at least 32-bits in precision. The precision in bits shall be a power of two.

6.5.9.6 The precision of size is implementation-defined. It shall be unsigned and shall be able to represent the maximum length of an array type. The precision in bits shall be a power of two.

6.5.9.7 The precision of uintptr is implementation-defined. It shall be able to represent the value of any pointer-type as an integer. It shall be unsigned.

6.5.9.8 The **`char`** type is equivalent to the **`u8`** type. However, there are limitations on its use, see «**??**: Arithmetic expressions» & «**??**: Access expressions».

6.5.9.9 The implementation is not required to provide the **`char`** type. If the implementation does not support the **`char`** type, it must print a diagnostic message and abort the translation phase for programs which attempt to utilize it.

6.5.9.10 The alignment of integer types shall be equal to their size in octets.

6.5.9.11 The default value of an integer type shall be zero.

The following table is informative.

Type	Size in bits	Minimum value	Maximum value
i8	8	-128	127
i16	16	-32708	32707
i32	32	-2147483648	2147483647
i64	64	-9223372036854775808	9223372036854775807
u8	8	0	255
u16	16	0	65535
u32	32	0	4294967295
u64	64	0	18446744073709551615
int	≥ 32	≤ -2147483648	≥ 2147483647
uint	≥ 32	0	≥ 4294967295
size	$*$	$*$	$*$
uintptr	$*$	$+$	$+$

$*$ implementation-defined

$+$ undefined

6.5.10 Floating point types

floating-type:
`f32`
`f64`

6.5.10.1 Floating-point types shall represent real numbers, i.e. numbers with both an integer and fractional part. Floating point types are considered *numeric types*.

6.5.10.2 The implementation shall represent floating-point types as IEEE 754-compatible floating-point numbers. **`f32`** shall be stored in the binary32 format, and **`f64`** shall use the binary64 format.

6.5.10.3 The alignment of float types shall be equal to their size in octets.

6.5.10.4 The default value of a float type shall be zero.

6.5.11 Rune types

6.5.11.1 The **`rune`** type represents a Unicode codepoint, encoded as a u32.

6.5.12 Enum types

enum-type:
    enum **{** *enum-values* **}**
    enum *integer-type* **{** *enum-values* **}**

enum-values:
    enum-value **,**$_{opt}$
    enum-value **,** *enum-values*

enum-value:
    name
    name **=** *expression*

6.5.12.1 Enum types, or *enumerated types*, are a subset of integer types which allows the user to assign a name to specific values. Enum types are considered *numeric types*.

6.5.12.2 If integer-type is specified, the underlying storage of the enum type is defined by the integer type specified. Otherwise, the underlying storage is **int**.

6.5.12.3 The size and alignment of an enum type is equivalent to those attributes of the underlying integer type.

6.5.12.4 If the enum-value does not specify a expression, the value assigned to that name is equal to the last value assigned to an enum-value of this enum type plus one. If no such previous value exists, zero is assigned.

6.5.12.5 An implicitly assigned value shall not exceed the precision of the underlying integer type; if it were to, a diagnostic message shall be shown instead per «5.5: Diagnostics».

6.5.12.6 expression, if specified, shall be evaluated in the translation environment and the resulting value shall be assigned to the corresponding enum-value. The expression shall be provided the enum's type's underlying integer type as a type hint. The result type must be assignable to the enum type's underlying integer type (ref «6.6.44: Assignment»).

6.5.12.7 A temporary scope shall be allocated while loading an enum type, and each value name, in order, shall be made available to that scope.

This allows the expression for each value to refer to previously declared values.

6.5.12.8 expression shall be limited to the «6.8: Translation compatible expression subset».

6.5.12.9 Each enum-value's name shall be unique within the set of all names of enum-values of the enum-type. Otherwise, a diagnostic message shall be printed and the translation phase shall be aborted.

6.5.12.10 Two enum types shall be considered equivalent if all of these conditions are met:

- The underlying storage type is the same.
- The set of assigned values and the names assigned to those values is the same, regardless of their order.

6.5.12.11 The default value of an enum type is undefined.

6.5.13 Pointer types

pointer-type:
    `*` *type*
    `nullable` `*` *type*

6.5.13.1 A pointer type is an indirect reference to an object of a secondary type. The notation of a pointer type is a `*` prefix before the secondary type.

6.5.13.2 A normal pointer type shall **always** refers to a valid secondary object. A pointer type prefixed with **`nullable`** is considered a *nullable pointer type*, and shall refer to either a valid secondary object or to a special value called *null*.

6.5.13.3 The representation of a pointer type shall be implementation-defined, but it shall have an implementation-defined length and alignment.

6.5.13.4 The default value of a nullable pointer type is null. The default value of a non-nullable pointer type is undefined.

6.5.13.5 A pointer type shall be equivalent to another pointer type only if they share the same secondary type and nullable status.

6.5.14 Other primitive types

The bool type

6.5.14.1 The **`bool`** type represents a boolean value, which may have one of two states: true or false.

6.5.14.2 The boolean type representation shall be equivalent to the **`uint`** type. Any non-zero value shall be interpreted as true, and zero shall be intepreted as false.

6.5.14.3 The default value of a boolean type is false.

The null type

6.5.14.4 The **`null`** type shall have the same representation as a pointer and can only store a specific, implementation-defined value (the *null* value).

6.5.14.5 There is no grammar for defining a value of type null, or a sub-type of null. It is for internal use only, as the type of the **`null`** constant.

The void type

6.5.14.6 The **`void`** type represents a non-existent value, and shall have no storage.

6.5.15 Struct and union types

struct-union-type:
    struct **{** *struct-union-fields* **}**
    union **{** *struct-union-fields* **}**

struct-union-fields:
    struct-union-field **,**$_{opt}$
    struct-union-field **,** *struct-union-fields*

struct-union-field:
    offset-specifier$_{opt}$ *name* **:** *type*
    offset-specifier$_{opt}$ *struct-union-type*
    offset-specifier$_{opt}$ *identifier*

offset-specifier:
    @offset **(** *expression* **)**

6.5.15.1 The *struct type* and *union type* are *aggregate types*, which *collect* multiple types, name them, and assign them *offset*s within their storage area. A struct type stores each value at a different offset; a union type stores all of its values at the the same offset. A type defined with the **struct** terminal is a struct type and uses the struct type class; if the **union** terminal is used the type is a union type with the union type class.

6.5.15.2 The struct-union-fields list denotes, in order, the subvalues which are collected by a struct or union, and potentially assigns a name to each.

6.5.15.3 For a struct type, the offset of each field is equal to the minimum *aligned* offset which would meet the alignment requirements of the field's type and which is greater than the offset of the previous field plus the size of the previous field. The implementation shall add *padding* to meet the alignment requirements of struct fields. For a union type, the offset of all members is zero.

6.5.15.4 The type of each struct or union field shall have a definite, nonzero size.

6.5.15.5 If given, the offset-specifier shall override the computed offset for a given field. If the user-defined offset for a field would not meet the alignment requirements for that type, the behavior is implementation-defined. The implementation shall either raise a diagnostic message and terminate the translation phase, or shall support unaligned memory accesses (perhaps at a cost to performance).

6.5.15.6 The expression given for the offset-specifier shall be limited to the «**??**: Translation compatible expression subset», and shall have an integer type and a positive or zero value.

6.5.15.7 The offset-specifier shall not be given for a **union** type.

6.5.15.8 A union type's size is the maximum size among its fields. A struct type's size is the maximum value of S for $S = O + Z$ among its fields, where O is that field's offset and Z is that field's size.

6.5.15.9 The default value of a struct or union type shall be defined as a value whose fields

assume the default values of their respective types. The default value of any fields which have overlapping storage (as defined by the bounds of their offset and their size) shall be undefined.

6.5.15.10 If the struct-union-type form of struct-union-field is given, the parent type shall collect the fields of the child type as its own. The offset of each field within the child type shall be the sum of the offset within the child type and the offset the child type occupies within the parent struct. The identifier form shall be interpreted in the same manner as a struct-union-type if it refers to a type alias of a struct or union type, otherwise a diagnostic message shall be printed and the translation phase shall abort.

Forward references: «6.5.21: Type aliases»

6.5.15.11 A struct or union type shall be equivalent to another struct or union type if their fields are of equivalent name, type, and offset, without respect to the order of their appearance in the program source.

The following types are equivalent:

```
struct { a:  int, b:  int }
struct { a:  int, struct { b:  int } }
```

6.5.15.12 Each field name (including names of embedded fields) shall be unique within the set of all field names of the struct-union-type.

6.5.16 Tuple types

tuple-type:
(*tuple-types* **)**

tuple-types:
type **,** *type* **,**$_{opt}$
type **,** *tuple-types*

6.5.16.1 A tuple type stores two or more values of arbitrary types in a specific order. It is similar to a struct type, but without names for each of its subvalues. Each value is stored at a given offset, possibly with padding added to meet alignment requirements.

6.5.16.2 The offset of each value is equal to the minimum *aligned* offset which would meet the alignment requirements of the value's type and which is greater than the offset of the previous value plus the size of the previous value type. The implementation shall add *padding* to meet the alignment requirements of tuple values.

6.5.16.3 The size of a tuple is the sum of the sizes of its value types plus any necessary padding. The alignment is the maximum alignment among its value types.

6.5.16.4 The type of each tuple value shall have a definite, nonzero size.

6.5.16.5 The default value of a tuple type shall be defined such that its values assume the default values of their respective types. If any of the subtypes do not have a default value, neither does the tuple type.

6.5.16.6 Two tuple types shall be equivalent to each other if they have the same value types in the same order.

6.5.17 Tagged union types

tagged-union-type:
(*tagged-types* **)**

tagged-types:
type **|** *type* **|**$_{opt}$
type **|** *tagged-types*

6.5.17.1 A tagged union stores a value of **one** of its constituent types, as well as a *tag* which indicates which of the constituent types is selected. The constituent types are defined by tagged-types.

6.5.17.2 The representation of a tagged union shall be a **`uint`**, in which the tag value is stored, followed by sufficient space to store any of the possible constituent types. Padding shall be inserted between the tag and the value area and after the value area if necessary to meet the maximum alignment among the tagged union members and the **`uint`** field.

6.5.17.3 The tag value shall be the type ID of the type which is selected from the constituent types. This value shall be stored at the **`uint`** field and shall indicate which type is stored in the value area.

It follows that the types `(A | B)` *and* `(B | A)` *are equivalent.*

6.5.17.4 The alignment of a tagged union type shall be the alignment of the **`uint`** type or the maximum alignment of the constituent types, whichever is greater.

6.5.17.5 The size of a tagged union type shall be the maximum size of its constituent types, plus the size of the **`uint`** type, plus any padding added per «6.5.17.2: Tagged union types».

6.5.17.6 If a member type among tagged-types is a tagged union type, it shall be reduced such that nested tagged union type is replaced with its constituent types in the parent union.

The types `(A | (B | (C | D)))` *and* `(A | B | C | D)` *are equivalent.*

6.5.17.7 The default value of a tagged union type is undefined.

6.5.17.8 A tagged union type shall be equivalent to another tagged union type if they share the same set of secondary types, without regard to order, and considering the secondary types of nested tagged unions as members of the set of their parent's secondary types.

6.5.18 Slice and array types

slice-array-type:
[ ] *type*
[*expression* **]** *type*
[∗] *type*
[_] *type*

6.5.18.1 An *array type* stores one or more items of a uniform secondary type. The number of items stored in an array type is an attribute of the array type and is specified during the translation phase. The secondary type shall have a definite, nonzero size.

6.5.18.2 The expression representation is used for array types of a determinate length, that is, with a determinate number of items. Such arrays are *bounded*. The expression must evaluate to a positive integer value, and shall be limited to the «**??**: Translation compatible expression subset».

6.5.18.3 An array type may be *unbounded*, in which case the number of items is not known. The ∗ representation indicates an array of this type.

6.5.18.4 An array may be bounded, but infer its size from context, using the _ representation. Such an array is said to be *context-defined*.

6.5.18.5 An array type may be *expandable*. This state is not represented in the type grammar, and is only used in specific situations. Array types are presumed to be non-expandable unless otherwise specified.

6.5.18.6 The representation of an *array type* shall be the items concatenated one after another, such that the offset of the *N*th item is determined by the equation $N \times S$, where *S* is the size of the secondary type.

6.5.18.7 A *slice type* stores a pointer to an unbounded array type, with a given *capacity*, and *length*, which respectively refer to the number of items that the unbounded array **may** store without re-allocation, and the number of items which are **currently valid**. The representation with no lexical elements between **[** and **]** indicates a slice type.

6.5.18.8 The representation of a slice type shall be equivalent to the following struct type:

```
struct {
        data: nullable *[*]type,
        length: size,
        capacity: size,
}
```

The type of the *data* field shall be a pointer to an unbounded array of the secondary type.

6.5.18.9 The alignment of an array type shall be equivalent to the alignment of the underlying type. The alignment of a slice type shall be equivalent to the alignment of the **size** type or «6.5.13: Pointer types», whichever is greater.

6.5.18.10 The size of a bounded array type shall be equal to $N \times S$, where N is the number of items and S is the size of the underlying type. The size of an unbounded array is undefined. The size of a slice type shall be equal to the size of the struct type defined by «6.5.18.6: Slice and array types».

6.5.18.11 The default value of an array type shall be equal to all of its members set to the default value of the underlying type. If the default value of the underlying type is undefined, the default value of the array type is undefined.

6.5.18.12 The default value of a slice type shall have the capacity and length fields set to zero and the data field set to null.

6.5.18.13 An array type shall be equivalent to another array type only if its length and secondary types are equivalent. A slice type shall only be equivalent to a slice type with the same secondary type.

6.5.19 String types

string-type:
`str`

6.5.19.1 A string stores a reference to a sequence of Unicode codepoints, encoded as UTF-8, along with its *length* and *capacity*. The length and capacity are measured in octets, rather than codepoints.

6.5.19.2 The representation of the string type shall be equivalent to the following struct type:

```
struct {
        data: *[*]const u8,
        length: size,
        capacity: size,
}
```

6.5.20 Function types

function-type:
    fntype-attr$_{opt}$ **fn** *prototype*

prototype:
    (*parameter-list*$_{opt}$ **)** *type*

fntype-attr:
    @noreturn

parameter-list:
    parameters **,**$_{opt}$
    parameters **...** **,**$_{opt}$
    parameters **,** **...** **,**$_{opt}$

parameters:
    parameter
    parameters **,** *parameter*

parameter:
    name **:** *type*
    _ **:** *type*

6.5.20.1 Function types represent a procedure which may be completed in the «5.4: Execution environment» to obtain a result and possibly cause side effects (see «5.4.6.1: Program execution»).

6.5.20.2 If the **@noreturn** form of fntype-attr is specified in the function-parameters, the type shall be **void** and the specified function shall not return to its caller.

6.5.20.3 The attributes of a function type are its *result type* and *input parameters*. A function type must have one result type (which may be **void**), and zero or more parameters. Each name shall be unique.

6.5.20.4 If the second form of parameters is used, the final parameter of the function type uses *Hare-style variadism*. If the third form is used, the function uses *C-style variadism*. The variadism of a function type affects the calling semantics for that function.

```
// Hare-style variadism:
fn(x: int, y: int, z: int...)

// C-style variadism:
fn(x: int, y: int, ...)
```

Forward references: «**??**: Call expressions»

6.5.20.5 The implementation is not required to support C-style variadism. If the implementation does not support C-style variadism, it must print a diagnostic message and

abort the translation environment for programs which attempt to utilize it.

6.5.20.6 The type of a parameter which uses Hare-style variadism shall be a slice of the specified type.

Therefore, in the case of `fn(x: int...)`*, the type of x shall be* `const []int`.

6.5.20.7 The size, alignment, default value, and storage semantics of function types is undefined. All function types shall be **`const`**, without regard to the use of **`const`** in the type description.

6.5.20.8 The function's result type, list of parameter types (in order), its variadism, and **`@noreturn`** status, are distinct characteristics of the function type, for the purpose of determining equivalency.

6.5.21 Type aliases

alias-type:
    identifier

unwrapped-alias:
    `...` *identifier*

6.5.21.1 A type alias assigns an identifier a unique type which is an alias for another type.

The grammar for an alias-type does not specify the underlying type. The underlying type is specified at the time it is declared, see «6.9: Declarations».

6.5.21.2 A type alias shall have the same storage, alignment, size, default value, and semantics as its underlying type.

6.5.21.3 Each type alias (uniquely identified by its identifier) shall be a unique type, even if it shares its underlying type with another type alias.

6.5.21.4 The **`...`** operator shall *unwrap* the type alias, and shall cause the statement to refer to the underlying type rather than the type alias itself.

This notably affects the relationship between type aliases and tagged unions. In the following example, union_a and union_b have different storage semantics, the former being a tagged union of two other tagged unions, and the latter being reduced to a single tagged union.

```
type signed = (i8 | i16 | i32 | i64 | int);
type unsigned = (u8 | u16 | u32 | u64 | uint);
type union_a = (signed | unsigned);
type union_b = (...signed | ...unsigned);
```

6.6 Expressions

6.6.1 An expression is a procedure which the implementation may perform to obtain a *result*, and possibly cause side-effects (see «5.4.6.1: Program execution»).

6.6.2 Expression types are organized into a number of classes and subclasses of expressions which define the contexts in which each expression type is applicable.

6.6.3 All expressions have a defined *result type*. It may be **`void`**.

6.6.4 Some expressions *terminate*. The semantics of terminating expressions vary between different expression types, and will be detailed as appropriate. If unspecified, expressions described by this expression are presumed to be non-terminating. Expression classes inherit their termination qualities from the more specific sub-expressions which they classify.

6.6.5 Some expressions may provide a *type hint* to other expressions which appear in their grammar, which those expressions may take advantage of to refine their behavior.

6.6.6 Constants

constant:
- *integer-constant*
- *floating-constant*
- *rune-constant*
- *string-constant*
- **`true`**
- **`false`**
- **`null`**
- **`void`**

6.6.6.1 Constants (also known as literals) shall describe a specific value of an unambiguous type. The result of the expression is the constant value.

6.6.6.2 The keywords **`true`** and **`false`** respectively represent the constants of the **`bool`** type.

6.6.6.3 The representation of **`true`** as an **`uint`**-equivalent (ref «6.5.14.2: Other primitive types») shall be one.

6.6.6.4 The **`null`** keyword represents the **`null`** value of the **`null`** type.

6.6.6.5 The **`void`** keyword represents the **`void`** value of the **`void`** type.

6.6.7 Floating constants

floating-constant: exactly:
decimal-digits **.** *decimal-digits* $exponent_{opt}$ $floating\text{-}suffix_{opt}$
decimal-digits $exponent_{opt}$ *floating-suffix*

floating-suffix: one of:
f32 f64

decimal-digits: exactly:
decimal-digit $decimal\text{-}digits_{opt}$

decimal-digit: one of:
0 1 2 3 4 5 6 7 8 9

exponent: exactly:
exponent-char $sign_{opt}$ *decimal-digits*

sign: one of:
+ -

exponent-char: one of:
e E

Floating constants represent an IEEE 754-compatible floating-point number in either the binary32 or binary64 format.

6.6.8 If the floating-suffix is not provided and a type hint is provided, the type shall be the type hint. If the floating-suffix is not provided and a type hint is not provided, the type shall be **f64**. Otherwise, the type shall refer to the type named by the suffix.

6.6.9 If the exponent is provided, the value of the constant shall be multiplied by 10 to the power of decimal-digits.

6.6.10 Integer constants

integer-constant: exactly:
`0x` *hex-digits* *integer-suffix*$_{opt}$
`0o` *octal-digits* *integer-suffix*$_{opt}$
`0b` *binary-digits* *integer-suffix*$_{opt}$
decimal-digits *exponent*$_{opt}$ *integer-suffix*$_{opt}$

hex-digits: exactly:
hex-digit *hex-digits*$_{opt}$

hex-digit: one of:
`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

octal-digits: exactly:
octal-digit *octal-digits*$_{opt}$

octal-digit: one of:
`0 1 2 3 4 5 6 7`

binary-digits: exactly:
binary-digit *binary-digits*$_{opt}$

binary-digit: one of:
`0 1`

integer-suffix: one of:
`i u z i8 i16 i32 i64 u8 u16 u32 u64`

Integer constants represent an integer value at a specific precision.

6.6.11 If the integer-suffix is not provided and a type hint is provided, the type shall be the type hint. If the integer-suffix is not provided and a type hint is not provided, the type shall be **`int`**. If the integer-suffix is provided, the type is specified by the suffix. Suffixes **`i`**, **`u`**, and **`z`** shall respectively refer to the **`int`**, **`uint`**, and **`size`** types; the remainder shall refer to the type named by the suffix.

6.6.12 If the number provided is not within the limits of the precision of the constant type, a diagnostic message shall be printed and the translation phase shall fail.

6.6.13 The prefixes **`0x`**, **`0o`**, and **`0b`** shall respectively cause the number to be interpreted with a hexadecimal, octal, or binary base. If no prefix is used, the number shall be interpreted with a decimal base.

6.6.14 If the exponent is provided, the value of the integer shall be multiplied by 10 to the power of decimal-digits.

6.6.15 Rune constants

rune-constant:
    ' *rune* **'**

rune:
    Any character other than **** or **'**
    escape-sequence

escape-sequence: exactly:
    named-escape
    \x *hex-digit* *hex-digit*
    \u *fourbyte*
    \U *eightbyte*

fourbyte: exactly:
    hex-digit *hex-digit* *hex-digit* *hex-digit*

eightbyte: exactly:
    fourbyte *fourbyte*

named-escape: one of:
    \0 **\a** **\b** **\f** **\n** **\r** **\t** **\v** **** **\'** **\"**

6.6.15.1 If a type hint is provided, the type shall be the type hint. Otherwise, the type shall be **rune**.

6.6.15.2 If the rune-constant is not an escape-sequence, the value of the rune shall be the Unicode codepoint representing rune.

6.6.15.3 A rune-constant beginning with **\x**, **\u**, or **\U** shall interpet its value as a Unicode codepoint specified in its hexadecimal representation by hex-digits.

6.6.15.4 A rune-constant containing a named-escape shall have a value based on the following chart:

Escape sequence	Unicode codepoint	Escape sequence	Unicode codepoint
\0	U+0000	**\a**	U+0007
\b	U+0008	**\f**	U+000C
\n	U+000A	**\r**	U+000D
\t	U+0009	**\v**	U+000B
\\	U+005C	**\'**	U+002C
\"	U+0022		

6.6.16 String constants

string-constant:
    " *string-chars* **"**
    string-constant *string-constant*

string-chars:
    string-char *string-chars*$_{opt}$

string-char:
    Any character other than **** or **"**
    escape-sequence

6.6.16.1 A string-constant expression shall have a result type of **`const str`**.

6.6.16.2 If the first form of string-constant is used, the string's *data* field shall refer to a UTF-8 encoded sequence of Unicode codepoints, ascertained by encoding the sequence of string-chars given in order, after interpreting escape codes per «6.6.15.17: Rune constants».

6.6.16.3 If the second form of string-constant is used, the string's *data* field shall refer to a UTF-8 encoded sequence of Unicode codepoints, ascertained by concatenating the *data* field of the first string-constant and the *data* field of the second string-constant.

6.6.16.4 The *length* and *capacity* fields shall be set to the length in octets of the encoded UTF-8 data.

6.6.17 Array literals

array-literal:
    [*array-members*$_{opt}$ **]**

array-members:
    expression **,**$_{opt}$
    expression **...** **,**$_{opt}$
    expression **,** *array-members*

Forward references: «**??**: Simple, complex, and compound expressions»

6.6.17.1 An array-literal expression produces a value of an array type. The type of each expression shall be uniform and shall determine the member type of the array value, and the length of the array type shall be defined by the number of members.

6.6.17.2 If a type hint has been provided to an array literal which is an array type (or a type alias which represents an array type), the member type will be inferred from this array type. The initializer expressions for each value among array-members shall receive this member type as a type hint. If the array-members is not specified, a type hint must be provided.

6.6.17.3 The execution environment shall evaluate the array-members, ordered such that

any side-effects of evaluating the arguments occur in the order that the members are listed, such that the *N*th member provides the value for the *N*th array member.

6.6.17.4 If the `...` form is used, the result's array type shall be expandable. If a type hint is available, it shall not be of a context-defined array type.

6.6.18 Enum literals

enum-literal: exactly:
identifier `::` *name*

6.6.18.1 An enum-literal expression produces a value of an enum type. The identifier shall be a type alias (see «6.5.21: Type aliases») which refers to an enum type. The result type shall be the type alias to which the identifier refers.

6.6.18.2 The enum type to which the type alias refers shall have an enum-value whose name is the enum-literal's name. The result of the enum-literal shall be the value assigned to this enum-value. If there is no such enum-value, a diagnostic message shall be printed and the translation phase shall abort.

6.6.19 Struct literals

struct-literal:
`struct` `{` *field-values* `,`$_{opt}$ `}`
identifier `{` *struct-initializer* `,`$_{opt}$ `}`

struct-initializer:
field-values
field-values `,` `...`
`...`

field-values:
field-value
field-values `,` *field-value*

field-value:
name `=` *expression*
name `:` *type* `=` *expression*
struct-literal

6.6.19.1 A struct-literal produces a value of a struct type. The first form is the *plain form*, and the second form is the *named form*.

6.6.19.2 If the plain form is given, the result type shall be a struct type defined by the field-values, in order, with their identifiers and types explicitly specified. The first form of field-value shall not be used in such a struct.

6.6.19.3 If the named form is given, the identifier shall identify a type alias (see «6.5.21:

Type aliases») which refers to a struct type. The result type shall be this alias type.

6.6.19.4 Each field-value shall specify a field by its name, and assign that field in the result value to the result of the expression given. The type of the named field, via the named type alias in the first form, or the given type in the second form, shall be provided to the initializer expression as a type hint. The field-values shall be evaluated in the order in which they appear in the struct-literal.

6.6.19.5 If `...` is not given, field-values shall be *exhaustive*, and include every field of the result type exactly once. Otherwise, a diagnostic message shall be printed and the translation phase shall abort.

6.6.19.6 If `...` is given, any fields of the result type which are not included in field-values shall be initialized to their default values. Each included field shall only be named once. If a field is omitted which does not have a default value, a diagnostic message shall be printed and the translation phase shall abort.

6.6.19.7 If the struct-literal form of the field-value is given, its fields shall be interpreted as fields of the parent struct.

The following values are equivalent:

```
struct { a:  int = 10, b:  int = 20 }
struct { a:  int = 10, struct { b:  int = 20 } }
```

6.6.20 Plain expressions

plain-expression:
- *identifier*
- *constant*
- *array-literal*
- *enum-literal*
- *struct-literal*

nested-expression:
- *plain-expression*
- **(** *expression* **)**
- **(** *tuple-items* **)**

tuple-items:
- *expression* **,** *expression* **,**$_{opt}$
- *expression* **,** *tuple-items*

Forward references: «**??**: Simple, complex, and compound expressions»

6.6.20.1 plain-expression is an expression class which represents its result value "plainly". In the case of constants and literals, the value is represented by the result of those expressions. In the case of an identifier, the expression produces the value of the identified object.

6.6.20.2 nested-expression is an expression class provided to allow the programmer to overcome undesirable associativity between operators.

6.6.20.3 The tuple-items form of nested-expression shall describe a *tuple expression*, and have a result type which is the tuple described by the types of its expressions in the order that they appear. If a type hint is available, the tuple items may be assigned to the types of the respective tuple sub-types in the order that they appear. The implementation shall cause the side-effects of each expression to occur in the order that they appear.

6.6.21 Allocations

allocation-expression:
alloc **(** *expression* **)**
alloc **(** *expression* **,** *expression* **)**
alloc **(** *slice-items* **...** **,** *expression* **)**
free **(** *expression* **)**

slice-items:
expression
slice-items **,** *expression*

6.6.21.1 An **alloc** expression allocates an object at runtime and initializes it to the first expression (the *initializer*). The *object type* of the allocation expression shall be based on its type hint, which may be absent; if so, it shall be a pointer type whose secondary type is the result type of the initializer.

6.6.21.2 In the case of a pointer type as the object type, the pointer's referent type shall have a defined, nonzero size. The execution environment shall provision storage sufficient to store a value of the referent type, and provide the pointer type's referent type as a type hint to the initializer. The new object's value shall be set to the initializer result.

6.6.21.3 In the case of a slice type as the object type, the execution environment shall choose a capacity which shall be greater than or equal to the number of items in the initializer, then provision an array of that length and set each Nth value to the Nth value of the initializer, for each value of N between 0 (inclusive) and the length of the initializer (exclusive). The result of the allocation-expression shall be a slice whose data field refers to this array, whose length is equal to the length of the initializer, and whose capacity is set to the allocated capacity.

6.6.21.4 If the second form of **alloc** is used, the object type shall be a slice type, and the second term shall be assignable to the **size** type. The second term shall specify the capacity, which shall be greater to or equal to the length of the initializer. The execution environment shall choose a capacity equal to or greater than this term, then provision an array of that length and set each Nth value to the Nth value of the initializer, for each value of N between 0 (inclusive) and the length of the initializer (exclusive). The result of the allocation-expression shall be a slice whose data field refers to this array, whose length is equal to the length of the initializer, and whose capacity is set to the selected capacity.

6.6.21.5 If the third form of **alloc** is used, the object type shall be a slice type, and the final term shall be assignable to the **size** type. The execution environment shall choose

a capacity equal to or greater than this term, then provision an array of that length and set each Nth value to the Nth value of slice-items (exclusive), for each value of N between 0 (inclusive) and the number of slice-items, then set each Nth value to the value of the last slice-items expression for each subsequent value of N up to the length given by the final term (exclusive). The result of the allocation-expression shall be a slice whose data field refers to the provisioned array, and whose length is equal to the final term, and whose capacity is set to the selected capacity.

6.6.21.6 In the **alloc** form, if the execution environment is unable to allocate sufficient storage for the requested type, the execution environment shall print a diagnostic message and abort. If the type is a nullable pointer type, **null** shall be returned instead of aborting.

6.6.21.7 The **free** form shall discard resources previously allocated with a **alloc** expression, freeing them for future use. The expression shall evaluate to a pointer type, in which case the object referred to by the pointer shall be freed, or a slice type, in which case the array referred to by its data field shall be freed.

6.6.22 Assertions

assertion-expression:

assert **(** *expression* **)**
assert **(** *expression* **,** *string-constant* **)**
static **assert** **(** *expression* **)**
static **assert** **(** *expression* **,** *string-constant* **)**
abort **(** *string-constant*$_{opt}$ **)**
static **abort** **(** *string-constant*$_{opt}$ **)**

Forward references: «**??**: Simple, complex, and compound expressions»

6.6.22.1 An assertion-expression is used to validate an assumption by the programmer by *asserting* its truth. The result type of a assertion-expression is **void**.

6.6.22.2 expression shall be an expression of type **bool**, which shall be provided to it as a type hint.

6.6.22.3 In the first two forms, this expression shall be evaluated in the execution environment, and if false, a diagnostic message shall be printed and the execution phase aborted. The programmer may provide the string-constant to be included in the diagnostic message.

6.6.22.4 In the **static** form, expression shall be limited to the «6.8: Translation compatible expression subset», shall be evaluated in the translation environment, and is otherwise equivalent to the other forms.

6.6.22.5 The **abort** form shall cause the execution environment to print a diagnostic message and abort. The programmer may provide the string-constant to be included in the diagnostic message. An abort is a terminating expression.

6.6.23 Calls

call-expression:
    postfix-expression **(** *argument-list*$_{opt}$ **)**

argument-list:
    expression **,**$_{opt}$
    expression **...** **,**$_{opt}$
    expression **,** *argument-list*

Forward references: «**??**: Simple, complex, and compound expressions»

6.6.23.1 A call-expression shall invoke a function in the execution environment and its result shall be a value of the type specified by the postfix-expression's function result type. This evaluation shall include any necessary side-effects per «5.4.6.1: Program execution».

6.6.23.2 The result type of the postfix-expression shall be restricted to a set which includes all function types which do not have the **`@init`**, **`@fini`**, or **`@test`** attributes set, as well as non-nullable pointers whose secondary type is included in the set.

The result type of the postfix-expression can be a function, a pointer to a function, a pointer to a pointer to a function, and so on.

6.6.23.3 The function invoked shall be the function object the postfix-expression refers to, selecting that object indirectly via any number of pointer types if appropriate.

6.6.23.4 The argument-list shall be a list of expressions whose types shall be assignable to the types of the invoked function's parameters, in the order that they are declared in the invoked function's result type. The types specified in the function's prototype shall be provided as type hints to each argument expression as appropriate.

6.6.23.5 The execution environment shall evaluate the argument-list, ordered such that any side-effects of evaluating the arguments occur in the order that the arguments are listed, to obtain the parameter values required to invoke the function.

6.6.23.6 If the invoked function uses Hare-style variadism, the argument-list shall provide zero or more arguments following the last non-variadic parameter, all of which must be assignable to the type of the variadic parameter.

6.6.23.7 If the final argument uses the **`...`** form, it must occupy the position of a variadic parameter and be of a slice or array type. The implementation shall interpret this value as the list of variadic parameters.

6.6.23.8 If the invoked function uses C-style variadism, the function may provide zero or more arguments following the final parameter. These arguments shall be of a type with a non-zero size, but are otherwise unconstrained.

6.6.23.9 The specific means by which the invoked function assumes control of the execution environment, and by which the arguments are provided to it, is implementation-defined.

This is generally provided by the target's ABI specification.

6.6.23.10 If the invoked function's result type has the **`@noreturn`** attribute, the call expression is considered to terminate.

6.6.24 Measurements

measurement-expression:
    size-expression
    length-expression
    offset-expression

size-expression:
    `size` **`(`** *type* **`)`**

length-expression:
    `len` **`(`** *expression* **`)`**

offset-expression:
    `offset` **`(`** *field-access-expression* **`)`**

Forward references: «6.6.25: Field access»

6.6.24.1 A measurement-expression is used to measure objects. The result type shall be **`size`**.

6.6.24.2 The **`size`** expression shall compute the *size* of the specified type.

6.6.24.3 The **`len`** expression shall compute the *length* of a bounded array, or the `length` field of a slice object, referred to by expression. If an unbounded array object is given, the translation environment shall print a diagnostic message and abort.

6.6.24.4 The object used for a length expression selected shall be the array or slice object the expression refers to, selecting that object indirectly via any number of pointer types if appropriate.

6.6.24.5 The **`offset`** expression shall determine the struct or tuple field which would be accessed by field-access-expression and compute its *offset*.

6.6.25 Field access

field-access-expression:
    postfix-expression **`.`** *name*
    postfix-expression **`.`** *integer-constant*

6.6.25.1 A field-access-expression is used to access fields of «6.5.15: Struct and union types» and «6.5.16: Tuple types». The result type of the postfix-expression shall be constrained to a set which includes all struct, union, and tuple types, as well as non-nullable pointers whose secondary type is included in the set.

The result type of the postfix-expression can be a struct or union or tuple, a pointer to a struct or union or tuple, a pointer to a pointer to a struct or union or tuple,

and so on.

6.6.25.2 The object from which the field is selected shall be the struct or union object the postfix-expression refers to, selecting that object indirectly via any number of pointer types if appropriate.

6.6.25.3 If the postfix-expression's result type is a union type, the first form shall be used. The result of the field-access-expression shall be the union's storage area interpreted as the type of the field named by name, and the result type of the expression shall be the type of the named field.

6.6.25.4 If the postfix-expression's result type is a struct type, the first form shall be used. The result of the field-access-expression shall be the value stored in the name field of the result of the postfix-expression, and the result type of the expression shall be the type of the named field.

6.6.25.5 If the postfix-expression's result type is a tuple type, the second form shall be used. The result of the field-access-expression shall be the Nth value stored in the tuple which is the result of the postfix-expression, and the result type of the expression shall be the type of the Nth value.

6.6.25.6 If the type of the struct object in the first term has the **`const`** flag, the result type shall also have the **`const`** flag set, regardless of the flag's value on the type of the named field.

6.6.26 Indexing

indexing-expression:
postfix-expression **`[`** *expression* **`]`**

6.6.26.1 An indexing-expression shall access a specific value of a slice or array type. The expression shall have an integer result type. The result type of the postfix-expression shall be constrained to a set which includes all slice and array types, as well as non-nullable pointers whose secondary type is included in the set.

The result type of the postfix-expression can be a slice or array, a pointer to a slice or array, a pointer to a pointer to a slice or array, and so on.

6.6.26.2 The object from which the field is selected shall be the slice or array object the postfix-expression refers to, selecting that object indirectly via any number of pointer types if appropriate.

6.6.26.3 The result type of an indexing-expression is the secondary type of the slice or array type given by the postfix-expression result type.

6.6.26.4 If the type of the array or slice object in the first term has the **`const`** flag, the result type shall also have the **`const`** flag set, regardless of the flag's value on the secondary type.

6.6.26.5 The execution environment shall compute the result of expression to obtain N for selecting the *N* per the algorithm given in «6.5.18: Slice and array types».

6.6.26.6 The execution environment shall perform a *bounds test* on the value of N to ensure

it falls within the acceptable range for the given slice or array type. It shall test that $N < Z$, where Z is the length of the bounded array type, or the `length` field of the slice, whichever is appropriate. For unbounded array types, the bounds test shall not occur. If the bounds test fails, a diagnostic message shall be printed and the execution environment shall abort.

The implementation may perform a bounds test in the translation environment if is able, print a diagnostic message, and abort the translation environment if it fails.

6.6.27 Slicing

slicing-expression:
postfix-expression **[** *expression*$_{opt}$ **..** *expression*$_{opt}$ **]**

6.6.27.1 A slicing-expression shall have a result type of slice, which is computed a subset of a slice or array object. The optional expressions shall have an integer result type. The result type of the postfix-expression shall be constrained to a set which includes all slice and array types, as well as non-nullable pointers whose secondary type is included in the set.

The result type of the postfix-expression can be a slice or array, a pointer to a slice or array, a pointer to a pointer to a slice or array, and so on.

6.6.27.2 The object from which the field is selected shall be the slice or array object the postfix-expression refers to, selecting that object indirectly via any number of pointer types if appropriate.

6.6.27.3 The first expression shall compute value L, and the second shall compute H. If absent, $L = 0$ and $H =$ `length`, where `length` shall be equal to the length of a bounded array type or the length of a slice type, represented in either case by the result of postfix-expression. If H is not specified, and postfix-expression is of an unbounded array type, the translation environment shall abort.

6.6.27.4 The resulting slice value shall have its `data` field set from, in the case of an array type, the address of the array; or in the case of a slice type, the `data` value of the source object; plus $L \times S$, where S is the size of the slice or array's secondary type.

6.6.27.5 The resulting slice value shall have its `length` field set to $H - L$, and its capacity field set to the `length` of the source object minus L. If the length of the object is undefined, the capacity shall be set to $H - L$ instead.

6.6.27.6 The secondary type of the resulting slice type shall be equivalent to the secondary type of the slice or array type given by postfix-expression. The resulting slice type shall inherit the const attribute from this type.

6.6.28 Slice mutation

slice-mutation-expression:
static$_{opt}$ **append** **(** *object-selector* **,** *append-values* **)**
static$_{opt}$ **delete** **(** *indexing-expression* **)**
static$_{opt}$ **delete** **(** *object-selector* **)**
static$_{opt}$ **insert** **(** *indexing-expression* **,** *append-values* **)**

append-values:
expression **,**$_{opt}$
expression **...** **,**$_{opt}$
expression **,** *append-values*

6.6.28.1 The **append** form shall append a new value, or values, to a slice object specified by the first argument. The expression's result type shall be a mutable slice type. This slice type's member type shall be the **append**'s *member type*. The result of an **append** shall be **void**.

6.6.28.2 The slice object shall be selected indirectly via any number of pointer types if appropriate.

6.6.28.3 The expression of the first and third cases of append-values shall have result types assignable to the **append**'s member type. The expression of the second case of append-values shall have a result type assignable to the result type of the **append**'s expression.

6.6.28.4 The **append** shall increase the slice object's capacity, and potentially re-allocate its data field, in order to ensure that the capacity is large enough to store the current contents of the slice as well as the values appended by the **append**.

6.6.28.5 The results of each expression in the append-values shall be appended to the slice object in the order in which they appear. If the second case of the append-values is specified, each of its members shall be appended to the slice object in order.

6.6.28.6 The side-effects of the **append**'s expression shall occur before the side-effects of the append-values. The side-effects of the append-values' expressions shall occur in the order in which they appear, except in the case of a value using **...**, which will be evaluated before other values.

6.6.28.7 The **delete** form shall delete a section of a slice object specified by the slicing-expression or indexing-expression's postfix-expression. The postfix-expression shall have a slice result type.

6.6.28.8 The **delete** form shall remove the members selected by the slicing-expression or indexing-expression from the data field of the slice object. If there is at least one member which follows the selected members, they shall be moved such that the first member following the selected members is moved to the index of the first selected member and each member following it maintains its position relative to the first selected member. The delete may decrease the slice object's capacity, and potentially re-allocate its data field, so long as the new capacity is large enough to store the new contents of the slice.

6.6.28.9 The **insert** form shall ensure that the slice has sufficient capacity to store the items from append-values by reallocating the slice if necessary, then moving the items from the index specified by indexing-expression (including the indexed item itself) forward by the number of places required to insert the append-values, then inserting the appended values into the new space. The side-effects of the indexing-expression shall occur first, followed by the side effects of each appended value the order that they appear.

6.6.28.10 The **static** forms of each of these expressions shall be equivalent to the non-static forms except that they shall never cause the underlying slice to be resized. In the static form, if an operation would require more space than the current capacity of the slice provides, the implementation shall print a diagnostic message and abort the execution environment.

6.6.29 Error propagation

error-propagation:
    postfix-expression **?**
    postfix-expression **!**

6.6.29.1 **?** and **!** are the error propagation operators. The postfix-expression shall have a result type which is either a type with the error flag set, or a tagged union type which has a type with the error flag set among its member types.

6.6.29.2 The result type of an error propagation expression shall be the same as the type of the postfix-expression, without its error cases, or **void** if no non-error types remain. The result is a tagged union whose member types are the subset of the original type which do not include the error flag; or, if there is only one such type, that type without a tagged union; or, if there are no such types, **void**. If the original type is not a tagged union type, the translation phase shall abort with a diagnostic message.

6.6.29.3 An error propagation expression shall perform an *error test* which checks if the result value of the postfix-expression is of a non-error type. If so, that value shall be the result of the error propagation expression.

6.6.29.4 In the **?** form, if the error test fails (i.e. an error is found), the error type shall be assignable to the current function's return type, and that value shall be returned from the function, causing the function to terminate.

6.6.29.5 In the **!** form, if the error test fails (i.e. an error is found), the execution environment shall print a diagnostic message and abort.

6.6.29.6 If a type hint is provided to the error propagation expression, the same type shall be provided to the postfix-expression.

6.6.30 Postfix expressions

postfix-expression:
    nested-expression
    call-expression
    field-access-expression
    indexing-expression
    slicing-expression
    error-propagation

object-selector:
    identifier
    indexing-expression
    field-access-expression

6.6.30.1 postfix-expression is an expression class for expressions whose operators use postfix notation.

6.6.30.2 object-selector defines a subset of postfix expressions which refer to objects, for use in other parts of the grammar.

6.6.31 Builtin expressions

builtin-expression:
    allocation-expression
    assertion-expression
    deferred-expression
    measurement-expression
    slice-mutation-expression
    postfix-expression

6.6.32 Unary arithmetic

unary-expression:
    builtin-expression
    unary-operator *unary-expression*

unary-operator: one of:
    + - ~ ! * &

6.6.32.1 A unary expression applies a unary-operator to a single value.

6.6.32.2 The **+** and **-** operators shall respectively perform unary positive and unary negation operations. The result type shall be equivalent to the type of unary-expression, which shall be of a signed numeric type.

6.6.32.3 The **~** operator shall perform a binary NOT operation, inverting each bit of the value. Its result type shall be equivalent to the type of unary-expression, which

shall be of an unsigned integer type.

6.6.32.4 The `!` operator shall perform a logical NOT operation. The result type, and the type of unary-expression, shall both be **`bool`**. If the unary-expression is **`true`**, the result shall be **`false`**, and vice-versa.

6.6.32.5 The `*` operator shall dereference a pointer, and return the object it references. The type of unary-expression shall be a pointer type, and the result type shall be the pointer's secondary type. The pointer type shall not be *nullable*.

6.6.32.6 The **`&`** operator shall take the address of an object. The result type shall be a pointer whose secondary type is the type of the object selected by the unary-expression. If the unary-expression is not an object-selector, the ensuing pointer shall only be valid within the current function.

The following table is informative.

Operator	Meaning
`+`	Positive
`-`	Negation
`~`	Binary NOT
`!`	Logical NOT
`*`	Dereference pointer
`&`	Take address

6.6.33 Casts and type assertions

cast-expression:
unary-expression
cast-expression **`:`** *type*
cast-expression **`as`** *type*
cast-expression **`is`** *type*

6.6.33.1 A cast expression interrogates or converts the type of an object. The first form illustrates the precedence. The second and third forms (**`:`** and **`as`**) have a result type specified by the type; and the fourth form (**`is`**) has a result type of **`bool`**.

6.6.33.2 Each form shall provide the specified type as a type hint to its cast-expression.

6.6.33.3 The second form is a *type cast*, and shall not fail. It shall cause the execution environment to convert or interpret the value as another type.

6.6.33.4 A type which may be cast to another type is considered *castable* to that type.

6.6.33.5 All types are castable to themselves. The set of other types which are castable to a given type are given by the following table:

Source type	Casts to
Floating types	Any numeric type
Integer types	Any numeric type or enum type
uintptr	Any pointer type or **null**
Any pointer type	Any pointer type, **uintptr**, or **null**
Enum types	Any enum type or numeric type
Array and slice types	Any array or slice type
Slice types	Any pointer to an array type
Type alias	Any type the underlying type could cast to
Any type	Any type alias with an underlying type it may be cast to
u8	**char**
char	**u8**
rune	**u32**
u32	**rune**
null	Any pointer type or **uintptr**
Tagged unions	See below

6.6.33.6 Tagged union types are mutually castable with any type which is found among its members, including the otherwise non-castable types of **bool** and **void**, as well as any other tagged union type.

6.6.33.7 When an integer type is cast to an integer of less precision, it shall be truncated towards the least significant bit.

6.6.33.8 When a signed type is cast to a unsigned type, the sign bit shall be copied to the most significant bit of the unsigned type, and vice versa.

6.6.33.9 When a floating type is cast to an integer, the resulting value shall be equal to $I = \lfloor F \rfloor$, where F is the source value. If the destination type has insufficient precision to represent I, it shall be truncated towards the least significant bit.

6.6.33.10 When casting an integer to a floating point type, if the destination type lacks the precision to represent the value, it shall be truncated towards zero.

6.6.33.11 Casting a pointer type to uintptr, and then back to the pointer type, shall yield the same pointer.

However, casting uintptr to any other integer type and back again may truncate towards the least significant bit and is not guaranteed to yield the same pointer.

6.6.33.12 The **const** flag shall not affect the rules for casting one type to another. The same holds for the error flag as well.

6.6.33.13 The third form is a *type assertion*. In this form, cast-expression shall be of a tagged union type, and type shall be a constituent type of that tagged union. The cast-expression shall be computed, and if the tag does not match type, a diagnostic message shall be printed and the execution environment shall abort. Otherwise, the result type is type.

6.6.33.14 The fourth form is a *type test*. In this form, cast-expression shall be of a tagged union type, and type shall be a constituent type of that tagged union. The result type is **bool**, and shall be **true** if the selected tag of cast-expression is type, or **false** otherwise.

6.6.34 Multiplicative arithmetic

multiplicative-expression:
cast-expression
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

6.6.34.1 A multiplicative-expression multiplies (`*`), divides (`/`), or obtains the remainder between (`%`) two expressions. The first form illustrates the precedence. The operands and result type shall be subject to the «6.7: Type promotion» rules.

6.6.34.2 In the case of division or modulus, the first term is the dividend, and the second term is the divisor. The result of the modulus shall have the same sign as the dividend.

6.6.34.3 A modulus (`%`) operation shall be performed with operands of integer types only.

6.6.34.4 If an operation would cause the result to overflow the result type, it is truncated towards the least significant bits in the case of integer types, and towards zero in the case of float types. Integer divisions which would produce a fractional part are rounded towards zero.

6.6.34.5 The implementation shall ensure that any side-effects of the first term shall occur before side-effects of the second term.

6.6.35 Additive arithmetic

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

6.6.35.1 An additive-expression adds (`+`) two operands, or subtracts (`-`) one from another. The first form illustrates the precedence. The operands and result type shall be subject to the «6.7: Type promotion» rules.

6.6.35.2 In the case of subtraction, the first term is the minuend, and the second term is the subtrahend.

6.6.35.3 If an operation would cause the result to overflow or underflow the result type, it is truncated towards the least significant bits in the case of integer types, and towards zero in the case of float types. In the case of signed types, this truncation will cause the sign bit to change.

6.6.35.4 The implementation shall ensure that any side-effects of the first term shall occur before side-effects of the second term.

6.6.36 Bit shifting arithmetic

shift-expression:
additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

6.6.36.1 A shift-expression performs a bitwise left-shift (**<<**) or right-shift (**>>**). The first form illustrates the precedence. The result type shall be the type of the first operand. Both operands shall be of unsigned integer types.

6.6.36.2 `shift-expression` **<<** N shall shift each bit towards the most significant bit N places, and set the least significant N bits to zero. The N most significant bits shall be silently discarded. If N is greater than the size in bits of the type, the result shall be zero.

This operation is equivalent to multiplying shift-expression by 2^N

6.6.36.3 `shift-expression` **>>** N shall shift each bit towards the least significant bit N places. The most significant bits shall be set to either zero or one depending on the signedness of shift-expression: If it is signed, then the N most significant bits shall be set to the value of the sign bit. If it unsigned, then the N most significant bits shall be set to zero. The N least significant bits shall be silently discarded. If N is greater than the size in bits of the type, the result shall be zero.

This operation is equivalent to dividing shift-expression by 2^N

6.6.36.4 The implementation shall ensure that any side-effects of the first term shall occur before side-effects of the second term.

6.6.37 Bitwise arithmetic

and-expression:
shift-expression
and-expression **&** *shift-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression **^** *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression **|** *exclusive-or-expression*

6.6.37.1 The implementation shall ensure that any side-effects of the first term shall occur before side-effects of the second term.

6.6.38 Logical comparisons

comparison-expression:
    inclusive-or-expression
    comparison-expression **<** *inclusive-or-expression*
    comparison-expression **>** *inclusive-or-expression*
    comparison-expression **<=** *inclusive-or-expression*
    comparison-expression **>=** *inclusive-or-expression*

equality-expression:
    comparison-expression
    equality-expression **==** *comparison-expression*
    equality-expression **!=** *comparison-expression*

6.6.38.1 A comparison-expression determines which operand is lesser than (**<**), greater than (**>**), less than or equal to (**<=**), or greater than or equal to (**>=**) the other. The operands shall be numeric, and are subject to the «6.7: Type promotion» rules. The result type shall be **bool**.

6.6.38.2 The result of the **<** operator shall be **true** if the first operand is mathematically less than the second operand and **false** otherwise.

6.6.38.3 The result of the **>** operator shall be **true** if the first operand is mathematically greater than the second operand and **false** otherwise.

6.6.38.4 The result of the **<=** operator shall be **true** if the first operand is mathematically less than or equal to second operand and **false** otherwise.

6.6.38.5 The result of the **>=** operator shall be **true** if the first operand is mathematically greater than or equal to second operand and **false** otherwise.

6.6.38.6 An equality-expression determines if two operands are equal to one another. The result type is **bool**. If the types of the **==** or **!=** operators are numeric, they shall be subject to «**??**: Type promotion». Otherwise, each operand must be of the same type, and that type must both be either **str**, **bool**, **rune**, or a pointer type.

6.6.38.7 The result of the **==** operator shall be **true** if the first operand is equal to second operand in value, and **false** otherwise.

6.6.38.8 The result of the **!=** operator shall be **true** if the first operand is not equal to second operand in value, and **false** otherwise.

6.6.38.9 If the type is **str**, an equality-expression shall be **true** if both strings have the same length and octets, and **false** otherwise.

6.6.38.10 The implementation shall ensure that any side-effects of the first term shall occur before side-effects of the second term.

6.6.39 Logical arithmetic

logical-and-expression:
    equality-expression
    logical-and-expression **&&** *equality-expression*

logical-xor-expression:
    logical-and-expression
    logical-xor-expression **^^** *logical-and-expression*

logical-or-expression:
    logical-xor-expression
    logical-or-expression **||** *logical-xor-expression*

6.6.39.1 For all cases of logical arithmetic, both terms shall be of the **bool** type, and the result type shall be **bool**.

6.6.39.2 **&&** shall compute a logical and operation, and shall be **true** if both terms are **true**, and false otherwise.

6.6.39.3 **^^** shall be a logical exclusive or operation, and shall be **true** if the terms are not equal to each other, and **false** otherwise.

6.6.39.4 **||** shall be a logical or operation, and shall be **true** if either term is **true**, and **false** otherwise.

6.6.39.5 If the first term of logical-and-expression is **false**, or the first term of logical-or-expression is **true**, the implementation shall ensure that the side-effects of the second term do not occur.

6.6.40 If expressions

if-expression:
    if *conditional-branch*
    if *conditional-branch* **else** *if-expression*
    if *conditional-branch* **else** *expression*

conditional-branch:
    (*expression* **)** *expression*

6.6.40.1 An if-expression chooses which, if any, expression to evaluate based on a logical criteria. In all forms, the result type of expression shall be **bool**. If a type hint is provided, the expression shall receive it as a type hint.

6.6.40.2 When executing a conditional-branch, the implementation shall evaluate the expression (the *condition*), and if **true**, the implementation shall execute the corresponding expression (the *branch*), ensuring that all side-effects occur. If the condition is **false**, the branch shall not be executed and shall not cause side-effects.

6.6.40.3 In the second form, the first conditional-branch shall be executed. If it was **false**,

the second-order if-expression shall be executed; otherwise not. In the latter case, any side-effects of the second condition or branch shall not occur.

6.6.40.4 In the third form, the conditional-branch shall be executed. If it was **`false`**, the expression shall be executed, and all of its side-effects shall occur.

6.6.40.5 In the first form, the result type is **`void`**. In the second form, the result type is **`void`** unless the third form is present in the N^{th}-order sub-expression for any value of N. In this case, and in the case of the third form used in the first order, if the result types of each branch are uniform, the result type of the expression as a whole shall be of that type. If the types are not uniform, the result type shall be a tagged union of the set of possible result types. The result value shall be selected from the result of the branch which is executed.

6.6.40.6 If a type hint is provided and all non-terminating branches are assignable to that type, the result type shall be the type given by the type hint. If the result types of all non-terminating cases are equivalent, the result type of the if expression shall be the result type of the cases. Otherwise, the result type shall be a tagged union of the set of the result types of the non-terminating cases.

If all cases terminate, the result type shall be **`void`** and the if expression shall terminate.

6.6.41 For loops

for-loop:
`for` **`(`** *for-predicate* **`)`** *expression*

for-predicate:
expression
binding-list **`;`** *expression*
expression **`;`** *expression*
binding-list **`;`** *expression* **`;`** *expression*

6.6.41.1 A for-loop executes its expression, the *body* of the loop, zero or more times, so long as a condition is true. Its result type is **`void`**.

6.6.41.2 In the first form, for-predicate specifies the *condition* with its expression. In the second form, the binding-list is the *binding* and the expression is the condition. In the second form, the first expression is the condition, and the second expression is the *afterthought*. In the third form, the binding-list is the binding, and the two expressions are respectively the condition and afterthought. The result type of the condition shall be **`bool`**, and this shall be provided as a type hint.

6.6.41.3 The implementation shall establish a new scope for the expression, then, if present, it shall evaluate the binding in this scope. The implementation shall then evaluate the condition. If it is **`true`**, the expression shall be evaluated and all of its side-effects shall occur; this process is an *iteration*. When the iteration is complete, the implementation shall evaluate the afterthought, if present, and then repeat the process, until the condition evaluates to **`false`**.

6.6.42 Switch expressions

switch-expression:
    switch **(** *expression* **)** **{** *switch-cases* **}**

switch-cases:
    switch-case **;**$_{opt}$
    switch-case **;** *switch-cases*

switch-case:
    case *case-options* **=>** *expression-list*
    case **=>** *expression-list*

case-options:
    expression **,**$_{opt}$
    expression **,** *case-options*

Forward references: «6.6.47: Compound expressions»

6.6.42.1 A switch statement evaluates a value (expression, the *switching expression*), then compares it with a number of switch-cases, taking whichever branch matches the value.

6.6.42.2 Each of the case-options specifies a value to compare with, given by expression. This expression shall be limited to the «6.8: Translation compatible expression subset», and its result type shall be equivalent to the result type of the switching expression.

6.6.42.3 Each switch-case introduces an implicit compound-expression which the provided expression-list gives the expressions of. The implementation shall evaluate the expression-list of the corresponding switch-case if any of the case-options is equal to the switching expression's result, setting the result of the overall switch expression to the result of the selected switch-case.

As such, the appropriate way to set the result of a switch statement is with a yield-expression. The semantics of deferred-expression, bindings, and so on, are also implicated.

6.6.42.4 The form of switch-case without case-options indicates any case which is not selected by the other cases. Only one case of this form shall appear in the switch expression.

6.6.42.5 The switch cases shall be *exhaustive*, meaning that every possible value of the switching expression is accounted for by a switch-case. It shall also be preicsely exhaustive: no two cases shall select for the same value.

6.6.42.6 The implementation shall ensure that side-effects of the switch value expression occur before those of the selected case, and that side-effects of non-selected cases do not occur.

6.6.42.7 If a type hint is provided, each branch shall receive it as a type hint.

6.6.42.8 If a type hint is provided and all non-terminating branches are assignable to that

type, the result type shall be the type given by the type hint. If the result types of all non-terminating cases are equivalent, the result type of the switch expression shall be the result type of the cases. Otherwise, the result type shall be a tagged union of the set of the result types of the non-terminating cases.

If all cases terminate, the result type shall be **`void`** and the switch expression shall terminate.

6.6.43 Match expressions

match-expression:
`match` **`(`** *expression* **`)`** **`{`** *match-cases* **`}`**

match-cases:
match-case **`,`**$_{opt}$
match-case **`,`** *match-cases*

match-case:
`case` *name* **`:`** *type* **`=>`** *expression-list*
`case` *type* **`=>`** *expression-list*
`case` **`=>`** *expression-list*

Forward references: «6.6.47: Compound expressions»

6.6.43.1 A match statement evaluates a value (expression, the *matching expression*), then selects and evaluates another expression based on its result type. The result type of the matching expression must be a tagged union or nullable pointer type, or an alias of either.

6.6.43.2 If the matching expression has a tagged union type, each match-case shall specify a type which is either a member of that tagged union, or another tagged union which supports a subset of the matching expression's type, or a type alias which refers to a qualifying type.

6.6.43.3 If the matching expression has a nullable pointer type, one match case shall be **`null`**, and another shall be the equivalent non-nullable pointer type, or a type alias which refers to a qualifying type.

6.6.43.4 Each match-case introduces an implicit compound-expression which the provided expression-list gives the expressions of. The implementation shall evaluate the expression-list of the corresponding match-case if the value of the matching expression is of the type specified by this match case, or can be assigned from it, setting the result of the overall match statement to the result of the selected match-case.

As such, the appropriate way to set the result of a match statement is with a yield-expression. The semantics of deferred-expression, bindings, and so on, are also implicated.

6.6.43.5 The form of match-case without a type indicates any case which is not selected by the other cases. Only one case of this form shall appear in the match expression.

6.6.43.6 The first form of match-case, if selected, shall cause the implementation to cast the match expression to the selected type and assign the resulting value to name. It shall insert this binding into the scope of the implicit compound-expression of the selected case.

6.6.43.7 The match cases shall be *exhaustive*, meaning that every possible type of the matching expression is accounted for by a match-case. It shall also be preicsely exhaustive: no two cases shall select for the same type.

6.6.43.8 The implementation shall ensure that side-effects of the match value expression occur before those of the selected case, and that side-effects of non-selected cases do not occur.

6.6.43.9 If a type hint is provided, each branch shall receive it as a type hint.

6.6.43.10 If a type hint is provided and all non-terminating branches are assignable to that type, the result type shall be the type given by the type hint. If the result types of all non-terminating cases are equivalent, the result type of the match expression shall be the result type of the cases. Otherwise, the result type shall be a tagged union of the set of the result types of the non-terminating cases.

If all cases terminate, the result type shall be **`void`** and the match expression shall terminate.

6.6.44 Assignment

assignment:
object-selector *assignment-op* *expression*
`*` *unary-expression* *assignment-op* *expression*
slicing-expression `=` *expression*

assignment-op: one of:
`= += -= *= /= %= <<= >>= &= |= ^= &&= ||= ^^=`

6.6.44.1 An assignment expression shall cause the object given by the first term to be assigned a new value based on the value given by the second term. The type of the object shall be provided as a type hint to the secondary expression(s).

6.6.44.2 If the assignment-op is `=`, the first term shall be assigned the value given by the second term. Otherwise, the assignment `e1 op= e1` shall be equivalent to the assignment `e1 = e1 op e2`, but the side effects of `e1` shall only occur once.

6.6.44.3 In the first form, the object-selector selects the object to be modified. The type of this object shall not be of a **`const`** type.

6.6.44.4 In the second form, the unary-expression shall have a result type of a non-nullable, non-const pointer type, and the object which is assigned shall be the secondary object to which the pointer object refers. The second term shall be assignable to the pointer's secondary type.

6.6.44.5 In the third form, the expression shall be of a slice type, and shall have a length equal to the slice given by slicing-expression. The first term shall not be of a **`const`** type, and the lengths of the two slices shall be equal. The contents of the slice

given by the second term shall be copied into the slice given by the first term.

6.6.44.6 The second term shall be *assignable* to the object. Assignability rules are stricter than castability rules. All types are assignable to themselves. The set of other types which are assignable to a given type are given by the following table:

Object type	May be assigned from
Mutable type	Constant types assignable to the object type
Signed integer types	Signed integer types of equal or higher precision
Unsigned integer types	Unsigned integer types of equal or higher precision
Floating-point types	Any floating-point type of equal or higher precision
Nullable pointer types	Non-nullable pointer type of the same secondary type
Nullable pointer types	`null`
Slice types	Array type of the same secondary type and definite length
Slice types	`[]void`
Array types of undefined size	Array types of defined size
Tagged union types	See notes
Type aliases	Any type assignable to the secondary type
`void`	Any type
`* void`	Any non-nullable pointer type
`nullable * void`	Any pointer type
`* const char`	`str`

The implementation shall perform any necessary conversion from the source type to the destination type.

6.6.44.7 Pointers to array types are mutually assignable if their secondary types are mutually assignable.

6.6.44.8 A pointer to a type is assignable to a pointer to a secondary type if the primary type is a struct type which contains the secondary type at offset zero, or if the type at offset zero is a type which would be assignable under these rules.

6.6.44.9 Tagged union types may be assigned from any of their constituent types. Tagged unions may also be assigned from any type which is assignable to exactly one of its constituent types. Additionally, tagged unions may be assigned from any other tagged union type, provided that the set of constituent types of the destination type is a superset of the set of constituent types of the source type.

6.6.44.10 `const` types have the same assignability rules as the equivalent non-const type. Types with the error flag set have the same assignability rules as the equivalent type with the flag unset.

In the context of an assignment expression, «6.6.44.2: Assignment» prevents the modification of objects with a const type. However, the assignability rules are referred to in many other contexts throughout the specification, and in these contexts, unless otherwise specified, non-const types are assignable to const types. For example, a binding which specifies a const type may use a non-const type for its expression.

6.6.44.11 For assignment of `* const char` from `str` types, see the notes on «6.6.33: Casts and type assertions».

6.6.44.12 The implementation shall ensure that any side-effects of the first term shall occur

before side-effects of the second term.

6.6.45 Variable binding

binding-list:
    static$_{opt}$ **let** *bindings*
    static$_{opt}$ **const** *bindings*

bindings:
    binding **,**$_{opt}$
    binding **,** *bindings*

binding:
    name **=** *expression*
    name **:** *type* **=** *expression*
    (*binding-names* **)** **=** *expression*
    (*binding-names* **)** **:** *type* **=** *expression*

binding-names:
    name **,** *name*
    name **,** *binding-names*

6.6.45.1 A binding-list shall cause one or more objects to become available in the present scope. Each object shall be identified by its name, and shall have its initial value set to the result of the expression or expression. The result type of a binding list expression is **void**.

6.6.45.2 In the first form of binding-list, the type of the object shall be equivalent to the result type of the expression. In the second form, the type shall be as indicated, and the result type of the expression shall be assignable to this type. In this second form, the type specified is used as a type hint for the expression.

6.6.45.3 The third and fourth forms of binding are the *tuple unpacking* form, and in this case, the type and expression shall be of a tuple type with a number of values equal to the number of times name is given. The implementation shall create separate bindings for each name, of the type of the corresponding tuple value, and initialize them to that value from the tuple.

6.6.45.4 If any name is _ (an underscore), a binding shall not be created, but any side-effects of the initializer shall still occur.

6.6.45.5 If the **const** form is used, the type of each binding shall be modified to *include* the **const** flag. If the **let** form is used, the type of each binding which uses the first form of binding shall be modified to *omit* the **const** flag.

6.6.45.6 If the **static** form is used, the variables shall be allocated *statically*, such that they are only initialized once and their previous value, accounting for any later mutations, is preserved each time the binding expression is encountered, including across repeated or recursive calls to the enclosing function. In this case, the initializer must use the «6.8: Translation compatible expression subset».

6.6.45.7 If a binding gives a name which is already defined in the current scope, the new binding shall *shadow* the earlier binding, causing any later references to this name to resolve to the newer binding.

6.6.45.8 If the second form of binding is used and the name is not already in use in this scope, the name shall be bound prior to the evaluation of the expression, and may be used in the expression *only* as the object of the unary addressing operator (**&**).

6.6.45.9 The type of the binding shall not use a type which has a zero or undefined size.

6.6.46 Deferred expressions

deferred-expression:
    defer *expression*

6.6.46.1 A deferred-expression causes another expression to be *deferred* until the termination of the current scope. The result type is **void**.

6.6.46.2 The implementation shall cause the expression to be evaluated upon the termination of the current scope, either due to normal program flow, or due to encountering a terminating expression.

6.6.46.3 If several expressions are deferred in a single scope, their side-effects shall occur in the reverse of the order that they appear in the program source.

6.6.46.4 If the current scope is terminated before a deferred-expression (*but not an expression which was already deferred*) would be evaluated, the side-effects of the expression shall not occur.

6.6.46.5 A deferred-expression shall not appear as a descendant of the expression tree formed by expression.

6.6.47 Compound expressions

expression-list:
    expression **;**
    expression **;** *expression-list*

compound-expression:
    $label_{opt}$ **{** *expression-list* **}**

label: exactly:
    : *name*

6.6.47.1 A compound-expression evaluates any number of expressions in sequence. If a label is present, the expression is considered *labelled*.

6.6.47.2 The result of a compound-expression shall be **void** unless the expression is selected by one or more yield-expressions. If the result types of the yield expressions are uniform, the compound-expression result type is that type. Otherwise, the result

type is the tagged union which is the sum of those types.

6.6.47.3 If a type hint is provided to an compound-expression, the hint shall be provided to the expression of any yield-expression which selects that compound-expression.

6.6.47.4 The expressions shall be evaluated such that the side-effects of each all occur in the order that each expression appears.

6.6.47.5 If any of the expressions terminate, the compound-expression is considered to terminate, unless the expression which would cause the compound-expression is a yield-expression which selects this compound-expressions scope. There shall be no expressions following the first terminating expression in a compound-expression.

6.6.47.6 The compound-expression shall establish a new scope whose parent is the scope in which the compound-expression resides.

6.6.48 Control statements

control-expression:
break *label*$_{opt}$
continue *label*$_{opt}$
return *expression*$_{opt}$
yield-expression

yield-expression:
yield
yield *expression*
yield *label*
yield *label* **,** *expression*

6.6.48.1 control-expression causes a *selected* compound-expression to terminate in a specific way. Control statements and yield expresisons are terminating expressions and their result type is **void**.

6.6.48.2 The rules for the selection of an applicable compound-expression vary based on the kind of control expression used. If a label is used, it shall select the corresponding labelled compound-expression from the anscestors of the control-expression, and the selected expression shall meet the requirements for the appropriate control expression type. Otherwise, the first qualifying compound-expression among the ancestors of the control-expression is used, ordered such that the nearest ancestor is considered first. If no suitable compound-expression is selected, a diagnostic message will be displayed and the translation phase shall terminate.

6.6.48.3 In the **break** and **continue** forms, the selected expression shall be the body of a for-loop. The **break** form shall cause the loop to abort without evaluating the *condition* or the *afterthought*. The **continue** form shall cause the loop to repeat immediately, running the *afterthought*, re-testing the *condition*, and repeating the loop if **true**.

6.6.48.4 The **return** form shall select the compound-expression which is the body of the function-declaration within which it appears. The expression shall be used as the

result value for the function, or, if absent, **`void`** shall be used. This expression shall receive the function's result type as a type hint, and the result shall be assignable to that type.

6.6.48.5 The yield-expression form shall select any compound-expression which is not eligible for the other forms; that is, any compound expression which is not a function or loop body. If an expression is provided, it shall not terminate, and shall be used as the result value for the compound-expression. See the description of compound-expression for details on the type semantics associated with a yield expression.

6.6.48.6 In any of these cases, the implementation shall ensure that side-effects do not occur for any expressions which are lexically situated *after* the control-expression within the selected compound-expression.

6.6.49 High-level expression class

expression:
- *assignment*
- *binding-list*
- *logical-or-expression*
- *if-expression*
- *for-loop*
- *switch-expression*
- *match-expression*
- *control-expression*
- *compound-expression*

6.7 Type promotion

6.7.1 The operands of some arithmetic expressions are subject to *type promotion*, to allow for arithmetic between disjoint types. The operand of lower precision may be *promoted*, or implicitly cast, to the precision of the more precise operand. Unless explicitly covered by the following cases, operands shall not be promoted, and the translation environment shall print a diagnostic message and abort for incompatible combinations of operand types.

6.7.2 For expressions where the result type is determined by type promotion, the result type shall be equivalent to the type of the operand which has the highest precision.

6.7.3 For expressions involving two integer types, the type of lower precision may be promoted to the type of higher precision only if the signedness is the same for each operand. A type cannot be promoted to **`uintptr`**.

6.7.4 For expressions involving floating-point types, **`f32`** may be promoted to **`f64`**.

6.7.5 For expressions involving pointer types, **`null`** may be promoted to any nullable pointer type, and a non-nullable pointer type may be promoted to a nullable pointer type with the same secondary type. Any pointer type may be promoted to a void pointer.

6.7.6 A mutable type may be promoted to a constant type which is otherwise equivalent to the mutable type, or any other type which that constant type may promote to.

6.7.7 A pointer type may be promoted to another pointer type if the secondary type of the pointer may be promoted to the secondary type of the second pointer.

6.7.8 An array type may promote to an array type with undefined size with an equivalent member type.

6.7.9 A non-aliased type A may promote to a type alias B if type A may promote to the underlying type of type B.

6.7.10 If one operand is a constant and the other is not, the constant shall receive the result type of the other operand as a type hint and type promotion shall be done between its result type and the other operand's result type.

6.7.11 If both operands are constants, they shall both receive the lowest-precision integer type which would hold both of them as a type hint.

6.8 Translation compatible expression subset

The translation-compatible expression subset is a subset of expression types which the implementation must be able to evaluate during the translation phase.

6.8.1 The following expression types are included:

- logical-or-expression
- logical-xor-expression
- logical-and-expression
- equality-expression
- comparison-expression
- inclusive-or-expression
- exclusive-or-expression
- and-expression
- shift-expression
- additive-expression
- multiplicative-expression
- cast-expression
- unary-expression
- field-access-expression
- indexing-expression
- measurement-expression
- nested-expression
- plain-expression

6.8.2 All terminals which are descendants of any of the listed terminals are included, and all non-terminals and terminals which are descendants of plain-expression are included.

6.8.3 The pointer dereference unary-expression (the $*$ operator) shall be excluded from the translation-compatible expression subset. Additionally, the implicit pointer type dereference semantics of field-access-expression and indexing-expression are not available.

6.8.4 The implementation is not required to use a conformant implementation of the storage semantics of types in the translation environment, provided that there are not observable side-effects in the execution environment as a result of any differences.

6.8.5 In a context where an expression is constrained to this subset, the use of an expression type outside of this set shall cause the translation environment to print a diagnostic message and abort.

6.9 Declarations

declarations:
    export$_{opt}$ *declaration* **;**
    export$_{opt}$ *declaration* **;** *declarations*

declaration:
    global-declaration
    constant-declaration
    type-declaration
    function-declaration

A declaration specifies the interpretation and attributes of a set of identifiers.

6.9.1 The identifiers shall be visible anywhere within the current translation unit. If the **export** keyword is used, the identifiers shall be part of the unit's exported interface.

6.9.2 The **export** keyword shall not be used with a function-declaration which uses the **@init**, **@fini**, or **@test** attributes.

6.9.3 Global declarations

global-declaration:
    let *global-bindings*
    const *global-bindings*

global-bindings:
    global-binding **,**$_{opt}$
    global-bindings **,** *global-binding*

global-binding:
    decl-attr$_{opt}$ *identifier* **:** *type*
    decl-attr$_{opt}$ *identifier* **:** *type* **=** *expression*

decl-attr:
    @symbol **(** *string-constant* **)**

6.9.3.1 In a global-declaration, sufficient space shall be reserved for each identifier in the global-bindings to store the type associated with it. That storage shall be initialized to the value of the expression and shall have alignment greater than or equal to the necessary alignment for the type. In the **const** form, the types shall have the constant flag enabled by default.

6.9.3.2 A global-binding's expression shall be limited to the «6.8: Translation compatible

expression subset», and shall be evaluated in the translation environment. The type of the value of the expression shall assignable to type.

6.9.3.3 The first form of global-binding is a *prototype*. In this form, the implementation shall not allocate storage for the global, and the programmer must arrange for storage to be provided elsewhere, e.g. during linking.

6.9.3.4 The interpretation of the **`@symbol`** form of decl-attr is implementation-defined.

6.9.4 Constant declarations

constant-declaration:
    `def` *constant-bindings*

constant-bindings:
    constant-binding **`,`**$_{opt}$
    constant-bindings **`,`** *constant-binding*

constant-binding:
    identifier **`:`** *type* **`=`** *expression*

6.9.4.1 In a constant-declaration, the identifiers in the constant-binding shall be available to the translation environment. No storage shall be allocated for them in the execution environment, and they shall not be addressable. References to them shall be equivalent to references to the expression associated with them, with a cast to type inserted.

6.9.4.2 A constant-binding's expression shall be limited to the «6.8: Translation compatible expression subset», and shall be evaluated in the translation environment. The type of the value of the expression shall assignable to type.

6.9.5 Type declarations

type-declaration:
    `type` *type-bindings*

type-bindings:
    identifier **`=`** *type* **`,`**$_{opt}$
    identifier **`=`** *type* **`,`** *type-bindings*

6.9.5.1 In a type-declaration, the identifiers shall declare type aliases. In a type-binding, the underlying type for the identifier shall be the type.

6.9.6 Function declarations

function-declaration:
    fndec-attrs$_{opt}$ **fn** *identifier prototype*
    fndec-attrs$_{opt}$ **fn** *name prototype* **=** *expression*

fndec-attrs:
    fndec-attr
    fndec-attr fndec-attrs

fndec-attr:
    @fini
    @init
    @test
    fntype-attr
    decl-attr

6.9.6.1 The first form of function-declaration is a *prototype*, and shall cause the identifier to refer to the function type described by the prototype and the function attributes. The implementation of this function shall be provided separately, or the translation phase shall fail.

6.9.6.2 The second form of function-declaration shall declare a function and its implementation. The result type of the expression shall be assignable to the prototype's result type. The function shall be available in the unit scope by its name, and available to other units by forming a fully-qualified identifier from the unit namespace and the name.

6.9.6.3 In the second form of function-declaration, each parameter in the prototype which uses the name form shall be available within the expression by its name. Those which use the _ form shall not be made available.

6.9.6.4 The **@fini** form of fndec-attr shall cause the function to be a finalization function. **@init** shall cause it to be an initialization function. If multiple fndec-attrs of the same type are specified, the last one shall override all previous ones.

References: «**??**: Initialization functions», «**??**: Finalization functions»

6.9.6.5 If **@init**, **@fini**, **@test**, or **@noreturn** are given, the result type shall be **void**.

6.9.6.6 The **@test** attribute indicates that a function is used for testing. The name of the function need not be unique within its namespace, and it shall not be inserted into the unit's scope. **@test** functions shall not be exported. Other semantics of **@test** functions are implementation-defined.

6.10 Units

sub-unit:
    imports$_{opt}$ *declarations*$_{opt}$

imports:
    use-statement
    use-statement *imports*

use-statement:
    `use` *identifier* `;`
    `use` *name* `=` *identifier* `;`
    `use` *identifier* `::` `{` *name-list* `}` `;`

name-list:
    name `,`$_{opt}$
    name `,` *name-list*

6.10.1 A unit, or translation unit, is composed of several source files as described by «5.3: Translation steps». Each source file is a sub-unit. A specific sub-unit may have no declarations, but the unit shall contain at least one declaration among its sub-units.

6.10.2 An import shall declare a dependency between this translation unit and another module of the namespace specified by the use-statement identifier. This shall cause the named module to be linked into the final program image as described by «5.3: Translation steps».

6.10.3 The first form of the use-statement shall cause the identifiers exported by the target module to become visible to this sub-unit in their fully-qualified form. Additionally, if the imported module has more than one namespace, identifiers of the form "x::y" shall be made available, where x is the most-specific namespace, and y is each of the exported members of the target module.

6.10.4 The second form of the use-statement shall cause the identifiers declared by the target module to become visible to this sub-unit in a rewritten form, with the fully-qualified namespace of the identifiers being visible under the namespace described by the name given in this form.

In the use statement `use foo = bar::baz;`*, identifiers in the namespace* `bar::baz` *will be visible under the namespace* `foo`*. For example, if the fully-qualified identifier* `bar::baz::bat` *exists, this sub-unit may refer to it as* `foo::bat`.

6.10.5 The third form of the use-statement shall cause only the identifiers listed in the name-list, qualified in the context of the target namespace, to become visible in their un-qualified form to this sub-unit.

If the use statement `use bar::baz::{bat}` *were specified in the same conditions as the previous example, the fully-qualified identifier* `bar::baz::bat` *may be referred to by its unqualified name* `bat` *in the scope of this sub-unit.*

6.10.6 The translation unit shall establish a top-level scope into which all unit-local declarations are inserted. Each sub-unit shall establish another scope whose parent scope is the top-level

unit scope, and in this sub-unit scope, each of the imports used by that sub-unit shall be made available.

In other words, declarations made in a sub-unit are visible to other members of that unit, but imports in a sub-unit are not visible to other sub-units.