Spawn Language Specification

Petr Makhnev

22.05.2024

Contents

Chapter 1

Introduction

Spawn takes its inspiration from many languages, some of them being Golang, Rust, Swift, and Kotlin. The language is built on the idea that the language should be practical, convenient for writing a large amount of code, be clear to read, and have a wide variety of tools to make development easier.

Spawn is a statically typed, general-purpose compiled language. The language uses Option/Result types for error handling, has design-safe references, immutability by default, and flow-typing.

Spawn is not an object-oriented language, instead it uses structures and interfaces to describe data and behavior. Some parts of the language can be considered object-oriented, for example, inheritance through composition. The language also has many functional programming elements, for example, first-class functions, closures, lambda literals, and so on.

The language currently supports three major operating systems (Windows, Linux, and macOS) with two architectures (x86_64 and aarch64).

1.1 Acknowledgments

We would like to thank Pandoc, its authors, and community, as this specification would be much harder to implement without Pandoc's versatility and support.

1.2 Feedback

If you have any feedback for this document, feel free to create an issue at our GitHub.

Chapter 2

Conformance

“Shall” is used to indicate requirements imposed on the implementation or program. “Shall not” is interpreted as a prohibition.

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

Chapter 3

Lexical elements

The initial elements of a program into which the source code is divided are called tokens. Tokens are indivisible minimal parts into which a program can be divided. The source code of programs in the language is encoded in UTF-8 encoding. If the source code cannot be decoded correctly, compilation ends with an error (**TODO**: which one?).

The language distinguishes between uppercase and lowercase letters.

In the source code, the presence of NULL character (U+0000) is implementation-dependent. It is recommended not to use it for stability and readability. If a character is needed inside a string literal, `\0` should be used instead.

3.1 Whitespaces and Comments

Whitespaces consisting of spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A) can be used for two purposes. Whitespaces separates tokens and `\n` can trigger insertion of semicolon (`;`). In other cases, they are ignored.

The language supports two types of comments, line and block. Comments are treated as whitespaces by the compiler.

Line comments are two consecutive slashes (`//`) without a whitespace. Everything after them until the end of the line is treated as comment content:

```
// comment text
// line 2
```

Block comments are represented by a slash (`/`) followed by a `*` and ending with a `*` followed by a slash (`/`). Such comments can occupy either one line or several:

```
/*
```

```
Block comment
with several lines
*/
```

Block comments can be nested as long as the balance of `/*` and `*/` is maintained.

Inside an inline comment, `/*` and `*/` have no special meaning, just as `//` does not have much meaning inside a block comment.

A comment cannot start inside a rune or string literal.

Unlike some languages, documentation uses inline comments, which may contain additional formatting and markup. See documenting code.

3.2 Tokens

There are several types of tokens: identifiers, keywords, literals, and punctuation. The comments described above are not tokens but act as delimiters for tokens.

3.3 Semicolons

In language, semicolons are optional in most cases and can be omitted. The language uses the following rules to automatically insert `;` where needed:

1. During lexical analysis, when source code is tokenized, `;` is automatically inserted when `\n` is encountered in the following places:
   - after an identifier
   - after a literal
   - after keywords: `return`, `break`, `continue`
   - after the following punctuation characters: `)`, `}`, `]`, `+`, `-`, `!`, `?`
2. To allow complex statements to occupy a single line, the semicolon can be omitted before a closing `)` and `}`.

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

3.4 Identifiers

Identifiers name program units such as variables, functions, or constants. An identifier is a set of one or more letters or digits. The first character of an identifier must be a letter.

```
identifier = letter (letter | digit)*
```

Unlike other languages, using keywords via escaping or prefixing is not supported.

Identifiers ending with two underscores (`__`) are considered reserved and cannot be used in user code.

3.5 Keywords

The following identifiers have special meaning and cannot be used as regular identifiers. Their spelling should be exactly as described below:

(**TODO**: finish + sort)

```
if
else
pub
fn
struct
interface
union
return
where
nil
unsafe
as
is
none
for
in
break
continue
comptime
or
defer
type
var
enum
module
import
const
match
select
mut
goto
spawn
chan
test
assert
extern
```

3.6 Soft keywords

Unlike regular keywords, soft keywords are interpreted as keywords only in certain contexts and can be used as regular identifiers in others. The following identifiers are considered soft keywords: (**TODO**: finish + sort)

```
map
chan
self
```

3.7 Integer literals

Integer literals describe an integer value in decimal notation. Other bases can be specified through certain prefixes. Binary integers begin with `0b` (`0B`), octal with `0o` (`0O`) and hexadecimal with `0x` (`0X`). Binary integers contain digits 1 or 0, octal integers contain digits 0 to 7, hexadecimal integers contain digits 0 to 9, and the letters A to F in any case.

```
decimal_digit = "0" ... "9"
binary_digit  = "0" | "1"
octal_digit   = "0" ... "7"
hex_digit     = "0" ... "9" | "A" ... "F" | "a" ... "f"

int_lit        = decimal_lit | binary_lit | octal_lit | hex_lit
decimal_lit    = "0" | ( "0" ... "9" ) ( "_"? decimal_digits )?
binary_lit     = "0" ( "b" | "B" ) "_"? binary_digits
octal_lit      = "0" ( "o" | "O" ) "_"? octal_digits
hex_lit        = "0" ( "x" | "X" ) "_"? hex_digits

decimal_digits = decimal_digit ( "_"? decimal_digit )*
binary_digits  = binary_digit ( "_"? binary_digit )*
octal_digits   = octal_digit ( "_"? octal_digit )*
hex_digits     = hex_digit ( "_"? hex_digit )*
```

Examples of valid integer literals:

```
0 100 2147483647 0b000_111 0o1564 0o100_100 0xFFFF_FFFF
```

A single 0 is considered a decimal zero.

Underscores (`_`) can be used as a number separator to improve readability; they are ignored and do not change the actual value of the number. Underscores may be used between two consecutive digits (or letters in the case of hexadecimal numbers), but the following are prohibited:

```
100__000 // error: successive underscores
100_     // error: number ends with an underscore
0_b1000  // error: underscore must be between digits
_100000  // treated as an identifier
```

> Note that the `0b_100` case is allowed because `0b` is treated as a number in this case.

Numbers starting with 0 are treated as regular decimal numbers with all leading zeros removed, unlike, for example, C or Go, where such numbers are treated as octal numbers.

Negative literals are expressed using a `-` sign before the number, for example, `-42`.

By default, during type inference, the type of numeric literal is inferred as `i32`; if the number overflows `i32`, then the compiler should give an error about the need to explicitly cast the literal to a wider type to avoid overflowing.

List of diagnostics:

- `E0154`: Integer literal too big to fit in `i32` type

3.8 Floating-point literals

A floating-point literal describes the decimal or hexadecimal representation of a floating-point value with unspecified precision. By default, such literals describe a value in decimal notation (without a prefix), but they can also represent hexadecimal values (with a `0x` prefix).

```
float_lit         = decimal_float_lit | hex_float_lit

decimal_float_lit = decimal_digits "." decimal_digits? decimal_exponent? |
                    decimal_digits decimal_exponent |
                    "." decimal_digits decimal_exponent?
decimal_exponent  = ( "e" | "E" ) ( "+" | "-" )? decimal_digits

hex_float_lit     = "0" ( "x" | "X" ) hex_mantissa hex_exponent
hex_mantissa      = "_"? hex_digits "." hex_digits? |
                    "_"? hex_digits |
                    "." hex_digits
hex_exponent      = ( "p" | "P" ) ( "+" | "-" )? decimal_digits
```

Decimal literals consist of an integer part (decimal digits), a decimal point, a real part (decimal digits), and an exponent (`e` or `E`) followed by an optional sign and decimal digits. The exponent describes the power to which 10 is raised. For example, 1.25e2 describes the number 1.25 * 10^2, which equals 125, and 1.25e-2 describes 1.25 * 10^-2, which equals 0.0125.

One of the integer parts may be omitted; for example, `0.` and `.5` are valid real literals. If there is an exponent, the dot can also be omitted (for example, `1e2`).

Hexadecimal literals begin with the prefix `0x` or `0X` followed by an integer part (hex digits), a dot, a real part (hex digits) and an exponent (`p` or `P`) followed by an optional sign and decimal digits. The exponent describes the power to which 2 is raised. For example, 0xFp2 describes 15 * 2^2, which equals 60, and 0xFp-2 describes 15 * 2^-2, which equals 3.75.

As with decimal literals, one of the integer parts may be omitted. If there is an exponent, the point can also be omitted.

Underscores (_) can be used as a number separator to improve readability; they are ignored and do not change the actual value of the number. Separators can be used after the base prefix and between two consecutive numbers (or letters in the case of hexadecimal numbers).

Negative literals are expressed using a `-` sign before the number, for example, `-3.14`.

Numbers starting with 0 are treated as regular decimal numbers; leading zeros have no effect on the value or number system of the number.

By default, during type inference, the type of real literal is inferred as `f64`.

3.9 Boolean literals

Boolean literals are literals of a boolean type. The language defines two boolean literals `true` and `false`, `true` describes truth value and `false` describes falsehood.

```
bool_lit = "true" | "false"
```

3.10 Strings literals

String literals are a set of characters enclosed in quotation marks. String literals can use either single or double quotes, the following lines are identical.

```
'Hello'
"World"
```

String literals are encoded in UTF-8.

Internally, strings cannot contain an unescaped `"` or `'` depending on the quotes used, and cannot contain an unescaped backslash not before a line break (`\n`). String literals can be multiline. The choice of quotes depends on the presence of other quotes in the string, so if the string contains `"`, then `'` can be used as a quote, and vice versa. The following strings show the possible options:

```
'Hello, "BMW"'
"Hello, 'BMW'"
'Hello, \'BMW\''
"Hello, \"BMW\""
```

Choosing the right quotation marks eliminates the need to escape the quotation mark in a string.

Strings can use `\` at the end of the line before a line break (`\n`), in this case line breaks after `\` will be ignored, and any whitespace at the beginning of the next line will be stripped:

```
println('This string will not include \
      backslashes or newline characters.')
// This string will not include backslashes or newline characters.
```

TODO: escaped UTF-8 characters

3.10.1 Raw strings

String literals prefixed with `r` are considered raw. Raw strings inherit the properties described above, however, backslashes within such a string have no special meaning and are processed as is, for example, `\n` inside a string will be interpreted as `\\n`:

```
r'Hello, \n World' == 'Hello, \\n World'
```

Escaping quotes within a string is still necessary, however:

```
println(r"Hello, \n \"BMW\"")
// Hello, \n "BMW"
```

TODO: Shouldn't we disable this or introduce a new syntax like in C# (`""` two quotes in a row) for escaping? TODO: What do we do with ¨inside a string?

Interpolation in raw lines is processed as is:

```
println(r'Hello, ${world}')
// Hello, ${world}
```

3.10.2 C strings

String literals prefixed with `c` define a C string. C strings represent a null-terminated pointer to an array of characters. Such string literals are of type `&u8` and act like a regular reference. C strings are used for C interop and can be safely passed to C functions that accept a const pointer to the string

(**TODO**: what if the pointer is not constant?).

```
libc.printf(c'Hello, %s', c'World')
```

Interpolation in C string literals is processed as is, like in raw strings:

A C string can be obtained from a regular string by calling the `c_str()` method:

```
str := 'Hello, %s'
libc.printf(str.c_str(), 'World'.c_str())
```

3.10.3 Interpolation

String interpolation is a way to insert expression values inside a string. String interpolation begins with `${` followed by an expression that will be evaluated and inserted into the string, and ends with `}`.

When performing interpolation, all expressions are converted to a string type via calling the `str()` method.

In interpolation can be used any expression.

Example of valid interpolation:

```
world := 'Earth'
println('Hello, ${world}') // Hello, Earth
println('Hello, ${person.name}') // Hello, John
```

3.11 Nil literal

The `nil` keyword describes a null pointer, which represents the absence of a pointer value. A null pointer is of type `*T`, where `T` is the type inferred from the context or from an explicit cast (`nil as *u8`).

3.12 Rune literals

Rune literals are a single character or escaped sequence enclosed in backticks (‘), representing an integer value representation of the Unicode codepoint. Quotes can contain any characters except line breaks or unescaped backticks.

A single character represents the Unicode character itself, multiple characters represent an escape sequence that encodes some Unicode characters.

Since Spawn supports UTF-8 out of the box, rune literals can directly use any UTF-8 characters, even if they occupy several bytes.

With escaped sequences, character can be specified in several ways: `\x` followed by exactly two hex digits, `\u` followed by exactly four hex digits, `\U` followed by exactly eight hex digits, and `\` followed by exactly three octal digits.

Each of these methods has its own valid segments. Octal escapes can represent values from 0 to 255, inclusive. Hexadecimal escapes follow this limitation by design. Escaping `\u` and `\U` representing Unicode codepoints above `0x10FFFF` or surrogate pairs (`0xD800` – `0xDFFF`) are considered invalid.

The following escapes have special meaning:

```
\a   U+0007 alert or bell
\b   U+0008 backspace
\f   U+000C form feed
\n   U+000A line feed or newline
```

```
\r   U+000D carriage return
\t   U+0009 horizontal tab
\v   U+000B vertical tab
\\   U+005C backslash
\`   U+0060 backtick  (valid escape only within rune literals)
```

Other escapes are considered invalid.

Rune literals use a signed 32-bit integer type.

```
`a`
`\`` // rune literal with single backtick
` `
` `
` `
`\n`
`\t`
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
`\'`         // illegal: unknown escape sequence
`\"`         // illegal: unknown escape sequence
`12`         // illegal: rune literal cannot contain more than one character
`\xa`        // illegal: too few hexadecimal digits
`\0`         // illegal: too few octal digits
`\400`       // illegal: octal value over 255
`\uDFFF`     // illegal: surrogate half
`\U00110000` // illegal: invalid Unicode code point
```

3.13 Byte literal

Byte literals use the same syntax as rune literals, but are prefixed with a `b` before the opening backtick (b‘a‘). Unlike rune literals, byte literals can only represent ASCII characters in the range 0 to 127, inclusive.

Byte literals support the following escapes: `\x` followed by exactly two hexadecimal digits and `\` followed by exactly three octal digits. `\u` and `\U` escape forms are not allowed. Octal and hexadecimal escapes can represent values from 0 to 127, inclusive. Escaping with special values described in rune literals can also be used in byte literals.

Byte literals use an unsigned 8-bit integer type (`u8`).

```
b`a`
```

```
b`\`` // byte literal with single backtick
b`\070`
b`\x0F`
b`\t`
b`\n`
b`\r`
b` `          invalid: byte literal cannot contain non-ASCII character
b` `          invalid: byte literal cannot contain non-ASCII character
b`\300`        invalid: octal value over 127
b`\xFF`        invalid: hex value over 127
b`\u1000`      invalid: byte literal cannot contain escape sequence with Unicode code
b`\U00010000`  invalid: byte literal cannot contain escape sequence with Unicode code
b`\x0F\x0F`    invalid: byte literal cannot contain more than one character
```

3.14 Code documentation

TODO: describe how to document code

Chapter 4

Type system

Spawn has a type system with the following basic properties:

- Static and flow type checking
- No unsafe implicit type conversions

Since the language is statically typed, type safety is checked at compile time for most types. In some cases, under the influence of the control flow graph, the type of the value can change, which is usually called flow typing, represented in Spawn as smart casts.

Implicit type casts are limited to safe upcasts that cannot cause data loss. Any other type casts must be explicit.

4.1 Type kinds

Spawn has the following type kinds:

- Primitive types
- Pointer types
- Function types
- String types
- Array types
- Map types
- Tuple types
- Unit types
- Channel types
- Never type

4.1.1 Primitive types

4.1.1.1 Boolean type

Boolean (or `bool`) type is a primitive data type that can only have two values: `true` and `false`:

```
a := true
b := false
```

`true` and `false` are special constants with the values `1` and `0` respectively.

4.1.1.1.1 ABI Boolean type has an identical memory representation to the `bool` type in C, which on major platforms will have a size of one byte.

4.1.1.2 Numeric types

Numeric types represent integers and real numbers.

The language defines the following architecture-independent signed integer types:

- `i8`: signed 8-bit integer type (**-128** -(2^7) to **127** (2^7 - 1)
- `i16`: signed 16-bit integer type (**-32768** -(2^15) to **32767** (2^15 - 1)
- `i32`: signed 32-bit integer type (**-2147483648** -(2^31) to **2147483647** (2^31 - 1)
- `i64`: signed 64-bit integer type (**-9223372036854775808** -(2^63) to **9223372036854775807** (2^63 - 1)
- `i128`: signed 128-bit integer type (**-170141183460469231731687303715884105728** -(2^127) to **170141183460469231731687303715884105727** (2^127 - 1)

The language defines the following architecture-independent unsigned integer types:

- `u8`: unsigned 8-bit integer type (**0** to **255** (2^8 - 1)
- `u16`: unsigned 16-bit integer type (**0** to **65535** (2^16 - 1)
- `u32`: unsigned 32-bit integer type (**0** to **4294967295** (2^32 - 1)
- `u64`: unsigned 64-bit integer type (**0** to **18446744073709551615** (2^64 - 1)
- `u128`: unsigned 128-bit integer type (**0** to **340282366920938463463374607431768211455** (2^128 - 1)

An n-bit integer type value has a width of n bits and is represented using two's complement arithmetic.

The language defines the following architecture-independent real types:

- `f32`: 32-bit floating point real type IEEE 754-2008
- `f64`: 64-bit floating point real type IEEE 754-2008

Real types are floating-point numbers that are represented using the IEEE 754-2008 format.

The language also defines the following platform-specific integer types:

- `usize`: an unsigned integer type that can represent a pointer to any object in memory. The size of the type depends on the processor architecture; on 32-bit systems it is 32 bits, on 64-bit systems it is 64 bits.
- `isize`: signed integer type with the same size as `usize`. `isize` is large enough to represent the difference between the two pointers.

Although `isize` and `usize` may be equivalent to the `i64` and `u64` types on certain architectures, they are not synonymous and cannot be used interchangeably. To use it, a type shall be explicitly cast.

4.1.2 Pointer types

Pointer types are divided into two parts, references and raw pointers.

4.1.2.1 References

```
ReferenceType = "&" "mut"? Type
```

References are a data type that stores a valid non-null reference to a value of the type specified by `Type`.

By default, all references are immutable, which means that multiple references to the same value can exist at the same time, but none of them can change that value.

To specify a mutable reference, `mut` keyword is used. The language does not impose restrictions on the number of references to the same value that can be changed.

References can be created using the `&` operator. To get a mutable reference, the `&mut` operator is used. To obtain a mutable reference, the variable from which it is taken must be mutable.

```
a := 1
ptr := &a // immutable reference

b := 2
ref := &mut b // error, cannot take mutable reference to immutable variable

mut c := 3
ref := &mut c // mutable pointer
```

Copying a reference does not copy the value it refers to but creates another reference to this value.

4.1.2.1.1 Default value

References have no value by default since they cannot be null. Fields of structures with a reference type are always required when creating a structure via a literal. The compiler should throw an error if such a field is omitted and does not have a default value.

4.1.2.2 Getting data from a reference

To obtain data from a reference, the `*` dereferencing operator is used:

```
a := 1
ref := &a
b := *ref // b == 1
```

The language by design guarantees that dereferencing is safe and cannot cause a panic or segmentation fault.

4.1.2.3 ABI

References have the same memory representation as C pointers, allowing them to be safely passed to C functions when needed.

4.1.2.4 Raw pointers

```
RawPointerType = "*" "mut"? Type
NilLiteral     = "nil"
```

Raw pointers are a data type that stores a pointer to a value of the type specified by `Type`. Unlike references, raw pointers can be null, point to invalid memory, and be misaligned. Their main purpose is to work with C code. They can also be used for low-level optimizations within the program.

Copying a pointer does not copy the value it refers to but creates another pointer to that value.

4.1.2.4.1 nil literal

For a null pointer, the language defines a special `nil` literal. It is completely equivalent to a null pointer in C.

Although the `nil` literal is inherently unsafe, its use does not require the presence of an `unsafe` block, since it can be inferred from the context that the function being called is either unsafe (and its call is already in an `unsafe` block), or works with C code.

Although raw pointers can store a null pointer, assigning `0` to an unsafe pointer is prohibited.

A `nil` literal cannot be assigned directly if its type is unknown, for example, when defining a variable:

```
a := nil
```

In this case, `nil` shall be explicitly cast to a concrete raw pointer type:

```
a := nil as *i32
```

List of diagnostics:

- `E0178`: Cannot assign bare `nil` to variable, cast it to a concrete unsafe pointer type first

4.1.2.4.2 Default value

The default value for a raw pointer is the null pointer `nil`.

4.1.2.4.3 Retrieving data from a pointer

Unlike references, the language cannot guarantee that pointer dereferences are safe, so any pointer dereferencing must be wrapped in an `unsafe` block:

```
a := 1
ptr := &a as *i32     // some unsafe-like C code
b1 := *ptr            // error, cannot dereference pointer outside unsafe block
b2 := unsafe { *ptr } // b2 == 1
```

If a raw pointer is invalid, dereferencing it will cause a panic.

4.1.2.5 ABI

Pointers have identical memory representations to C pointers, allowing them to be safely passed to C functions when needed.

4.1.2.5.1 Pointer arithmetic

Pointer arithmetic is only possible with raw pointers; to use it on references, it shall be explicitly cast to a raw pointer (`&a as *mut u8`).

```
a := 1
ref := mem.calloc(2, mem.size_of[i32]()) as *i32
second_element_ptr := ref + 1
```

Pointer arithmetic is defined only for array-like pointers, that is, obtaining a pointer to some element of an array through pointer arithmetic at its beginning or another valid pointer to this array. If the resulting pointer is outside the bounds of the array, then dereferencing it will lead to undefined behavior.

```
arr := [1, 2, 3]
ptr := arr.raw() // pointer to the first element of the array
ptr = ptr + 1    // pointer to the second element of the array
ptr = ptr + 10   // pointer to the eleventh element of the array, still safe
*ptr             // undefined behavior, pointer is out of bounds
```

4.1.2.5.2 Reference and pointer conversion from one to another

Converting a reference to a raw pointer is trivial and does not require an `unsafe` block. For conversion, the `as` operator is used, on the right is written the type of pointer into which the reference must be converted:

```
a := 1
ref := &a
ptr := ref as *i32
```

Converting a pointer to a reference requires more care. We cannot check that a pointer points to valid memory, only that it has the correct alignment and is not null.

For these checks, the standard library provides a function `mem.assume_safe()`, which converts a pointer passed to it into a reference, checking that it is not null and has the correct alignment. If the pointer is null or has incorrect alignment, the function panics.

Since this function is marked `unsafe`, it must be called in an `unsafe` block:

```
ret_ptr := nil as *i32
libc.get_c_data(ret_ptr)
ret_ref := unsafe { mem.assume_safe(ret_ptr) } // ret_ref has type `&i32`
```

4.1.3 Function types

A function type defines the set of all functions that have the same parameter and return type. A function type has no default value and must always be initialized correctly.

```
FunctionType = 'fn' Signature
```

Function types inherit all the rules of function signatures – for example, the last argument may be a type preceded by `...`, which defines a variadic parameter that can take zero or more arguments.

4.1.4 String types

The string type represents a set of string values. The value of a string is a potentially empty sequence of bytes. The number of bytes is called the string length and is always greater than or equal to zero. Strings are immutable; once created, their contents cannot be changed.

The length of the string in the case of a regular and raw string can be obtained through the `len()` method. Specific bytes of a string can be obtained through square brackets at indexes from 0 to `str.len() - 1`. Access beyond these limits will result in a panic, except in compilation mode when safety checks are disabled.

4.1.4.1 C-string type

C strings are compile-time known strings represented by string literals with the prefix `c`:

```
c"Hello World"
```

Such strings have an ABI that fully corresponds to strings in the C language, including the null terminator at the end of the string. Such strings are of type `*u8` and behave like regular pointers.

4.1.5 Array types

Arrays are divided into two kinds, dynamic and fixed. Dynamic arrays can grow, that is, increase their size if necessary. Fixed arrays have a length known at compile time and cannot grow. For both types, the length determines the number of elements in the array, which cannot be less than zero.

4.1.5.1 Fixed array

```
FixedSizeArrayType = '[' Expression ']' Type
```

The length of a fixed array is part of the type and must be known at compile time and greater than zero. The length of an array can be obtained through the `len()` method. Retrieving elements is done via `arr[<index>]` where `index` is an expression with a value from 0 to `arr.len() - 1`. Access beyond these limits will result in a panic, except in compiler mode when safety checks are disabled.

A fixed-size array can be created in two ways:

- Via array literal which is explicitly converted to a fixed-size array type
- Via composite literal for a fixed-size array

```
a1 := [1, 2, 3] as [3]i32
a2 := [1 as u8, 2, 3] as [3]u8
a3 := [true, false, true] as [3]bool
a4 := [5]i32{}
a5 := [10]&Foo{init: || &Foo{}}
```

Fixed arrays can be nested, thereby forming an array with multiple dimensions:

```
const N = 10

[8][8]u8          // 8x8 array of u8
[1][2][3]i32      // 1x2x3 array of i32
[N * 2][N * 3]u64 // 20x30 array of u64
```

4.1.5.2 Dynamic array

```
ArrayType = '[' ']' Type
```

Dynamic arrays, unlike fixed ones, can change their size. As with fixed arrays, the `len()` method is used to get the length. The length of a dynamic array is always greater than or equal to zero. Retrieving elements is done via `arr[<index>]` where `index` is an expression with a value from 0 to `arr.len()`

`- 1`. Access beyond these limits will result in a panic, except in compilation mode when safety checks are disabled.

A dynamic array can be created in two ways:

- Via array literal
- Via composite literal for a dynamic array

```
a1 := [1, 2, 3]       // []i32
a2 := [1 as u8, 2, 3] // []u8
a3 := []bool{}
a4 := []bool{cap: 10}
a4 := []&Foo{len: 10, init: || &Foo{}}
```

As with fixed arrays, dynamic arrays can be nested to create arrays with multiple dimensions. Unlike fixed arrays, where the length of each inner array is the same, in a dynamic array, each internal array can have a different length.

```
matrix := [][]usize{len: 5, init: |i| []usize{len: 5, init: |j| i * j}}
println(matrix)
// [[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8],
//  [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]
```

4.1.6 Map types

A map is a group of elements in no particular order that has a specific type called an element type associated with a set of unique keys of another type called a key type.

```
MapType = 'map[' Type ']' Type
```

To use a certain type as a map key, it must implement the `MapKey` interface:

```
pub interface MapKey {
    Hashable
    Equality
}
```

That is, define a comparison operator for equality and a method for calculating a hash for a value of this type. All built-in types can be used as keys since these interfaces are implemented for them out of the box. For custom types, users need to implement them manually.

```
map[string]i32
map[Node]string
map[&mut Node]&mut Tree
```

The number of map elements is called its length. The `len()` method is defined to get the length of the map.

To get or add an element, square brackets are used, as with array indexing, with one difference, the type of expression in brackets must match the type of the map key:

```
mp := map[string]i32{}
mp["John"] = 24 // add new key-value ("John" -> 24)

println(mp["John"]) // retrieve "John" value
```

If the key is not found, the program panics.

To remove key from a map, `remove()` method is used.

A map can be created in two ways:

- Via map literal
- Via composite literal for map

```
mp1 := map[string]i32{}         // empty map[string]i32
mp2 := map[&Node]string{}       // empty map[&Node]string
mp3 := { 'a': 1, 'b': 2 }       // map[string]i32
mp4 := { 'a': 1 as u8, 'b': 2 } // map[string]u8
```

4.1.7 Tuple types

A tuple type defines a set of types combined into a single type.

```
TupleType = '(' ')' | '(' Type ',' TypeList ')'
```

A type set can have zero, two, or more types. The tuple type with zero types is called `unit` and has its own separate type. A tuple with one element is not allowed.

Tuple type can be created via tuple literal.

4.1.8 Option type

Option is a data type that can either contain a value or be empty.

```
OptionType = '?' Type
```

The empty state of an option type is described by `none` literal. The Option type often uses the discriminant elision optimization. So, for example, an optional reference `?&Foo` is represented in memory as just a reference `&Foo`. Comparing to `none` for this type is equivalent to checking for a pointer to NULL in C.

An option type instance can be created using the `as` operator:

```
a := 1
opt := a as ?i32
```

Any type `T` is automatically converted to `?T` when necessary, for example, when returning from a function with type `?string`, the return value of type `string` will be automatically converted to `?string`.

```
fn foo() -> ?string {
    return 'Hello' // valid, 'Hello' is automatically wrapped in `?string`
}
```

4.1.9 Result type

Result is a type that can either contain a value or an error.

`ResultType = '!' (Type | ('[' Type (',' Type)? ']'))?`

The Result type can have the following forms:

- `![T, E]` is the full form of the Result type, where `T` is the value type and `E` is the error type
- `!T` is a short form of a Result type, where the error type is not specified, equivalent to `![T, Error]`, where Error is the base error interface
- `![E]` is a shortened form of the Result type, where the value type is not specified, equivalent to `![unit, E]`
- `!` — bare Result type, equivalent to `![unit, Error]`

By default, unless otherwise specified, the error type is the base interface `Error` with the following definition:

```
pub interface Error {
    fn msg(self) -> string

    fn source(self) -> ?Error {
        return none
    }
}
```

Any type that implements the `Error` interface can be used as an error type in a Result type, and can also be returned from a function that does not specify a specific error type (`!T` or `!`). Built-in type string implements the `Error` interface by default.

Any type `T` is automatically converted to `![T, E]` when necessary, for example, when returning from a function with type `!string`, the return value of type `string` will be automatically converted to `!string`.

```
fn foo() -> !string {
    return 'Hello' // valid, 'Hello' is automatically wrapped in `!string`
}
```

Any E type must be wrapped in a call to the `error` built-in function to convert the error value to a Result in error state:

```
fn bar() -> !string {
    return error('Error message')
}
```

Equivalent to:

```
fn bar() -> Result[string, Error] {
    return 'Error message' as Error as Result[string, Error]
}
```

A result value of functions that return a Result type must be used. Otherwise, the compiler will throw an error. This behavior is similar to `must_use` attribute.

Ignoring the Result type will result in a compilation error.

Call considered to be ignored if it is placed in an expression statement that is not the last statement of an if expression, match arm, or `or {}` block.

```
fn get_data() -> !string { ... }

fn foo() -> !string {
    get_data() // error, Result type must be used

    data := get_data() // ok, Result type is used

    data1 := get_data() or {
        panic('Error: ${err.msg()}')
    }

    data2 := get_data() or { 'Default value' }

    data3 := get_data().unwrap()

    data4 := get_data()!

    return get_data()
}
```

4.1.10 Unit types

The Unit type describes a tuple with zero elements. This type has size 0 and can be used to describe the absence of data. This type has one valid value specified through parentheses: `()`.

By default, functions return type `unit`, in which case `return ()` is automatically inserted at the end of the function. This is also true when the `unit` type is specified explicitly as the return type.

```
fn foo() -> unit {
    return () // this return can be omitted
}
```

The Unit type can be used for optimizations since the compiler knows that the type has a single value and has zero sizes. For example, the `Set[T]` type is implemented as `Map[T, unit]`, thanks to this the compiler can optimize the layout.

4.1.11 Channel types

Channels provide a mechanism for communication between concurrently executing functions by sending and receiving values of a specific type.

```
ChannelType = 'chan' Type
```

A channel can be created via composite literal for a channel:

```
ch1 := chan i32{}         // unbuffered channel
ch2 := chan i32{cap: 10} // buffered channel with capacity 10
```

When creating a channel, channel size can be specified. The channel size determines the size of the internal buffer in which elements will be placed when they are sent to the channel. When the channel size is zero, any sending to the channel blocks the thread until another thread reads the passed value from the channel. Such a channel is called unbuffered. If the length is greater than zero, such a channel is called buffered. In this case, the thread will be blocked only when the internal buffer is full and a sending is being made, or it is empty, and a receiving from the channel is taking place.

Channels are a first-in-first-out queue. That is, if values 1, 2, 3, 4 were sent to the channel, then when received by another thread, it will receive the values in the same order: 1, 2, 3, 4.

To close a channel, the `close()` method is used. The `len()` method is used to get the length and the `cap()` method is used to get the maximum number of elements in the buffer.

4.1.12 Never type

Never type is a bottom type that has no values. This type can only be used as a function return type to indicate that the function does not return control, that is, it calls an exit-like function (`exit`, `abort`, and so on) or ends with an infinite loop.

For example, the `panic` function uses this type because it itself calls `exit` internally, thereby always interrupting the program's flow of execution:

```
fn panic(msg string) -> never {
    // ...
    exit(1)
}
```

The compiler must check that all execution paths terminate in these ways:

```
fn my_exit(cond bool) -> never {
    if cond {
        exit(1)
    }

    println("bye") // error, mush end with exit-like
                   // function call or infinity loop
}
```

4.2 Context types

A context type describes a type expected in some context. For example, in the assignment `a = x`, the context type of the expression `x` is the type of the variable `a`.

The context type is used to define concrete types for `nil`, `none` and array literals, and to infer the type of enumerations for enum fetch expressions or type of parameters for lambda literals.

Having a context type allows the compiler to automatically infer types for instantiation, as well as to generate automatic wrapping types into Option or Result types.

The following expressions are of the context type:

- Right side of assign statement.
- Expression of return expression
- Expression of send statement
- Each argument of call expression
- Expression of each key in composite literal
- Second and further expressions in array literal
- Second and further key values in map literal
- Expression on the left and right side of in expression
- Expression inside cast operator
- Expression inside index expression

4.2.1 Context type for assign statement

For the assignment `a = x`, the context type of the expression `x` is the type of the variable `a`.

Example:

```
mut a := 1 as u8 // a has type u8
a = 10
//  ^^ context type of `10` is `u8`
```

For the assignment `a, a1, ..., an = x`, the context type of the expression `x` is a tuple type with the types of all variables on the left.

Example:

```
mut a, b := 1, 2 as u8, u16
a, b = foo()
//     ^^^^^ context type of `foo()` is `(u8, u16)`
```

For the assignment `a, a1, ..., an = x1, x2, ..., xn`, the context type of the expression $\texttt{x}_\text{i}$ is the type of the variable $\texttt{a}_\text{i}$.

Example:

```
mut a, b := 1, 2 as u8, u16
a, b = 10, 20
//         ^^ context type of `20` is `u16`
//     ^^ context type of `10` is `u8`
```

4.2.2 Context type for return expression

For the expression `return x`, the context type of expression `x` is the return type of the enclosing function.

Example:

```
fn foo() -> u8 {
    return 10
    //     ^^ context type of `10` is `u8`
}
```

For the expression `return x1, x2, ..., xn` in an enclosing function with return type `(T1, T2, ..., Tn)`, the context type of expression $\texttt{x}_\text{i}$ is type $\texttt{T}_\text{i}$.

Example:

```
fn foo() -> (u8, u16) {
    return 10, 20
    //         ^^ context type of `20` is `u16`
    //     ^^ context type of `10` is `u8`
}
```

For the expression `return (x1, x2, ..., xn)` in enclosing a function with return type `(T1, T2, ..., Tn)`, the context type of expression $\texttt{x}_\text{i}$ is type $\texttt{T}_\text{i}$.

Example:

```
fn foo() -> (u8, u16) {
    return (10, 20)
    //          ^^ context type of `20` is `u16`
    //      ^^ context type of `10` is `u8`
}
```

4.2.3 Context type for send statement

For the expression `ch <- x` where `ch` has type `chan T`, the context type of the expression `x` is type `T`.

Example:

```
ch := chan u8{}
ch <- 10
//    ^^ context type of `10` is `u8`
```

4.2.4 Context type for call expressions

For the expression `f(x1, x2, ..., xn)`, the context type of the expression x_i is the type of the i-th argument of the function `f`.

Example:

```
fn foo(a u8, b u16) {}

foo(10, 20)
//      ^^ context type of `20` is `u16`
//  ^^ context type of `10` is `u8`
```

For expression `f(<non-variadic-args>, x1, x2, ..., xn` for variadic function `f(<non-variadic-params>, args ...T)`, context type of expression x_i is type `T`.

Example:

```
fn foo(a u8, b u16, ...c u32) {}

foo(10, 20, 30, 40)
//              ^^ context type of `40` is `u32`
//          ^^ context type of `30` is `u32`
//      ^^ context type of `20` is `u16`
//  ^^ context type of `10` is `u8`
```

4.2.5 Context type for composite literals

For the expression `T{f1: x1, f2: x2, ..., fn: xn}`, the context type of the expression x_i is the type of the field f_i of the structure `T`.

Example:

```
struct Foo {
    a u8
    b u16
}

f := Foo{ a: 10, b: 20 }
```

```
//                   ^^ context type of `20` is `u16`
//            ^^ context type of `10` is `u8`
```

4.2.6 Context type for array literals

For the expression `[first, x1, x2, ..., xn]`, if no context type is available for the entire expression (for example, if a literal is used to initialize a new variable), the context type of the expression $\texttt{x}_\mathrm{i}$ is the type of the first element of `first`.

Example:

```
a := [10 as u8, 20, 30]
//                   ^^ context type of `30` is `u8`
//              ^^ context type of `20` is `u8`
```

For the expression `[x1, x2, ..., xn]`, if the context type `[]T` is available for the entire expression (for example, if a literal is assigned to a variable of known type), the context type of the expression $\texttt{x}_\mathrm{i}$ is type `T` .

Example:

```
mut arr := []u8{}
arr = [10, 20, 30]
//     ^^  ^^  ^^ for all `10`, `20`, `30` context type is `u8`
```

4.2.7 Context type for map literals

For the expression `{first_key: first_value, k1: v1, k2: v2, ..., kn: vn}`, if no context type is available for the entire expression (for example, if a literal is used to initialize a new variable), the context type of the expression `ki` is the type of the first key is `first_key`. The context type of the expression `vi` is the type of the first value `first_value`.

Example:

```
mp := { "a": 10 as u8, "b": 20 }
//                          ^^ context type of `20` is `u8`
//                     ^^ context type of `"b"` is string
```

For the expression `{k1: v1, k2: v2, ..., kn: vn}`, if the context type `map[K]V` is available for the entire expression (for example, if the literal is assigned to a variable with a known type), the context type of the expression `ki` is the type `K`. The context type of expression `vi` is the type `V`.

Example:

```
mut mp := map[string]u8{}

mp = { "a": 10, "b": 20 }
```

```
//          ^^        ^^ for all values context type is u8
//     ^^^       ^^^ for all keys context type is string
```

4.2.8 Context type for in expression

For the expression `x in arr`, where `arr` has type `[]T`, the context type of the expression `x` is type `T`.

Example:

```
enum Colors {
    red
    green
    blue
}

fn main() {
    arr := [Colors.red, Colors.green, Colors.blue]
    for .red in arr {
//      ^^^^ context type of `.red` is `Colors`
    }
}
```

For the expression `x in [k1, k2, ..., kn]` where `x` has type `[]T`, the context of the expression $\mathtt{x}_i$ is type `T`.

Example:

```
enum Colors {
    red
    green
    blue
}

fn main() {
    color := Colors.red
    if color in [.red, .green, .blue] {
        //       ^^^^  ^^^^^^  ^^^^^ context type is `Colors`
    }
}
```

4.2.9 Context type for index expression

For the expression `x[i]`, where `x` has type `map[K]V`, the context type of the expression `i` is type `K`.

Example:

```
enum Colors {
    red
    green
    blue
}

fn main() {
    mut mp := map[Colors]u8{}
    mp[.red] = 10
    // ^^^^ context type of `.red` is `Colors`
}
```

4.2.10 Context type for expression in cast operator

For the expression `[x1, x2, ..., xn] as T`, where `T` is a fixed or dynamic array type `[N]T` or `[]T`, the context type of the expression $\mathtt{x}_\mathrm{i}$ is type `T`.

Example:

```
a1 := [10, 20, 30] as [3]u8
//     ^^  ^^  ^^ for all `10`, `20`, `30` context type is `u8`

a2 := [10, 20, 30] as []u8
//     ^^  ^^  ^^ for all `10`, `20`, `30` context type is `u8`
```

4.3 Type identity

The two types are said to be equivalent in the following cases:

- Two primitive types are identical if their names are the same
- Two named types (structures, unions, enumerations, interfaces) are identical if their names are the same
- Two references are identical if their inner types are identical and their mutability matches
- Two pointers are identical if their inner types are identical and their mutability is the same
- Two fixed arrays are equivalent if their internal types are identical and their lengths are the same
- Two arrays are identical if their inner types are identical
- Two maps are identical if their key and value types are identical
- Two channels are identical if their inner types are identical
- Two tuples are identical if the number and types of their elements are identical
- Two function types are identical if they have the same number of parameters and return values, the types of their parameters and return values are identical, and both functions either have or do not have variadic parameters. Parameter names do not affect type equivalence

- The two `unit` types are always identical since `unit` has only one value
- Two instantiated types are identical if their inner types and generic arguments are identical

4.4 Assignability

A value x of type V is assignable to a variable of type T ("x is assignable to T") if one of the following conditions applies:

- T and V are equivalent types
- T is an interface type and V implements T
- T is a union type and V is one of the union variants
- T and V are a reference type. V is assignable to T where T is not a mutable reference or T is a mutable reference and V is also mutable reference (in other words, `&V` and `&mut V` can be assigned to `&T`, but only `&mut V` can be assigned to `&mut T`)
- T is `&void` type and V is any reference type
- T is `&mut void` type and V is any mutable reference type
- T and V are a pointer type. V is assignable to T where T is not a mutable pointer or T is a mutable pointer and V is also mutable pointer (in other words, `*V` and `*mut V` can be assigned to `*T`, but only `*mut V` can be assigned to `*mut T`)
- T is `*void` type and V is any reference or pointer type
- T is `*mut void` type and V is any mutable reference or pointer type
- T and V are integer types, and V can be lossless converted to T
- T and V are float types, and V can be lossless converted to T
- T is an enum type with backed type identical to V
- x is `nil` literal and T is a pointer type
- T is `Option[X]` type and V is assignable to `X` type
- T is `Result[X, E]` type and V is assignable to `X` type

4.5 Method sets

The method set of a type determines the methods that can be called on an operand of that type. Every type has a (possibly empty) method set associated with it:

- The method set of a defined type T consists of all methods declared with receiver type T.
- The method set of a reference to a defined type T (where T is neither a reference nor an interface) is the set of all methods declared with receiver &T, &mut T or T.

In a method set, each method must have a unique non-blank method name.

4.6 Type layout

The layout of a type is its size, alignment, and the relative offsets of its fields. This section describes the guaranteed layout of types in memory.

4.6.1 Size and alignment

Each value in memory has a size and alignment. The alignment of a value specifies valid memory addresses for the value to be placed. A value with alignment `n` shall be placed at an address that is a multiple of `n`. Alignments measured in bytes and shall be at least one and always have a power of 2. Compiler intrinsic `align_of[T]` returns the alignment of type `T`.

The size of a value is the offset in bytes between successive elements in an array with that item type including alignment padding. The size of a type is always a multiple of its alignment. Size can be zero, in which case the type is called a zero-sized type. Compiler intrinsic `size_of[T]` returns the size of type `T`.

Primitive types have a fixed size and alignment:

Type	`size_of[T]()`	`align_of[T]()`
bool	1	1
u8	1	1
u16	2	2
u32	4	4
u64	8	8
u128	16	16
i8	1	1
i16	2	2
i32	4	4
i64	8	8
i128	16	16
f32	4	4
f64	8	8
rune	4	4

`usize` and `isize` types have a size and alignment that depend on the target, but big enough to hold any address on the target platform. For example, on a 32-bit platform, `usize` and `isize` have a size of 4 bytes and on a 64-bit platform, they have a size of 8 bytes.

Most primitives are generally aligned to their size, although this is platform-specific behavior. In particular, on x86 `u64` and `f64` are only aligned to 32 bits.

4.6.2 References and Pointers layout

References and pointers have the same layout, mutability does not affect the layout. Reference and pointer types have the same size and alignment as the `usize` type.

4.6.3 Fixed array layout

Fixed array `[N]T` has a size equal to `N * size_of[T]()` and an alignment equal to `align_of[T]()`. The size of a fixed array is always a multiple of the size of its element type.

4.6.4 Tuple layout

Tuple `(T1, T2, ..., Tn)` is represented as a structure with fields of types `T1, T2, ..., Tn`. The size and alignment of a tuple are the same as the size and alignment of the structure with the same fields.

The exception is a tuple with a zero element (`unit` type), which has a size of zero and an alignment of 1.

4.6.5 Enum layout

Enumeration type has a layout that depends on the backed type of the enumeration. If the enumeration has a backed type `u8`, then the size and alignment of the enumeration are the same as the size and alignment of the `u8` type. Since default backed type is `i32`, default size and alignment of enumeration are the same as the size and alignment of the `i32` type.

4.7 Zero-sized types

Zero-sized types are types that have a size of zero. These types are used to represent the absence of data and are used in various optimizations like discriminant elision.

The following types are zero-sized:

- Unit type or tuple with zero elements
- Structure with zero fields

4.8 Default methods for all types

The language defines the following methods, which are available for all types:

4.8.1 `str() -> string`: returns the string representation of the value

The `str` method returns a string representation for any type.

```
struct Foo {
    x i32
}

fn main() {
    f := Foo{ x: 42 }
    println(f.str())
    // Foo{
    //    x: 42
    // }
}
```

If a type defines its own `str` method, then the default implementation is not generated and the explicitly defined method is used.

```
struct Foo {
    x i32
}

fn (f Foo) str() -> string {
    return 'Foo with x = ${f.x}'
}

fn main() {
    f := Foo{ x: 42 }
    println(f.str())
    // Foo with x = 42
}
```

The `str` method is generated according to the following rules:

- If the type is an alias, interface, union, then the `str` method is generated as `$TYPE_NAME(inner_type.str())`, for example, `Option(42)`.
- If the type is a struct, then the `str` method is generated as `$TYPE_NAME{\nfield1: value1\n field2: value2\n ...}`. For example, `Foo{       a: 1       b: 2   }`
- If the type is an enum, then the `str` method returns the name of the enumeration variant. For example, `red`.
- For functional types, their representation in Spawn code is returned. For example: `fn (i32) -> i32`.
- For references, the method returns a string representation in the form `&inner_type.str()` or `&mut inner_type.str()`, for example, `&42` or `&mut "hello"`.

- For tuples, the method returns a string representation in the form `(field1.str(), field2.str(), ...)`, for example, `(1, 2, 3)`.
- If the type above contains a string type and needs to display its value, then it should be wrapped in quotes, for example, `StringOrInt("hello")`.

All primitive types, arrays, maps, channels, and strings shall have an explicit implementation of the `str()` method in the standard library.

4.8.2 `equal(self) -> bool`: equality test

The `equal` method tests the equality of two values with the same type:

```
struct Foo {
    x i32
}

fn main() {
    f1 := Foo{ x: 42 }
    f2 := Foo{ x: 42 }
    println(f1 == f2) // true
}
```

If a type defines its own `equal` method, then the default implementation is not generated and the explicitly defined method is used.

```
struct Foo {
    x i32
}

fn (f1 Foo) equal(f2 Foo) -> bool {
    return f1.x == f2.x
}

fn main() {
    f1 := Foo{ x: 42 }
    f2 := Foo{ x: 42 }
    println(f1 == f2) // true
}
```

The `equal` method is generated according to the following rules:

- If the type is a structure, then the `equal` method is generated as `field1 == other.field1 && field2 == other.field2 && ...`.
- If the type is an alias, value determines equivalence then.
- If the type is an interface or union, then equivalence is determined by the equivalence of the internal type identifier that is currently stored in the type and, if they are equal, by the equivalence of the internal values.
- Two enumeration options are equal if their names and the types to which they are bound are equal.

- Two pointers are equal if their integer representations are equal.
- Two references are equal if the dereferenced values are equal.
- Two tuples are equal if their lengths are equal and each value inside them is equal.

All primitive types, arrays, maps, channels, and strings shall have an explicit implementation of the `equal` method in the standard library.

Chapter 5

Type inference

5.1 Smart casts

Spawn supports a limited form of flow-sensitive typing called smart casts. Flow-sensitive typing means that the type of the value can change depending on the control flow graph. This allows avoiding unnecessary explicit type casting when the runtime type is guaranteed to match the expected type.

Flow-sensitive typing can be defined as a special case of Data-flow analysis.

5.1.1 Smart cast types

There are two main types of smartcasts:

- interface type smartcast to a specific type
- smartcast `?T` to `none` or specific type `T`

These two types of smartcasts use a common subset of constructs, but they do not overlap in analysis.

An example of an interface type smartcast:

```
interface Speaker {
    fn speak(self)
}

struct Dog {
    name string
}

fn (d Dog) speak() {
    println('Woof!')
}
```

```
fn main() {
    s := Dog{ name: "Rex" } as Speaker

    if s is Dog {
        println(s.name) // s has type Dog
    } else {
        // here `s` is definitely not a Dog

        if s is Dog {
            // this block is unreachable
        }
    }
}
```

?T smartcast example:

```
fn main() {
    arr := [1, 2, 3]
    first := arr.first_or_none()
    if first != none {
        // first is of type i32 and we can call its `hex()`
        // method without unwrapping
        println(first.hex())
    }
}
```

5.2 Least Upper Bound (LUB)

Spawn uses the Least Upper Bound (LUB) algorithm to determine the common type of two types for `if` and `match` expressions. The LUB algorithm is used in the following cases:

- For the type of if expression
- For the type of match expression

The LUB algorithm is defined as follows:

- If the types are identical, the common type is the same type
- If one type is untyped `none` and the other is a specific type `T`, the common type is `?T` (`Option[T]`).
- If one type is `T` and the other is `?T`, the common type is `?T`
- If one type is untyped `nil` and the other is a raw pointer type `*T`, the common type is `*T`
- If one type is `never` and the other is a specific type `T`, the common type is `T`
- If one type is `T` and other is `R` and `T` and `R` implement the same interface,

the common type is the interface type. If `T` and `R` implement several common interfaces, LUB process is failed and the compiler reports an error

$LUB(T1, T2) = LUB(T2, T1)$ means that the LUB operation is commutative, for example, $LUB(none, i32) = ?i32$ and $LUB(i32, none) = ?i32$.

5.3 Instantiation

Instantiation is the process of creating a specific type from a generic type. Function instantiation can be thought of as the process of creating a specific function type from a generic function type.

The process of instantiation is carried out by substituting specific types into places of generic types.

The specific types can be declared explicitly or can be inferred automatically by the compiler based on context or arguments.

```
GenericArguments = '[' TypeList ']'
```

In the case where a function call contains GenericArguments, the compiler uses the described types to instantiate the function. Also, if the type in composite literal contains GenericArguments, then the compiler uses the described types to instantiate the structure.

5.3.1 Mapping Generic Type → Concrete Type

The instantiation process begins by inferring type pairs for each generic type. For each generic type described in a function or type definition, a specific type is inferred, either based on GenericArguments or based on context and arguments.

For a function with generic parameters `T1, T2, ... Tn` that is called as follows `foo[U1, U2, ... Un]()`, generic types are inferred by mapping the generic type $\mathtt{T}_\mathrm{i}$ to a specific type U_i.

For example:

```
fn filter[T](arr []T, val T) -> []T { ... }

fn main() {
    arr := [1, 2, 3]
    res := filter[i32](arr, 2)
}
```

GenericArguments `[i32]` is mapped to GenericParameters `[T]` and the compiler infers that type `T` is instantiated by type `i32`. Thus, the compiler has inferred all the necessary generic parameters for the function instance.

In case GenericArguments are not provided, the compiler attempts to infer types based on context and arguments, a process called type reifying.

5.3.2 Type reifying

Type reifying is the process of inferring the specific types with which a function or type should be instantiated based on context and arguments. Type reifying is considered successful if it was able to infer all missing types. Otherwise, the program is considered invalid.

5.3.2.1 Basic example

Let's look at a simple example to understand the general idea:

```
fn filter[T](arr []T, val T) -> []T { ... }

fn main() {
    arr := [1, 2, 3]
    res := filter(arr, 2)
}
```

In this example, GenericArguments are omitted, but knowing the function arguments and their types, the compiler can infer that type `T` must be instantiated by type `i32`.

In a simplified form, the compiler sees that the passed array is of type `[]i32`, while the first parameter of the `filter` function is of type `[]T`, from which the compiler can conclude that `T` must be instantiated with type `i32` . The second argument of the function is of type `i32`, and the second parameter of the function is of type `T`, since the type `T` is already inferred as `i32`, the compiler deducing that `T` should again be instantiated by type `i32`, considers this call is correct and instantiates the `filter` function with type `i32`.

However, the following call will be invalid:

```
fn main() {
    arr := [1, 2, 3]
    res := filter(arr, "2")
}
```

Because from the second argument the compiler inferred that `T` must be instantiated by type `string`, which does not correspond to the previously inferred type `T` as `i32`.

Before the algorithm is described, we will consider the equations for solving the problem above.

We have two arguments and two parameters of the function, for each argument-parameter pair we write the equation:

1. $[]T \equiv_A typeof(arr)$

2. $T \equiv_A typeof(2)$

Replacing the *typeof* function with the actual types, we get:

1. $[]T \equiv_A []i32$
2. $T \equiv_A i32$

Now we can start solving the system of equations. We begin with the first equation. Since both sides of the equation are arrays, we can unwrap them:

$T \equiv_A i32$

Now, since the left side of the equation is a type variable, we found the solution for it. We can substitute the found value into the second equation:

$i32 \equiv_A i32$

And since this equation is true, the system of equations is solved, in a result the type `T` is inferred as `i32`. Since all type variables are found, the inference process is considered successful.

5.3.2.2 Algorithm

TODO: Describe the algorithm

Chapter 6

Declarations and scopes

Declaration is a statement that introduces a new non-blank name into the program and associates it with a type or value.

Every identifier in a program must be declared. Identifier cannot be declared twice in the same block.

The `init` identifier shall only be used for `init` function.

An identifier declared in a block cannot be redeclared (shadowed) in the inner blocks.

```
fn main() {
    a := 1
    if a == 1 {
        a := 2 // error
    }
}
```

If declaration uses the `pub` keyword, then the declared entity is visible outside the current module.

If declaration uses the `extern` keyword, where appropriate, then the declared entity is treated as external.

6.1 Blank identifier

Blank identifier is defined by the special identifier `_` and is used to ignore entire expressions, or to ignore parts of expressions when unpacking.

```
BlankIdent = "_"
```

Blank identifier can only be used in certain places:

6.1.1 As a variable name when unpacking:

```
fn foo() -> (i32, i32) {
    return 1, 2
}

_, b := foo()
```

In this case, only the second value will be assigned to the variable `b`, the first value will be ignored.

`_` cannot be used in a normal variable declaration:

```
_ := 1 // error
```

6.1.2 As the left side of an assignment

In this case, the right expression will be ignored:

```
_ = foo()
```

Or when unpacking:

```
_, b = foo()
```

6.1.3 As a parameter name

If the parameter is not used, then its name can be replaced with `_`:

```
fn foo(_ i32) {}
```

6.1.4 As a receiver name

If the receiver is not used, then its name can be replaced with `_`:

```
fn (_ Foo) foo() {}
```

6.1.5 As a placeholder in a for-in loop

```
for _ in 0 .. 10 {} // iterate 10 times
for _, el in arr {} // iterate over array elements without index
for i, _ in arr {}  // iterate over array indexes without element
```

In all other cases, the use of `_` is prohibited.

Placeholder can be defined multiple times in one block:

```
fn bar(_ i32, _ i64) {}
fn main() {
    _, a := foo()
    b, _ := foo()
}
```

List of diagnostics:

- `E0177`: Cannot declare variable `_`, this name is reserved for placeholders
- `E0179`: Placeholder `_` cannot be mutable
- `E0180`: Wrong usage of placeholder
- `E0182`: Placeholder `_` is not needed here (warning)
- `E0183`: Placeholder `_` is invalid here

6.2 Predeclared identifiers

The following identifiers are predeclared in the global scope:

Types:

```
i8 i16 i32 i64 i128 isize
u8 u16 u32 u64 u128 usize
f32 f64
bool
rune
```

Zero values

```
nil none
```

Functions:

```
print println eprint eprintln
todo unreachable
panic recover
```

6.3 Uniqueness of identifiers

Given a set of identifiers, an identifier is called unique if it is different from every other in the set. Two identifiers are different if they are spelled differently or if they appear in different modules and are not exported. Otherwise, they are the same.

6.4 Enumerations

```
EnumDeclaration         = Attributes? 'pub'? 'enum' identifier EnumBackedType?
                          '{' EnumVariantDeclaration* '}'
EnumBackedType          = 'as' Type
EnumVariantDeclaration  = Attributes? identifier DefaultEnumVariantValue? semi?
DefaultEnumVariantValue = '=' Expression
```

Enumerations are a non-zero set of named constants bound to that enumeration and representing integer values. Each enumeration option has a value one greater than the previous one, unless it has an explicit value. The first enumeration option has value 0 if it has no explicit value.

```
enum Color {
    red   // value: 0
    green // value: 1
    blue  // value: 2
}
```

With explicit values:

```
enum Color {
    red    = 1
    yellow = 2
    green  = 25
    blue   // value: 26
}
```

By default, each enumeration option has `i32` type, which imposes restrictions on the maximum and minimum values of the enumeration option. If the value does not fit within the boundaries of `i32`, then the compiler should throw an error.

6.4.1 Backed type

An enumeration definition can have an optional backed type, which specifies the type of the enumeration variants. Backed type is defined after the enumeration name using the `as` keyword and the type.

```
enum Color as u8 {
    red
    blue
}
```

Backed type must be an integer type `u8`, `u16`, `u32`, `u64`, `usize`, `i8`, `i16`, `i32`, `i64` or `isize`. When specifying a backed type, each enumeration variant must fit into this type, otherwise the compiler must throw an error.

Backed type also specified a type to which the enumeration value can be implicitly converted. For example, if the backed type is `u8`, then the enumeration value can be used in expressions where `u8` is expected.

```
enum Color as u8 {
    red
    blue
}

fn take_u8(value u8) {
    // ...
}
```

```
fn main() {
    take_u8(Color.red) // ok
}
```

In other cases the compiler should throw an error and require an explicit cast.

List of diagnostics:

- `E0099`: Enum with no variants
- `E0206`: Enum variant out of range for a backed type
- `E0207`: Enum backed type must be one of `u8`, `u16`, `u32`, `u64`, `usize`, `i8`, `i16`, `i32`, `i64` or `isize`

6.4.2 Flag enum

The `flag` attribute changes the default behavior for enum variant values. Each enumeration option now has a value equal to `1 << i`, where `i` is the ordinal number of the enumeration option. The first enumeration option has the value `1`, the second `2`, the third `4` and so on. Variants of such enumerations cannot have explicit values.

```
#[flag]
enum Access {
    read    // value: 0b0001
    write   // value: 0b0010
    execute // value: 0b0100
}
```

Flag enums are also called bit fields because each enumeration variant has a value representable as a bitmask that is unique throughout the enumeration.

For example, using the `Access` enum from the example above, user can create a variable that can hold multiple enum values at once:

```
a := Access.read | .write // value: 0b0011
```

`.read` set the first bit to `1` and `.write` set the second bit to `1`. Now, we can check for the presence of a certain value in a variable using the `&` operator:

```
if a & .read != 0 {
    println('read access')
}

if a & .write != 0 && a & .execute != 0 {
    println('read and execute access')
}
```

To make working with such enums easier, flag enums have additional methods not available for regular enums:

- `has(flag T) -> bool` — checks for the presence of one or more flags in the enumeration
- `any(flag T) -> bool` — checks for the presence of at least one flag in the enumeration
- `set(flag T)` — sets a flag in the enumeration
- `toggle(flag T)` — toggles the flag in the enumeration
- `clear(flag T)` — resets the flag in the enumeration

Example:

```
#[flag]
enum Access {
    read    // value: 0b0001
    write   // value: 0b0010
    execute // value: 0b0100
}

fn main() {
    mut a := Access.read | .write // value: 0b0011
    if a.has(.read) {
        println('read access')
    }

    a.toggle(.write)
    if a.any(.write | .execute) {
        println('write or execute access')
    }

    a.set(.execute)
    if a.has(.execute) {
        println('execute access')
    }
}
```

Since the values of enum variants grow exponentially, an enum cannot have more than `size_of[BackedType]() * 8 - if BackedType is signed { 1 } else { 0 }` variants. Thus, the default definition cannot have more than 31 options for `i32`. In order for an enum to store up to 64 variants, it must be defined with backed type `u64`.

If the next variant value exceeds the maximum value for the backed type, then the compiler should throw an error.

6.5 Const declaration

A constant declaration associates a list of identifiers (constant names) with expressions that initialize their values.

```
ConstDeclaration = Attributes? ExternHeader? 'pub'? 'comptime'? 'const'
                   ( ConstDefinition | '(' ConstDefinitions? ')' )
ConstDefinitions = ConstDefinition (semi ConstDefinition)* semi?
ConstDefinition  = identifier '=' Expression
```

Constants can only be defined within the scope of a module, not within functions or a block of code.

The type of each constant is determined by the type of the expression that initializes its value. The expression does not need to be compile-time computable. If the value of a constant cannot be calculated at compile time, it will be initialized before executing the `main` function.

The initialization expression can use other constants, but if there is a circular dependency between two constants, the compiler should throw an error.

```
const FIRST = SECOND
const SECOND = FIRST + 10 // error
```

6.6 Module variable declaration

A module variable declaration defines a variable that is available throughout the module.

```
ModuleVarDeclaration = Attributes? ExternHeader? 'pub'? 'var'
                       ( ModuleVarDefinition | '(' ModuleVarDefinitions? ')' )
ModuleVarDefinitions = ModuleVarDefinition (semi ModuleVarDefinition)* semi?
ModuleVarDefinition  = identifier '=' Expression
```

Unlike constants, module variables can change their value over time. Module variables can only be defined within the scope of the module, not within functions or code blocks.

The type of each module variable is determined by the type of the expression that initializes its value.

The initialization expression can use other module variables, but if there is a circular dependency between two variables, the compiler should throw an error.

```
var first = second
var second = first + 10 // error
```

Module variable can be declared as "thread local" with the `thread_local` attribute. Such variables are stored in thread-local storage and have a separate instance for each thread.

6.7 Function declaration

A function declaration associates an identifier representing the name of a function with the function itself.

```
FunctionDeclaration = Attributes? ExternHeader? 'pub'? 'fn' identifier
                      GenericParameters? Signature WhereConstraints? Block?
```

A function with a signature that has a return type must end with terminating statement.

```
fn foo(name string) -> i32 {
    if name == "John" {
        return 10
    }

    // error: missing return statement
}
```

Terminating statements can be omitted if the function has no return type or has the following return type:

- unit type
- empty tuple type
- `?unit`
- `!`
- `!unit`
- `![unit, <any error type>]`

A function may not have a body, in which case it defines a function implemented outside Spawn, for example, in the C library:

```
extern fn puts(s *u8) // implemented in libc
```

In this case function must have the `extern` keyword before the `fn` keyword. After `extern` keyword can be specified the calling convention, for example, `extern "C"`.

If a function defines GenericParameters, the function is called a generic function. Before use, it must be instantiated:

```
fn less[T: Ordered](a T, b T) -> bool {
    return a < b
}

fn main() {
    less(1, 2)
    less[string]("a", "b")
}
```

When instantiating a generic, the arguments can be inferred by the compiler or specified explicitly. If the compiler could not output them on its own, it should throw an error.

Bounds for generics can also be specified in `where`:

```
fn less[T](a T, b T) -> bool where T: Ordered {
    return a < b
}
```

6.7.1 Variadic functions

A function can have a variadic parameter, which is a parameter that can accept an arbitrary number of arguments. The variadic parameter shall be the last parameter in the function signature. A function with a variadic parameter is called a variadic function.

To define a variadic parameter, the `...` symbol is used before the parameter type:

```
fn sum(a i32, b i32, args ...i32) -> i32 {
    res := a + b
    for arg in args {
        res += arg
    }
    return res
}
```

6.7.2 Parameters default values

Function parameters can have default values. If a parameter has a default value, then all subsequent parameters shall also have default values.

When calling a function with default parameters, arguments for parameters with default values can be omitted:

```
fn foo(a i32, b i32 = 10, c i32 = 20) {
    println(a)
    println(b)
    println(c)
}

fn main() {
    foo(1) // 1 10 20
    foo(1, 2) // 1 2 20
    foo(1, 2, 3) // 1 2 3
}
```

If a function is variadic, the function shall not have default parameters:

```
fn foo(a i32, b i32 = 10, c i32 = 20, args ...i32) {} // error
```

6.8 Method declarations

A method is a function with an additional parameter called receiver. Method declaration associates an identifier representing the method name with the method itself. The method is associated with the base type defined in the receiver.

```
MethodDeclaration = Attributes? 'pub'? 'fn' '(' Receiver ')' identifier
                    GenericParameters? Signature WhereConstraints? Block?
Receiver          = identifier Type
```

The receiver is defined as an additional parameter located before the method name. This parameter specifies a single non-variadic parameter called the receiver. The receiver type must be type T or a reference to type T followed by an optional list of generic parameters enclosed in square brackets. Type T is called the base type of the receiver; this type does not need to be an interface and shall be defined in the current module.

If the receiver name is not used within a method, it should be renamed to _ (blank identifier) and potentially the entire method should be changed to static method.

Methods can be mutable or immutable.

The method name must be unique for the underlying receiver type, the following case is invalid:

```
fn (b Bar) foo() {}
fn (b &Bar) foo() {}
```

Because the second method also attaches the `foo` method to the `Bar` type, which already has a method with that name.

If the underlying receiver type is a generic type, then the receiver type must describe all types defined by the type and have the same names (or _ if the type is not used inside a method). Syntactically, this looks like instantiation of a generic type:

```
struct Point[T] {
    x T
    y T
}

fn (p Point[T]) distance(other Point[T]) -> usize { ... }
fn (p Point[_]) print() {}
```

All constraints for generic types are inherited from the type where they are defined:

```
struct Point[T: Ordered] {
    x T
    y T
}

fn (p Point[T]) set_x(x T) { ... }
//                      ^ must implement Ordered as well
```

6.9 Static method declaration

A static method is a function directly bound to a type, accessible through the name of that type. Unlike methods, calling a static method does not require an instance of the type, only its name. Static method declaration associates an identifier representing the method name with the method itself.

```
StaticMethodDeclaration = Attributes? 'pub'? 'fn' identifier '.' identifier
                          GenericParameters? Signature WhereConstraints? Block?
```

The type to which a method is bound is defined as the type name before the method name, followed by a dot:

```
fn System.default_user()
```

TODO: what if the type is a generic type?

Static and instance methods cannot have the same names. The following code is invalid:

```
fn (s System) name() -> string
fn System.name() -> string
```

6.10 Type alias declaration

Type alias declaration creates a type alias for an existing type.

```
TypeAliasDeclaration = Attributes? ExternHeader? 'pub'? 'type' identifier
                       GenericParameters? WhereConstraints? '=' Type
```

An alias type can have its own methods and also inherit all methods of the base type.

```
type MyInt = i32

fn (a MyInt) foo() { ... }

fn main() {
    a := 100 as MyInt
    a.foo()
    println(a.str())
}
```

A type alias can be used in any context where the underlying type is expected.

```
type MyInt = i32

fn take_i32(a i32) { ... }

fn main() {
    a := 100 as MyInt
    take_i32(a)
}
```

Conversely, a base type can be used in contexts where a type alias is expected. The compiler will automatically convert the type to the expected one.

```
type MyInt = i32

fn take_my_int(a MyInt) { ... }

fn main() {
    a := 100
    take_my_int(a)
}
```

6.11 Struct declaration

A structure is a collection of named elements called fields, each of which has a name and a type. Field names can be specified explicitly or implicitly. Field names in the structure must be unique.

```
StructDeclaration      = Attributes? ExternHeader? 'pub'? 'struct' identifier
                         GenericParameters? WhereConstraints?
                         '{' FieldDeclaration* '}'
FieldDeclaration       = Attributes? (PlainFieldDeclaration | EmbeddedDefinition)
                         semi?
PlainFieldDeclaration = identifier Type DefaultFieldValue?
DefaultFieldValue      = '=' Expression
EmbeddedDefinition     = RefType | identifier | SelectorExpression | GenericType
```

For example:

```
pub struct ContinueOutLoopInspection {}

#[rename("CamelCase")]
struct Person {
    #[json("UserName")]
    name string
    age  i32 = 0

    data_cb fn () -> Data
}
```

A unit-like struct is a structure with no fields:

```
struct Empty {}
```

Such structures have only one valid value: `Empty{}`.

6.11.1 Field visibility

By default, all fields in a structure are private and can only be accessed from methods of the structure itself. To make a field public, the field must be prefixed with the `pub` keyword:

```
struct Person {
    pub name string
    pub age  i32
}
```

6.11.2 Field attributes

Each field can have one or more attributes that specify meta-information, which can then be obtained through compile-time reflection:

```
struct Person {
    #[json("UserName")]
    name string

    #[json(skip)]
    #[sql(skip)]
    age i32
}
```

6.11.3 Field default value

A field that has an explicit name and type can have a default value. The default value will be used when creating a structure instance via composite literal if a value for this field was not specified explicitly.

```
struct Person {
    name string = "John"
    age  i32 = 0
}

fn main() {
    p := Person{}
    println(p.name) // John
    println(p.age)  // 0
}
```

6.11.4 Embedded fields

If a field definition contains only a type without explicitly specifying a name, then it is called an embedded field. The type of such a field can be one of the following types: a structure type, a structure reference type, a generic type with an inner structure type, an alias of a structure type.

The field name is inferred from the unqualified type name:

```
struct Foo {
    Bar                  // Bar
    &Qux                 // Qux
    mod.MyBar            // MyBar
    mod.Baz[i32, string] // Baz
}
```

The following structure is invalid because field names must be unique within the structure:

```
struct Foo {
    Bar
    &Bar
    mod.Bar
}
```

Fields embedding implements a form of inheritance through composition. When embedding structure A into structure B, all (TODO: public?) fields of structure A will be accessible from the instance of structure B via dot notation and can also be set when creating structure B via a composite literal:

```
struct Data {
    name string
    age  i32
}

struct Person {
    Data
    height i32
}

fn main() {
    p := Person{
        name: "Alice"
        age: 25
        height: 170
    }
    println(p.name) // Alice
}
```

When embedding struct A into struct B, all struct A's methods will also be accessible via call as struct B's own methods.

```
struct Data {
    name string
    age  i32
}

fn (d Data) as_string() -> string {
    return '${d.name}: ${d.age}'
}

struct Person {
    Data
    height i32
}

fn (p Person) print() {
    println(p.as_string())
    println('height:', p.height)
}

fn main() {
    p := Person{
        name: "Alice"
        age: 25
        height: 170
    }
    p.print()
    // Alice: 25
    // height: 170
}
```

A structure T cannot contain a field of type T or a type that contains T as one of its components, either explicitly or implicitly in the form of an array `[]T` or simply `T`.

```
struct T1 {
    T1 // illegal, T1 directly embedded in T1
    T2 // illegal, T2 has T1 type field
    T3 // illegal, T2 has []T1 type field

    T4 // legal, T1 component in T4 is reference
    T5 // legal, T1 component in T5 is part of function type
}

struct T2 {
    t1 T1
}

struct T3 {
    t1 []T1
}

struct T4 {
    t1 &mut T1
}

struct T5 {
    t1 fn () -> T1
}
```

If embedded struct S has field X and struct Y that embeds it also has field X, field Y.X will shadow field S.X then. To access field S.X from Y, a full path shall be used:

```
struct S {
    x i32
}

struct Y {
    S
    x i32
}

fn main() {
    mut y := Y{}
    y.x = 10   // set Y.x
    y.S.x = 20 // set S.x
}
```

6.11.5 C-union-like structs

Structures that have the `c_union` attribute are treated as C-style unions. In such structures, all fields share a common memory space with the size of the largest field. There can be only one active field in the structure at one time.

```
struct VecXY {
    x i32
    y i32
}

#[c_union]
struct Vec {
    VecXY
    arr [2]i32
}

fn main() {
    v := Vec{}
    v.x = 10
    unsafe {
        println(v.x) // 10
        println(v.arr[0]) // 10
    }
}
```

Access to structure fields must always be in an `unsafe` block. Assignment is allowed outside the `unsafe` block at the same time.

If at a particular moment the structure contains an active field X, then accessing the Y field may lead to undefined behavior.

A composite literal for such a structure can contain only one field:

```
v := Vec{ VecXY: VecXY{ x: 10, y: 20 } }
```

All rules for embedding structures apply to such structures as well.

TODO: field type restriction

6.12 Interface declaration

An interface defines a set of methods that are inherent to a set of types. An interface type defines a set of types. A variable with an interface type can store a value of any type from the set of these types. This type is said to “implement” an interface. An interface type variable has no default value and must always be initialized.

```
InterfaceDeclaration        = Attributes? 'pub'? 'interface' identifier
                              GenericParameters? WhereConstraints? '{'
                              InterfaceMember* '}'
InterfaceMember             = (InterfaceMethodDefinition |
                              InterfaceEmbeddedDefinition) semi?
InterfaceMethodDefinition   = Attributes? 'fn' identifier GenericParameters?
                              Signature WhereConstraints? Block?
InterfaceEmbeddedDefinition = identifier | SelectorExpression | GenericType
```

An interface defines a set of interface elements. An interface element is either an interface method or an embedded interface type.

6.12.1 Basic interfaces

In the simplest form, interfaces define a set of (potentially empty) methods. The set of types inherent in an interface is determined by a set of types, where each of these types has in its set of all the methods of the interface. Interfaces containing only methods are called basic interfaces.

```
interface ReadWriter {
    fn read(&mut self, buf &mut []u8) -> !i32
    fn write(&mut self, buf []u8) -> !i32
}
```

Interface method names must be unique within the entire interface and cannot have `_` as their name.

An interface can be implemented by more than one type.

Any type that is part of an interface's type set implements that interface. Any type can implement any number of different interfaces. For example, any type implements an interface without methods. This type is defined under the name `any` for convenience.

6.12.2 Embedded interfaces

Interfaces can also embed other interface types. In this case, embedding interface E into interface T is called embedding E into T. In the case of embedding, the resulting type set T will be the intersection of the type sets E and T. In other words, the type set T will contain only those types that implement explicit methods of the T interface and all methods of the embedded interface E.

```
interface Reader {
    fn read(&mut self, buf &mut []u8) -> !i32
}

interface Writer {
    fn write(&mut self, buf []u8) -> !i32
}

interface ReadWriter {
    Reader
    Writer
}
```

In this example, `ReadWriter` matches the following interface exactly:

```
interface ReadWriter {
    fn read(&mut self, buf &mut []u8) -> !i32
    fn write(&mut self, buf []u8) -> !i32
}
```

If an embedded interface E defines a method X, then when E is embedded into T, if T also contains a method X, the methods must have the same signature or the compiler must throw an error.

6.12.3 Method receivers

Interface methods specify the optional receiver as the first parameter. This receiver can have three types:

- `self` — the receiver does not impose restrictions on the receiver type of the implementing type; this means that when implementing this type, the receiver can be anything (by value, by reference, by mutable reference)
- `&self` — the receiver requires that the implementing type must be passed by immutable reference
- `&mut self` — the receiver requires that the implementing type must be passed by mutable reference

If an implementing type does not follow these rules, it is considered not to implement that interface method.

```
interface Hasher {
    fn write(&mut self, buf []u8)
    //       ^^^^^^^^^ mutable receiver
}

struct MyHasher {}

fn (h MyHasher) write(buf []u8) {
//    ^^^^^^^^ non-matching receiver type
    println('writing bytes: ${buf}')
}
```

The `MyHasher` type does not implement the `Hasher` interface, since the `write` method requires a receiver passed by mutable reference, while `MyHasher` is passed by value.

6.12.4 Instance methods

The interface methods that define the receiver are called instance methods. Such methods can only be called on a variable of an interface type, and their implementation requires the presence of an instance method on the implementing type.

6.12.5 Static methods

Interface methods that do not define a receiver are called static methods. Such methods can be called on an interface type, and their implementation requires the presence of a static method on the implementing type.

```
interface Hasher {
    fn read(&mut self, buf &mut []u8) -> !i32
    fn round() -> u64
}

struct MyHasher {}

fn (h &mut MyHasher) write(buf []u8) {
    println('writing bytes: ${buf}')
}

fn MyHasher.round() -> u64 {
    return 10
}
```

Such methods cannot be called through an interface type and are used to define restrictions on generic types:

```
fn hash[T: Hasher](h T) {
    T.round()
}
```

In this case, because `Hasher` defines a `round()` method, it can be used inside the `hash()` function. In the resulting code, a specific method of type `T` will be called, which was passed as an argument when calling the `hash()` function.

6.12.6 Default method implementation

By default, interface methods define only a name and signature and are required for implementation. However, an interface can define methods with a default implementation. Such methods are called default methods.

```
interface Hasher {
    fn write(&mut self, buf []u8)

    fn write_string(&mut self, str string) {
        self.write(str.bytes())
    }
}
```

Such methods are not required for implementation; when calling such a method, if the type does not have an implementation, the default implementation described in the body of the interface will be used.

```
struct MyHasher {}

fn (h &mut MyHasher) write(buf []u8) {
    println('writing bytes: ${buf}')
}

fn main() {
    h := MyHasher{} as Hasher
    h.write_string("hello") // writing bytes: [104 101 108 108 111]
}
```

Such a method can be overloaded, in which case when it is called, the overloaded version of the method will be called.

```
struct MyHasher {}

fn (h &mut MyHasher) write(buf []u8) {
    println('writing bytes: ${buf}')
}

fn (h &mut MyHasher) write_string(str string) {
    println('writing string: ${str}')
}

fn main() {
    h := MyHasher{} as Hasher
    h.write_string("hello") // writing string: hello
}
```

Static interface methods can also have a default implementation. In this case, they can be called through the interface type.

```
interface Hasher {
    fn write(&mut self, buf []u8)

    fn round() -> u64 {
        return 10
    }
}
```

In this case, in the example with generic type restrictions, the `round()` method will be called on a specific type `T` if it implements its own `round()` method, otherwise the default method will be called.

```
struct MyHasher {}

fn (h &mut MyHasher) write(buf []u8) { ... }

fn MyHasher.round() -> u64 {
    return 20
}

struct OurHasher {}

fn (h &mut OurHasher) write(buf []u8) { ... }

fn hash[T: Hasher](h T) {
    println(T.round())
}

fn main() {
    hash(MyHasher{})  // 20, called `MyHasher.round()`
    hash(OurHasher{}) // 10, called default `Hasher.round()`
}
```

6.13 Union declaration

Union declaration defines a type that can store one of several types. The union type is defined by the `union` keyword followed by the union name and a list of types that can be stored in the union.

```
UnionDeclaration = Attributes? 'pub'? 'union' identifier GenericParameters?
                   WhereClause? '=' TypeUnionList
TypeUnionList    = Type ( '|' Type)*
```

The Union type does not define indirection, so the size of the union is equal to the size of the largest type in the union, and a union type cannot contain struct with field of type union:

```
union Foo = Bar | Baz

struct Bar {
    x Foo
}
```

This code is invalid because the `Bar` struct contains a field of type `Foo`, which is a union type one of the types of which is `Bar`. In this case size of `Foo` cannot be determined.

Internal representation of the union type is a pair of discriminant and data.

```
union StringOrInt = string | i32
```

In C representation of the `StringOrInt` union will be:

```
struct StringOrInt {
    int id; // discriminant
    union {
        string _str;
        int    _int;
    };
}
```

Discriminant is an opaque integer that indicates the type of the data stored in the union. The type of the discriminant is not specified and is determined by the compiler.

6.13.1 Discriminant elision

Union types with exactly two types are targeted for discriminant elision optimization.

Option-like unions The union type with two types where one of the types is a zero-sized type.

```
struct Empty {}

union OptionString1 = string | Empty
union OptionString2 = string | unit
union OptionString3 = string | ()
```

Payload of Option-like unions The non-zero-sized type in the Option-like union. In the example above, the payload is `string`.

Niche The niche of a type determines invalid bit-patterns that will be used by layout optimizations.

For example, `&mut T` has at least one niche, the “all zeros” bit-pattern. While all niches are invalid bit-patterns, not all invalid bit-patterns are niches. For example, the “all bits uninitialized” is an invalid bit-pattern for `&mut T`, but this bit-pattern cannot be used by layout optimizations, and is not a niche.

The next types have niches:

- `&T` and `&mut T` have a niche of all bits zeros (NULL pointer)
- Any function types (`fn ()`) have a niche of all bits zero (NULL function pointer)
- `string` has a niche of all bits zero (nil string)
- Interface type with inner discriminant value is equal to -1
- Union type with inner discriminant value is equal to -1

Option-like union types where the payload has at least one niche **shall be optimized** to representation with the same memory layout as the payload type. This is called discriminant elision, as there is no explicit discriminant value stored anywhere. Instead, niche values are used to represent the unit variant.

```
union OptionString2 = string | unit
```

Represented in memory as just a string, with the unit variant represented by nil-string (niche value):

```
struct string {
    data *u8;
    size usize;
};

void main() {
    // s1 := "hello" as OptionString2
    string s1 = (string){.data = "hello", .size = 5};
    // if s1 is string { ... }
    if (s1.data != NULL) { ... }

    // s2 := () as OptionString2
    string s2 = (string){.data = NULL, .size = 0};
    // if s2 is unit { ... }
    if (s2.data == NULL) { ... }
}
```

The next union type is not Option-like since it has two zero-sized types:

```
struct Foo {}
struct Bar {}
union Baz = string | Foo | Bar
```

6.14 Extern declarations

Extern declarations describe structures, type aliases, functions, constants, and module variables that are defined externally. For example, in some object files written in another language.

Extern functions are defined without a body. Extern constants and module variables are defined with a dummy value, the real value will be obtained during linking. Extern structures can define fields; the field list does not have to contain all the fields from the original structure but must have the same names.

Extern declarations in the definition can specify an ABI (Application Binary Interface). Currently, only the C ABI is supported, which is also the default ABI.

Extern functions can define C-style variadic via `args ...any`:

```
extern "C" fn printf(format string, args ...any)
```

6.14.1 Extern list

Extern list defines a block of declarations, each of which is treated as extern.

```
ExternHeader = 'extern' StringLiteral?
ExternList   = Attributes? ExternHeader '{' (TopLevelDeclaration semi?)* '}'
```

The list of declarations within a block can only contain the following declarations:

- Functions
- Structs
- Module variables
- Constants
- Type alias declaration

For example:

```
extern "C" {
    const PI = 0 // real value will be obtained in link-time

    type size_t = u64

    fn puts(s *u8)

    struct Point {
        x i32
        y i32
    }
}
```

Since the C ABI is the default, the `"C"` string can be omitted.

6.15 Generic parameters

Functions, methods, type aliases, structs, unions and interfaces may be *parameterized* by types and constants. These parameters are listed in square brackets (`[...]`), usually immediately after the name of the declaration and before its definition

```
GenericParameters            = '[' GenericParameterList ']'
GenericParameterList         = GenericParameter (',' GenericParameter)* ','?
GenericParameter             = (PlainGenericParameter | ConstantGenericParameter)
                               GenericParameterDefaultValue?
GenericParameterDefaultValue = '=' Expression
PlainGenericParameter        = identifier (':' Constraints)?
ConstantGenericParameter     = 'const' identifier 'as' Type
```

Generic parameters in scope within the definition where they are declared.

6.15.1 Const generics

Const generic allows types to be parameterized by constant values. The const identifier introduces a name for the constant parameter, and all instances of the item must be instantiated with a value of the given type.

The only allowed types of const parameters are `u8-u128`, `usize`, `i8-i128`, `isize`, `rune` and `bool`.

```
fn double[const N as usize]() {
    println("doubled: ${N * 2}")
}

const SOME_CONST = 10

fn main() {
    double[10]()
    double[-53]()
    double[SOME_CONST]()
}
```

Const parameter can be used as part of any runtime expression:

```
fn foo[const N as usize](a [N]i32) {
    for i := 0; i < N; i++ {
        println(a[i])
    }
}
```

As well as part of field type in struct:

```
struct SimdVector[const N as usize] {
    inner [N]i32
}
```

And can be passed to another generic parameter:

```
fn foo[const N as usize, T](a [N]T) {
    for i := 0; i < N; i++ {
        println(a[i])
    }
}

fn bar[const N as usize]() {
    foo[N, i32]([1, 2, 3])
}
```

6.15.2 Where clauses

Where clauses provide another way to specify bounds on type parameters as well as a way to specify bounds on types that aren't type parameters.

```
WhereConstraints = semi? WhereClause semi?
WhereClause      = 'where' semi? WhereClauseItem (',' WhereClauseItem)* ','?
WhereClauseItem  = identifier ':' Constraints
Constraints      = Type ('+' Type)*
```

For example:

```
fn foo[T](a T, b T) -> bool where T: Ordered {
    return a < b
}
```

Equivalent to:

```
fn foo[T: Ordered](a T, b T) -> bool {
    return a < b
}
```

Chapter 7

Expressions

An expression specifies the computation of a value by applying operators and functions to operands. Computations may involve side effects such as updating memory or performing I/O. All expressions have a defined result type.

```
Expression =
    identifier
  | comptime_identifier
  | OrExpression
  | AndExpression
  | ConditionalExpression
  | AddExpression
  | MulExpression
  | TakeAddrExpression
  | UnaryExpression
  | ReceiveExpression
  | RangeExpression
  | FunctionLiteral
  | LambdaExpression
  | MatchExpression
  | SelectExpression
  | IfExpression
  | CompileTimeIfExpression
  | InExpression
  | IsExpression
  | AsExpression
  | EnumFetchExpression
  | UnpackingExpression
  | MutExpression
  | SpawnExpression
  | ReturnExpression
  | UnsafeExpression
  | CompileTimeExpression
  | BreakExpression
  | ContinueExpression
  | GotoExpression
  | SelectorExpression
  | CompileTimeSelectorExpression
  | CallExpression
  | IndexExpression
  | OrBlockExpression
  | OptionPropagationExpression
  | ResultPropagationExpression
  | TypeInitializer
  | ListExpression
  | Block
  | TupleLiteral
  | ArrayLiteral
  | MapLiteral
  | Literal
  | ParenthesesExpression
```

7.1 Qualified identifiers

A qualified identifier is an identifier qualified with a module name prefix. Both the module name and the identifier must not be blank.

```
SelectorExpression = Expression ('.' | '?.') identifier
QualifiedType      = identifier '.' identifier
```

Qualified identifier accesses an identifier in a different module, which must be imported. The identifier must be exported and declared in that module.

7.2 Selector expression

A selector expression accesses a field or method.

```
SelectorExpression = Expression '.' identifier
```

In `x.f`, identifier `f` is called the (field or method) selector, this identifier must not be blank identifier. Type of this expression is type of `f`. If `x` is a module name, `x.f` refers to the qualified identifier `x.f`.

A selector `f` may denote a field or method `f` of a type T, or it may refer to a field or method `f` of a nested embedded field of T. The number of embedded fields traversed to reach f is called its depth in T. The depth of a field or method f declared in T is zero. The depth of a field or method f declared in an embedded field A in T is the depth of f in A plus one.

7.3 Safe selector expression

Safe selector expression is the selector expression, that uses the `?.` operator instead of `.`.

```
SafeSelectorExpression = Expression '?.' identifier
```

The left expression shall have an option type. If the left side of the expression evaluates to `none`, the selector expression evaluates to `none` without evaluating the right side.

Safe selector expression has the same type as the right side of the expression wrapped in the option type.

```
fn main() {
    arr := [1, 2, 3]
    el := arr.get_or_none(5)?.hex()
    // type of el is ?string since hex() returns string
}
```

If `get_or_none` returns `none`, then `hex()` method will not be called and the result of the expression will be `none`. Otherwise, the method will be called on unwrapped value of `arr.get_or_none(5)` and after evaluation, the result will be wrapped back in the option type.

The code above is equivalent to:

```
fn main() {
    arr := [1, 2, 3]
    el_tmp := arr.get_or_none(5)
    el := if el_tmp != none { el_tmp.unwrap().hex() as ?string } else { none }
}
```

7.4 Composite literals

Composite literals construct new composite values each time they are evaluated. They consist of the type of the literal followed by a brace-bound list of keyed elements.

```
TypeInitializer     = TypeInitializerType '{' UnpackingExpression? KeyValue* '}'
TypeInitializerType = ArrayType | FixedSizeArrayType | MapType |
                      ChannelType | identifier | QualifiedType |
                      GenericType
KeyValue            = identifier ':' Expression
```

TypeInitializerType must be a struct, fixed or dynamic array, map, channel, or a type alias of one of these types. UnpackingExpression can be used only with structs.

7.4.1 For structs

A composite literal with a structure type creates a new instance of the structure and initializes its fields. Each key in the list must correspond to a field name in the structure and cannot be duplicated. The value type for the key must be assignable to the field type.

```
struct Point {
    x i32
    y i32
}

p := Point{ x: 10, y: 20 }
```

The literal shall describe all fields of the structure itself and all embedded fields being initialized that do not have a default value unless the initializer contains an UnpackingExpression.

```
struct Person {
   name string
   age  i32 = 0

   child &Person
}

p := Person{ name: "Alice" } // error, child field is required
                             // since &Person has no zero-value

p2 := Person{ child: &p }    // error, name field is required

p3 := Person{ name: "Alice", child: &p }          // ok
p3 := Person{ name: "Alice", age: 20, child: &p } // ok
```

UnpackingExpression describes unpacking values from another structure:

```
struct Point {
    x i32
    y i32
}

p := Point{ x: 10, y: 20 }
p2 := Point{ ...p, y: 50 }
```

In this case, all fields that are not explicitly described in the literal will be initialized with the values from the p structure. The code above is equivalent to the following:

```
p2 := Point{ x: p.x, y: 50 }
```

The type of the expression in unpacking expression must be identical to the type of the structure.

Taking the address of such a composite literal creates a reference to the resulting structure value.

```
ref := &Point{ x: 10, y: 20 }
mut_ref := &mut Point{ x: 10, y: 20 }
```

The value of a structure can be allocated both on the stack and on the heap.

7.4.2 For dynamic arrays

A composite literal with a dynamic array type creates a new instance of an empty dynamic array.

The next fields can be specified inside the literal:

len – specifies the length of the created array

```
a1 := []i32{len: 5} // array of zeroes with 5 elements
```

All elements are initialized with zero values, if the type does not have a zero value (for example, a reference), then the compiler should throw an error and suggest to add explicit initialization of elements via `init`, which will be discussed later.

For example, structs shall be initialized with all fields:

```
struct Point {
    x i32
    y i32
}

a1 := []Point{len: 5}                   // error: Point struct must be
                                        //        initialized
a2 := []Point{len: 5, init: || Point{}} // ok
```

cap – sets the number of pre-allocated elements

```
a1 := []i32{cap: 5} // array with zero elements but with
                    // allocated memory for 5 elements
```

This field allows allocating a certain number of elements in advance, which will allow avoiding reallocating memory when adding elements within the allocated amount:

```
mut a1 := []i32{cap: 5}
a1.push(1) // fast, no reallocation
a1.push(2) // fast, no reallocation
a1.push(3) // fast, no reallocation
a1.push(4) // fast, no reallocation
a1.push(5) // fast, no reallocation
a1.push(6) // first reallocation
```

When specifying a `len` field, the `cap` field will always be set to `len` if the passed value is less than `len`:

```
a1 := []i32{len: 5, cap: 2}
println(a1.cap()) // 5
```

init – specifies the lambda that will be called `len` times to initialize the array elements

```
a1 := []i32{len: 5, init: || 1} // [1, 1, 1, 1, 1]
```

The lambda passed to `init` must be of type `fn (usize) -> ElementType` or `fn () -> ElementType`. If the type being passed has an argument, during initialization, the index of the current element that is being initialized will be passed there.

```
a1 := []usize{len: 5, init: |i| i * 2} // [0, 2, 4, 6, 8]
```

7.4.3 For fixed arrays

A composite literal with a fixed array type creates a new instance of a fixed array.

```
a := [5]i32{} // [0, 0, 0, 0, 0]
```

All elements are initialized with zero values, if the type does not have a zero value (for example, a reference), then the compiler should throw an error and suggest to add explicit initialization of elements via `init`, which will be discussed later.

For example, structs shall be initialized with all fields:

```
struct Point {
    x i32
    y i32
}

a1 := [5]Point{}                 // error: Point struct must be initialized
a2 := [5]Point{init: || Point{}} // ok
```

The next field can be specified inside the literal:

init – specifies the lambda that will be called `len` times to initialize the array elements

```
a := [5]i32{init: || 1} // [1, 1, 1, 1, 1]
```

The lambda passed to `init` must be of type `fn (usize) -> ElementType` or `fn () -> ElementType`. If the type being passed has an argument, during initialization, the index of the current element that is being initialized will be passed there.

```
a := [5]usize{init: |i| i * 2} // [0, 2, 4, 6, 8]
```

7.4.4 For maps

A composite literal with a map type creates a new empty map instance.

```
m := map[i32]string{} // map[i32]string
```

The next field can be specified inside the literal:

cap – sets the number of pre-allocated elements

```
m := map[i32]string{cap: 5} // map[i32]string with allocated memory for 5 elements
```

This preallocated memory will lead to fewer reallocations when adding elements to the map.

7.4.5 For channels

A composite literal with a channel type creates a new instance of the channel.

```
ch := chan i32{} // chan i32
```

An additional `cap` field can be specified within the literal, which sets the size of the channel buffer.

```
ch := chan i32{cap: 5} // chan i32 with buffer size 5
```

7.5 Array literals

An array literal creates a new instance of the array and initializes its elements with the values from the list.

```
ArrayLiteral = '[' ListExpression? ']'
```

To specify a specific array type, the first element must be explicitly converted to the desired type:

```
arr1 := [1 as u8, 2, 3]          // []u8
arr2 := [1 as IntOrString, 2, 3] // []IntOrString
```

All other elements must be assignable to the type of the first element.

Array literals can be nested, thereby defining multidimensional arrays:

```
arr := [[1, 2],
        [3, 4]] // [][]i32
```

By default, an array literal creates a dynamically sized array, but it can also be used to create a fixed-size array. To do this, it must be explicitly converted to a fixed array type:

```
arr := [1, 2, 3] as [3]i32 // [3]i32
```

If the number of elements in the literal does not match the size of the array, then the compiler should throw an error.

7.5.1 Empty array literal

An empty array literal `[]` is allowed only when the type of the array can be inferred from the context. If the type cannot be inferred, then the empty array literal is invalid.

7.6 Map literals

A map literal creates a new map instance and initializes it with values from a list of key-value pairs.

```
MapLiteral  = '{' MapKeyValue* '}'
MapKeyValue = Expression ':' Expression
```

The type of the key and value is inferred from the first key-value pair:

```
mp1 := { 1: 2, 3: 4 }       // map[i32]i32
mp2 := { "hello": "world" } // map[string]string
```

The remaining key-value pairs must be assignable to the key type and value type of the first pair.

An empty map literal `{}` is not allowed and will be treated as a code block. To create an empty map, composite literal for maps should be used.

7.7 Tuple literals

A tuple literal creates a new instance of the tuple and initializes its elements with the values from the list.

```
TupleLiteral = '(' ListExpression? ')'
```

The type of the created tuple is inferred from the types of the elements:

```
t1 := (1, 2, 3) // (i32, i32, i32)
t2 := (1, "hello") // (i32, string)
```

The evaluation of a tuple literal evaluates its tuple initializers in left-to-right order.

7.8 Function literals

A function literal represents an anonymous function.

```
FunctionLiteral = 'fn' GenericParameters? Signature WhereConstraints? Block
```

Function literals can be either used immediately or assigned to a variable for further calling:

```
fn () {
    println('Hello, world!')
}()

double := fn (x i32) -> i32 {
    return x * 2
}

println(double(10)) // 20
println(double(20)) // 40
```

Function literals are closures, meaning they can access variables from a surrounding function:

```
fn make_counter() -> fn () -> i32 {
    count := 0
    return fn () -> i32 {
        count += 1
        return count
    }
}
```

In this case, the captured variable will be shared across all function calls and will live as long as there is at least one closure that references it.

The number of captured variables is unlimited.

7.9 None literals

The `none` literal is used to represent the absence of a value in Option types.

```
NoneLiteral = 'none'
```

The `none` literal is untyped and can be used only when the type can be inferred from the context:

```
fn get_none() -> ?i32 {
    return none // ok
}

a := none          // error: type of none cannot be inferred
b := none as ?i32 // ok
```

7.10 Lambda expressions

Lambda expressions are a shorthand for creating function literals. They are defined by the `|` symbol followed by a list of parameters, `|`, optional result type, and the function body as expression:

```
LambdaLiteral         = LambdaSignature Expression
LambdaSignature       = LambdaParameters ('->' Type)?
LambdaParameters      = '||' | ('|' LambdaParametersList? '|')
LambdaParametersList = LambdaParameter (',' LambdaParameter)*
LambdaParameter       = 'mut'? identifier '...'? Type?
```

Lambda expressions can be used in the same way as function literals:

```
double := |x i32| x * 2

println(double(10)) // 20
println(double(20)) // 40
```

The type of lambda parameters can be omitted if it can be inferred from the context:

```
fn take_fn(f fn (i32) -> i32) {
    println(f(10))
}

take_fn(|x| x * 2) // 20
```

If the type cannot be inferred, then it must be specified explicitly:

```
double1 := |x| x * 2 // error, type of x cannot be inferred
double2 := |x i32| x * 2 // ok
```

The return type is inferred from the body of the lambda:

```
double := |x f64| x * 2 // fn (f64) -> f64
```

7.11 "or"-block expression

"or"-block expression defines a block of code that will be executed if the expression evaluates to `none` or one of the error values in the Result type.

```
OrBlockExpression ::= Expression 'or' Block
```

The expression to the left of `or` shall be of type `?T` or `!T`, that is, be either Option or Result.

The block of code will only be executed if the expression to the left of `or` has the value `none` or one of the error values in the Result type.

```
fn get_or_none(cond bool) -> ?i32 {
    if cond {
        return 10
    }
    return none
}
```

```
fn main() {
    res1 := get_or_none(true) or { 20 } // 10
    res2 := get_or_none(false) or { 20 } // 20
}
```

If or-block is used as a value, the last expression in the block must be of type `T`:

```
res := get_or_none(false) or { "hello" } // error: expected i32, got string
```

Either the block must end with terminating statements:

```
res1 := get_or_none(false) or { return }
res2 := get_or_none(false) or { panic('not found') }
```

When `or block` is used for a Result type, inside the block there is an implicit variable `err`, which contains the error value:

```
fn get_or_error(cond bool) -> !i32 {
    if cond {
        return 10
    }
    return error('not found')
}

fn main() {
    res1 := get_or_error(false) or {
        println(err.msg())
        //      ^^^ implicit variable
        10
    }
}
```

7.12 “if” expression

The “if” expression specifies the conditional execution of two branches according to the value of the boolean expression. If the boolean expression is true, then the first branch (true branch) is executed, otherwise, the second branch “else” (false branch) is executed (if it exists).

```
IfExpression = 'if' Condition Block ElseBranch?
Condition    = VarDeclaration | Expression
ElseBranch   = IfExpression | ('else' Block)
```

Condition expression shall have a type `bool`:

Unlike some other languages, `if` is an expression, not a statement. When used as an expression, `if` must have both branches, and the types of both branches must have LUB (Least Upper Bound) type.

The type of each branch is determined by the type of the last expression within the branch. If the branch ends with a `return`, `break` and `continue` type is `never`.

When `if` is used as an expression, each branch shall end with an expression.

Example of a valid `if` expression:

```
a := if true {
    1
} else {
    2
}
```

Example of `if-else-if` expression:

```
a := if true {
    1
} else if false {
    2
} else {
    3
}
```

> Note that when used as an expression, `if` has the highest precedence so the following code is valid:
>
> ```
> a := if true { 1 } else { 2 }.str()
> ```
>
> And it will be equivalent to:
>
> ```
> a := (if true { 1 } else { 2 }).str()
> ```

When using `if` as a statement there are no restrictions on branch types, and also the `else` branch can be omitted:

```
if true {
    println('Hello')
}
```

7.13 "if" unwrapping expression

`if` expression has special syntax for working with Option/Result types.

If expression of VarDeclaration evaluates to non-`none`/non-error value, then the first branch (non-none/non-error branch) is executed, otherwise, if there is one, the "else" branch (none/error branch) is executed.

Expression of VarDeclaration shall have an Option or Result type.

For example:

```
fn find_user() -> ?i32 { ... }

fn main() {
    user_id := if id := find_user() {
        id
    } else {
        -1
    }
}
```

If `find_user()` returns `none`, then the second branch will be executed and `user_id` will be equal to `-1`, otherwise `user_id` will be equal to the `id` variable. This variable is only visible inside an `if` expression and cannot be used outside it.

When working with `Result` types, the `else` branch contains an implicit variable `err`, which contains the error value:

```
fn eval_expr(expr string) -> !i32 { ... }

fn main() {
    expr := '1 + 2'
    eval_res := if res := eval_expr(expr) {
        res
    } else {
        println(err.msg())
        -1
    }
}
```

List of diagnostics:

- `E0021`: Missing condition in `if` expression
- `E0125`: Unnecessary parentheses for `if` condition (warning)
- `E0126`: `if` expression must have `else` branch
- `E0155`: `if` expression body must end with expression, `return` statement or `break`/`continue`
- `E0166`: `if` condition must have `bool`

7.14 "match" expression

`match` expression allows matching a value against multiple values and execute the appropriate code depending on the matching result.

A `match` expression consists of several arms, each consisting of one or more expressions against which the value or type is compared, and a block of code or expression that is executed if there is a match.

If none of the variants match, then the `else` block is executed, if any. Match expression shall have no more than one `else` branch.

```
MatchExpression         = 'match' Expression? '{' MatchArm* MatchElseArm? '}'
MatchArm                = MatchArmExpressionList '->' MatchArmBody (',' | semi)?
MatchArmExpressionList = MatchArmExpression (',' MatchArmExpression)*
MatchArmExpression     = Expression | Type
MatchArmBody           = Expression | Block
MatchElseArm           = 'else' '->' MatchArmBody (',' | semi)?
```

The type of each arm condition shall be identical to the type of the matched expression unless the arm contains a range expression. In this case, the type of the range element shall be assignable to the type of the matched.

If arms contain multiple variants, they are checked in the order they are defined, from left to right.

If a `match` expression is used as an expression, each arm shall be expression or block that ends with an expression.

The type of each branch is determined by the type of the last expression within the branch. If the branch ends with a `return`, `break` and `continue` type is `never`.

The types of all branches must have LUB (Least Upper Bound) type.

Example of a valid `match` expression:

```
b := 1
a := match b {
    1 -> 1
    2, 3, 4 -> 2
    else -> 3
}
```

With blocks:

```
b := 1
a := match b {
    1 -> {
        println('One')
        1
    }
    2 -> {
        println('Two')
        2
    }
    else -> {
        println('Three')
        3
    }
}
```

The matching expression can be omitted, in which case the expression `true` is used implicitly.

For example:

```
b := 1
a := match {
    b > 0 -> 1
    b < 0 -> -1
    else -> 0
}
```

When using `match` as an expression, the `else` branch is required, unless the branches cover all possible cases for the type of expression being matched. This is possible when matching an enum or a union type.

> Note that when used as an expression, `match` has the highest precedence so the following code is valid:
>
> ```
> a := match b {
>    1 -> 1
>    else -> 2
> }.str()
> ```
>
> And it will be equivalent to:
>
> ```
> a := (match b {
>    1 -> 1
>    else -> 2
> }).str()
> ```

When using `match` as a statement, there are no restrictions on the types of branches, and also the `else` option can be omitted:

```
match b {
    1 -> println('One')
    2 -> println('Two')
}
```

Except when matching enums or union types, in which case the compiler must throw an error if not all the enumeration or union variants have been covered and there is no `else` arm.

7.14.1 Type match expression

If match arm contains a type rather than a value, then the match is based on the type of the expression rather than its value. The type of matched expression must be an interface or a union.

Example:

```
user := get_user()
a := match user {
    Admin -> 'Admin ${user.name}'
    User -> 'User ${user.name}'
    else -> 'Unknown user type'
}
```

In every single arm with at least one type, all other expressions in that arm shall also be types. If the same code must be executed for both types and values, then two separate arms must be used.

Diagnostic list:

- `E0040`: `match` expression can have only one `else` branch
- `E0156`: Empty `match` expression (warning)
- `E0157`: `match` expression with only `else` branch is not allowed
- `E0158`: `match` expression with type arms can only be used with interface types
- `E0159`: `match` arm expressions cannot have both types and values
- `E0160`: Cannot match type `name` because it is not implemented interface from `match` expression
- `E0161`: `match` expression for enums must be exhaustive
- `E0162`: `match` expression already matches all variants of enum `E`, so `else` branch is never executed
- `E0163`: Unnecessary parentheses for `match` expression (warning)
- `E0164`: Expression matched several times in `match` expression
- `E0165`: `match` expression body must end with expression, `return` statement or `break`/`continue`

7.15 “spawn” expression

Spawn expression begins executing a function call in a separate thread of execution in the same address space.

```
SpawnExpression = 'spawn' Expression
```

The expression must be a function or method call.

The value of the function, as well as the arguments, are evaluated as usual in the calling thread; however, unlike normal execution, program execution does not wait for the end function execution. Instead, the function begins its execution independently on a different thread. When the function completes its execution, the thread is destroyed.

Spawn expression returns a special handle through which the created thread can be controlled. `join()` method can be called on this handle to wait for the completion of the spawned function and to get the result of the function. This method will block the calling thread until the function finishes executing:

```
import time

fn main() {
    h := spawn fn () -> i32 {
        time.sleep(1 * time.SECOND)
        return 42
    }()

    println(h.join().unwrap())
}
```

Which can also be written with channels:

```
import time

fn main() {
    ch := chan i32{}

    spawn fn () {
        time.sleep(1 * time.SECOND)
        ch <- 42
    }()

    println((<-ch).unwrap())
}
```

7.16 Call expressions

Call expressions invoke a function or method with a list of arguments.

```
CallExpression = Expression GenericArguments? '(' ArgumentList? ')'
ArgumentList   = Argument (',' Argument)* ','?
Argument       = Expression | NamedArgument
NamedArgument  = identifier ':' Expression
```

Given an expression f with a function type:

```
f(x1, x2, ..., xn)
```

calls the function `f` with arguments x1, x2, …, xn. Each argument passed to the function must be assignable to the corresponding parameter of the function.

Arguments can be passed by position or by name:

```
f(1, 2, 3)          // positional arguments
f(x: 1, y: 2, z: 3) // named arguments
```

```
f(1, y: 2, z: 3)    // mixed arguments
```

Named and positional arguments can be mixed, but all positional arguments shall come before named arguments.

The type of the expression is the result type of the function that is being called.

A method invocation is similar, but the method itself is specified as a selector upon a value of the receiver type for the method.

```
math.sqrt(2.0) // function call
a := 10
a.sqrt()       // method call with receiver `a`
```

If `f` is a generic function, it must be instantiated before it can be called or used as a function value.

Arguments are evaluated before the expression `f` is called. The order of evaluation is from left to right. After evaluation, they passed by value to the function and the called function starts its execution. The return value is passed by value back to the caller when the function returns.

Method call `x.method()` is valid if the method set of (the type of) `x`contains `method` and the argument list can be assigned to the parameter list of `method`.

If `x` is addressable and `&x`'s method set contains `method`, `x.m()` is shorthand for `(&x).m()`. In other words, compiler will automatically take the address of `x` and call the method on the reference.

This is called automatic (de)referencing. When the compiler sees a method call on a value, it will automatically take the address of the value and call the method on the reference. This also works for method calls on references when the method is defined on the value; in this case, the compiler will automatically dereference the reference.

7.16.1 Passing arguments to variadic parameter

If a function has a variadic parameter, within the function body, it is treated as a `[]T` array. This array may be empty if no arguments are passed to the variadic parameter, or have one or more elements if arguments are passed.

If the final argument is an assignable array of type `T`, it can be passed directly to the variadic parameter with prepending `...` to the argument:

```
fn sum(nums ...i32) -> i32 { ... }

fn main() {
    nums := [1, 2, 3]
    println(sum(...nums)) // 6
}
```

In this case, array `nums` is passed to the variadic parameter `nums` of the function `sum` without creating a new array.

7.17 Unpacking expression

Unpacking expression allows passing elements of a dynamic array to a variadic parameter of a function or method.

```
UnpackingExpression = '...' Expression
```

For example:

```
fn sum(nums ...i32) -> i32 { ... }

fn main() {
    nums := [1, 2, 3]
    println(sum(...nums)) // 6
}
```

Unpacking expression is also used in composite literals to unpack values from another structure.

```
struct Point {
    x i32
    y i32
}

fn main() {
    p := Point{ x: 10, y: 20 }
    p2 := Point{ ...p, y: 50 }
}
```

7.18 "select" expression

A "select" statement chooses which of a set of possible send or receive operations will proceed.

It looks similar to a "match" expression but with the cases all referring to communication operations.

```
SelectExpression   = 'select' '{' SelectArm* SelectElseArm? '}'
SelectArm          = SelectArmStatement '->' SelectArmBody (',' | semi)?
SelectArmStatement = VarDeclaration | AssignStatement | SendStatement | Expression
SelectArmBody      = Expression | Block
SelectElseArm      = 'else' '->' SelectArmBody (',' | semi)?
```

There can be at most one else case, and it may appear anywhere in the list of arms.

Execution of a “select” statement proceeds in several steps:

1. For all the cases in the statement, the channel operands of receive operations and the channel and right-side expressions of send statements are evaluated exactly once, in source order, upon entering the “select” statement. The result is a set of channels to receive from or send to, and the corresponding values to send. Any side effects in that evaluation will occur irrespective of which (if any) communication operation is selected to proceed. Expressions on the left side of the variable declaration or assignment are not yet evaluated.
2. If one or more of the communications can proceed, a single one that can proceed is chosen via a uniform pseudo-random selection. Otherwise, if there is an else case, that case is chosen. If there is no else case, the “select” statement blocks until at least one of the communications can proceed or, if there is a timeout case, until the timeout is reached.
3. Unless the selected case is the default case, the respective communication operation is executed.
4. If the selected case is a VarDeclaration or an assignment, the left side expressions are evaluated and the received value (or values) are assigned.
5. The statement list of the selected case is executed.

7.19 Enum fetch expression

The enum fetch expression allows referring to an enumeration variant by its name. This is the shorthand for `EnumName.variant_name` selector expression.

```
EnumFetchExpression = '.' identifier
```

Since the expression specifies only the name of the variant, the compiler must infer the context type. Context type shall be an enum type, and enum shall have a variant with the specified name.

Diagnostic list:

- `E0098`: Unknown enum variant
- `E0103`: Cannot determine a type of enum to fetch

7.20 Index expression

```
IndexExpression = Expression "[" Expression "]"
```

Index expression allows accessing an element of an array, slice, string, map, or tuple element by index. An expression in square brackets is called an index or key if the indexed expression is of type `map`.

```
Expr[Index]
```

If `Expr` is an array type:

- index `Index` must be of type `usize`
- the index must be in the range from 0 to the length of the array, not inclusive, if it goes beyond the limits, program panics
- `a[x]` returns array element `a` at index `x` with an array element type

If `Expr` is of fixed array type:

- if the index is constant, then it must be in the range from 0 to the length of the array, not inclusive, otherwise the compiler must throw an error
- otherwise, if the index is not constant and goes beyond the limits, program panics
- `a[x]` returns array element `a` at index `x` with an array element type

If `Expr` is of reference to a (fixed) array type:

- `a[x]` is equivalent to `(*a)[x]`

If `Expr` is of reference to a map type:

- `a[x]` is equivalent to `(*a)[x]`

If `Expr` is a raw pointer type:

- `a[x]` is equivalent to `*(a + x)`
- the expression must be in an unsafe block because there is an implicit dereference of the raw pointer

If `Expr` is of type string:

- if the value of the string and the index are constant, then the index must be in the range from 0 to the length of the string, not inclusive, otherwise the compiler must throw an error
- if the index goes beyond the limits, program panics
- `a[x]` returns **bytes** of string `a` at index `x` with type `u8`
- `a[x]` cannot be assigned a value because strings are immutable

7.21 Range expression

```
RangeExpression = Expression? (".." | "..=") Expression? ("step" Expression)?
```

`range` expression creates a range of values from a given starting value to an ending value with an optional step. The start and end values can be omitted, in which case they will be equal to `0` and the maximum value of the type that determines the type of the segment. The optional step can be omitted, in which case it will be defaulted to `1`.

If the `range` expression is specified with the `..=` operator, then the range will include the end value, otherwise the end value will not be included.

`range` expression constructs a `Range` structure having the following definition:

```
struct Range[T] {
    start T = 0
    end   T = T.MAX
    step  T = 1
}
```

Next definition shows how `range` expression is represented as `Range` structure:

```
2..10        == Range{ start: 2, end: 10, step: 1 }
2..=10       == Range{ start: 2, end: 10 + 1, step: 1 }
2..10 step 2 == Range{ start: 2, end: 10, step: 2 }

..10      == Range{ start: 0, end: 10, step: 1 }
1..       == Range{ start: 1, end: max_usize, step: 1 }
..        == Range{ start: 0, end: max_usize, step: 1 }
.. step 2 == Range{ start: 0, end: max_usize, step: 2 }
```

By default, the type of range elements is `i32`; to define a range with a different type, left value should be explicitly cast to the desired type:

```
for i in 0 as u8 .. 10 {
    println(i)
}
```

7.22 Code block expression

```
BlockExpression = "{" Statement* "}"
```

A block expression is an expression consisting of zero or more statements enclosed in curly braces and separated by line breaks or semicolons. These expressions are essentially equivalent to the function body in their syntax.

```
{
    println('Hello')
    println('World')
}
```

When a block is executed, all the statements inside it are executed, one after another. The last expression in the block is the result of the block's execution. A block has a last statement if and only if the last statement in the block is an expression. If the block has no statements, or the last statement is not an expression (such as a `for` loop or assignment), then the block does not have a last expression.

If a block is used as an expression and does not have a final expression, then the compiler should throw an error. If a block has a final expression, then the type of that expression will specify the type of the block, which must be compatible with the expected type:

```
mut a := 10
b := {
    a := 20
    a // block has a last expression with type i32
}
```

7.23 "return" expression

"return" expression in function `F` terminates execution of `F` and optionally provides a value that will be returned to the calling function. Any deferred functions will be executed before `F` returns.

```
ReturnExpression = 'return' Expression?
```

If the surrounding function does not have a return value, or it is specified as a **`unit`**, **`never`**, or bare Result type, then the **`return`** statement cannot have an expression. The compiler should throw an error otherwise.

```
fn foo() {
    return
}
```

If the surrounding function returns a tuple type, then when returning values, they can be listed separated by commas without parentheses:

```
fn foo() -> (i32, i32) {
    return 1, 2
}
```

This is equivalent to:

```
fn foo() -> (i32, i32) {
    return (1, 2)
}
```

Diagnostic list:

- `E0075`: Return statement with expression in function with no return type
- `E0076`: Return statement missing expression

7.24 "break" expression

"break" expression is used to terminate the execution of the nearest loop within a single function. Ending a loop means that execution will continue from the first statement immediately after the loop.

```
BreakExpression = 'break' identifier?
```

If expression has a label, that label must be bound to one of the loops that contains `break`:

```
outer: for {
    for i in 0 .. 10 {
        if i == 5 {
            break outer
        }
    }
}
```

If an expression is defined in a lambda function and the label defines a loop outside the lambda function, then the compiler should throw an error.

- `E0146`: Cannot break/continue to label since label is not directly attached to any enclosing loop
- `E0167`: Cannot break/continue to label since label is outside current function literal

7.25 "continue" expression

"continue" expression is used to skip the current iteration of the nearest loop within a single function. Skipping an iteration means that execution will continue with the first statement inside the loop.

```
ContinueExpression = 'continue' identifier?
```

If expression has a label, that label must be bound to one of the loops that contains `continue`:

```
outer: for {
    for i in 0 .. 10 {
        if i == 5 {
            continue outer
        }
    }
}
```

If an expression is defined in a lambda function and the label defines a loop outside the lambda function, then the compiler should throw an error.

Diagnostic list:

- `E0146`: Cannot break/continue to label since label is not directly attached to any enclosing loop
- `E0167`: Cannot break/continue to label since label is outside current function literal

7.26 "goto" expression

"goto" expression passes control to a label inside the current function.

```
GotoExpression = 'goto' identifier
```

TODO: rules for goto

7.27 Option propagation expression

Option propagation expression is used to propagate the `none` value up the call stack.

```
OptionPropagationExpression = Expression '?'
```

If the expression evaluates to `none`, option propagation expression will return this `none` value to the caller. Otherwise, the value of the expression will be returned.

```
fn get_user() -> ?User { ... }

fn foo() -> ?User {
    user := get_user()?
    // ...
    return User{ ... }
}
```

This code is equivalent to:

```
fn foo() -> ?User {
    user_tmp := get_user()
    if user_tmp == none {
        return none
    }
    user := user_tmp.unwrap()
    // ...
    return User{ ... }
}
```

To use option propagation expression, the return type of the enclosing function shall be an Option type.

7.28 Result propagation expression

Result propagation expression is used to propagate the error value up the call stack.

```
ResultPropagationExpression = Expression '!'
```

If the expression evaluates to an error, the result propagation expression will return this error value to the caller. Otherwise, the value of the expression will be returned.

```
fn get_user() -> !User { ... }

fn foo() -> !User {
    user := get_user()!
    // ...
    return User{ ... }
}
```

This code is equivalent to:

```
fn foo() -> !User {
    user_tmp := get_user()
    if user_tmp.is_error() {
        return user_tmp.unwrap_err()
    }
    user := user_tmp.unwrap()
    // ...
    return User{ ... }
}
```

Error propagation can be used only in functions that return a Result with an error type that is compatible with the error type of the propagated error.

- `!`, `!i32` can be propagated only when enclosing function result type has default error type Error.
- `![MyError]`, `![i32, MyError]` can be propagated both when enclosing a function result type has error type `MyError` or default error type Error.

7.29 "comptime" expression

A "comptime" expression is an expression (including code blocks) that is executed during compilation, the expression itself will be replaced by the result of the execution.

```
CompileTimeExpression = 'comptime' Expression
```

Expression shall be executable at compile time, which means that it cannot contain any runtime-specific operations, such as I/O operations, network operations, and so on.

TODO: valid nodes for compile time expression

Chapter 8

Statements

Statements are executable pieces of code that can be used in various contexts, such as functions, blocks, loops, and so on.

```
Statement =
    ExpressionStatement
  | VarDeclaration
  | AssignStatement
  | SendStatement
  | ForStatement
  | CompileTimeForStatement
  | AssertStatement
  | LabeledStatement
  | DeferStatement
  | IndDecStatement
```

8.1 Terminating statements

Terminal statements interrupt the regular execution of a block. The following statements are terminal:

- Expression statement containing a `return` or `break` expression.
- A block that ends with one of the terminal statements.
- Calling `no return` functions (including the built-in `panic` function)
- Expression statement containing If expression in which:
  - there is an `else` branch
  - both branches end with terminal statements
- For statement in which:
  - the body of the loop does not contain `break`
  - does not have a loop termination condition
  - does not have a range clause

- Expression statement containing Match expression in which:
  - there is an `else` branch
  - all branches end with terminal statements
- Label statement with an internal terminal statement

Other statements are not terminal.

8.2 Assignments

An assignment replaces the current value stored in a variable with a new value, indicated by the expression to the right of the assignment operator. Assign statement can have multiple operands to the left and right of the assignment operator.

```
AssignOp = '=' | '+=' | '-=' | '|=' | '^=' | '*=' | '/=' | '%=' | '<<=' |
           '>>=' | '>>>=' | '&=' | '&^='
AssignStatement = Expression AssignOp Expression
```

For assignments with multiple operands, the following restrictions apply:

- If there is more than one operand on the left, then there must be exactly the same number of operands on the right, or one operand of tuple type with the same number of elements.
- If there is one operand on the left, then there must be exactly one operand on the right.

Each operand on the left must be addressable, be a key in the map, or be a blank identifier (for `=` assignments only).

The operands may be in parentheses, the following examples are equivalent:

```
a, b, c = 1, 2, 3
(a, b, c) = (1, 2, 3)
```

Blank identifiers can be used as placeholders when assigning tuples:

```
a, _, c = 1, 2, 3
```

In this case, the value that should be assigned to the second operand is ignored.

If there is only an empty identifier on the right, then the expression on the right is executed, but its value is explicitly ignored.

```
_ = foo()
```

8.2.1 Operator assignments

Operator assignments are shorthand for assignment with an operator.

The expression `A op= B` is equivalent to `A = A op B`, except that `A` is evaluated only once. For example, if `A` is a function call that returns a reference, then `A` will be evaluated only once.

```
fn get_ref() -> &mut i32 { ... }

fn main() {
    get_ref() += 1
    // equivalent to
    ref := get_ref()
    ref = ref + 1
}
```

Operators can be overloaded with the following expansions:

- `A += B` is equivalent to `A.add_assign(B)`
- `A -= B` is equivalent to `A.sub_assign(B)`
- `A *= B` is equivalent to `A.mul_assign(B)`
- `A /= B` is equivalent to `A.div_assign(B)`
- `A %= B` is equivalent to `A.rem_assign(B)`

Unlike ordinary assignments, in the case of operator assignments, there must be exactly one operand on the left and right. The left argument cannot be an empty identifier.

8.3 Variable declaration

Variable declaration creates one or more variables with appropriate names and initialization expressions.

```
VarDeclaration    = VarDefinitionList ':=' Expression
VarDefinitionList = VarDefinition (',' VarDefinition)*
VarDefinition     = 'mut'? identifier
```

When creating multiple variables, the right side must have the same number of expressions, or one expression, which must be a tuple with the same number of elements.

```
a, b := 1, 2
c, d := foo()
```

A variable cannot be set without specifying its value. As with assignment, `_` can be used as a placeholder to ignore a value.

```
_, b := foo()
```

The type of each variable is determined by the type of the expression that initializes its value.

The variable being created must have a unique name in the current and all external scopes.

```
a := 100
if true {
    a := 200 // error
}
```

8.4 Labeled statement

Labeled statement is used to bind a label to a statement for later use within a `goto`, `break` or `continue` statement.

```
Label            = identifier ':'
LabeledStatement = Label Statement?
```

The label must be unique within the function, otherwise the compiler must throw an error.

```
fn main() {
    print('Hello')
    if cond {
        goto end
    }
    end: print('World')
}
```

8.5 Expression statement

Expression statement is an expression that ends with a semicolon (including an implicit one). The value of such an expression is evaluated but ignored.

```
ExpressionStatement = Expression semi
```

8.6 For statement

A “for” statement specifies the execution of a code block sequentially a specified number of times until the condition is true.

There are three different variations of the `for` statement:

8.6.1 For statement with optional condition

The simplest form of a loop, a block of code is executed as long as the condition is true.

```
for a > 10 {
    a--
}
```

The condition can be omitted, in which case `true` is implicitly used, which means the loop will run indefinitely until `break` or `return` is called.

```
for {
    a--

    if a == 0 {
        break
    }
}
```

The condition is checked before each iteration, including the first one, and must be of type `bool`.

8.6.2 For statement with for clause

A “for” statement with `ForClause` is also executed as long as the condition is true, but also allows an optional *init* and a *post* statement, such as an assignment or increment/decrement.

```
ForClause = VarDeclaration? ';' Expression? ';' ForClausePostStatement?
ForClausePostStatement = AssignStatement | IndDecStatement
```

If an init statement is not empty, it is executed exactly once before the first iteration of the loop. The post-statement is executed after each iteration before checking the condition if the body of the loop has been executed.

```
for i := 0; i < 10; i++ {
    println(i)
}
```

Although each of the three `ForClause` elements can be omitted, semicolons must be present in any case. If it only contains a condition, they can be omitted, turning the `for clause` statement into a `for` statement with an optional condition.

```
for i < 10 { ... } // same as for ; i < 10; i++ { ... }
```

Diagnostic list:

- `E0168`: Unnecessary parentheses for `for` statement (warning)

8.7 IncDec statement

Increment (`++`) and decrement (`--`) statement increment or decrement their operand by one. The operand must be addressable, that is, have an address or be an index in the map:

```
IncDecStatement = Expression ('++' | '--')
```

`a++` is equivalent to `a += 1`, `a--` is equivalent to `a -= 1`.

8.8 Defer statement

A “Defer” statement calls a function whose execution is deferred until the outer function finishes executing by calling a return statement, reaching the end of the function, or the current thread panics.

```
DeferStatement = 'defer' Expression
```

The expression must be a function or method call.

Execution of the defer statement evaluates the value of the function as well as its arguments as usual, but execution of the function itself is delayed until the outer function has finished executing. Deferred functions are executed in reverse order.

```
fn foo() {
    defer fn () {
        println("world")
    }()
    defer fn () {
        println("hello")
    }()
}

fn main() {
    foo()
    // hello
    // world
}
```

8.9 Send statement

Send statement sends a value to the channel. The type of the expression must be a channel. The type of value sent must be compatible with the type of the channel.

```
SendStatement = Channel '<-' Expression
Channel       = Expression
```

The channel and the expression being sent are evaluated before the value is sent to the channel. Sending a value to a channel blocks the current thread if the channel is not buffered or if the buffered channel is full. Sending to a closed channel leads to runtime panic.

```
ch := chan i32{}
ch <- 42
```

8.10 Assert statement

Assert statement checks the condition and panics if it is false.

```
AssertStatement = 'assert' Expression (',' Expression)?
```

The second expression is optional and is used as an error message in case of panic. It shall be of type `string`.

```
assert a > 0, 'a must be greater than 0'
```

If the condition is false, the program will panic with the error message provided or a default message. The default message consists of the condition expression and the file and line number where the panic occurred.

For example:

```
assert a > 0
```

Will panic with the message:

```
panic: assertion failed: a > 0 at test.sp:1:1
```

Chapter 9

Operators

9.1 Unary operators

```
UnaryExpr = unary_op Expression
unary_op = "+" | "-" | "!" | "~" | "*" | "&" | "&mut" | "<-"
```

9.1.1 Unary arithmetic operators

The `+`, `-` and `~` operators can only be applied to primitive types.

- Unary `+` creates a copy of the operand and returns it unchanged.
- Unary `-` returns the negative value of the operand.
- Unary `~` returns the bitwise negation of the operand. The bit negation of `x` is defined as `-(x + 1)`.

These operators cannot be overridden.

9.1.2 Logical negation operator

The `!` operator performs logical negation on an operand of type `bool` and returns a result of type `bool`. If the type of the operand is not `bool`, then the compiler should throw an error.

The operator cannot be overridden.

9.1.3 Address operators

For an operand `x` of type `T`, the `&` operator creates a reference to the value `x` of type `&T`. Using `&mut` creates a mutable reference to the value `x` of type `&mut T`. To use `&mut` the operand must be mutable.

The operand must be addressable, that is, have an address in memory, such as being a variable, a reference dereference, an index operator, or a structure field.

As an exception, the operand may be a composite structure literal.

```
a := 1
ref := &a
mut_ref := &mut a // error, a is not mutable

b := 2
ref := &mut b // mutable pointer

p := &Point { x: 1, y: 2 }
```

For an operand `x` of type `&T` or `&mut T`, the `*` operator dereferences the reference and returns a value of type `T`.

For an operand `x` of type `*T` or `*mut T`, the `*` operator dereferences the pointer and returns a value of type `T`. Such dereferencing is only allowed inside an `unsafe` block. If the pointer is null, then dereferencing will cause a panic.

```
a := 1
ref := &a
b := *ref // b == 1

c := 2
ptr := &c as *i32
d := *ptr // error, pointer dereference requires unsafe block
e := unsafe { *ptr } // ok

nil_ptr := nil as *i32
f := unsafe { *nil_ptr } // causes runtime panic
```

The operator cannot be overridden.

Diagnostic list:

- `E0064`: Pointer dereference requires unsafe block
- `E0144`: Cannot take mutable reference to immutable variable

9.1.4 Dereference operator

For an operand of type `&T` or `*T`, the `*` operator dereferences a reference or pointer and returns a value of type `T`.

For an operand with a raw pointer type, the `*` operator must be in an `unsafe` block, otherwise the compiler must throw an error.

9.1.5 Receive operator

For an operand of type channel type `chan T`, the `<-` operator takes the value from the channel and returns it with type `?T`. The expression blocks the thread of execution until a value is received from the channel. Receiving from a closed channel returns control immediately, returning the value from the channel, or `none` if the channel has no more values.

To handle errors associated with receiving from a channel, standard methods and constructs for the `Option` type are used.

```
a := <-ch or {
    println('Channel closed')
    return
}
```

9.2 Binary operators

9.2.1 Logical operators

9.2.1.1 Logical disjunction (OR)

```
OrExpression = Expression "||" Expression
```

The `||` operator performs a lazy logical OR on two expressions of type `bool`. Operator laziness means that the second expression will only be evaluated if the first expression evaluates to `false`.

Both operands must be of type `bool`, otherwise the compiler must throw an error. The logical OR expression type is `bool`.

9.2.1.2 Logical conjunction (AND)

```
AndExpression = Expression "&&" Expression
```

The `&&` operator performs a lazy logical AND on two expressions of type `bool`. Operator laziness means that the second expression will only be evaluated if the first expression evaluates to `true`.

Both operands must be of type `bool`, otherwise the compiler must throw an error. The type of the Boolean AND expression is `bool`.

9.3 Containment-checking operators

```
InExpression = Expression ("in" | "!in") Expression
```

The `in` operator checks whether a value exists in the container. The `!in` operator is the negation of the `in` operator. This operator can be overloaded with the following expansion:

- `a in b` is equivalent to `b.contains(a)`
- `a !in b` is equivalent to `!(b.contains(a))`

> **Note** Even though `a` is the first expression in the code entry, after expansion, the expression `b` will be evaluated first.

When overloading this method, it must return type `bool` otherwise the compiler must throw an error.

Example of overloaded `in` operator:

```
struct Point {
    x i32
    y i32
}

fn (p Point) contains(p2 Point) -> bool {
    return p.x == p2.x && p.y == p2.y
}
```

9.4 Type checking operators

```
IsExpression = Expression ("is" | "!is") Type
```

The `is` operator performs a type check on the expression `E` against the type `T`. The `!is` operator is the negation of the `is` operator.

The check is performed at runtime, and returns `true` if the type `E` matches the type `T`, otherwise it returns `false`.

The `E` expression must be of an interface or union type, otherwise the compiler must throw an error.

9.5 Cast operator

```
CastExpression = Expression ("as" | "as?") Type
```

The cast operator describes the binary expression `E as T`, where `E` is the expression that needs to be cast to type `T`.

The `as` cast expression is an *unchecked* cast expression that does a type cast at runtime if the runtime type `E` matches the type `T` or panics if the types do not match.

The result of an unchecked cast is always an expression of type `T`.

The `as?` cast expression is a *checked* cast expression. Like unchecked cast, checked cast performs a type cast at runtime, but in case of a type mismatch it does not panic, but returns `none`.

The result of a checked cast is always an expression of type `?T`.

If `E` is of type `U` and `U` is equivalent to `T`, that is, the expression is already of type `T`, then the compiler should throw an error if expression is outside an `unsafe` block, otherwise the cast is allowed.

9.5.1 Numeric cast

9.5.1.1 Integer cast

If the type of expression `E` is one of the integer types, and the type `T` is another integer type, then the type cast shall check at runtime that the value of `E` is within the range of type `T`, otherwise the cast fails and panics if the cast is unchecked, or returns `none` if the cast is checked.

If `E` is an integer literal, the check shall be done at compile time.

If all values of `E` is representable in `T`, then *unchecked* cast is always non-panicking.

If all values of `E` is representable in `T`, then *checked* cast is prohibited and shall be replaced with *unchecked* cast.

9.5.1.2 Float cast

Casting `f32` to `f64` will produce lossless f64 value. Casting `f64` to `f32` will produce the closest `f32` value.

This cast cannot use *checked* cast.

9.5.1.3 Float to integer cast

Casting returns the value of the floating-point number truncated towards zero.

- `NaN` will return 0.
- Values larger than the maximum integer value, including `INFINITY`, will saturate to the maximum value of the integer type.
- Values smaller than the minimum integer value, including `NEG_INFINITY`, will saturate to the minimum value of the integer type.

This cast cannot use *checked* cast.

9.5.1.4 Integer to float cast

Casting returns the closest floating-point number to the integer value.

- on overflow, infinity (of the same sign as the input) is produced, note: with the current set of numeric types, overflow can only happen on `u128 as f32` for values greater or equal to `MAX_F32 + (0.5 ULP)`

This cast cannot use *checked* cast.

9.5.2 Enum to integer cast

Casting returns the value of the variant. If a backed type of the enum is specified, the cast shall check at runtime that the value of the variant is within the range of the backing type, otherwise the cast fails and panics if the cast is unchecked, or returns `none` if the cast is checked.

9.5.3 Integer to enum cast

If the type of expression `T` is an integer type, and the type `E` is an enumeration, then the cast shall check at runtime that the value of `E` is a valid variant of the enumeration, otherwise the cast fails and panics if the cast is unchecked, or returns `none` if the cast is checked.

If `E` is an integer literal, the check shall be done at compile time.

9.5.4 Concrete type-to-interface type cast

If the expression `E` is of a concrete type and the type `T` is an interface type, then the compiler shall check at compile time that the concrete type implements the interface type `T`. This cast returns the value of `E` wrapped to the interface type `T`.

This cast cannot use *checked* cast.

9.5.5 Interface type to concrete type cast

If the expression `E` is of an interface type and the type `T` is a concrete type, then first, compiler shall check at compile time that the type `T` is implementing the interface type of `E`, and at runtime shall check that the current value stored in `E` is of type `T`.

This cast returns the value of `E` or, if current value in interface is not `E`, panics if the cast is unchecked, or returns `none` if the cast is checked.

9.5.6 Concrete type to union type cast

If the expression `E` is of a concrete type and the type `T` is a union type, then the compiler shall check at compile time that the concrete type is one of the types in the union type `T`. This cast returns the value of `E` wrapped to the union type `T`.

This cast cannot use *checked* cast.

9.5.7 Union type to concrete type cast

If the expression `E` is of a union type and the type `T` is a concrete type, then the compiler shall check at compile time that the union type `E` contains the concrete type `T` as one of its types. At runtime, the cast check that the current value stored in `E` is of type `T`.

This cast returns the value of `E` or, if the current value in the union is not `T`, panics if the cast is unchecked, or returns `none` if the cast is checked.

9.5.8 Reference to pointer type cast

If the expression `E` is of reference type and the type `T` is a raw pointer type, then the cast shall return the address of the value of `E`.

This cast is no-op and cannot use *checked* cast.

9.5.9 Pointer to integer type cast

If the expression `E` is of raw pointer type and the type `T` is an integer type, then the cast shall return the address of the value of `E` as an integer.

This cast is no-op and cannot use *checked* cast.

9.5.10 None literal cast

If the expression `E` is a `none` literal, the type `T` shall be an `Option`.

9.5.11 Nil literal cast

If the expression `E` is a `nil` literal, the type `T` shall be a raw pointer type.

9.5.12 Unsafe casts

The following casts are considered unsafe and require an `unsafe` block.

9.5.12.1 Integer to pointer type cast

If the expression `E` is of integer type and the type `T` is a raw pointer type, then the cast shall return the pointer to the address of the value of `E`.

This cast cannot use *checked* cast.

9.5.12.2 Pointer to pointer type cast

If the expression `E` is of raw pointer `*T1` type and the type `T` is a raw pointer `*T2` type, then the cast shall return the pointer to the address of the value of `E` as a pointer to the type `T2`.

9.5.12.3 Void reference to non-void reference cast

If the expression `E` is of type `&void` and the type `T` is a reference to a non-void type, then the cast shall return the reference of type `T`.

This cast cannot use *checked* cast.

Chapter 10

Attributes

An attribute is general, free-form metadata that can be attached to any declaration in a file or to the file itself. Attributes are used to provide additional information to the compiler and other tools like formatters, linters, and IDEs. The exact interpretation of attributes is up to the tools that use them.

```
Attributes          = Attribute (semi Attribute)* semi?
Attribute           = '#[' AttributeExpression ']'
AttributeExpression = AttributeName AttributeValue?
AttributeName       = identifier | 'unsafe' | SelectorExpression
AttributeValue      = '(' ArgumentList? ')'
```

The following attributes cannot be obtained via compile-time-reflection because they are removed early in the compilation:

- `enable_if`

All other attributes can be obtained through compile-time reflection.

The following attributes are reserved and have special meaning:

Code generation attributes:

- Attribute `cold`
- Attribute `hot`
- Attribute `inline`
- Attribute `no_inline`
- Attribute `track_caller`
- Attribute `c_union`
- Attribute `align`
- Attribute `thread_local`

Conditional compilation attributes:

- Attribute `enable_if`

Diagnostics attributes:

- Attribute `must_use`
- Attribute `unsafe`

Extern functions attributes:

- Attribute `link_name`
- Attribute `include` and `include_if`
- Attribute `include_path` and `include_if_path`
- Attribute `library` and `library_if`
- Attribute `library_path` and `library_path_if`
- Attribute `cflags` and `cflags_if`

ABI attributes:

- Attribute `used`
- Attribute `no_mangle`
- Attribute `export_name`

Testing attributes:

- Attribute `ignore`
- Attribute `should_panic`
- Attribute `run_if`

10.1 `cold` attribute

```
#[cold]
```

Attribute `cold` is used to mark functions that are unlikely to be called. The compiler may place such functions in a separate segment.

- Shall be applied to functions

Example:

```
#[cold]
fn foo() {}
```

10.2 `hot` attribute

```
#[hot]
```

Attribute `hot` is used to mark functions that are likely to be called frequently. The compiler may place such functions in a separate segment.

- Shall be applied to functions

Example:

```
#[hot]
fn foo() {}
```

10.3 `inline` attribute

```
#[inline]
```

Attribute `inline` is used to mark a function as a candidate for inlining. Inlining is the process of replacing call expression with the function body.

The compiler may ignore this attribute and not inline the function.

- Shall be applied to functions

Example:

```
#[inline]
fn foo() {}
```

10.4 `no_inline` attribute

```
#[no_inline]
```

Attribute `no_inline` is used to mark a function that must not be inlined. In this case, the compiler must leave all function calls in the code.

- Shall be applied to functions

Example:

```
#[no_inline]
fn foo() {}
```

10.5 `track_caller` attribute

```
#[track_caller]
```

Attribute `track_caller` allows the function to obtain a `Location` which indicates the topmost untracked caller that ultimately led to the invocation of the function.

- Shall be applied to non-extern, non-main functions

Example:

```
#[track_caller]
fn foo() {
    println(Location.caller())
}
```

```
fn bar() {
    foo()
}

fn main() {
    bar() // prints the location of the call in `bar`
}
```

10.6 `c_union` attribute

`#[c_union]`

Attribute `c_union` is used to mark a structure as a C union. The structure fields in such structs behave like union fields in C; that is, they share the same memory location with size equal to the size of the largest field.

- Shall be applied to structures

Example:

```
#[c_union]
struct MyUnion {
    a i32
    b f32
}
```

10.7 `align` attribute

`#[align]`

Attribute `align` is used to specify the alignment of the structure.

- Shall be applied to structures
- Shall have a single argument that is a power of two
- The alignment of the structure is set to the specified value

Example:

```
#[align(16)]
struct MyStruct {
    a i32
    b f32
}
```

10.8 `thread_local` attribute

`#[thread_local]`

Attribute `thread_local` is used to mark a variable as a thread-local variable. Thread-local variables are stored in a separate memory location for each thread.

- Shall be applied to variables

Example:

```
#[thread_local]
var some_tls_var = 42
```

10.9 enable_if attribute

```
#[enable_if]
```

Attribute `enable_if` is used to conditionally compile a declaration, or exclude file from compilation.

- Attribute may be applied to any declaration or file.
- Attribute shall have a single argument that is a compile-time boolean expression.

Example:

```
module foo

#[enable_if(linux)] // enable this file only on Linux

#[enable_if(amd64)] // enable this function only on amd64
fn foo() {}

#[enable_if(arm64)] // enable this function only on arm64
fn foo() {}
```

10.10 must_use attribute

```
#[must_use]
```

Attribute `must_use` is used to indicate a function whose call result must be used or explicitly ignored.

- Shall be applied to functions or methods
- The function shall have a return type that is not `unit`

Example:

```
#[must_use]
fn foo() -> i32 {
    return 42
}
```

```
fn main() {
    foo()      // error: result of `foo` must be used
    a := foo() // ok
    _ = foo()  // ok
}
```

10.11 unsafe attribute

`#[unsafe]`

Attribute `unsafe` is used to mark a function as unsafe. Unsafe functions can be called only from within an `unsafe` block.

- Shall be applied to functions

Example:

```
#[unsafe]
fn foo() {}

fn main() {
    unsafe {
        foo()
    }
}
```

10.12 link_name attribute

`#[link_name]`

Attribute `link_name` is used to specify the name of the function, struct, constant, or variable in the generated object file.

- Shall be applied to **extern** functions, constants, variables, and structures.
- Shall have a single argument that is a string literal with the correct identifier name.

Example:

```
#[link_name("add")]
extern fn _add() {}
```

10.13 include and include_if attributes

`#[include]`, `#[include_if(cond)]`

Attributes `include` and `include_if` are used to include a C header file in the current module.

- Shall be applied to the file
- `include` shall have a single argument that is a string literal with the path to the header file
- `include_if` shall have two arguments: first is a compile-time boolean expression, and the second is a string literal with the path to the header file
- Path to the header file shall be in `<>` if the file is a system header: `#[include("<stdio.h>")]`

Example:

```
module foo

#[include("<stdio.h>")]
#[include_if(windows, "<windows.h>")]
```

10.14 `include_path` and `include_path_if` attributes

`#[include_path]`, `#[include_path_if(cond)]`

Attributes `include_path` and `include_path_if` are used to add a path to the list of directories that will be searched for header files.

- Shall be applied to the file
- `include_path` shall have a single argument that is a string literal with the path to the directory
- `include_path_if` shall have two arguments: first is a compile-time boolean expression, and the second is a string literal with the path to the directory

Example:

```
module foo

#[include_path("/usr/include")]
#[include_path_if(windows, "thirdparty/lib/include")]
```

10.15 `library` and `library_if` attributes

`#[library]`, `#[library_if(cond)]`

Attributes `library` and `library_if` are used to add a library to the list of external dependencies that will be linked with the current module.

- Shall be applied to the file
- `library` shall have a single argument that is a string literal with the name of the library

- `library_if` shall have two arguments: first is a compile-time boolean expression, and the second is a string literal with the name of the library
- The name of the library shall be specified without the `-l` prefix and without the file extension: `#[library("m")]`, `#[library_if(darwin, "backtrace")]`

Example:

```
module foo

#[library("m")]
#[library_if(darwin, "backtrace")]
```

10.16 `library_path` and `library_path_if` attributes

`#[library_path]`, `#[library_path_if(cond)]`

Attributes `library_path` and `library_path_if` are used to add a path to the list of directories that will be searched for libraries.

- Shall be applied to the file
- `library_path` shall have a single argument that is a string literal with the path to the directory
- `library_path_if` shall have two arguments: first is a compile-time boolean expression, and the second is a string literal with the path to the directory

Example:

```
module foo

#[library_path("/usr/lib")]
#[library_path_if(windows, "thirdparty/lib")]
```

10.17 `cflags` and `cflags_if` attributes

`#[cflags]`, `#[cflags_if(cond)]`

Attributes `cflags` and `cflags_if` are used to add additional flags to the C compiler command line when compiling the current module.

- Shall be applied to the file
- `cflags` shall have a single argument that is a string literal with the flags
- `cflags_if` shall have two arguments: first is a compile-time boolean expression, and the second is a string literal with the flags

Example:

```
module foo

#[cflags("-DNODEBUG")]
#[cflags_if(windows, "-DWIN32")]
```

10.18 `used` attribute

```
#[used]
```

Attribute `used` is used to mark a private function or variable as used. The compiler must not remove such items from the generated object file even if they are not referenced. However, the linker is still free to remove such an item.

Note that all public functions and variables are considered used by default.

- Shall be applied to functions or variables

Example:

```
#[used]
var some_variable = 42

#[used]
fn some_function() {}
```

10.19 `no_mangle` attribute

```
#[no_mangle]
```

Attribute `no_mangle` is used to prevent the compiler from mangling the name of any top level declaration. In the resulting object file, the name of the declaration will be the same as the identifier in the source code.

Also, this declaration will be publicly exported from the object file, simular to `used` attribute.

Example:

```
#[no_mangle]
fn foo() {}
```

10.20 `export_name` attribute

```
#[export_name]
```

Attribute `export_name` is used to specify the name of the function or variable in the generated object file.

- Shall be applied to functions or variables

- Shall have a single argument that is a string literal with the correct identifier name

Example:

```
#[export_name("add")]
fn _add() {}
```

10.21 `ignore` attribute

`#[ignore]`

Attribute `ignore` is used to mark a test function that should be ignored by the test runner.

- Shall be applied to test functions
- Attribute prevents the test function from being executed

Example:

```
#[ignore]
test "this test will be ignored" {
    t.assert_eq(1, 2, "1 is not equal to 2")
}
```

10.22 `should_panic` attribute

`#[should_panic]`

Attribute `should_panic` is used to mark a test function that should panic when executed.

- Shall be applied to test functions
- May have an optional argument that is a string literal with the expected panic message
- If an attribute has an argument, the test function must panic with the same message to pass the test
- Attribute indicates that the test function should panic when executed to pass the test; otherwise the test will fail

Example:

```
#[should_panic]
test "this test should panic" {
    panic("test panic")
}

#[should_panic("oh no")]
test "this test should panic with message" {
    panic("oh no")
}
```

10.23 run_if attribute

```
#[run_if]
```

Attribute `run_if` is used to mark a test function that should be executed only if the specified condition is true.

- Shall be applied to test functions
- Shall have a single argument that is a compile-time boolean expression
- If the condition is false, the test function will be ignored

Example:

```
#[run_if(windows)]
test "this test will be executed only on Windows" {
    t.assert_eq(1, 1, "1 is equal to 1")
}
```

10.24 File attributes

File attributes are attributes that are applied to the entire file. They are placed at the beginning of the file before any other declarations, but after the module declaration and imports.

```
module foo

import bar

#[enable_if(windows)]

fn zoo() {}
```

Chapter 11

Mutability

Spawn is a language with immutability default. This means that, for example, variables cannot be changed after initialization by default.

Mutability in Spawn comes in two different forms:

- Mutability at the value (variable) level (keyword `mut`)
- Mutability at the type level (references `&mut` or raw pointers `*mut`)

Value-level mutability is essentially the ability to change the value in a particular variable/parameter. Mutability at the reference or pointer level is the ability to change the value pointed to by a reference or pointer.

Unlike variables, fields of structures or C-unions are mutable by default and cannot be made immutable.

Mutability at the type level takes precedence over mutability at the value level, meaning that a mutable variable holding an immutable reference cannot be used to change the data it points to.

The two types of variability described can occur together; consider the following example:

```
fn main() {
    mut a := 10
    mut b := &mut a
}
```

In this example, `a` is mutable at the value level, meaning that by assigning a new value to `a`, we change just the value of the variable. `b` also has value-level mutability, but also has type-level mutability because it contains a mutable reference to `a`.

In this case, the variable `b` is said to be fully mutable, since we can change both its value (assign a reference to another value) and the value to which it points.

11.1 Method mutability

Methods can be defined as mutable or immutable. A mutable method can change the data in the type it is defined for, while an immutable method cannot.

The mutability of a method is determined by the type of the receiver. Receiver with type `T` or `&T` defines an immutable method and receiver with type `&mut T` defines a mutable method.

Let's look at an example of a method that changes data in a structure:

```
struct Point {
    x i32
    y i32
}

fn (p &mut Point) set_x(x i32) {
    p.x = x
}

fn main() {
    mut p := Point{ x: 1, y: 2 }
    p.set_x(10)
}
```

In this example, the `set_x` method defines a receiver of type `&mut Point`, which means that the method can change the values of the `Point` structure pointed to by the `p` reference.

Calling such a method is valid only for variables that either store a mutable reference to `Point` or are mutable by value and store a value of type `Point`.

This method cannot be called on a variable that stores an immutable reference to `Point` even if the variable itself is mutable. Also, such a method cannot be called on a variable that is immutable by value.

```
fn main() {
    p1 := &Point{ x: 1, y: 2 }
    p1.set_x(10)
//  ^^ error, cannot call mutable method on immutable reference

    p2 := Point{ x: 1, y: 2 }
    p2.set_x(10)
//  ^^ error, cannot call mutable method on immutable value
}
```

The following case shows a situation where the variable itself is mutable, but stores an immutable reference:

```
fn main() {
    mut p := &Point{ x: 1, y: 2 }
    p.set_x(10)
}
```

In this case, calling the `set_x` method is also invalid, since although the variable is mutable, it stores an immutable reference to `Point`. Mutability in this case allows changing a reference to another but not changing the value to which it points.

When assigning to a complex expression, all its individual parts must be mutable:

```
struct Data {
    x i32
}

struct Person {
    data &Data
}

fn main() {
    common_data := &Data{}
    mut mp := { "John": Person{ data: common_data } }
    mp["John"].data.x = 10
    //         ^^^^ error, data is immutable reference
}
```

In the example above, the `data` field in the `Person` structure stores an immutable reference, so the assignment `mp["John"].data.x = 10` is invalid.

The following table shows on what values the method can be called with a particular receiver:

Receiver type	Variable type	Variable mutability
`T`	`T`	any
`T`	`&T`	any
`T`	`&mut T`	any
`&T`	`T`	any
`&T`	`&T`	any
`&mut T`	`T`	must be mutable
`&mut T`	`&mut T`	any

The following table shows when a method with a particular receiver **cannot be** called:

Receiver type	Variable type	Variable mutability
`&mut T`	`&T`	any
`&mut T`	`T`	immutable

Chapter 12

Naming conventions

There are three different naming styles that are used in the language:

- `CamelCase`
- `snake_case`
- `SCREAMING_SNAKE_CASE`

12.1 CamelCase

12.2 snake_case

12.3 SCREAMING_SNAKE_CASE

Used to name constants:

```
const (
    MAX_VALUE         = 100
    JANUARY           = 1
    SOME_BIG_CONSTANT = 1000000
)
```

Chapter 13

Compile-time reflection

Compile-time reflection is a mechanism for generating new code based on information about fields, enumeration variants, and attributes. Reflection allows getting structured access to the attributes of functions, structures, constants, fields, enumeration options, and other top-level elements.

Compile-time reflection uses `if`/`for`/other constructs preceded by the `comptime` keyword:

```
comptime if attr := foo.find_attr("get") {
    // run code if `foo` has `get` attribute
}

comptime for field in Bar.fields { // iterate over fields of `Bar`
    // ...

    comptime if field.name == "x" { // check if field name is "x"
        comptime break              // stop iteration
    }
}
```

Such constructs create new code based on the code they contain.

13.1 Get attribute from functions, structs, and constants

To get an attribute from a function, field, enum variant, structure or constant, used the `comptime if` construct and the `find_attr` method. The `find_attr` method must be called on the name of a function, structure or constant and takes the name of an attribute:

```
#[get("/")]
fn foo() {}

fn main() {
    comptime if attr := foo.find_attr("get") {
        println(attr.args[0])
    }
}
```

The `find_attr` method is defined only in compile-time context and returns a `reflection.AttributeInfo` structure which contains information about the attribute or `none` if the attribute is not found. The body of `comptime if` will only be executed if the attribute is found. Otherwise, if there is `else` branch, it will be executed instead.

13.2 Iterate over compile-time arrays

The language defines the following compile-time arrays:

- `Struct.fields` — array of structure fields
- `Struct.methods` — array of structure methods
- `Struct.attrs` — array of structure attributes
- `StructField.attrs` — array of structure field attributes
- `Function.attrs` — array of function attributes
- `Constant.attrs` — array of constant attributes
- `Enum.variants` - array of enumeration options
- `Enum.attrs` — array of enumeration attributes
- `EnumVariant.attrs` — array of enumeration variant attributes

Iterating through compile-time arrays is done using the `comptime for` statement:

```
struct Bar {
    x i32
    y i32
}

fn main() {
    comptime for field in Bar.fields {
        println(field.name)
    }
}
```

If the expression to the right of `in` is not a compile-time array, then the compiler should throw an error.

The `comptime for` statement defines `comptime break` and `comptime continue` expressions that behave similarly to regular break and continue expression, but work only within the `comptime for` statement.

```
fn main() {
    comptime for field in Bar.fields {
        comptime if field.name == "x" {
            $break
        }
    }
}
```

Their execution occurs at the moment when the execution of compile-time code during compilation reaches the corresponding instruction.

```
struct Bar {
    a i32
    x i32
    y i32
}

fn main() {
    comptime for field in Bar.fields {
        comptime if field.name == "x" {
            println("Found x field, stopping iteration")
            $break
        }

        println(field.name)
    }
}
```

Will be compiled to:

```
fn main() {
    println("a")
    println("Found x field, stopping iteration")
}
```

13.3 Compile-time selector expression

Any selector expression starting with a variable name from `comptime if`/`comptime for` will be executed at compile time. So, for example, `attr.args[0]` will be replaced with the value of the attribute argument, in this case `"/"`. The selector can only be used within a `comptime if` or `comptime for` statement.

Such selector expressions can be used in other `comptime if` conditions:

```
// ...
```

```
fn main() {
    comptime if attr := foo.find_attr("get") {
        comptime if attr.args[0] == "/" {
            println("Root path")
        }
    }
}
```

Will be compiled to:

```
// ...

fn main() {
    println("Root path")
}
```

13.4 Compile-time variables

Compile-time variables are variables whose values are determined by some compile-time expression.

Such variables can only be defined within a `comptime if` or `comptime for` statement.

Such variables serve as temporary storage for the results of compile-time evaluations and can be used in other compile-time expressions within the same `comptime if` or `comptime for` statement.

Such variables do not appear in the final code and cannot be used during program execution.

Such variables follow the normal rules for variable mutability.

```
fn main() {
    comptime if attr := foo.find_attr("get") {
        $attr_value := attr.args[0]
        comptime if $attr_value == "/" {
            println("Root path")
        }
    }
}
```

13.5 Type checking

13.5.1 reflection.Type

The `reflection.Type` type defines the type description at compile time. It contains type information and also provides the following methods:

- `is_ref()` - returns `true` if the type is a reference.
- `is_mut_ref()` - returns `true` if the type is a mutable reference.
- `is_pointer()` - returns `true` if the type is a pointer.
- `is_mut_pointer()` - returns `true` if the type is a mutable pointer.
- `is_array()` - returns `true` if the type is an array.
- `elem()` - returns the type of the element if the type is a pointer, fixed or dynamic array, or channel, otherwise the compiler should throw an error.
- `is_map()` - returns `true` if the type is a map.
- `key()` - returns the type of the key if the type is a map, otherwise the compiler should throw an error.
- `value()` - returns the value type if the type is a map, otherwise the compiler should throw an error.
- `is_fn()` - returns `true` if the type is a function.
- `arg(index usize)` - returns the type of the argument by index if the type is a function, otherwise the compiler should throw an error.
- `ret_type()` - returns the return type if the type is a function, otherwise the compiler should throw an error.
- `is_channel()` - returns `true` if the type is a channel.
- `is_interface()` - returns `true` if the type is an interface.
- `is_struct()` - returns `true` if the type is a structure.
- `is_enum()` - returns `true` if the type is an enum.
- `is_fn()` - returns `true` if the type is a function.
- `is_tuple()` - returns `true` if the type is a tuple.
- `is_option()` - returns `true` if the type is an option.
- `is_result()` - returns `true` if the type is a result.
- `is_union()` - returns `true` if the type is a union.

Methods that return `reflection.Type` can be used in contexts that expect types, for example, in generic arguments to a function call.

```
struct Bar {
    ages  []i32
    names []string
}

fn encode[T: reflection.Struct](val T) -> string {
    mut res := ""
    comptime for field in T.fields {
        comptime if field.typ.is_array() {
            comptime elem_type := field.typ.elem()
            res += encode_array[$elem_type](val.$(field.name))
            //                  ^^^^^^^^^^ will be i32 or string
        }
    }
    return res
}
```

Fields of type `reflection.Type` can be checked for existence within a specific type via the standard `is`/`!is` operator:

```
struct Bar {
    x i32
    y i32
}

fn (b Bar) foo() -> string {}

fn main() {
    comptime for field in Bar.fields {
        comptime if field.typ is i32 {
            println("Field is i32")
        }
    }

    comptime if foo_method := Bar.find_method("foo") {
        comptime if foo_method.typ is fn () -> string {
            println("Method returns string")
        }

        comptime if foo_method.ret_type is string {
            println("Method returns string")
        }
    }
}
```

13.6 Compile-time field/method selector

Compile-time field/method selector allows creating code to access a structure field or type method at compile time. The selector can only be used within a `comptime if` or `comptime for` statement.

```
CompileTimeSelectorExpression = Expression '.' '$(' Expression ')'
```

Inside `$()` there can be any compile-time expressions, including compile-time variables, if their value is of type string and the structure has a field with that name.

```
struct Bar {
    x i32
    y i32
}

fn main() {
    b := Bar{ x: 10, y: 20 }
    comptime for field in Bar.fields {
        println(b.$(field.name))
    }
}
```

Will be compiled to:

```
fn main() {
    println(b.x)
    println(b.y)
}
```

Through the selector, field values can also be changed:

```
struct Bar {
    x i32
    y i32
}

fn main() {
    mut b := Bar{ x: 10, y: 20 }
    comptime for field in Bar.fields {
        b.$(field.name) = 0
    }
    println(b.x)
}
```

Will be compiled to:

```
fn main() {
    b.x = 0
    b.y = 0
    println(b.x) // 0
}
```

Calling a method on a structure instance:

```
struct Bar {
    x i32
    y i32
}

fn (b Bar) foo() -> i32 {
    return b.x + b.y
}

fn main() {
    b := Bar{ x: 10, y: 20 }
    comptime if foo_method := Bar.find_method("foo") {
        comptime if foo_method.typ is fn () -> i32 {
            println(b.$(foo_method.name)())
        }
    }
}
```

A method cannot be called if its type is unknown, so it is necessary to check the type of the method before calling it using the `comptime if` and `is` operator. If this check is not performed, the compiler should throw an error.

The above example will be compiled to:

```
fn main() {
    b := Bar{ x: 10, y: 20 }
    println(b.foo())
}
```

13.7 Working on generic types

The compile-time arrays described above can also be obtained for generic types:

```
fn encode_struct[T: reflection.Struct](val T) -> string {
    mut res := ""
    comptime for field in T.fields {
        res += field.name + ":"
        res += val.$(field.name).str() + ", "
    }
    return res
}

struct Bar {
    x i32
    y i32
}

struct Foo {
    a i32
    b i32
}

fn main() {
    b := Bar{ x: 10, y: 20 }
    println(encode_struct(b)) // x:10, y:20

    f := Foo{ a: 30, b: 40 }
    println(encode_struct(f)) // a:30, b:40
}
```

To access some compile-time arrays on a generic type, it must define constraints:

- To access `fields` - `T: reflection.Struct`
- To access `variants` - `T: reflection.Enum`

If these restrictions are not met, the compiler should throw an error.

`attrs` and `methods` can be got from any generic type without restrictions.

Chapter 14

Built-in functions

The language provides a set of built-in functions that are always available in the program. These function names are reserved and cannot be redefined.

14.1 Print functions

`print`, `println`, `eprint`, `eprintln` functions are used to print data to the standard output and standard error. All these functions accept any number of arguments of any type. Type-to-string conversion is done using the `str()` methods.

14.2 Todo marker function

`todo` function is used to mark a function or a block of code as unfinished. If execution reaches a `todo` function, runtime panic occurs.

14.3 Unreachable marker function

`unreachable` function is used to mark a block of code as unreachable. If execution reaches an `unreachable` function, runtime panic occurs. In optimized builds, the compiler may use this information to optimize the code, reaching an `unreachable` function is undefined behavior.

Chapter 15

Modules

Spawn programs are created from separate modules. A module consists of one or more source code files, each of which defines the module's constants, variables, functions, and types. Within the module, all these definitions are freely available to each other.

Each source file consists of a module name definition followed by zero or more import declarations. Each import declaration defines a module on which the current module will have a dependency and whose definitions will be available in the current module.

List of imports followed by zero or more definitions of functions, types,

```
SourceFile = ModuleClause ImportDecl* TopLevelDecl*
```

15.1 Module clause

A module clause specifies the name of the module to which the current file will belong.

```
ModuleClause = "module" ModuleName
ModuleName   = identifier
```

ModuleName cannot be blank identifier.

A set of files with the same name forms a module. The module name must be unique within the project.

All files that define module `X` must be in a folder named `X`, otherwise the compiler must throw an error. Within a module folder, all files must define the same module, otherwise the compiler must throw an error. The exception is the `main` module, which can be defined in any folder and serve as the entry point to the program. When compiling a module as a dependency, these files will be ignored.

15.2 Exported identifiers

Each module can export one or more definitions, the exported definition will be accessible from outside the module. To export a definition, the `pub` keyword is used, which is described in the grammar for each definition.

For example, the `bar` function in the following example will be available outside the `foo` module:

```
module foo

// bar can be accessed from other modules
pub fn bar() {}
```

15.3 Import declarations

Import declaration specifies that the file containing the import depends on the functionality of the imported module and also gives access to the exported-identifiers of that module. To access them, the imported module name is used.

```
Import                      = 'import' ImportPath ImportAlias? SelectiveImportList?
ImportPath                  = identifier ('.' identifier)*
ImportAlias                 = 'as' identifier
SelectiveImportList         = '{' SelectiveImportListElements? '}'
SelectiveImportListElements = identifier (',' identifier)* ','?
```

Importing a submodule is no different from importing a module, except that when importing a submodule, the user must specify the full name of this module.

```
import foo.bar
```

When importing a module, duplicate names of modules with previously imported modules are not allowed:

```
import foo
import bar.foo // error, duplicate import with `foo` name
```

In this case, the import must specify an alias, the name through which the module symbols will be accessed:

```
import foo
import bar.foo as my_foo
```

To access module symbols, dot notation is used:

```
import mod

fn main() {
    mod.foo()
}
```

To use `foo()` without specifying a module name, when importing, user can specify a list of symbols that will be imported globally in the current file:

```
import mod { foo }

fn main() {
    foo() // ok
    mod.Bar{}
}
```

The number of symbols imported in this way is not limited. When importing the name X, if a symbol with the same name is defined in a module, the compiler should throw an error because the imported symbol will conflict with an already defined symbol in the current module.

Other symbols of an imported module are accessed through the module name.

Chapter 16

Runtime panics

Runtime errors such as out-of-bounds array access or arithmetic operations leading to overflow cause runtime panic. Panic causes the program to immediately stop and display an error message with a program trace to the panic point.

```
fn main() {
    arr := []i32{len: 10}
    println(arr[10])
}
```

Shows the following error message:

```
panic: index out of bounds, index: 10, len: 10 at main.sp:3:13

thread "main" (259):
bounds_check()
   y/builtin/array.sp:1360 at 0x10427493b
&Array[i32].get_or_panic()
   y/builtin/array.sp:587 at 0x10427679f
main.main()
   main.sp:3 at 0x1042766cf
main()
   out.c:4363 at 0x10426e717
```

Panic can be caused manually by calling the built-in `panic` function:

```
fn main() {
    panic('Something went wrong')
}
```

During the execution of function F, if a runtime panic or an explicit call to the `panic` function occurs, then function F terminates its execution. All deferred functions are executed as usual. After the deferred functions of function F are complete, all deferred functions in the function that is called function F are called, and so on up to the top function in the current thread. At this point, the program terminates and an error message is displayed. This sequence is called panicking.

16.1 Recovering panics

The `recover` built-in function allows a program to change the behavior of a panicked thread. Let's imagine that function G postpones the execution of function D:

```
fn G() {
    defer D()
}
```

Function D in turn calls the `recover` function:

```
fn D() {
    if err := recover() {
        // ...
    }
}
```

And in the current thread, panic arises. When execution reaches a call to function D, the `recover` function is called. The return value will correspond to the value that was passed to the `panic` function. If D completes normally without causing another panic, then the panic ends. In this case, the state between the call to the G function and the panic will be discarded and the program will continue to execute normally. If function G deferred functions other than D, they will be executed as usual.

The `recover` function returns none if the current thread is not panicking or `recover` was called outside a deferred function.

The following function shows the possibility of creating a wrapper function that will catch any panics and display an error message:

```
fn protect(cb fn ()) {
    defer fn () {
        if err := recover() {
            println('recovered from panic: ', err)
        }
    }()

    cb()
}
```

Chapter 17

Unsafe operations

Unsafe operations are operations that can lead to undefined behavior that is not recognized at compile time. Such operations are performed in the `unsafe` block.

```
unsafe {
    // unsafe operations
}
```

The following operations are considered unsafe:

- Raw pointer dereference
- Raw pointer indexing
- Calling a function marked with the `unsafe` attribute
- Call extern function
- Write or read from extern variable
- Write or read to a structure field with the `c_union` attribute

Chapter 18

Conditional compilation

Conditional compilation allows to include or exclude parts of code or entire files depending on compilation options or environment.

18.1 File-based conditional compilation

A file with the following name structure is considered to be a file that will be included in the build only for a specific operating system:

```
file_<os>.sp
```

Therefore, the following files will be included in the build for the corresponding operating systems:

```
file_linux.sp   // used for Linux
file_windows.sp // used for Windows
file_darwin.sp  // used for macOS
file_freebsd.sp // used for FreeBSD
```

If a file has the following name structure, then it will be included in the build for a specific architecture:

```
file_<arch>.sp
```

Therefore, the following files will be included in the build for the corresponding architectures:

```
file_x86_64.sp // used for x86_64
file_x86.sp    // used for x86
file_arm64.sp  // used for arm64
```

Only one condition can be defined in a filename; if a file contains multiple conditions, the compiler will use the last condition if it is valid.

18.2 #[enable_if] file attribute

The #[enable_if] attribute at the file level defines the condition under which the file will be included in the build:

```
module main

#[enable_if(linux && amd64)]

fn foo() {}
```

In this example, the file will be included in the build for Linux and x86_64 only.

The enable_if condition can contain any expressions evaluated at the compile-time.

18.3 #[enable_if] attribute on declarations

The #[enable_if] attribute on declarations allows them to be included or excluded from the build depending on conditions.

```
#[enable_if(debug)]
fn log(s string) {
    println(s)
}

#[enable_if(!debug)]
fn log(s string) {}
```

In this example, the log function when built in debug mode will be included with a body that displays a message on the screen; in release mode the function will be empty.

18.4 "comptime if" expression

The comptime if expression behaves the same as a usual "if" expression, however, it is evaluated at compile time.

```
CompileTimeIfExpression = 'comptime if' Condition Block CompileTimeElseBranch?
CompileTimeElseBranch   = 'else' (CompileTimeIfExpression | Block)
```

If the condition is true, then the "if" body will be placed in the current block of code, otherwise it will be skipped. If the "if" expression contains an "else" block and the condition is false, then the "else" block will be placed in the current block of code.

```
fn foo() {
    comptime if linux {
        println('Linux')
    } else {
        println('Not Linux')
    }

    name := comptime if linux { 'Linux' } else { 'Not Linux' }
    println(name)
}
```

On Linux, this function will look like this:

```
fn foo() {
    println('Linux')
    name := 'Linux'
    println(name)
}
```

On other operating systems:

```
fn foo() {
    println('Not Linux')
    name := 'Not Linux'
    println(name)
}
```

Chapter 19

Program initialization and execution

19.1 The zero value

During initialization of a fixed array, or array with a given length, if no value is specified for an element, the element will be assigned the default value for its type. This value is called zero-value.

The following types have zero-value:

- `bool` — `false`
- `i8-i128`, `u8–u128` — `0`
- `f32`, `f64` — `0.0`
- `rune` — `0`
- `string` — `""`
- `*T`, `*mut T` — `nil`
- Array type — empty array
- Fixed array type — array with zero-value elements
- Map types — empty map
- Enum types — first option
- Channel types — empty channel
- Tuple types — tuple with zero-value elements
- Unit type — `()`

The following fields do not have zero-value and must be initialized explicitly:

- `&T`, `&mut T`
- function types
- interface types
- union types

- struct types

If the field is not explicitly initialized or a value for the element is not provided when initializing the array, the compiler should throw an error.

```
struct Foo {
    ref &i32
}

fn main() {
    f := Foo{} // error, field `ref` must be initialized
    // Foo{ ref: &num }

    arr := [10]&i32{} // error, no init provided for array element
    // [10]&i32{init: || &num}
}
```

The #[required] attribute specify that a field must be initialized even if its type has a zero-value:

```
struct Foo {
    #[required]
    num i32
}

fn main() {
    f := Foo{} // error, field `num` must be initialized
    // Foo{ num: 0 }
}
```

19.2 Module initialization

Within one module, initialization of module variables and constants occurs in steps. Each step selects the earliest variables and constants that have no dependencies on other variables or constants or have dependencies only on variables and constants that have already been initialized.

For initialization, a graph of variable dependencies and constants is built, since constants and variables differ little within the framework of initialization, they are processed together in a single graph. The vertices of the graph are variables and constants, and the edges are the dependencies between them. If a vertex has no outgoing edges, then it has no dependencies on other variables and constants and can be initialized first. Next, variables are initialized that have dependencies only on already initialized variables and constants. The process continues until all variables and constants have been initialized.

If a circular dependency occurs during initialization of a variable or constant, the compiler should throw an error.

19.2.1 init function

A module can define a special function `init()`, it must be private and have no arguments or return value.

```
fn init() {}
```

A module and a file can contain multiple `init()` functions. Although `init` is not a defined identifier, it cannot be used to define the `init()` function for purposes other than describing module initialization.

Package initialization is completed after all module variables have been initialized, and all `init()` functions have been called in the order in which they are defined within one file (if the file contains several functions) and in the order in which the module files were given to the compiler.

19.3 Program initialization

All modules on which the `main` module depends are initialized in steps. If a module imports other modules, they are initialized first. If a module is imported multiple times, it is initialized only once.

Initialization of all modules occurs within the same thread. `init()` functions can launch their own threads, but the `init()` functions are launched one after another only as each of them completes.

19.4 Program execution

A complete program is created by linking the `main` module with modules that are imported (transitively) by the `main` module. The main module must be named `main` and define a function `main`.

A `main` function is subject to the following restrictions:

- Has name `main`
- No `extern` header
- No functions parameters
- No return value
- No generic parameters
- No where clause
- Has a body

```
fn main() { ... }
```

Program execution begins with initialization followed by a call to the `main` function. The moment this function finishes executing, the program ends. Any running threads will not be awaited and will be terminated.

Chapter 20

Testing

Tests are described in test files. Test files are files that have a name (including an extension) ending with `_test.sp`. Test files can contain test functions, as well as other definitions necessary for testing.

20.1 Test declaration

A test declaration defines a function that represents a unit test. Test declarations can only be used in test files.

```
TestDeclaration = Attributes? 'test' StringLiteral Block
```

The test name is defined as a string literal, which means it can have spaces and punctuation marks.

```
test "value in range" {
    // ...
}
```

Inside each test, an implicit variable `t` is defined and has type `testing.Tester` and providing methods for testing.

```
test "value in range" {
    t.assert_eq(2 + 2, 5, "2 + 2 must be 5")
}
```

Chapter 21

Compiler intrinsics

Compiler intrinsics are functions that are built into the compiler and are available in the standard library in `intrinsics` module.

Intrinsics are used to perform operations that cannot be implemented in the language itself.

Next intrinsics shall be defined:

21.1 `size_of[T]()`

Returns the size of the type `T` in bytes.

```
println(intrinsics.size_of[i32]())  // 4
println(intrinsics.size_of[u128]()) // 16
```

21.2 `align_of[T]()`

Returns the alignment of the type `T` in bytes.

```
println(intrinsics.align_of[i32]())  // 4
println(intrinsics.align_of[u128]()) // 16
```

21.3 `offset_of[T](field_name)`

Returns the offset of the field `F` in the structure `T`.

```
struct Foo {
    x i32
    y i32
}
```

```
println(intrinsics.offset_of[Foo]("x")) // 0
println(intrinsics.offset_of[Foo]("y")) // 4
```

21.4 `likely(expr)`

Informs the compiler that the expression `expr` is likely to be true. Compiler can use this information to better arrange the generated code.

```
if intrinsics.likely(a > b) {
    // ...
}
```

21.5 `unlikely(expr)`

Informs the compiler that the expression `expr` is likely to be false. Compiler can use this information to better arrange the generated code.

```
if intrinsics.unlikely(a > b) {
    // ...
}
```

21.6 `caller_location() -> Location`

Returns the caller location of the function in which this function is called. Function shall have `#[track_caller]` attribute, otherwise return value is `Location` of the current file and line where `caller_location()` is called.

```
#[track_caller]
fn foo() {
    loc := intrinsics.caller_location()
    println(loc.file) // test.sp
    println(loc.line) // 10
}

fn main() {
    foo()
}
```

Chapter 22

Allocators

By default, Spawn uses the GC for memory management. However, in some cases, such as embedded systems, GC may not be a valid option. Also, on hot paths, GC can cause performance issues. In such cases, Spawn provides allocators as a way to manually manage memory.

Allocators can be used in conjunction with the GC to manage memory in a specific section of the code (for example, when processing a game frame), or completely replace the GC when it is disabled.

The main allocator interface is the `Allocator` interface:

```
interface Allocator {
    // alloc allocates a block of memory of the given size and
    // returns a pointer to it.
    // If the allocation fails, it panics.
    fn alloc(&mut self, size usize) -> &mut u8

    // safe_alloc allocates a block of memory of the given size
    // and returns a pointer to it.
    // If the allocation fails, it returns `none`.
    fn safe_alloc(&mut self, size usize) -> ?&mut u8

    // dealloc deallocates a block of memory that was previously
    // allocated by this allocator.
    fn dealloc(&mut self, ptr &mut void)

    // safe_dealloc deallocates a block of memory that was previously
    // allocated by this allocator.
    // Before deallocating, it checks whether the allocator owns the
    // pointer.
    // If the allocator does not own the pointer, it returns `false`
    // and does not deallocate.
    fn safe_dealloc(&mut self, ptr &mut void) -> bool

    // owns returns whether the given pointer was allocated by this
    // allocator.
    // If allocator cannot determine whether it owns the pointer,
    // it returns `.unsure`.
    fn owns(&self, ptr &void) -> OwnsResult

    // reset resets the allocator to its initial state.
    // This is useful for allocators that reuse the same memory.
    fn reset(&mut self)

    // dispose frees all memory used by the allocator.
    fn dispose(&mut self)
}
```

It defines a basic set of methods that every allocator must implement.

The main allocator used in conjunction with GC is called `GCAllocator`. This allocator is used by default for all allocations in the program. When GC is disabled, the main allocator becomes `SystemAllocator`, which uses functions such as `malloc` and `free` for allocations and deallocations.

The user can override the global allocator by defining an allocator variable with the `global_allocator` attribute:

```
#[global_allocator]
var global_alloc = MyAllocator{}
```

In this case, all allocations in the program will use `MyAllocator` instead of `GCAllocator` or `SystemAllocator`.

22.1 Local allocators

Local allocators can be used to manage memory in a specific area of code. Local allocators are passed into functions manually, meaning there are no implicit parameters that the compiler automatically adds to functions.

Consider the following function:

```
fn frame(alloc Allocator) {
    arr := []u8{len: 1024}
    // ...
}
```

If the first argument of any function is an allocator, then the compiler will automatically use it for all allocations within the function.

If the allocator is not passed as the first function argument, it will be ignored and the global allocator will be used. To tell the compiler to use an allocator as a local one, you need to use the `allocator` attribute:

```
#[allocator("alloc")]
fn frame(data []u8, alloc Allocator) {
    // ...
}
```

22.1.1 Implicit allocations

Some constructs in the language can allocate memory implicitly, for example, when converting a structure into an interface or when creating a variable by taking an address (`a := &Foo{}`). Thanks to local allocators, this is not a problem, since for all such allocations, the local allocator will be implicitly used.

22.2 Block scoped allocators

Local allocators affect all allocations within a function, but sometimes it is useful to execute part of the code with a different allocator, for example, using the `StackAllocator`, so as not to allocate memory on the heap. To do this, block attributes are used:

```
fn frame(alloc Allocator) {
    arr := []u8{len: 1024}
    // ...
    mut stack_alloc = StackAllocator.new(1024)
    #[allocator("stack_alloc")] {
        // code that uses stack allocator
    }
    stack_alloc.dispose() // do nothing, no need to free memory on stack
}
```

The `allocator` attribute tells the compiler to use the `stack_alloc` allocator instead of the `alloc` allocator inside the block.

22.3 Allocators in structs

Allocators can be used in structures to manage memory in all methods of the structure. To do this, the structure must define an allocator field and specify the `allocator` attribute for it:

```
struct Foo {
    data []u8

    #[allocator]
    alloc Allocator
}
```

Now in structure methods, all allocations will implicitly use `alloc`:

```
fn (f Foo) bar() {
    mut arr := []u8{len: 1024}
    // ...
}
```

Without the attribute, the allocator will not be used in the default structure methods. To use an allocator in certain methods, you should use the `allocator` attribute:

```
#[allocator("f.alloc")]
fn (f Foo) bar() {
    mut arr := []u8{len: 1024} // uses `f.alloc`
    // ...
}

fn (f Foo) baz() {
    mut arr := []u8{len: 1024} // uses global allocator
    // ...
}
```

An allocator from a structure can also be used as a local allocator:

```
fn (f Foo) bar() {
    mut arr := []u8{len: 1024} // uses global allocator
    #[allocator("f.alloc")] {
        // code that uses f.alloc
    }
}
```

22.4 Allocator arguments in functions

If a function takes an allocator as the last argument and has a default value equal to the global allocator, then when using GC, this argument can be omitted:

```
fn foo(a i32, b i32, alloc Allocator = global) {
    // ...
}
```

In this case, if the allocator is not passed, then the global allocator will be used, which is safe when using GC.

However, when the GC is disabled, the global allocator becomes `SystemAllocator`, which is not safe to use by default due to possible memory leaks. In this case, if a custom global allocator is not installed, the compiler will throw an error that the allocator was not passed to the function.

```
foo(1, 2) // error, allocator not passed
foo(1, 2, alloc: my_alloc) // ok
```

Thanks to this, we can be sure that when the GC is disabled, all functions that allocate memory will use a specific custom allocator.

All above also applies to structures that define a field to an allocator with a default value equal to the global allocator.