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 only covers the language itself (its grammar and semantics). The document does not specify any parts of the runtime library, nor does it specify 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 will be published there as well. You may also contact the editor via email to Drew DeVault <sir@cmpwn.com>.

Contents

1 Introduction **4**

1.1 Copyright 4

2 Scope **5**

3 Terms and definitions **6**

4 Conformance **7**

5 Program environment **8**

5.2 Translation environment 8

5.3 Translation steps 8

5.4 Execution environment 9

5.4.4 The freestanding environment 9

5.4.5 The hosted environment 9

5.4.6 The test environment 10

5.4.7 Program execution 11

5.5 Diagnostics 11

6 Language **12**

6.1 Notation 12

6.2 Lexical analysis 13

6.3 Keywords 15

6.4 Attributes 15

6.5 Identifiers 16

6.6 Types 17

6.6.8 Integer types 18

6.6.9 Floating point types 19

6.6.10 Rune types 19

6.6.11 Flexible types 19

6.6.12 Other primitive types 20

6.6.13 Pointer types 21

6.6.14 Struct and union types . . . 22
6.6.15 Tuple types . . . 24
6.6.16 Tagged union types . . . 24
6.6.17 Slice and array types . . . 25
6.6.18 String types . . . 27
6.6.19 Function types . . . 27
6.6.20 Type aliases . . . 28
6.7 Expressions . . . 29
6.7.4 Literals . . . 30
6.7.5 Floating literals . . . 31
6.7.9 Integer literals . . . 32
6.7.14 Rune literals . . . 33
6.7.15 String literals . . . 34
6.7.16 Array literals . . . 34
6.7.17 Struct literals . . . 35
6.7.18 Tuple literals . . . 36
6.7.19 Plain expressions . . . 37
6.7.20 Allocations . . . 37
6.7.21 Assertions . . . 38
6.7.22 Calls . . . 39
6.7.23 Measurements . . . 40
6.7.24 Field access . . . 41
6.7.25 Indexing . . . 41
6.7.26 Slicing . . . 42
6.7.27 Appending . . . 43
6.7.28 Inserting . . . 44
6.7.29 Deleting . . . 45
6.7.30 Error propagation . . . 46
6.7.31 Postfix expressions . . . 47
6.7.32 Variadic expressions . . . 47
6.7.33 Builtin expressions . . . 48
6.7.34 Unary arithmetic . . . 48
6.7.35 Casts and type assertions . . . 49
6.7.36 Multiplicative arithmetic . . . 51

6.7.37 Additive arithmetic . . . 51
6.7.38 Bit shifting arithmetic . . . 52
6.7.39 Bitwise arithmetic . . . 53
6.7.40 Logical comparisons . . . 53
6.7.41 Logical arithmetic . . . 54
6.7.42 If expressions . . . 55
6.7.43 For loops . . . 56
6.7.44 Switch expressions . . . 57
6.7.45 Match expressions . . . 58
6.7.46 Assignment . . . 60
6.7.47 Binding expressions . . . 62
6.7.48 Defer expressions . . . 63
6.7.49 Compound expressions . . . 64
6.7.50 Control expressions . . . 65
6.7.51 High-level expression class . . . 66
6.8 Type promotion . . . 66
6.9 Translation compatible expression subset . . . 67
6.10 Result type reduction algorithm . . . 68
6.11 Flexible type promotion algorithm . . . 68
6.12 Declarations . . . 69
6.12.4 Global declarations . . . 70
6.12.5 Constant declarations . . . 71
6.12.6 Type declarations . . . 72
6.12.7 Function declarations . . . 73
6.13 Units . . . 74

A Language syntax summary **76**

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

© Drew DeVault, Ember Sawady, et al., 2020–2022

This document is licensed under the terms of CC-BY-ND. Free redistribution of this document is permitted, but derivative works are not allowed. Software which implements this specification are not considered derivative works; you may freely apply this specification as such without restriction.

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 expects 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, unless the implementation supports unaligned memory accesses.*

3.3 **character**: a single Unicode codepoint 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.

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 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.

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 text encoded in the UTF-8 format.

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*. Every module contains one or more *declarations*, any number of which may be *exported* for use by other modules.

Forward references: «A: Units»

5.3 Translation steps

5.3.1 The list of source files constituting the translation unit are identified in an implementation-defined manner. 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: «A: Lexical analysis»

5.3.3 Syntax analysis is conducted on the token stream, mapping the tokens to a sub-unit.

Forward references: «A: Units»

5.3.4 Logical analysis is conducted on the sub-units. 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 dependencies and translated into a single program image which is suitable for interpretation by the execution environment.

5.4 Execution environment

5.4.1 Three execution environments are defined: *freestanding*, *hosted*, and *test*. The implementation must support a freestanding environment; support for any other environment is optional.

5.4.2 During *program startup*, the execution environment shall first initialize all global declarations to their initial values in an implementation-defined manner. The next steps are dependent on the execution environment.

5.4.3 Behavior during *program teardown* is dependent on the execution environment.

5.4.4 The freestanding environment

5.4.4.1 In the freestanding environment, behavior during program startup after global declarations are initialized, behavior during program teardown, as well as behavior of initialization and finalization functions, is undefined.

Forward references: «6.12.7: Function declarations»

5.4.5 The hosted environment

5.4.5.1 In the hosted environment, after global declarations are initialized during program startup, the execution environment shall call all initialization functions in an order such that the initialization functions for a given module are run before those of any module that depends on it, then transfer control to the program *entry point* in an implementation-defined manner. The ordering of initialization functions within a module shall be undefined.

5.4.5.2 Program teardown in the hosted environment shall cause the execution environment to call all finalization functions in an order such that the finalization functions for a given module are run after those of any module that depends on it, then terminate. The ordering of finalization functions within a module shall be undefined.

5.4.5.3 The program entry point shall be a function named `main` in the root namespace. The declaration shall not have any attributes. The function shall have no parameters and a result type of **`void`**. The declaration shall be exported, unless the declaration is a prototype, in which case it need not be exported.

The signature of a conformant entry point follows:

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

The program shall provide this declaration in the root namespace.

5.4.5.4 Alternatively, the entry point may be declared with a different name and/or in a different namespace if a recognized form of **`@symbol`** is supplied to the declaration. The set of recognized forms, if any exist, are implementation-defined.

If an implementation used the symbol "main" for its entry point, the following signature would be a valid entry point declaration:

```
export @symbol("main") fn any::other::name() void;
```

5.4.5.5 The identifier `main` shall not be used for any root namespace declaration which would not be a valid entry point, unless the declaration uses any of **`@symbol`**, **`@init`**, **`@fini`**, or **`@test`**. The implementation-defined set of recognized forms of **`@symbol`** shall not be used in any namespace for a declaration which would not be a valid entry point.

5.4.5.6 Returning from the program entry point shall begin the program teardown process.

Forward references: «6.12.4: Global declarations», «6.12.7: Function declarations»

5.4.6 The test environment

5.4.6.1 In the test environment, after global declarations are initialized during program startup, the execution environment shall call all initialization functions in an order such that the initialization functions for a given module are run before those of any module that depends on it, then transfer control to the *test runner*. The ordering of initialization functions within a module shall be undefined.

5.4.6.2 Program teardown in the hosted environment shall cause the execution environment to call all finalization functions in an order such that the finalization functions for a given module are run after those of any module that depends on it, then terminate. The ordering of initialization functions within a module shall be undefined.

5.4.6.3 The test runner shall call any number of test functions in an unspecified order, and then begin the program teardown process.

The implementation should provide a means for the user to specify which tests to run.

5.4.6.4 Prior to calling each test function, as well as after calling the final test function, the test runner shall reset the execution environment in an implementation-defined manner.

Example *Resetting the execution environment may involve setting the floating point rounding mode to the default, clearing all floating point exceptions, resetting all signal handlers to their defaults, and routing standard I/O streams to known locations.*

5.4.6.5 Each test function designates one test, which may either *pass* or *fail*. By default, if control returns from a test function, the test passes; if the test function aborts (such as from an assertion-expression or a failed type assertion), the test fails. The implementation may declare functions in the runtime library which change the pass and fail conditions for the running test. Aborting from a test function shall not terminate the program; the next test function shall always be executed.

5.4.6.6 The test runner shall keep track of which tests pass and which tests fail, and print this information in an implementation-defined manner, possibly alongside other messages.

Forward references: «6.12.7: Function declarations»

5.4.7 Program execution

5.4.7.1 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.

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

Forward references: «A: 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.

Example *On a Unix-like 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 A: 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 will use the key phrase "one of".

6.1.3 Most nonterminals are tolerant of white-space characters inserted between their terminals. However, some are not—these will 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 **`;`**
    switch-case **`;`** *switch-cases*

6.1.5 When the `U+XXXX` notation is used, where `XXXX` denotes any number of hexadecimal digits, the hexadecimal digits are to be interpreted as the value of the denoted Unicode codepoint, as specified by ISO/IEC 10646.

Example `U+000A` *denotes the Unicode codepoint with the value 10 (line feed).*

6.1.6 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.

rawstring-char:
    Any character other than `

6.2 Lexical analysis

token:
    comment
    integer-literal
    floating-literal
    rune-literal
    string-section
    keyword
    name
    operator
    attribute
    invalid-attribute
    annotation

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

comment: exactly:
    `//` *comment-chars*

comment-chars: exactly:
    comment-char *comment-chars*$_{opt}$

comment-char:
    Any character other than `U+000A`

annotation:
    `#` `[` *identifier* *annotation-tokens*$_{opt}$ `]`

annotation-tokens:
    `(` ...balanced token list... `)`

6.2.1 A token is the smallest unit of meaning in the Hare grammar. The lexical analysis phase processes a 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 codepoints `U+0009` (horizontal tabulation), `U+000A` (line feed), and `U+0020` (space). Any number of white-space characters may be inserted between tokens, either to disambiguate from subsequent tokens, or for aesthetic purposes. This white-space 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 white-space therein is not*

discarded.

6.2.3 The lexical analysis process shall repeatedly consume Unicode characters from the source file input until there are no more characters to consume. White-space characters shall be discarded. When a non-white-space character is encountered, it shall mark the beginning of a token: the longest sequence of characters which constitutes a token shall then be consumed and emitted to the token stream, unless the token is a comment, in which case it shall be discarded. If no token can be formed, a diagnostic message shall be printed and the translation phase shall abort.

6.2.4 An annotation is used to provide information to third-party tooling. The "balanced token list" shall include an arbitrary number of valid lexical tokens which must be *balanced*, such that each of the following "opening" tokens are paired with the corresponding "closing" token, for the tokens `(` and `)`, `[` and `]`, and `{` and `}`.

6.2.5 The translation environment shall unconditionally discard annotations, and shall not use them to inform the translation process in any way.

This stipulation prohibits Hare compilers from using annotations as the basis of vendor language extensions.

6.2.6 The translation environment may impose an upper limit on the level of nested balanced token constructs in an annotation permitted by the implementation.

6.2.7 Third-party tools may interpret annotations in any manner of their choosing. Tools shall define a namespace for annotations pertraining to that tool, and discard annotations outside of the namespaces they are designed to work with. Tools shall further define a grammar which applies to the balanced list of tokens in whatever manner is necessary to suit their use-case.

Annotations are designed to lend additional contextual information to third-party tools that perform some kind of analysis of Hare source code, for instance to provide inputs to code generation tools, to tune the behavior of linters, and so on. Some examples:

```
let x: nullable *int = // ...
#[lint::disable("warn-unsafe-cast")]
do_work(x: *int);

#[json::generate]
type position = struct {
        #[json::name("X")]
        x: int,
        #[json::name("Y")]
        y: int,
        #[json::name("Z")]
        z: int,
};
```

6.3 Keywords

keyword: one of:

abort align alloc append as assert bool break case const continue def defer delete else enum export f32 f64 false fn for free i16 i32 i64 i8 if insert int is len let match never null nullable offset opaque return rune size static str struct switch true type u16 u32 u64 u8 uint uintptr union use vaarg vaend valist vastart void yield _

6.3.1 Keywords 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, such as «A: Identifiers» and labels.

6.4 Attributes

attribute: one of:

@fini @init @offset @packed @symbol @test @threadlocal

invalid-attribute: exactly:

@ *name*

6.4.1 Attributes are terminals with special meaning. They are case-sensitive.

6.4.2 invalid-attribute isn't used anywhere in the syntax. If an invalid-attribute is consumed during lexical analysis, the program is invalid, and the translation phase shall print a diagnostic message and abort.

The purpose of invalid-attribute is to disallow the use of a keyword or name immediately after an attribute, unless the tokens are separated by white-space. Thus, the following program is invalid:

```
@testfn test() void = void;
```

6.5 Identifiers

identifier:
    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
    _

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

alnum:
    decimal-digit
    nondigit

6.5.1 An *identifier* is a user-defined sequence of **name** components which denote a namespace, object, function, type alias, or enumeration value.

6.5.2 An identifier is only meaningful within a specific *scope* of the program. The scope is defined by the region of the program it encompasses: it may be a translation unit, a sub-unit, a function, enum-values, a for-loop, or a compound-expression. After an identifier is *inserted* into a scope, it is considered *visible* within the region that defines its scope.

6.5.3 Scopes may overlap one another. For any two overlapping scopes, one scope will encompass a region which is contained entirely within the region encompassed by the other scope.

6.5.4 Scopes which encompass a for-loop or compound-expression may be *labelled*. A labelled scope has a name associated with it; the name need not be unique.

6.5.5 Every module in a program, except for one, is assigned a unique identifier; this is the module's *namespace*. The exempt module may also be assigned an identifier, or it may instead be assigned the *root namespace*. Unless otherwise specified, identifiers within a program don't designate namespaces.

6.5.6 The implementation may define the maximum length of an identifier and/or name.

6.6 Types

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

storage-class:
    primitive-type
    pointer-type
    struct-union-type
    tuple-type
    tagged-union-type
    slice-array-type
    function-type
    alias-type
    unwrapped-alias

primitive-type:
    integer-type
    floating-type
    bool
    never
    opaque
    rune
    str
    valist
    void

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

6.6.2 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, their error flag or lack thereof, or their default values, are distinct. Each type class may impose additional distinguishing characteristics on their types, which are specified in their respective sections.

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

6.6.4 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.

6.6.5 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.

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

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

6.6.8 Integer types

integer-type: one of:
`i8 i16 i32 i64 u8 u16 u32 u64 int uint size uintptr`

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

6.6.8.2 Signed integer types shall be represented in two's complement form, such that arithmetic on signed operands behaves identically to arithmetic on unsigned operands. The most significant bit of a signed integer shall determine the sign of the value: the value shall be positive if the most significant bit is unset, otherwise it shall be negative. Zero shall be positive.

6.6.8.3 For integer types whose size is greater than one octet, the order in which its octets are represented is implementation-defined, provided that the significance strictly either increases or decreases as the memory address increases.

6.6.8.4 If an operation on an integer type would cause the result to overflow, it is truncated towards the least significant bit.

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

6.6.8.6 The width of **`int`** and **`uint`** are implementation-defined. **`int`** shall be signed, and **`uint`** shall be unsigned. Both types shall have the same width, which, in bits, shall be a power of two no smaller than 32.

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

6.6.8.8 The width of **`uintptr`** is implementation-defined. It shall be able to represent the value of any pointer-type (including *null*). It shall be unsigned.

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

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

The following table is informative.

Type	Width in bits	Minimum value	Maximum value
i8	8	-128	127
i16	16	-32768	32767
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	$*$	0	$*$
uintptr	$*$	0	$*$

$*$ implementation-defined

6.6.9 Floating point types

floating-type:
f32
f64

6.6.9.1 Floating point types shall represent approximations to real numbers. Floating point types are considered *numeric types*.

6.6.9.2 The bit layout of floating point types shall be implementation-defined.

6.6.9.3 **f32** shall have a *width* of 32 and a size and alignment of 4. **f64** shall have a width of 64 and a size and alignment of 8.

6.6.9.4 The alignment of floating point types shall be equal to their size in octets.

6.6.9.5 If an operation on a floating point type would cause the result to overflow or underflow, it is truncated towards zero.

6.6.9.6 The default value of a floating point type shall be positive zero.

6.6.10 Rune types

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

6.6.11 Flexible types

6.6.11.1 Flexible types are used during the translation phase as the result types of literals whose final result types have not yet been determined. The three classes of flexible type are flexible integers, flexible floats, and flexible runes. The size and alignment of flexible types shall be undefined. Flexible integers and flexible floats are considered numeric types.

6.6.11.2 A flexible integer shall have a maximum and minimum value associated with it,

each of which may assume any value in the inclusive range -9223372036854775808 to 18446744073709551615. The minimum value shall represent the value of the smallest integer literal which may have this result type, and the maximum value shall represent the largest. The minimum value shall always be less than or equal to the maximum, and the type shall be considered unsigned if and only if the minimum value is greater than or equal to zero.

6.6.11.3 The *default type* for a flexible float shall be **`f64`**. The default type for a flexible rune shall be **`rune`**. The default type for a flexible integer shall be **`int`** if the maximum and minimum values are representable as **`int`**, otherwise **`i64`** if the maximum and minimum values are representable as **`i64`**, otherwise **`u64`** if the maximum and minimum values are representable as **`u64`**, otherwise the translation environment shall print a diagnostic message and abort.

6.6.11.4 A flexible type may be *lowered* to a different type by the «6.11: Flexible type promotion algorithm», causing all instances of that type within the translation environment to be replaced by the new type. If a flexible type hasn't been lowered by the end of the translation phase, it shall be lowered to its default type.

6.6.11.5 There is no grammar for defining flexible types. They are for internal use only, as the result type of integer-literal, floating-literal, and rune-literal.

6.6.11.6 Each literal with a flexible result shall have a distinct type.

6.6.11.7 If a flexible type is embedded within another type, it shall first be lowered to its default type.

This is intended to ease implementation, in order to prevent a type's ID, size, and alignment from changing partway through the translation phase.

6.6.12 Other primitive types

The bool type

6.6.12.1 The **`bool`** type represents a boolean value, which may have one of two states: true (represented as one) or false (represented as zero).

6.6.12.2 The boolean type shall have the same representation, size, and alignment as **`u8`**.

6.6.12.3 If a boolean object has a value which isn't true or false, the interpretation of the value is unspecified.

6.6.12.4 The default value of a boolean type is false.

The never type

6.6.12.5 The **`never`** type has no representable values, and no storage. It is used as the result type of expressions which are guaranteed to *never* return to their caller.

6.6.12.6 The size and alignment of **`never`** shall be undefined.

6.6.12.7 If an expression whose result type is **`never`** returns to its caller, behavior is undefined.

This isn't possible under normal circumstances, but conditions such as incorrect external function implementations and invalid function pointer casts can cause this

to occur.

The null type

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

6.6.12.9 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 result type of the **`null`** expression.

The opaque type

The **`opaque`** type represents an object whose size, alignment, and representation are unknown. As such, the opaque type has an undefined size and alignment, and can't store any value.

No value exists whose type is **`opaque`**. *Rather,* **`opaque`** *is intended to be used as the secondary type of a pointer or slice type, to exchange data whose representation is generic or unknown.*

The valist type

6.6.12.10 The **`valist`** type is provided for compatibility with the C programming language as specified by ISO/IEC 9899. Implementation support is optional: implementations which do not provide C ABI compatibility must parse this type, print a diagnostic message, and abort.

6.6.12.11 The size and alignment of **`valist`** shall be an implementation-defined non-zero multiple of eight.

6.6.12.12 Assigning to a **`valist`** object from an object-selector shall initialize a copy of the object-selector, exactly as though a new **`valist`** were initialized with **`vastart`**, followed by the same sequence of **`vaarg`** uses that had been previously used to reach the present state of the object-selector.

If any previously evaluated **`vaarg`** *expression on the object-selector violated a runtime constraint, behavior of all further* **`vaarg`** *expressions on the newly created object is undefined.*

Forward references: «6.7.32: Variadic expressions»

6.6.12.13 A **`valist`** object shall only be valid within the function it was initialized in.

The void type

6.6.12.14 The **`void`** type represents an object with no storage. Only one value with this type exists (the *void* value); this value is also the default value.

6.6.12.15 The size and alignment of **`void`** shall be zero.

6.6.13 Pointer types

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

6.6.13.1 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.

6.6.13.2 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*. A non-nullable pointer type shall **always** refer to a valid secondary object.

6.6.13.3 A pointer type's secondary type must have a non-zero size. Its size may be undefined. The secondary type shall not be **`never`**.

6.6.13.4 The representation of a pointer type shall be implementation-defined, and it shall have an implementation-defined size and alignment.

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

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

6.6.14 Struct and union types

struct-union-type:
`struct` **`@packed`**$_{opt}$ **`{`** *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* **`)`**

6.6.14.1 The *struct type* and *union type collect* multiple types, name them, and assign them *offset*s within their storage area. A union type stores all of its values at the same offset; a struct type may store its values at different offsets. 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.

6.6.14.2 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.

6.6.14.3 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. Padding shall additionally be added to the end of a struct or union type whose alignment is

non-zero, such that the total size of the struct or union type modulo its alignment is zero. A struct or union type's alignment is the maximum alignment among its fields.

6.6.14.4 For a struct type using the **`@packed`** modifier, the offset of each field shall be computed without respect to alignment, such that each field's offset is equal to the offset of the previous field plus the size of the previous field. No additional padding shall be added to the end of the struct type in this case. If the alignment of the struct fields or the struct type itself would not meet the alignment requirements for their respective 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).

6.6.14.5 The type of each struct or union field shall have a definite size.

6.6.14.6 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).

6.6.14.7 The expression given for the offset-specifier shall be limited to the «6.9: Translation compatible expression subset», and shall have an integer type and a positive or zero value.

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

6.6.14.9 The default value of a struct type shall be defined as a value whose fields assume the default values of their respective types. If any field's default value is undefined, the struct type's default value shall also be undefined.

6.6.14.10 The default value of a union type shall be undefined.

6.6.14.11 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, or an alias thereof, otherwise a diagnostic message shall be printed and the translation phase shall abort.

Forward references: «6.6.20: Type aliases»

6.6.14.12 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 } }
```

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

6.6.15 Tuple types

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

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

6.6.15.1 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.

6.6.15.2 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. Padding shall additionally be added to the end of a tuple type whose alignment is non-zero, such that the total size of the tuple type modulo its alignment is zero.

6.6.15.3 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.

6.6.15.4 The type of each tuple value shall have a definite size.

6.6.15.5 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.

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

6.6.16 Tagged union types

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

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

6.6.16.1 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.

6.6.16.2 The representation of a tagged union shall be a **`u32`**, in which the tag value is stored, followed by sufficient space to store any of the possible constituent types. Padding shall be inserted prior to the tag if necessary to meet the maximum alignment among the tagged union members and the **`u32`** field.

The representation of `(u8 | i16)` *is equivalent to*

```
struct {
        tag: u32,
        union {
                unsigned: u8,
                signed: i16,
        },
}
```

And the representation of `(u64 | []u8)` *is equivalent to*

```
struct {
        pad: u32,
        tag: u32,
        union {
                unsigned: u64,
                slice: []u8,
        },
}
```

6.6.16.3 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 **`u32`** field and shall indicate which type is stored in the value area.

6.6.16.4 The alignment of a tagged union type shall be the alignment of the **`u32`** type or the maximum alignment of the constituent types, whichever is greater.

6.6.16.5 The size of a tagged union type shall be the maximum size of its constituent types, plus the size of the **`u32`** type, plus any padding added per «6.6.16.2: Tagged union types».

6.6.16.6 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.*

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

6.6.16.8 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.

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

6.6.17 Slice and array types

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

6.6.17.1 An *array type* stores one or more items of a uniform secondary type. The number of items stored in an array type (its *length*) is an attribute of the array type and is specified during the translation phase. The secondary type shall have a definite, non-zero size.

6.6.17.2 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 «6.9: Translation compatible expression subset».

6.6.17.3 An array type may be *unbounded*, in which case the length is not known. The `*` representation indicates an array of this type.

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

6.6.17.5 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.

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

6.6.17.7 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 secondary type shall not have a size of zero, and shall not be **`never`**. The representation with no lexical elements between **`[`** and **`]`** indicates a slice type.

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

```
struct {
        data: nullable *opaque, // See notes
        length: size,
        capacity: size,
}
```

If the secondary type has a definite size, the type of the *data* field shall be a nullable pointer to an unbounded array of the secondary type, otherwise, it shall be a nullable pointer to an unbounded array whose secondary type is unknown.

6.6.17.9 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 «6.6.13: Pointer types», whichever is greater.

6.6.17.10 The size of a bounded array type shall be equal to $N \times S$, where N is the length 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 «6.6.17.6: Slice and array types».

6.6.17.11 The default value of a bounded array type shall be an array whose members are all set to the default value of the secondary type. If the array is unbounded, or if the

default value of the secondary type is undefined, the default value of the array type is undefined.

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

6.6.17.13 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.

6.6.18 String types

6.6.18.1 The **`str`** type (or *string type*) 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.

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

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

6.6.18.3 The default value of a string type shall have the length field set to zero and the data field set to null.

6.6.19 Function types

function-type:
    `fn` *prototype*

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

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

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

parameter:
    name **`:`** *type*
    type

6.6.19.1 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 «5.4.7.1: Program execution»).

6.6.19.2 The attributes of a function type are its *result type* and *parameters*. A function type must have one result type, and zero or more parameters. Within a list of parameters, no two parameters which use the name form may have the same name. The type of every parameter shall have a definite size.

6.6.19.3 If the second form of parameter-list is used, the final parameter of the function type uses *Hare-style variadism*. If the third or fourth 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: «6.7.22: Calls»

6.6.19.4 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.

6.6.19.5 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`.

6.6.19.6 The size, alignment, default value, and storage semantics of function types is undefined.

6.6.19.7 The function's result type, list of parameter types (in order), and its variadism, are distinct characteristics of the function type, for the purpose of determining equivalency.

6.6.20 Type aliases

alias-type:
    identifier

unwrapped-alias:
    ... *identifier*

6.6.20.1 A type alias assigns an identifier a unique type which is an alias for another type or a name for a set of enum values.

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

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

6.6.20.3 The underlying type of a type alias shall not be **`never`**.

6.6.20.4 A type alias that represents an enum type shall have a default value of zero only if one of the enum values is equal to zero, otherwise its default value is undefined. A type alias that doesn't represent an enum type shall have the same default value as its underlying type.

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

6.6.20.6 An unwrapped-alias shall refer to the underlying type of the given type alias, 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);
```

6.7 Expressions

6.7.1 An expression is a procedure which the implementation may perform to obtain a *result*, and possibly cause side-effects (see «5.4.7.1: Program execution»). All expressions have a defined *result type*.

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

6.7.3 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.

6.7.4 Literals

literal:
    integer-literal
    floating-literal
    rune-literal
    string-literal
    array-literal
    struct-literal
    tuple-literal
    `true`
    `false`
    `null`
    `void`

6.7.4.1 Literals describe a specific value of an unambiguous type (which may be a flexible type).

6.7.4.2 The keywords **`true`** and **`false`** respectively represent the values of the **`bool`** type.

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

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

6.7.5 Floating literals

floating-literal: exactly:
nonzero-decimal-digits **.** *decimal-digits* *decimal-exponent*$_{opt}$ *floating-suffix*$_{opt}$
nonzero-decimal-digits *decimal-exponent*$_{opt}$ *floating-suffix*
0x *hex-digits* **.** *hex-digits* *binary-exponent* *floating-suffix*$_{opt}$
0x *hex-digits* *binary-exponent* *floating-suffix*$_{opt}$

floating-suffix: one of:
f32 f64

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

nonzero-decimal-digits: exactly:
0
nonzero-decimal-digit *decimal-digits*$_{opt}$

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

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

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

binary-exponent: exactly:
binary-exponent-char *sign*$_{opt}$ *decimal-digits*

sign: one of:
+ -

decimal-exponent-char: one of:
e E

binary-exponent-char: one of:
p P

Floating literals represent an approximation to a real number.

6.7.6 If the floating-suffix is not provided, the result shall be a flexible float. Otherwise, the type shall refer to the type named by the suffix.

6.7.7 decimal-digits shall be interpreted as base 10. hex-digits shall be interpreted as base 16, in

a case-insensitive manner.

6.7.8 In the first two forms, if the decimal-exponent is provided, the value of the literal shall be multiplied by 10 to the power of decimal-digits. In the third and fourth forms, the value of the literal shall be multiplied by 2 to the power of decimal-digits. If sign is provided within the exponent, it shall assume the given sign.

6.7.9 Integer literals

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

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`

positive-decimal-exponent:
decimal-exponent-char **`+`**$_{opt}$ *decimal-digits*

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

Integer literals represent an integer value.

6.7.10 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. Otherwise, the type shall be a flexible integer with both the maximum value and minimum value set to the number provided.

6.7.11 If the number provided is not representable within the type, or if the number is not representable as a flexible integer, a diagnostic message shall be printed and the translation phase shall fail.

6.7.12 octal-digits shall be interpreted as base 8. binary-digits shall be interpreted as base 2.

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

6.7.14 Rune literals

rune-literal: exactly:
`'` *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 \\ \' \"`

6.7.14.1 The result type of a rune-literal shall be a flexible rune type.

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

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

6.7.14.4 A rune-literal 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+0027
`\"`	U+0022		

6.7.15 String literals

string-literal:
    string-section *string-literal*$_{opt}$

string-section: exactly:
    " *string-chars*$_{opt}$ **"**
    `**  *rawstring-chars*$_{opt}$  **`

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

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

rawstring-chars: exactly:
    rawstring-char *rawstring-chars*$_{opt}$

rawstring-char:
    Any character other than **`**

6.7.15.1 A string-literal expression shall have a result type of **`const str`**.

6.7.15.2 If the first form of string-section 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 «6.7.14.17: Rune literals».

6.7.15.3 If the second form of string-section is used, the string's *data* field shall refer to a UTF-8 encoded sequence of Unicode codepoints, ascertained by encoding the sequence of rawstring-chars given in order.

6.7.15.4 If a string-literal consists of more than one string-section, the string's *data* field shall refer to a UTF-8 encoded sequence of Unicode codepoints, ascertained by concatenating the *data* fields of the string-sections given in order.

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

6.7.16 Array literals

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

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

6.7.16.1 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.

6.7.16.2 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.

6.7.16.3 The array-members shall be evaluated in the order in which they appear in the array-literal, and the *N*th item shall provide the value for the *N*th array member.

6.7.16.4 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.

6.7.17 Struct literals

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

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

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

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

6.7.17.1 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*.

6.7.17.2 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.

6.7.17.3 If the named form is given, the identifier shall identify a type alias (see «6.6.20: Type aliases») which refers to a struct or union type. The result type shall be this alias type.

6.7.17.4 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.

6.7.17.5 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.

6.7.17.6 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.

6.7.17.7 If the named type is a union type, the field-values shall be empty and `...` provided. The union type must have a default value.

6.7.17.8 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 } }
```

6.7.18 Tuple literals

tuple-literal:
`(` *tuple-items* `)`

tuple-items:
expression `,` *expression* `,`*opt*
expression `,` *tuple-items*

6.7.18.1 A tuple-literal produces a value of a tuple type. The result type shall be the tuple type described by the types of its expressions in the order that they appear.

6.7.18.2 If a type hint is available and the hint is a tuple type (or a type alias which represents a tuple type), the tuple items shall receive as hints the types of the respective tuple sub-types in the order that they appear.

6.7.18.3 The tuple-items shall be evaluated in the order in which they appear in the tuple-literal, and the *N*th item shall provide the value for the *N*th tuple item.

6.7.19 Plain expressions

plain-expression:
    identifier
    literal

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

6.7.19.1 plain-expression is an expression class which represents its result value "plainly". In the case of an identifier, the expression produces the value of the identified object.

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

6.7.20 Allocations

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

6.7.20.1 An **alloc** expression allocates an object at runtime and initializes its value to the first expression (the *initializer*). The result type of the *initializer* provides the allocation's *object type*. The result type of the allocation expression is, usually, a pointer type whose secondary type is the object type.

6.7.20.2 The first form is the object allocation form. If the allocation expression is provided a type hint of a pointer type, the secondary type of this pointer type shall be provided to the initializer as its type hint. The execution environment will allocate sufficient storage for the object type and initialize its value using the initializer expression, then set its result to a pointer to the new object. The object type must have a defined size which is greater than zero.

6.7.20.3 The second form is the copy allocation form. In this form, the initializer will provide an object type which is either a slice or array type (or an alias thereof). The execution environment will allocate storage sufficient to store an array equal in length to the initializer, then copy the initializer's slice or array items into this array. The result value shall be set to a slice object whose secondary type is equal to the secondary type of the initializer's result type, whose data field refers to the new array, and whose length and capacity fields are set to the length of the array.

6.7.20.4 The third form is the slice allocation form. In this form, the initializer will provide an object type which is either a slice or array type (or an alias thereof), and the second expression provides the desired *capacity* for a new slice. The capacity 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 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.

6.7.20.5 In the slice allocation form, if the array specified by the initializer is *expandable*, the second expression shall be used as the *length* instead of the capacity. The execution environment shall choose a capacity equal to or greater than this value, then for each Nth value of the allocated array from the length of the initializer (L, inclusive) to the specified *length* (L') shall be initialized to the value at $L - 1$ in the initializer. The length field of the resulting slice value shall be set to the L'.

The following allocates a slice of length 10 with all values set to zero:

```
let x:  []int = alloc([0...], 10)
```

6.7.20.6 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 hint is a nullable pointer type, the result type of the allocation expression shall also be nullable, and **null** shall be returned instead of aborting if sufficient storage cannot be provided.

6.7.20.7 The **free** form shall discard resources previously allocated with a **alloc** expression, freeing them for future use. If the expression's result type is a pointer type, and the result value compares unequal to **null**, then the object referred to by the pointer shall be freed. If the expression's result type is a slice type or **str**, then the array object referred to by its `data` field shall be freed. Otherwise, the expression's result value shall be **null**, and no additional side-effects shall occur.

6.7.21 Assertions

assertion-expression:
assert **(** *expression* **)**
assert **(** *expression* **,** *expression* **)**
abort **(** *expression*$_{opt}$ **)**

static-assertion-expression:
static *assertion-expression*

6.7.21.1 An assertion-expression is used to validate an assumption by the programmer by *asserting* its truth.

6.7.21.2 In the first two forms, the first expression shall be evaluated in the execution environment. If the expression evalutes to false, a diagnostic message shall be printed and the execution phase aborted. The expression shall have type **bool**, which shall be provided to it as a type hint. The result type of these forms is **void**.

6.7.21.3 In the second **assert** form, and in the **abort** form if present, the final expression shall have type **str**, which shall be provided to it as a type hint. The contents of

the string shall be included in the diagnostic message.

6.7.21.4 In the **abort** form, the execution environment shall unconditionally print a diagnostic message and abort. The result type of this form is **never**.

6.7.21.5 A static-assertion-expression is identical to an assertion-expression, except that the assertion is run in the translation environment rather than the execution environment, and the result type is always **void**.

6.7.22 Calls

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

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

6.7.22.1 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 «5.4.7.1: Program execution».

6.7.22.2 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.

6.7.22.3 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.

6.7.22.4 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.

6.7.22.5 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.

6.7.22.6 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.

6.7.22.7 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.

6.7.22.8 If the invoked function uses C-style variadism, the function may provide zero or more arguments following the final parameter.

6.7.22.9 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.

6.7.23 Measurements

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

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

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

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

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

Forward references: «6.7.24: Field access»

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

6.7.23.2 The **`align`** expression shall compute the *alignment* of the specified type. If type is a type of undefined alignment, a diagnostic message shall be printed and translation shall abort.

6.7.23.3 The **`size`** expression shall compute the *size* of the specified type. If type is a type of undefined size, a diagnostic message shall be printed and translation shall abort.

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

6.7.23.5 The object used for a length expression shall be the array, slice, or **`str`** object the expression refers to, selecting that object indirectly via any number of non-nullable pointer types if appropriate.

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

6.7.24 Field access

field-access-expression:
    postfix-expression **.** *name*
    postfix-expression **.** *integer-literal*

6.7.24.1 A field-access-expression is used to access fields of «6.6.14: Struct and union types» and «6.6.15: 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.

6.7.24.2 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.

6.7.24.3 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.

6.7.24.4 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.

6.7.24.5 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.

6.7.24.6 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.

6.7.25 Indexing

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

6.7.25.1 An indexing-expression shall access a specific value of a slice or array type. The expression shall have a result type of **`size`**, which shall be provided to it as a type hint. The result type of the postfix-expression shall be constrained to a set which includes all slice and array types whose secondary type has definite size, 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.

6.7.25.2 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.

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

6.7.25.4 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.

6.7.25.5 The execution environment shall compute the result of expression to obtain N for selecting the *N* per the algorithm given in «6.6.17: Slice and array types».

6.7.25.6 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 it is able, and print a diagnostic message and abort the translation environment if it fails.

6.7.26 Slicing

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

6.7.26.1 A slicing-expression shall have a result type of slice, which is computed a subset of a slice or array object. Each optional expressions shall, if present, have a result type of **`size`**, which shall be provided to it as a type hint. 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.

6.7.26.2 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.

6.7.26.3 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.

6.7.26.4 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.

6.7.26.5 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.

6.7.26.6 The execution environment shall perform a *bounds test* on the values of L and H to ensure they fall within the acceptable range of the given slice or array type. It shall test that $L < H + 1$. If postfix-expression isn't of an unbounded array type, it shall also test that $H <$ `length` $+ 1$. 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 it is able, and print a diagnostic message and abort the translation environment if it fails.

6.7.26.7 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.

6.7.27 Appending

slice-mutation-expression:
append-expression
insert-expression
delete-expression

append-expression:
static$_{opt}$ **append** **(** *object-selector* **,** *expression* **)**
static$_{opt}$ **append** **(** *object-selector* **,** *expression* **...** **)**
static$_{opt}$ **append** **(** *object-selector* **,** *expression* **,** *expression* **)**

6.7.27.1 An append-expression shall append some number of values to the slice object specified by the first term, which shall not be provided a type hint. Valid result types for object-selector shall either be a slice type whose secondary type has a definite size, or a non-nullable pointer to a valid result type. The selected object shall be mutable and non-const. The member type of this slice shall be the append-expression's *member type*. The result type of an append-expression shall be **void**.

6.7.27.2 In the first form, the type of the second term shall be assignable to the member type, which shall be provided to it as a type hint.

6.7.27.3 In the second form, the type of the second term shall be a non-expandable bounded array, a pointer to a non-expandable bounded array, or a slice. In all cases, the secondary type of the array or slice type shall be the append-expression's member type. The second term shall be provided the object type as a type hint.

6.7.27.4 In the third form, the second term shall be an expandable array whose member type is the append-expression's member type, and the third term shall be assignable to **size**. The second term shall be provided the object type as a type hint, and the third term shall be provided **size** as a type hint. The length of the expandable

array shall be given by the third term.

6.7.27.5 The second term of an **append** expression is used to obtain the *append values*, of which there may be none. In the first form, the value of the second term shall be used as the sole append value. In the second and third forms, the data of the array or slice given in the second term shall be used as the append values.

6.7.27.6 The second term shall be evaluated after all other terms. If present, the third term shall be evaluated after the object-selector.

6.7.27.7 The append values shall be obtained after the second term is evaluated. Once obtained, the append values shall not change as a result of any side-effects.

6.7.27.8 After the append values are obtained, the execution environment shall ensure that the capacity of the object is at least $N \geq L_1 + L_2$, where L_1 is the object length and L_2 is the number of append values, reallocating the underlying storage if necessary. If sufficient storage cannot be allocated, the execution environment shall print a diagnostic message and abort. The `length` field of the slice object shall be set to $L_1 + L_2$.

6.7.27.9 After ensuring sufficient space is available for the new items, the execution environment shall assign values of the slice object from index $N = L_1$ (inclusive) to $N = L_1 + L_2$ (exclusive) from each consecutive append value.

6.7.27.10 The **static** forms shall be equivalent to the non-static forms except that they shall never cause the underlying slice to be reallocated. If the operation would require more space than the capacity of the slice provides, the execution environment shall print a diagnostic message and abort.

6.7.28 Inserting

insert-expression:

static$_{opt}$ **insert** **(** *indexing-expression* **,** *expression* **)**
static$_{opt}$ **insert** **(** *indexing-expression* **,** *expression* **...** **)**
static$_{opt}$ **insert** **(** *indexing-expression* **,** *expression* **,** *expression* **)**

6.7.28.1 An insert-expression shall insert some number of values into the slice object specified by the first term, which shall not be provided a type hint. The indexing-expression specifies both the slice object and an index at which the new values shall be inserted (the *insertion index*). The selected object shall be mutable and non-const, and shall not have an array type. The secondary type of this slice shall be the expression's *member type*. The member type shall have a definite size. The result type of an insert-expression shall be **void**.

6.7.28.2 The bounds test described in «6.7.25: Indexing» shall be performed on the first term, except it shall instead test that $N < Z + 1$.

6.7.28.3 In the first form, the type of the second term shall be assignable to the member type, which shall be provided to it as a type hint.

6.7.28.4 In the second form, the type of the second term shall be a non-expandable bounded array, a pointer to a non-expandable bounded array, or a slice. In all cases, the

secondary type of the array or slice type shall be the insert-expression's member type. The second term shall be provided the object type as a type hint.

6.7.28.5 In the third form, the second term shall be an expandable array whose member type is the insert-expression's member type, and the third term shall be assignable to **`size`**. The second term shall be provided the object type as a type hint, and the third term shall be provided **`size`** as a type hint. The length of the expandable array shall be given by the third term.

6.7.28.6 The second term of an insert-expression is used to obtain the *insert values*, of which there may be none. In the first form, the value of the second term shall be used as the sole insert value. In the second and third forms, the data of the array or slice given in the second term shall be used as the insert values.

6.7.28.7 The second term shall be evaluated after all other terms. If present, the third term shall be evaluated after the bounds test on the first term is performed.

6.7.28.8 The insert values shall be obtained after the second term is evaluated. Once obtained, the insert values shall not change as a result of any side-effects.

6.7.28.9 After the insert values are obtained, the execution environment shall ensure that the capacity of the object is at least $N \geq L_1 + L_2$, where L_1 is the object length and L_2 is the number of insert values, reallocating the underlying storage if necessary. If sufficient storage cannot be allocated, the execution environment shall print a diagnostic message and abort. The `length` field of the slice object shall be set to $L_1 + L_2$.

6.7.28.10 After ensuring sufficient space is available for the new items, the execution environment shall copy or move each item N such that from $N = I$ (inclusive) to $N = I + L_2$ (exclusive), where I is the insertion index, each item is placed at $N + L_2$.

6.7.28.11 The execution environment shall then assign values of the object slice from index $N = I$ (inclusive) to $N = I + L_2$ (exclusive) from each consecutive insert value.

6.7.28.12 The **`static`** forms shall be equivalent to the non-static forms except that they shall never cause the underlying slice to be reallocated. If the operation would require more space than the capacity of the slice provides, the execution environment shall print a diagnostic message and abort.

6.7.29 Deleting

delete-expression:

`static`$_{opt}$ **`delete`** **`(`** *indexing-expression* **`)`**
`static`$_{opt}$ **`delete`** **`(`** *slicing-expression* **`)`**

6.7.29.1 A delete-expression shall remove some number of values from the specified slice object. The selected object shall be mutable and non-const, and shall not have an array type. The result type of a delete-expression shall be **`void`**.

6.7.29.2 Should a indexing-expression be used, it shall specify both the slice object and an index L, and H shall be set to $L + 1$. Should a slicing-expression be used, it shall

specify both the slice object and a range spanning from L to H. In either case, the appropriate bounds test shall be performed, as described in «6.7.25: Indexing» and «6.7.26: Slicing».

6.7.29.3 The execution environment shall then copy or move each item N in the slice such that from $N = H$ (inclusive) to $N = Z$ (exclusive), where Z is the value of the slice object's `length` field, each item is placed at $N + L - H$. It shall then subtract $H - L$ from the value of the slice object's `length` field.

6.7.29.4 If the **`static`** form is used, the slice object's capacity shall not be modified, and its data shall not be reallocated. Otherwise, the execution environment may decrease the slice object's capacity, re-allocating its data if necessary. This re-allocation, if performed, shall not fail. The slice object shall be updated to reflect the new capacity if necessary. If a non-**`static`** delete-expression sets the value of the slice object's `length` field to zero, the `capacity` field's value shall also be set to zero, and the array object referred to by its `data` field shall be freed.

6.7.30 Error propagation

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

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

6.7.30.2 The result type of an error propagation expression shall be the same as the type of the postfix-expression, without its error cases, or **`never`** 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, **`never`**. If the original type is not a tagged union type, the translation phase shall abort with a diagnostic message.

6.7.30.3 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.

6.7.30.4 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. This form shall not be used within the expression tree formed by a defer-expression.

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

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

6.7.31 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
    (*object-selector* **)**

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

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

6.7.32 Variadic expressions

variadic-expression:
    vastart ( )
    vaarg (*object-selector* **)**
    vaend (*object-selector* **)**

6.7.32.1 Variadic expressions are provided for compatibility with the C programming language as specified by ISO/IEC 9899. Implementation support is optional: implementations which do not provide C ABI compatibility must parse these expressions, print a diagnostic message, and abort.

These expressions are used only for C compatibility. "Hare-style" variadism is handled separately.

6.7.32.2 The **vastart** expression shall have a result type of **valist**, and may only be used within a function body which uses C-style variadism. It will initialize a **valist** in an implementation-defined manner, such that the first use of **vaarg** on the new object would return the first variadic parameter.

6.7.32.3 The **vaarg** expression accepts an object-selector which must be of the **valist** type. The expression shall also be provided a type hint, which shall have a defined size and alignment.

6.7.32.4 When the **vaarg** expression is evaluated, the following conditions (*runtime constraints*) must hold:

1. Any previous evaluation of a **vaarg** expression on the same object-selector didn't violate any

runtime constraints

2. There exists a variadic parameter that has yet to be consumed by the object-selector
3. The type of the next variadic parameter and the type hint have the same size and alignment
4. The type of the next variadic parameter is assignable to the type hint (ignoring type flags), or the type of the next variadic parameter and the type hint are both integer types, or the type of the next variadic parameter is either a pointer type or the null type and the type hint is a pointer type, or either the type of the next variadic parameter or the type hint has a size of zero

6.7.32.5 If no runtime constraints are violated, the result of the **`vaarg`** expression shall be the next variadic parameter from the **`valist`** object, and the parameter shall be consumed by the object such that another **`vaarg`** expression on the same object-selector would yield the successive variadic parameter, if one exists. If any runtime constraint is violated, behavior of the **`vaarg`** expression is undefined.

6.7.32.6 The **`vaend`** expression accepts an object-selector which must be of the **`valist`** type. The object must have previously been initialized with **`vastart`**. The implementation shall finalize the **`valist`** object in an implementation-defined manner. Any further evaluated **`vaarg`** expression on the object is invalid, unless the object is first re-initialized with **`vastart`**. The result type of this expression is **`void`**.

6.7.33 Builtin expressions

builtin-expression:
allocation-expression
assertion-expression
measurement-expression
slice-mutation-expression
postfix-expression
static-assertion-expression
variadic-expression

6.7.34 Unary arithmetic

unary-expression:
builtin-expression
compound-expression
match-expression
switch-expression
unary-operator *unary-expression*

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

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

6.7.34.2

6.7.34.3 The **-** operator shall perform a unary negation operation. If the type of unary-expression is a flexible integer, the result type shall be a flexible integer whose minimum value is the result of negating the operand type's maximum value, and whose maximum value is the result of negating the operand type's minimum value. If the resulting flexible integer type isn't representable, a diagnostic message shall be printed and the translation environment shall abort. Otherwise, if the type of unary-expression isn't a flexible integer, the result type shall be equivalent to the type of unary-expression, which shall be of a numeric type.

6.7.34.4 The **~** operator shall perform a bitwise 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 integer type.

6.7.34.5 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.

6.7.34.6 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*, and the pointer's secondary type shall have a defined size.

6.7.34.7 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
-	Negation
~	Bitwise NOT
!	Logical NOT
*****	Dereference pointer
&	Take address

6.7.35 Casts and type assertions

cast-expression:
unary-expression
cast-expression **:** *type*
cast-expression **as** *assertion-target*
cast-expression **is** *assertion-target*

assertion-target:
type
null

6.7.35.1 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**.

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

6.7.35.3 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.

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

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

Result	Castable from
Any numeric type	Floating types
Any numeric type or enum type	Integer and enum types
Any pointer type or the null type	**uintptr**
Any pointer type, **uintptr**, or the null type	Any pointer type
Any array or slice type	Array types
Any slice type or pointer to an array type	Slice types
Any type the underlying type of the source could cast to	Type aliases
Any type alias with an underlying type the source may be cast to	Any type
Any integer type	**rune**
rune	Any integer type
Any pointer type or **uintptr**	the null type
See below	Tagged unions
Tagged unions	See below
See below	Flexible types

6.7.35.6 Tagged union types are mutually castable with any type which is found among its members, including otherwise non-castable types and other tagged union types.

6.7.35.7 When a flexible type is cast to another type, the «6.11: Flexible type promotion algorithm» shall be applied to them, and the flexible type shall be castable to the other type if the promotion succeeds.

6.7.35.8 When an integer type is cast to an integer with a smaller width, it shall be truncated towards the least significant bit.

6.7.35.9 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.

6.7.35.10 When a floating type is cast to an integer, the resulting value shall be equal to the nearest integer with smaller or equal absolute value. If the destination type has insufficient width to represent the resulting integer, it shall be truncated towards the least significant bit.

6.7.35.11 When casting an integer to a floating point type, if the destination type has insufficient width to represent the value, it shall be truncated towards zero.

6.7.35.12 Casting a pointer type to **uintptr**, and then back to the pointer type, shall yield the same pointer. Likewise, casting the null type to a **uintptr** and then back to a nullable pointer type shall yield **null**.

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.*

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

6.7.35.14 The third form is a *type assertion*. In this form, cast-expression shall be of a tagged union type or a nullable pointer type.

In the former case, assertion-target shall be type and shall be one of constituent types of 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.

In the latter case assertion-target shall either be a type that is a nullable pointer type or **null**. If it is **null** and the value of cast-expression does not equal null, a diagnostic message shall be printed and execution environment shall abort. If assertion-target is not **null** and the value of cast-expression equals null, a diagnostic message shall be printed and the execution environment shall abort. The result type of a type assertion with **null** as assertion-target shall be the null type. The result type of other type assertions shall be type.

6.7.35.15 The fourth form is a *type test*. In this form, The result type is **bool**, and shall be **true** if and only if the type assertion from cast-expression to assertion-target would succeed, or **false** otherwise.

6.7.36 Multiplicative arithmetic

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

6.7.36.1 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 «6.8: Type promotion» rules.

6.7.36.2 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.

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

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

6.7.37 Additive arithmetic

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

6.7.37.1 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 «6.8: Type promotion» rules.

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

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

6.7.38 Bit shifting arithmetic

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

6.7.38.1 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.

6.7.38.2 `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.

6.7.38.3 `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.

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

6.7.39 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*

6.7.39.1 An and-expression performs a bitwise and operation.

6.7.39.2 An exclusive-or-expression performs a bitwise exclusive or operation.

6.7.39.3 An inclusive-or-expression performs a bitwise inclusive or operation.

6.7.39.4 The operands and result type shall be subject to the «6.8: Type promotion» rules.

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

6.7.40 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*

6.7.40.1 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 «6.8: Type promotion» rules. The result type shall be **bool**.

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

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

6.7.40.4 The result of the **<=** operator shall be **true** if the first operand is mathematically

less than or equal to second operand and **false** otherwise.

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

6.7.40.6 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 «6.8: 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.

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

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

6.7.40.9 The method in which two floating point values are compared or check for equality is implementation-defined. Two floating point values with the same representation may compare unequal.

6.7.40.10 Two **str** objects shall be equal if both strings have the same length and octets. Otherwise, they shall not be equal.

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

6.7.41 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*

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

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

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

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

6.7.41.5 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.

6.7.42 If expressions

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

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

6.7.42.1 An if-expression chooses which, if any, expression to evaluate based on a logical criteria. In conditional-branch, the result type of the first expression shall be **bool**.

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

6.7.42.3 In the second form, the conditional-branch shall be executed. If the condition is **false**, the expression (the *false branch*) shall be executed, ensuring that all side-effects occur. If the condition is **true**, the false branch shall not be executed and shall not cause side-effects. The result value shall be selected from the result of the branch which is executed. If the if-expression is provided a type hint, the executed branch shall receive it as a type hint.

6.7.42.4 An **else** is associated with the inner-most if-expression which is allowed by the syntax.

6.7.42.5 The first form shall behave semantically as though the second form were used with **void** as the false branch.

6.7.42.6 If a type hint is provided and all branches are assignable to that type, the result type shall be the type given by the type hint. Otherwise, the result type shall be the «6.10: Result type reduction algorithm» applied to the result types of all branches.

6.7.43 For loops

for-loop:
    for *label*$_{opt}$ **(** *for-predicate* **)** *expression*

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

label:
    : *name*

6.7.43.1 A for-loop executes its expression, the *body* of the loop, zero or more times, so long as a condition is true.

6.7.43.2 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 third form, the first expression is the condition, and the second expression is the *afterthought*. In the fourth 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.

6.7.43.3 If the condition uses the «6.9: Translation compatible expression subset», and its result value is **true**, and the scope of the for-loop is never terminated as a side-effect of a **break** or **yield** expression, the result type of the for-loop shall be **never**, otherwise, the result type shall be **void**.

6.7.43.4 The afterthought shall not have a result type whose error flag is set, nor shall it have a tagged union result type which has a type with the error flag set among its members.

6.7.43.5 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 evaluation is an *iteration*. When an iteration completes, the implementation shall evaluate the afterthought, if present, and then re-evaluate the condition and perform another iteration if it is **true**.

6.7.43.6 If a label is present, the established scope shall be labelled with its name.

6.7.44 Switch expressions

switch-expression:
    switch *label*$_{opt}$ **(** *expression* **)** **{** *switch-cases* **}**

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

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

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

Forward references: «6.7.49: Compound expressions»

6.7.44.1 A switch expression evaluates a value (expression, the *switching expression*), then compares it with a number of switch-cases, taking whichever branch compares equal to the value. The switching expression's result type must either be of an integer type or be a **str**, **bool**, **rune**, a flexible rune type, or a pointer type.

6.7.44.2 Each of the case-options specifies a value to compare with, given by expression. This expression shall be limited to the «6.9: Translation compatible expression subset», and its result type shall be assignable to the result type of the switching expression.

6.7.44.3 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 expression is with a yield-expression. The semantics of defer-expression, bindings, and so on, are also implicated.

6.7.44.4 If the switch-expression has a label, its name shall be used to label the scope of each case's implicitly introduced compound-expression.

6.7.44.5 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.

6.7.44.6 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 precisely exhaustive: no two cases shall select for the same value.

6.7.44.7 If the value of the switching expression doesn't compare equal to any of the switch-cases, a diagnostic message shall be printed and the execution environment shall abort.

This isn't possible under normal circumstances, but certain operations such as invalid casts can cause this to occur.

6.7.44.8 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.

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

6.7.44.10 If a type hint is provided and all branches are assignable to that type, the result type shall be the type given by the type hint. Otherwise, the result type shall be the «6.10: Result type reduction algorithm» applied to the result types of all branches.

6.7.45 Match expressions

match-expression:
match *label*$_{opt}$ **(** *expression* **)** **{** *match-cases* **}**

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

match-case:
case **let** *name* **:** *type* **=>** *expression-list*
case **let** **(** *binding-names* **)** **:** *type* **=>** *expression-list*
case *type* **=>** *expression-list*
case **=>** *expression-list*

Forward references: «6.7.49: Compound expressions», «6.7.47: Binding expressions»

6.7.45.1 A match expression 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.

6.7.45.2 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.

6.7.45.3 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.

6.7.45.4 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 expression to the result of the selected match-case.

As such, the appropriate way to set the result of a match expression is with a

yield-expression. The semantics of defer-expression, bindings, and so on, are also implicated.

6.7.45.5 If the match-expression has a label, its name shall be used to label the scope of each case's implicitly introduced compound-expression.

6.7.45.6 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.

6.7.45.7 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.

6.7.45.8 The second form of match-case is the *tuple unpacking* form, and in this case, the type shall be of a tuple type with a number of values equal to the number of items given in binding-names. The implementation shall cast the match expression to the selected type and create separate bindings for each name in binding-names, of the type of the corresponding tuple value, and initialize them to that value from the tuple. It shall insert this binding into the scope of the implicit compound-expression of the selected case. If any binding-name is _ (an underscore), a binding for that name shall not be created, but bindings to any names within the tuple shall be created.

6.7.45.9 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 precisely exhaustive: no two cases shall select for the same type.

6.7.45.10 If the type of the matching expression isn't accounted for by any of the match-cases, a diagnostic message shall be printed and the execution environment shall abort.

This isn't possible under normal circumstances, but certain operations such as invalid casts can cause this to occur.

6.7.45.11 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.

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

6.7.45.13 If a type hint is provided and all branches are assignable to that type, the result type shall be the type given by the type hint. Otherwise, the result type shall be the «6.10: Result type reduction algorithm» applied to the result types of all branches.

6.7.46 Assignment

assignment:
    assignment-target *assignment-op* *expression*

assignment-target:
    object-selector
    indirect-assignment-target
    slicing-assignment-target

indirect-assignment-target:
    ***** *unary-expression*
    (*indirect-assignment* **)**

slicing-assignment-target:
    slicing-expression
    (*slicing-assignment-target* **)**

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

Forward references: «6.7.47: Binding expressions»

6.7.46.1 An assignment expression shall cause the object given by assignment-target 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. The result type of an assignment shall be **void**.

6.7.46.2 If the assignment-op is **=**, the assignment-target 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.

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

6.7.46.4 In the indirect-assignment-target form, the unary-expression shall have a result 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.

6.7.46.5 In the slicing-assignment-target form, the expression shall be either of a slice type and have a length equal to the slice given by slicing-expression, or an expandable array. The assignment-op shall be **=**. 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 or expandable array given by the second term shall be copied into the slice given by the first term.

6.7.46.6 The type of the object being assigned to shall have definite size and alignment.

6.7.46.7 The second term shall be *assignable* to the object. Assignability rules are different 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 lower width
Unsigned integer types	Unsigned integer types of equal or lower width
Floating point types	Any floating point type of equal or lower width
Nullable pointer types	Non-nullable pointer type of the same secondary type
Nullable pointer types	the null type
Slice types	Array type of the same secondary type and definite length
Slice types	Pointer to array type of the same secondary type and definite length
`[]opaque`	Slice types
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
`*opaque`	Any non-nullable pointer type
`nullable*opaque`	Any pointer type
Pointers to array types	See notes
Any type	**`never`**

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

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

6.7.46.9 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.

6.7.46.10 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.

6.7.46.11 **`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.

6.7.46.12 If at least one of the types is flexible, the «6.11: Flexible type promotion algorithm» shall be applied to them, and they shall be considered mutually assignable if the promotion succeeds.

In the context of an assignment expression, «6.7.46.2: 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.

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

6.7.47 Binding expressions

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

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

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

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

binding-name:
    name
    _

6.7.47.1 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. The result type of a binding list expression is **void**.

6.7.47.2 A binding's expression shall be evaluated before any names are made available in the present scope.

Because the binding is only inserted into the scope when the binding-list is evaluated, it isn't visible to any expressions which lexically precede it, even if they reside in the same scope.

6.7.47.3 In the first form of binding, 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.

6.7.47.4 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 binding-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. If any binding-name is **_** (an underscore), a binding for that name shall not be created, but bindings to any names within the tuple shall be created and any side-effects of the initializer shall still occur.

6.7.47.5 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. In either case, the type of each binding shall have a definite size. If the type of a binding which uses the **`const`** or **`let`** form is a flexible type, it shall first be lowered to its default type.

6.7.47.6 If the **`static`** form is used, the objects 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 «6.9: Translation compatible expression subset».

6.7.47.7 If the **`def`** form is used, the objects shall not have any storage allocated for them in the execution environment. References to bindings created using this form shall be equivalent to references to the expression associated with them, with a cast to type inserted if the second form of binding is used. The tuple unpacking form of binding shall not be used with this form. The expression associated with each binding shall be limited to the «6.9: Translation compatible expression subset», and shall be evaluated in the translation environment.

6.7.47.8 If a given name already refers to an object visible in the present scope, the new binding shall *shadow* the earlier binding, causing any references to the name which lexically follow this binding-list within the present scope to resolve to the newly bound object.

6.7.48 Defer expressions

defer-expression:
`defer` *expression*

6.7.48.1 A defer-expression causes another expression to be *deferred* until the current scope *terminates*. The result type is **`void`**.

6.7.48.2 The expression shall not have a result type whose error flag is set, nor shall it have a tagged union result type which has a type with the error flag set among its members.

6.7.48.3 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.

6.7.48.4 Names and identifiers within the deferred expression are resolved during evaluation of the defer-expression.

Thus, a binding created after a defer-expression is evaluated can't be referenced within the deferred expression. This is also true of shadowed bindings.

6.7.48.5 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.

6.7.48.6 If a scope is terminated before a defer-expression within the scope (*but not an expression which was already deferred*) would be evaluated, the side-effects of the

expression shall not occur.

6.7.48.7 Before a scope terminates, all active nested scopes shall also be terminated, in order, from smallest region encompassed to largest.

6.7.48.8 Before a control-expression is evaluated (but after its provided expression is evaluated, if present), the scope associated with its selected expression is terminated.

Forward references: «6.7.50: Control expressions»

6.7.48.9 Before a function with a result type of **never** is called by a call-expression (but after all arguments have been evaluated), as well as before the execution environment aborts for any reason, the active function scope most recently made active is terminated.

6.7.48.10 As a special case, if a deferred expression is currently being executed, then if a scope outside of the inner-most deferred expression would terminate, the scope within the inner-most deferred expression which encompasses the largest region shall terminate instead.

This is intended to prevent exponential code generation when deferred expressions may abort.

6.7.49 Compound expressions

expression-list:
expression **;** *expression-list*$_{opt}$
binding-list **;** *expression-list*$_{opt}$
defer-expression **;** *expression-list*$_{opt}$

compound-expression:
label$_{opt}$ **{** *expression-list* **}**

6.7.49.1 A compound-expression evaluates a list of expressions in sequence.

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

6.7.49.3 The implementation shall establish a new scope for the compound-expression. The expressions shall then shall be evaluated such that the side-effects of each all occur in the order that each expression appears.

6.7.49.4 If a label is present, the established scope shall be labelled with its name.

6.7.49.5 None of the items in the expression-list shall have a result type whose error flag is set, nor shall any of the items have a tagged union result type which has a type with the error flag set among its member types.

6.7.49.6 Only the final item in the expression-list is permitted to have the result type **never**.

6.7.49.7 If the final item in the expression-list doesn't have the result type **never**, the compound-expression shall behave exactly as though `yield void;` were appended to the end of the expression-list.

The implicitly added yield-expression guarantees that all compound-expressions

will end with an expression whose result type is **never**.

6.7.49.8 The result type of a compound-expression shall be the «6.10: Result type reduction algorithm» applied to the result types of the expressions used in all all yield-expressions which select this compound-expression.

If no yield-expression selects this compound-expression (and a yield-expression wasn't implicitly appended to the expression-list), the result type is **never**.

6.7.50 Control expressions

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

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

6.7.50.1 A control-expression causes control to jump to another part of the program, possibly affecting the value and/or result of the *selected* expression or function. The result type is **never**.

6.7.50.2 The selected expression or function shall be selected from a pool which contains the current function, as well as all expressions within the current function with an associated scope which is active. Furthermore, if a label is provided, all expressions whose associated scope is unlabelled or whose associated scope's label doesn't match the provided label shall be excluded from the pool. The means by which an expression or function is selected from this pool varies based on the form of control-expression used. If no suitable expression or function can be selected, a diagnostic message shall be printed and the translation phase shall abort.

6.7.50.3 In the **break** and **continue** forms, if a label isn't provided, the selected expression shall be the for-loop whose associated scope encompasses the smallest region. If a label is provided, the selected expression shall be the expression whose scope encompasses the smallest region, and if this expression isn't a for-loop, a diagnostic message shall be printed and the translation phase shall abort. The **break** form shall cause the loop to end without evaluating the *condition* or the *afterthought*. The **continue** form shall cause the loop to repeat immediately, running the *afterthought* if present, re-testing the *condition*, and repeating the loop if **true**.

6.7.50.4 The **return** form shall select the current function. If an expression is given, its result shall be used as the result of the function, otherwise, **void** shall be used. The result type shall be assignable to the function's result type, which shall be provided to the expression as a type hint, if the expression is present.

6.7.50.5 In the yield-expression form, if a label isn't provided, the selected expression shall be the compound-expression whose associated scope encompasses the smallest region. If a label is provided, the selected expression shall be the expression whose scope encompasses the smallest region, and if this expression isn't a compound-expression, a diagnostic message shall be printed and the translation phase shall abort. A yield-expression that isn't provided an expression is equivalent to a yield-expression provided the expression **`void`**. The provided expression shall be used as the result value for the selected compound-expression.

A yield-expression may select the implicit compound-expression introduced by a switch-expression or match-expression.

6.7.50.6 If a control-expression is used within the expression tree formed by a defer-expression, and the selected expression isn't a descendent of the same tree, the translation environment shall print a diagnostic message and abort.

6.7.51 High-level expression class

expression:
assignment
logical-or-expression
if-expression
for-loop
control-expression

6.8 Type promotion

6.8.1 The operands of some arithmetic expressions are subject to *type promotion*, to allow for arithmetic between disjoint types. The operand with the smaller width may be *promoted*, or implicitly cast, to the width of the other 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.

6.8.2 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 largest width.

6.8.3 For expressions involving at least one flexible type, the result type shall be determined by the «6.11: Flexible type promotion algorithm».

6.8.4 For expressions involving two integer types, the type with smaller width may be promoted to the type with larger width only if the signedness is the same for each operand.

6.8.5 Expressions involving **`uintptr`** and **`size`** promote to **`uintptr`**, expressions involving **`uintptr`** and the null type promote to **`uintptr`**, and expressions involving **`uintptr`** and a pointer type promote to that pointer type. All other expressions involving **`uintptr`** shall cause the translation environment to print a diagnostic message and abort.

6.8.6 An integer type may be promoted to an enum type whose storage is the same as the integer's storage.

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

6.8.8 For expressions involving pointer types, the null type 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 an **`opaque`** pointer.

6.8.9 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.

6.8.10 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.

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

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

6.9 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.

6.9.1 The following expressions 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

6.9.2 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.

6.9.3 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.

6.9.4 The expression used for a length-expression is not required to be translation compatible as long as the result type is either an array type or an array type's alias, indirected via any number of non-nullable pointer types or their aliases if appropriate.

6.9.5 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.

6.9.6 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.

6.10 Result type reduction algorithm

6.10.1 The result type reduction algorithm shall operate on a list of types. It shall perform the following reductions in order:

1. Replace all tagged unions with their members
2. Remove all duplicate types
3. Remove all pointer types such that an equivalent nullable pointer type exists
4. Remove all mutable types such that an equivalent constant type exists
5. If the null type and exactly one pointer type remain, replace both with an equivalent nullable pointer type

6.10.2 If the null type and at least one other type remain after these reductions, the translation phase shall print a diagnostic message and abort.

6.10.3 If exactly one type remains, it shall be the result of the result type reduction algorithm. If more than one type remains, the result of the result type reduction algorithm shall be a tagged union containing the remaining types.

6.11 Flexible type promotion algorithm

6.11.1 The flexible type promotion algorithm shall operate on two types, at least one of which shall be a flexible type.

6.11.2 If both operands are flexible integers, they shall both be lowered to a flexible integer with the minimum value set to the smaller of their minimum values and the maximum value set to the larger of their maximum values, and that type shall be the result.

6.11.3 Otherwise, if both operands are flexible floats or both are flexible runes, they shall both be lowered to a new flexible float or flexible rune respectively, and that type shall be the result. If both operands are flexible types with different storages, the promotion shall fail.

6.11.4 If one operand is a tagged union type and promotion between exactly one of its members and the flexible type succeeds, the side-effects of that promotion shall occur and that member shall be the result of the promotion. Otherwise, if the flexible type's default type is a member of the tagged union, the flexible type shall be lowered to its default type and the promotion shall succeed. Otherwise, the promotion shall fail.

6.11.5 If one operand is a flexible float and the other is a non-flexible floating type, the flexible float shall be lowered to the other floating type, which shall be the result of the promotion. Otherwise, the promotion shall fail.

6.11.6 If one operand is a flexible integer and the other is a non-flexible signed integer type, the promotion shall succeed if both the minimum and maximum value fields of the flexible integer are within the limits of the other type. If one operand is a flexible integer and the other is a non-flexible unsigned integer type, the promotion shall succeed if the maximum value of the flexible integer is within the limits of the other type. If the promotion succeeds, the flexible integer shall be lowered to the other type, which shall be the result of the promotion.

6.11.7 If one operand is a flexible rune and the other is **`rune`**, the flexible rune shall be lowered into a **`rune`**. If one operand is a flexible rune and the other is an integer type, the promotion shall succeed if the flexible rune value can be represented as the specified integer type without any data loss.

6.12 Declarations

declarations:
    `export`$_{opt}$ *declaration* **`;`** *declarations*$_{opt}$
    static-assertion-expression **`;`** *declarations*$_{opt}$

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

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

6.12.2 If the **`export`** keyword is used, the declaration is considered to be *exported*, allowing other modules to access it. An exported declaration may be visible within the scopes of multiple translation units.

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

6.12.4 Global declarations

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

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

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

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

6.12.4.1 In a global-declaration, sufficient space shall be reserved for each global-binding 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.

6.12.4.2 The identifier of each global-binding shall be inserted into the current translation unit's scope, for use by any other declaration in the translation unit. If the identifier already refers to an object visible in the translation unit's scope, a diagnostic message shall be printed and the translation phase shall be aborted.

6.12.4.3 A global-binding's expression shall be limited to the «6.9: Translation compatible expression subset», and shall be evaluated in the translation environment. If specified, the type of the value of the expression shall be assignable to type. If not specified, the type of the global-binding shall be the result of the expression. Bindings whose type has undefined size shall not be provided an expression. If the type of a binding is a flexible type, it shall first be lowered to its default type.

6.12.4.4 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, the manner of which is implementation-defined.

6.12.4.5 The interpretation of the **@symbol** form of decl-attr is implementation-defined. **@symbol** shall not be used alongside **@init**, **@fini**, or **@test**.

The purpose of this directive is to allow users to customize the symbol name emitted for targets like ELF.

6.12.4.6 The interpretation of **@threadlocal** is implementation-defined.

The purpose of this directive is to store a separate copy of a global for each thread, similar to thread_local in C.

6.12.5 Constant declarations

constant-declaration:
    def *constant-bindings*

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

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

6.12.5.1 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. References to them shall be equivalent to references to the expression associated with them, with a cast to type inserted if the first form is used.

When the second form is used, expression may have a flexible result type.

6.12.5.2 The identifier of each constant-binding shall be inserted into the current translation unit's scope, for use by any other declaration in the translation unit. If the identifier already refers to an object visible in the translation unit's scope, a diagnostic message shall be printed and the translation phase shall be aborted.

6.12.5.3 A constant-binding's expression shall be limited to the «6.9: Translation compatible expression subset», and shall be evaluated in the translation environment. If the first form of constant-binding is given, the type of the value of the expression shall be assignable to type.

6.12.6 Type declarations

type-declaration:
    type *type-bindings*

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

type-binding:
    identifier **=** *type*
    identifier **=** *enum-type*

enum-type:
    enum *enum-storage*$_{opt}$ **{** *enum-values* **}**

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

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

enum-storage:
    integer-type
    rune

6.12.6.1 In a type-declaration, the identifiers shall declare type aliases. In the first form of type-binding, the underlying type for the identifier shall be the type. In the second form, the underlying type shall be enum-storage, if specified. Otherwise, the underlying type shall be **int**.

6.12.6.2 The identifier of each type-binding shall be inserted into the current translation unit's scope, for use by any other declaration in the translation unit. If the identifier already refers to an object visible in the translation unit's scope, a diagnostic message shall be printed and the translation phase shall be aborted.

6.12.6.3 In the second form of type-binding, each enum-value shall be made available to the translation unit's scope, via an identifier comprised of the components of the type alias' identifier followed by the enum-value's name. No storage shall be allocated for them in the execution environment. References to them shall be equivalent to references to the enum-value's assigned value.

6.12.6.4 Each enum-value is assigned the value of its expression, if present. Otherwise, the value assigned to it is equal to the value of the nearest lexically preceding enum-value of this enum type plus one. If no such previous value exists, zero is assigned.

6.12.6.5 If an implicitly assigned enum-value would not be representable in the underlying

integer type, a diagnostic message shall be shown as per «5.5: Diagnostics».

6.12.6.6 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 «6.7.46: Assignment»).

6.12.6.7 The implementation shall establish a new scope for the enum-values, and each enum-value name shall be made available in the scope.

This allows the expression for each value to refer to other values within the enum.

6.12.6.8 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.

6.12.6.9 expression shall be limited to the «6.9: Translation compatible expression subset».

6.12.7 Function declarations

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

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

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

6.12.7.1 The identifier of each function-declaration which does not use **@init**, **@fini**, or **@test** shall be inserted into the current translation unit's scope, for use by any other declaration in the translation unit. If the identifier already refers to an object visible in the translation unit's scope, a diagnostic message shall be printed and the translation phase shall be aborted.

6.12.7.2 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 programmer must arrange for the implementation of this function to be provided separately, the manner of which is implementation-defined. **@init**, **@fini**, and **@test** shall not be used on prototypes.

6.12.7.3 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.

6.12.7.4 In the second form of function-declaration, the implementation shall establish a new scope for the expression. For each of the prototype's parameters, in the order that they appear in the program source, the implementation shall resolve the parameter's type (within the newly established scope), and then, if the name form is used, insert the parameter's name into the scope. The prototype's result type shall be resolved outside of the newly established scope. The expression shall be translated after all named parameters have been made visible.

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

6.12.7.6 Functions declared with **`@test`**, **`@init`**, or **`@fini`** shall accept no parameters, shall return void, and need not have unique names.

6.13 Units

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

imports:
use-directive *imports*$_{opt}$

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

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

6.13.1 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 (excluding static assertions) among its sub-units.

6.13.2 Each translation unit shall establish a scope into which all of the unit's declarations are inserted. Each sub-unit shall also establish a scope, into which any imports shall make declarations from other modules 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.

6.13.3 A use-directive shall declare a dependency between this module and another module (the *target module*) whose namespace is specified by the use-directive's identifier. This shall cause the named module to be included in the final program, as described by «5.3: Trans-

lation steps». All dependencies of the target module also become dependencies of this module.

6.13.4 If a name or identifier which would be inserted into this sub-unit's scope by a use-directive already refers to an object visible in the scope, a diagnostic message shall be printed and the translation phase shall be aborted.

6.13.5 If a use-directive would cause a module to depend on itself, a diagnostic message shall be printed and the translation phase shall be aborted.

6.13.6 The first form of use-directive shall cause all declarations exported by the target module to be made available to this sub-unit's scope, via an identifier comprised of the last component of the namespace identifier followed by the components of the declaration's identifier.

Example *In the use directive* `use bar::baz;`*, identifiers in the module whose namespace is* `bar::baz` *will be made visible as identifiers whose first component is* `baz`*. For example, if* `bar::baz` *exports a declaration named* `bat`*, it is made visible to this sub-unit as* `baz::bat`.

6.13.7 The second form of use-directive shall cause all declarations exported by the target module to be made available to this sub-unit's scope, via an identifier comprised of the given name followed by the components of the declaration's identifier.

Example *In the use directive* `use foo = bar::baz;`*, identifiers in the module whose namespace is* `bar::baz` *will be made visible as identifiers whose first component is* `foo`*. For example, if* `bar::baz` *exports a declaration named* `bat`*, it is made visible to this sub-unit as* `foo::bat`.

6.13.8 The third form of use-directive shall cause only the declarations named in the member-list, each of which shall name a declaration exported by the target module, to be inserted into this sub-unit's scope (with the same name they were initially declared with). For each enum type alias inserted into the scope, all of the enum's members shall also be inserted, each with the same name that they have in the unit scope of the target module.

Example *If* `bar::baz` *exports a declaration named* `bat`*, the use directive* `use bar::baz::{bat}`*, will cause the name* `bat` *in this sub-unit's scope to refer to said declaration.*

6.13.9 The fourth form of use-directive shall cause all declarations exported by the target module to be made available to this sub-unit's scope, with the same names they were initially declared with.

A Language syntax summary

Lexical analysis

token:
    comment
    integer-literal
    floating-literal
    rune-literal
    string-section
    keyword
    name
    operator
    attribute
    invalid-attribute
    annotation

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

comment: exactly:
    `//` *comment-chars*

comment-chars: exactly:
    comment-char *comment-chars*$_{opt}$

comment-char:
    Any character other than `U+000A`

annotation:
    `#` `[` *identifier* *annotation-tokens*$_{opt}$ `]`

annotation-tokens:
    `(` ...balanced token list... `)`

Keywords

keyword: one of:

```
abort align alloc append as assert bool break case const continue
def defer delete else enum export f32 f64 false fn for free i16
i32 i64 i8 if insert int is len let match never null nullable
offset opaque return rune size static str struct switch true type
u16 u32 u64 u8 uint uintptr union use vaarg vaend valist vastart
void yield _
```

Attributes

attribute: one of:

```
@fini @init @offset @packed @symbol @test @threadlocal
```

invalid-attribute: exactly:

`@` *name*

Identifiers

identifier:

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
_
```

decimal-digit: one of:

```
0 1 2 3 4 5 6 7 8 9
```

alnum:

decimal-digit
nondigit

Types

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

storage-class:
    primitive-type
    pointer-type
    struct-union-type
    tuple-type
    tagged-union-type
    slice-array-type
    function-type
    alias-type
    unwrapped-alias

primitive-type:
    integer-type
    floating-type
    bool
    never
    opaque
    rune
    str
    valist
    void

integer-type: one of:
    i8 i16 i32 i64 u8 u16 u32 u64 int uint size uintptr

floating-type:
    f32
    f64

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

struct-union-type:
    struct @packed$_{opt}$ **{** *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* **)**

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

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

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

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

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

function-type:
    fn *prototype*

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

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

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

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

alias-type:
    identifier

unwrapped-alias:
    ... *identifier*

Expressions

literal:
    integer-literal
    floating-literal
    rune-literal
    string-literal
    array-literal
    struct-literal
    tuple-literal
    true
    false
    null
    void

floating-literal: exactly:

nonzero-decimal-digits **.** *decimal-digits* *decimal-exponent*$_{opt}$ *floating-suffix*$_{opt}$
nonzero-decimal-digits *decimal-exponent*$_{opt}$ *floating-suffix*
0x *hex-digits* **.** *hex-digits* *binary-exponent* *floating-suffix*$_{opt}$
0x *hex-digits* *binary-exponent* *floating-suffix*$_{opt}$

floating-suffix: one of:

f32 f64

decimal-digits: exactly:

decimal-digit *decimal-digits*$_{opt}$

nonzero-decimal-digits: exactly:

0
nonzero-decimal-digit *decimal-digits*$_{opt}$

nonzero-decimal-digit: one of:

1 2 3 4 5 6 7 8 9

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

decimal-exponent: exactly:

decimal-exponent-char *sign*$_{opt}$ *decimal-digits*

binary-exponent: exactly:

binary-exponent-char *sign*$_{opt}$ *decimal-digits*

sign: one of:

+ -

decimal-exponent-char: one of:

e E

binary-exponent-char: one of:

p P

integer-literal: exactly:

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

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`

positive-decimal-exponent:

decimal-exponent-char **`+`**$_{opt}$ *decimal-digits*

integer-suffix: one of:

`i u z i8 i16 i32 i64 u8 u16 u32 u64`

rune-literal: exactly:

`'` *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 \\ \' \"`

string-literal:
    string-section *string-literal*$_{opt}$

string-section: exactly:
    " *string-chars*$_{opt}$ **"**
    `** *rawstring-chars*$_{opt}$ **`

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

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

rawstring-chars: exactly:
    rawstring-char *rawstring-chars*$_{opt}$

rawstring-char:
    Any character other than **`**

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

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

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

struct-initializer:
    field-values **,**$_{opt}$
    field-values **,** **...**
    ...

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

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

tuple-literal:
    (*tuple-items* **)**

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

plain-expression:
    identifier
    literal

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

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

assertion-expression:
    assert **(** *expression* **)**
    assert **(** *expression* **,** *expression* **)**
    abort **(** *expression*$_{opt}$ **)**

static-assertion-expression:
    static *assertion-expression*

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

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

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

align-expression:
    align **(** *type* **)**

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

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

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

field-access-expression:
    postfix-expression **.** *name*
    postfix-expression **.** *integer-literal*

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

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

slice-mutation-expression:
    append-expression
    insert-expression
    delete-expression

append-expression:
    $\mathbf{static}_{opt}$ **append** **(** *object-selector* **,** *expression* **)**
    $\mathbf{static}_{opt}$ **append** **(** *object-selector* **,** *expression* **...** **)**
    $\mathbf{static}_{opt}$ **append** **(** *object-selector* **,** *expression* **,** *expression* **)**

insert-expression:
    $\mathbf{static}_{opt}$ **insert** **(** *indexing-expression* **,** *expression* **)**
    $\mathbf{static}_{opt}$ **insert** **(** *indexing-expression* **,** *expression* **...** **)**
    $\mathbf{static}_{opt}$ **insert** **(** *indexing-expression* **,** *expression* **,** *expression* **)**

delete-expression:
    $\mathbf{static}_{opt}$ **delete** **(** *indexing-expression* **)**
    $\mathbf{static}_{opt}$ **delete** **(** *slicing-expression* **)**

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

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

object-selector:
    identifier
    indexing-expression
    field-access-expression
    (*object-selector* **)**

variadic-expression:
    vastart ( )
    vaarg (*object-selector* **)**
    vaend (*object-selector* **)**

builtin-expression:
    allocation-expression
    assertion-expression
    measurement-expression
    slice-mutation-expression
    postfix-expression
    static-assertion-expression
    variadic-expression

unary-expression:
    builtin-expression
    compound-expression
    match-expression
    switch-expression
    unary-operator *unary-expression*

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

cast-expression:
    unary-expression
    cast-expression **:** *type*
    cast-expression **as** *assertion-target*
    cast-expression **is** *assertion-target*

assertion-target:
    type
    null

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

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

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

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*

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*

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*

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

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

for-loop:
    for $label_{opt}$ **(** *for-predicate* **)** *expression*

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

label:
    : *name*

switch-expression:
    switch $label_{opt}$ **(** *expression* **)** **{** *switch-cases* **}**

switch-cases:
    switch-case $switch\text{-}cases_{opt}$

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

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

match-expression:
    match *label*$_{opt}$ **(** *expression* **)** **{** *match-cases* **}**

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

match-case:
    case let *name* **:** *type* **=>** *expression-list*
    case let (*binding-names* **) :** *type* **=>** *expression-list*
    case *type* **=>** *expression-list*
    case => *expression-list*

assignment:
    assignment-target *assignment-op* *expression*

assignment-target:
    object-selector
    indirect-assignment-target
    slicing-assignment-target

indirect-assignment-target:
    ***** *unary-expression*
    (*indirect-assignment* **)**

slicing-assignment-target:
    slicing-expression
    (*slicing-assignment-target* **)**

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

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

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

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

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

binding-name:
    name
    _

defer-expression:
    defer *expression*

expression-list:
    expression **;** *expression-list*$_{opt}$
    binding-list **;** *expression-list*$_{opt}$
    defer-expression **;** *expression-list*$_{opt}$

compound-expression:
    label$_{opt}$ **{** *expression-list* **}**

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

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

expression:
    assignment
    logical-or-expression
    if-expression
    for-loop
    control-expression

Declarations

declarations:
    export$_{opt}$ *declaration* **;** *declarations*$_{opt}$
    static-assertion-expression **;** *declarations*$_{opt}$

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

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

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

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

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

constant-declaration:
    def *constant-bindings*

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

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

type-declaration:
    type *type-bindings*

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

type-binding:
    identifier **=** *type*
    identifier **=** *enum-type*

enum-type:
    enum *enum-storage*$_{opt}$ **{** *enum-values* **}**

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

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

enum-storage:
    integer-type
    rune

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

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

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

Units

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

imports:
    use-directive *imports*$_{opt}$

use-directive:
    use *identifier* **;**
    use *name* **=** *identifier* **;**
    use *identifier* **::** **{** *member-list* **}** **;**
    use *identifier* **::** ***** **;**

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