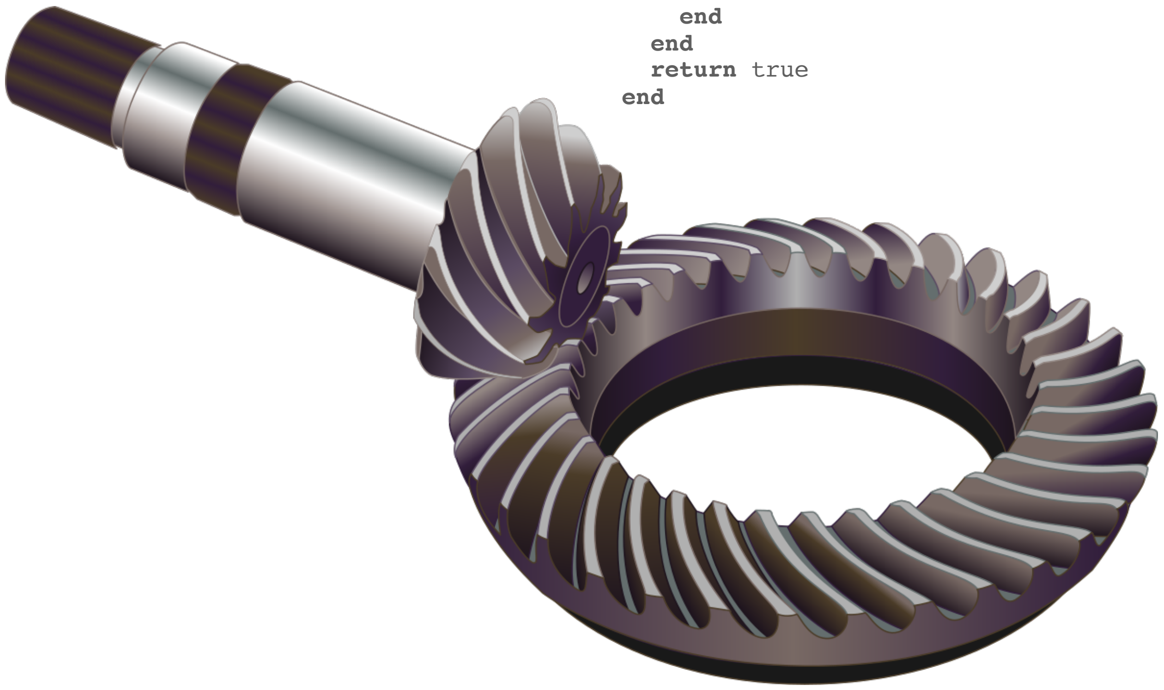
# GEAR

## The programming language

```
func isPrime(number) do
  for var i in (3..sqrt(number)).step(2) do
    if number%i = 0 then
      return false
    end
  end
  return true
end
```



```
let primes := [2] +  [n for n in (3..<10000).step(by:2) where isPrime(n)]

// array of prime numbers
```

**J. de Haan**

Gear is a so-called multi-paradigm programming language. It is imperative, functional, class oriented and partially dynamic. It supports type inference, and in fact doesn't have a static type system, but still is type-safe. Gear is case sensitive!

As a sort of tradition, every language's first program prints 'Hello world!'. In Gear, this program can be written in a single line:

```
print('Hello world!')
```

This is a full program; there's no need for a main() function or some other entry point. A program can be a simple statement.

If you want to make it more interesting you can also use:

- a regular function

```
func helloworld() do
  print('Hello world!')
end
helloWorld()
```

- a lambda expression

```
var helloWorld := lambda() do
  print('Hello world!')
end
helloWorld()
```

- a function with named parameter and string interpolation

```
func greetings(to name) do
  print('Hello $(name)!')
end
greetings(to: 'world')
```

Gear doesn't look like C-type programs, meaning there are no curly braces to create blocks of code. It aims for readability.

Semicolons are not used to separate statements from each other.

An assignment is done via the ':=' token, and testing for equality just requires a single '=' token.

Gear supports functions, records, classes, higher order functions, arrays, dictionaries, sets and much more.

Gear runs on GearVM, which (currently) is built into the compiler, so compile and run at the same time!

# TABLE OF CONTENTS

# THE BASICS
# VARIABLES & EXPRESSIONS

Gear is a general purpose language that aims for readability and simplicity, yet provides powerful contructs to create compact programs. Since it takes some of the syntax of Pascal, Swift and Python, it will be recognizable and easy to learn.

Gear uses variables and constants to store and refer to values through an identifying name. As the name implies, constants, after receiving a value cannot be changed during their lifetime.

## DECLARING VARIABLES AND CONSTANTS

In Gear a variable is declared as either a variable or a constant. Variables are declared using the keyword ‘var’, and constants are declared using ‘let’. The expression to be asigned to the variable can be any expression, from function calls to class constructors to calculations.

```
var planet := 'Neptune'
let Pi := 3.1415926
var radius := 2
var area := Pi * radius^2
var ready := false
let number := number(of:'12345') * 8
var initialValue := nil
```

Each variable declaration needs to be preceded by the keyword 'var' or 'let'. Constant and variable names can only contain the letters 'A' to 'Z', 'a' to 'z', '_' and contain digits '0' to '9' if they are not the first chracter.

You may have noticed we talk about declarations, and not so much about statements. Should we omit the keyword 'var', it would be an assignment statement, which is only allowed after declaration of the variable. Also, note that we don't use any type information. All variable types are inferred from the expression. So, under water, variable planet is of type String, Pi is a Number, as are radius and area, ready is a Boolean and number is a Number, while variable initial is nil.

After a constant gets a value other than nil, it becomes immutable and cannot be changed anymore. Variables can be changed, however, after their initialization (other than nil) they cannot change type anymore. So, if a variable is initialized with a string value, it's type remains string type.

Multiple variables can be declared in a variable declaration. Most notably, you can assign a value to multiple variables in 1 declaration.

```
var x:=1, y:=2, z:=3, s:='Saturn'
var a,b,c:=1
let u,v:=4, w:=5, p:='Pluto'
var d,e,f:=1, s1,s2:=''
var w:=0, x,y,z:=1, done:=false
```

The declarations must be separated by a comma.

## DECLARING VALUE PROPERTIES

Next to constants and variables, there's a third variable type called value, and the associated keyword is 'val'. A value variable or property is always calculated at the moment it is used. The difference with a normal variable is that the latter gets a value which remains the same until you explicitly give it a different value. The example makes things more clear. Given the following:

```
let Pi := 3.1415926
var radius := 2
var area := Pi * radius^2
print(area)  // prints 12.5663704
radius := 1
print(area)  // still prints 12.5663704
```

If we now change the value of radius, the variable area will not change because of this.

However, if we define the same code but now with a value, like this:

```
val area do
  var result := Pi * radius^2
  return result
end

print(area)  // prints 12.5663704
radius := 1  // assignment of value 1 to radius
print(area)  // now prints 3.1415926
```

Variable area is automatically recalculated, and now gets a new value.

The body of a value variable is very flexible and accepts any statement or declaration code. However, if the value variable returns a single expression, it is allowed to write

```
val area := Pi * radius^2
```

Note: It is not allowed to declare multiple values in one declaration!

Changing the value of variable radius is called an assignment. An assignment looks like a variable declaration, but doesn't have the keywords let, var or val in front of it. Assignments assign new values to variables.

So, in short, once you've declared a variable, constant or value property you can't redeclare with the same name. You cannot change it to store values of another type. Assigning new values to constants or value properties is not allowed.

## COMMENTS

A Gear comments is similar to a comment in C. Single-line comments start with two forward-slashes (//) and run unto the end of a line:

```
// This is a single line comment.
```

Multiline comments begin with a forward-slash followed by an asterisk (/*) and end with an asterisk followed by a forward-slash (*/):

```
/* This is a
multi-line comment */
```

Multi-line comments can be nested.

```
/* This is a /*nested*/
multi-line comment */
```

## NEWLINES

NewLines are vry important in Gear. Not only are they used as separator of statements and declarations, but also very important to increase readability.

Each statement or declaration needs to be followed/finalized by a NewLine. This implies that the last line of a program or module is always an empty line. It also implies that you cannot put 2 or more statements on 1 line. For example this is not allowed:

```
var hello := 'Hello' print(hello)// a declarations and a statement on 1 line
```

It will result in a compile time error. The correct way is:

```
var hello := 'Hello'
print(hello)
```

NewLines are typically not visible and are usually made from the Carriage Return (ASCII 13) and/or LineFeed (ASCII 10) characters.

Especially, in function declarations they’re important to increase readability. See the chapter on functions for more info.

## TYPES

Gear supports the regular types Number, String, Character and Boolean. The types are not explicitly available as such; they are internal to the compiler. Gear is type safe, in the sense that a variable cannot change from one type to another. A variable of type string can never change to a number.

### String and Character

A string is a sequence of characters between single quotes, which internally in Gear is represented as of type String.

```
let helloWorld := 'Hello world!'
var code := '123'
```

If you want the string to contain a single quote, like in the string 'it's me', then you need to repeat the single quote, like this 'it''s me'.

```
let label := 'it''s me'
print(label)        // prints: it's me
```

Next to the string type, Gear also supports the type Character. While strings are stored as pointers to a memory location, and can have any size, characters are stored directly on the stack and can have a maximum size of 4 bytes. This means that utf8 based characters can be used. Characters are defined with double quotes, whereas strings are declared in single quotes.

```
let emoji := "🤪"
```

An emoji is a utf8 character consisting of 4 bytes, so perfectly fits the maximum size.

Strings can be added to each other, or concatenated using the '+' operator.

```
let hi :=  'Hello ' + 'world' + "!" // 'Hello world!'
```

Note that in expressions it is possible to add two characters to each other. This always returns a string.

```
let s := "abcd" + "efgh"  // s is a string ‘abcdefgh’
```

It is also allowed to add a character to a string.

```
s := s + "i"
```

If you take the indexed character of a string, it is in fact a character.

```
let c := s[2] // c is a character, not a string
```

Next to Characters, Numbers and Booleans can be added to strings as well.

```
let hiNumber :=  'Hello ' + 42 // 'Hello 42'
let truth :=  'It is ' + true // 'It is true'
```

Sometimes adding other value types to a string can become messy and less readable.

```
let bananas := 4, strawberries := 7
let fruits := 'There are ' + bananas + ' bananas and '
  + strawberries + ' strawberries.'
```

This can be done easier by using so-called string interpolation.

```
let fruits :=
  'There are $(bananas) bananas and $(strawberries) strawberries.'
```

The values must appear in parentheses and must be preceded by a dollar $ sign. You can even make calculations within the parentheses.

```
let bowl := 'The bowl contains $(bananas + strawberries) pieces of fruit.'
```

Multiple interpolations can appear in a string however nested interpolation is not allowed. Use of functions in interpolated strings is also allowed. Strings may contain '\n' and '\t' for newlines and tabs.

```
print('Ascii code of A:\t$(ord("A")).\n')
```

The function ord() belongs to the standard available functions of Gear. It returns the ordinal value of a character.

Indexing of a string starts at 0. So, the first character of a string is string[0].

Use the range operator to copy part of a string.

```
var string := 'Hello world!'
let substring := string[6..10] // 'world', 10 inclusive
let otherstring := string[0..<5] // 'Hello', 5 not inclusive
```

The length of a string can be determined by the function length().

```
let size := length(string)
```

Gear supports creating a character from a number through the chr() function. The argument must be a number between 0 and 255, otherwise a runtime error is generated.

```
print(chr(65)) // 'A'
```

Using this function it is possible to create unicode characters such as emojis.

```
let emoji := chr(240)+chr(159)+chr(166)+chr(129)
print(emoji)
// 🦁
```

An easier way of doing this is to use the built-in character creator #. The same emoji can be created with

```
let emoji := #240#159#166#129
print(emoji)
print(#65) // 'A'
```

Also in this case the decimal number after the # must be between 0 and 255. If not, a compile error (not a runtime error) is generated. This only works for decimal numbers.

Alternatively, unicode characters can be created using hexadecimal numbers. A hexadecimal unicode numbers starts with ‘0u’ and has a maximum length of 8 hexadecimal numbers, which is 4 bytes.

```
print(0uF09FA4AA) //🤪
```

However, unicode also works with codepoints, and the above unicode number may be replaced with the following notation, using a ‘0U+’codepoint.

```
print(0U+1F92A) //🤪
```

The codepoint for this crazy smiley is 1F92A. All smiley unicode codepoints can be found here:

https://unicode.org/emoji/charts/full-emoji-list.html

Finally, it’s possible to create characters by adding unicode codepoints together.

```
print(0U+0065 + 0U+0301) // é
```

This adds the unicode codepoints of the letter e and the ’ in order to create é.

Of course there’s also a single codepoint for this:

```
print(0U+00E9) // é
```

## Number

Numbers can be written as decimal, hexadecimal, octal or binary. Hexadecimal numbers start with ‘0x’, octal numbers start with ‘0o’ and binary numbers start with ‘0b’. Consider the decimal number 27:

```
Hexadecimal:  0x1B          // 27
Octal:        0o33          // 27
Binary:       0b11011       // 27
```

print(0x1B) would print the number 27.

All numbers are internally represented as a 64 bit double precision floating point number. Decimal numbers can have the following forms:

```
-1                  integer
999                 integer
3.1415              floating point
2.1e-2 or 2.1E-2    exponent
```

## Boolean

There are of course two Boolean values ‘true’ and ‘false’. They are predefined and you need to use lowercased letters.

## Nil

The lowercased ‘nil’ is a value. You can assign nil to variables. By assigning nil to a variable, e.g. in cases that you don’t know the initial value and thus it’s type, you provide a way to use the variable as a forward declared variable. Note that nil cannot be used in calculations. This will result in a Gear runtime error.

## Tuples

A tuple is a finite ordered list (sequence) of elements. A tuple is written by listing the elements within parentheses (), and separated by commas; for example, (2, 7, 4, 1, 7) denotes a 5-tuple. In Gear, a variable can contain a tuple. Consider the following exmple:

```
let one2four := (one: 'one', two: 2, three: 'three', four: 4)
```

The items of a tuple need to have a name. Other than that the values can be anything.

You access a tuple's element by putting the name after a dot '.' and you can assign a new value to a tuple element by using its name.

```
print(one2four.three) // prints three
one2four.three := 3
```

Tuples can also be used as return values from functions.

```
func swap(left, right) do
  return (left: right, right: left)
end

let swapped := swap(8, 42)
print(swapped.left) // 42
```

In this case the variables are returned inside a tuple. By using the '.' in front of the variable, the variable becomes the name of the tuple item. In below function the variables are returned as a tuple.

```
func calculate(.scores) do
  var min := scores[0], max := scores[0]
  var sum := 0
  let num := scores.count

  for var score in scores do
    if score < min then
      min := score
    end
    if score > max then
      max := score
    end
    sum += score
  end
  let avg := sum/num

  return (.num, .min, .max, .sum, .avg) // a tuple is returned
end
```

```
let statistics := calculate(scores: [6.5, 7, 7.2, 5.9, 5.5, 6.4, 6, 5.8])

print('num: $(statistics.num)')
print('min: $(statistics.min)')
print('max: $(statistics.max)')
print('sum: $(statistics.sum)')
print('avg: $(statistics.avg)')
```

The tuple elements can be used now as members using dot syntax.

A tuple may even consist of function expressions.

```
var formula := (square: x=>x^2, root: x=>x^0.5)
print(formula.square(5))    // 25
print(formula.root(16))     // 4
```

See the chapter on functions for a detailed description.

## Printing values

The print function in Gear is very versatile. It can print any item or object. Multiple values can be printed, separated by commas. Interpolated strings can be printed as well.

```
print('Hello ', 42, ' !', [1,2,3]) // the last item is an array
print('Hello $(42)!')
```

In a standard print call automatically a newline terminates the print statement. This can be manipulated by changing the terminator of the print statement.

```
let a := 99, b := '\twhatif\t', c := 101
print(a, b, c)

print(a, b, c, terminator: '')
print(' this is on the same line')
print('this is on the next line')

for i in 1..10 do
  print(terminator: '|', i)
end
print() // empty line
```

## Formatted output

For the display of numbers, it is possible to provide an output format. The format itself is provided as a tuple with the following elements:

```
let format := (precision: ##, digits: #, width: ##, format: specifier)
```

A format is applied to a number (variable) by adding a left curved arrow '<~', followed by the format. The result is always returned as a string.

```
let format := (precision: 12, format: fmtGeneral)
print(pi()<~format) // returns the string 3.14159265359
```

You can also directly apply the format as a tuple.

```
print(pi()<~(digits: 3)) // returns the string 3.142
```

However, by using a variable holding the tuple it is possible to change a value between subsequent printing instructions.

```
format.digits := 2
print((3.12345E3)<~format) // 3123.45
```

Next to precision and digits there is the 'width' property. It's an optional property, which denotes the total space (width) the value takes. If the value is smaller than width, the result will be prepended with spaces up to the width.

```
print((pi())<~(digits: 5, width:10))
sss3.14159  first 3 spaces, since total size is 7
```

Finally, there is the format specifier property. It can be any of the values 'fmtGeneral, fmtExponent, fmtFixed, fmtNumber or fmtCurrency'. The default format is 'fmtFixed', which applies a fixed floating point number.

Some examples:

```
print((1762354)<~(precision: 10, digits: 3, format: fmtExponent))
// 1.762354000E+006
// precision  E digits

print((1762354)<~(precision: 10, digits: 3, format: fmtFixed))
// 1762354.000

print((1762354)<~(precision: 10, digits: 3, format: fmtGeneral))
// 1762354

print((1762354)<~(precision: 10, digits: 3, format: fmtNumber))
// 1,762,354.000
```

Finally, the format 'fmtCurrency'. When this format is used, the numbers are printed as currency.

```
print((1762354)<~(precision: 10, digits: 3, format: fmtCurrency))
// 1,762,354.000$ default Dollar sign added
```

You can change the default currency symbol by adding the currency property in this case.

```
print((1762354)<~( digits: 3, format: fmtCurrency, currency: '€'))
// 1,762,354.000€ Euro sign added
```

Create your standard currency format, and use it for all your book keeping:

```
let Euros := (digits: 2, format: fmtCurrency, currency: '€'))
print(amount<~Euros)
```

## EXPRESSIONS AND CALCULATIONS

### Basic arithmetic

Gear features the basic arithmetic operators you know from other languages, such as addition, subtraction, multiplication, division, modulo (or remainder), left and right number shifting and logical operations.

The following shows which operations are allowed.

Addition: a+b

```
Number + Number => Number          // 8+9 => 17
String + Number => String          // 'Hello ' + 42 => 'Hello 42'
String + String => String          // 'Hello ' + 'world' => 'Hello world'
String + Character => String       // 'Hello ' + "🥳" => 'Hello 🥳'
String + Boolean => String         // 'It''s '  + true => 'It's true'
```

Subtraction: a-b

```
Number - Number => Number          // 8-9 = -1
```

Multiplication: a*b

```
Number * Number => Number          // 8*9 = 72
```

Division: a / b

```
Number / Number => Number          // 72/9 = 8
```

A division by zero results in a Gear runtime error.

Modulo or remainder: a%b

```
Number % Number => Number          // 72%7 => 2
```

Gear supports, only for numbers:

```
Negation:            -a            // -4
```

```
Shift left:          a << b         // 4<<2 => 16
Shift right:         a >> b         // 36>>2 => 9
Power:               a^b            // 4^3 => 64
Bitwise not:         !a             // !15 => -16
Bitwise and:         a&b            // 15 & 16 => 0
Bitwise or:          a|b            // 15 | 16 => 31
Bitwise xor:         a~b            // 15 ~ 7 => 8 (~ tilde)
```

### Logical (Boolean) operators

```
not:          not true => false,  not not true => true
and:          true and false => false
or:           true or false => true
```

### Comparison operators

```
=             a = b         a equals b
<>            a <> b        a is not equal to b
>=            a >= b        a is greater than or equal to b
>             a > b         a is greater than b
<=            a <= b        a is less than or equal b
<             a < b         a is less than b
in            a in b        a is element of b (b is a range, array or set)
not in        a not in b    a is not an element of b
is            a is b        a is instance of b (b is a class or record name)
```

Regarding the 'not in' operator, this is just an easier way of using the combination of not and in. Consider

```
let b1 := not (a in 1..10)
```

or

```
let b1 := a not in 1..10
```

The result is exactly the same, just easier to read and understand.

## Constant folding for numbers

Gear supports constant folding voor constant number expressions. If a binary expression contains both left and right a constant number, then the compiler precalculates the resulting operation and emits a constant value in the code table, instead of pushing the two values and emit an operation on them. For example

```
let a := 10 * 5
```

will be translated by the compiler in

```
let a := 50
```

Note that binary expressions are evaluated from left to right, so an expression such as

```
let b := a + 100 + 50
```

cannot be folded as

```
let b := a + 150
```

You either need to write it as

```
let b := a + (100 + 50)
```

or

```
let b := 100 + 50 + a
```

Now in both cases the constants will be folded and stored as a constant 150 on the expression stack. So, instead of this bytecode

```
constant 100
constant 50
add
```

Folding results in

```
constant 150
```

This not only saves a couple of bytes, but also the operation at runtime is faster.

Also, for unary expressions code folding takes place.

```
let value := -5
```

normally would be a 2-byte instruction:

```
constant 5
negate
```

However after folding it will result in a single constant instruction:

```
constant -5
```

Note that code folding only works on numbers!

## If-expression

If-expressions are supported one way or the other in many languages. For example in c-like languages they appear as conditional expressions with a ternary operator.

C-like language: condition ? evaluated-when-true : evaluated-when-false

Example: Y = X > 0 ? A : B

Which means that if X is greater than zero than Y becomes A otherwise Y becomes B. In order to create better readibility Gear supports the If-Expression:

```
Y := if X > 0 then A else B
```

The ‘then’ and the ‘else’ part are required.

An if-expression is to be treated as a normal expression, which means you can assign it to variables.

```
let a := 7, b := 13
let max := if a>b then a else b
print('Maximum = $(max)')
```

If-expressions should be used wisely, and preferably not concatenated in endless if-expressions, for readibility. However, with a few if-then’s, it works fine:

```
func fibonacci(n) do
  return if n=0 then 0
    else if n=1 then 1    // note ‘else if’ instead of ‘elseif’
    else fibonacci(n-1) + fibonacci(n-2)
end
```

instead of:

```
func Fibonacci(n) do
  if n = 0 then
    return 0
  elseif n = 1 then    // note ‘elseif’ instead of ‘else if’
    return 1
  else
    return Fibonacci(n-1) + Fibonacci(n-2)
  end
end
```

## True-false-expression

Gear also supports a variant of the if-expression, called the true-or-false expression. The expression checks whether a boolean expression is true or false, and returns a value (or expression) based on the truthness or falseness of the boolean expression evaluation.

```
let heads := 1
let tails  := 0
randomize()
var toss := random(2) // number between 0 and 2, excluding 2

var throw := toss = heads?
  true: 'heads thrown'
  false: 'tails thrown'

print(throw)
```

So far, the above can be achieved with the if-expression as well, so what’s new here? Well the true and false sections can be interchanged, like this:

```
var throw := toss = heads?
  false: 'tails thrown'
  true: 'heads thrown'
```

Now false and true have changed places! This makes it very flexible. It’s a variant to the ternary expression, which is far more readable and flexible!

## Match-expression

A match expression consists of 1 or more if-parts and a mandatory final else-clause. 256 if clauses are allowed.

```
match value
  if expression: result_expression
  if expression: result_expression
  if expression: result_expression
  etcetera (max 256)
  else default_expression
```

The expression after the ‘if’ is used for pattern matching. It is very flexible and can take the following forms:

```
match value
  if operator value [, value]: // operator: < = > <= => <>
  if value [, value]: // operator = applied
  if in range, array or set:
  if is class:
```

Here are a few examples to further explain.

```
let age := 35
print(match age
  if < 0: 'undefined'
  if in 0..<10: 'child'
  if in 10..<20: 'teenager'
  if in 20..<30: 'young adult'
  if in 30..<40: 'settled'
  if in 40..<65: 'middle aged'
  else 'pensionado'
)

var someCharacter := 'u'
print(someCharacter, match someCharacter
  if 'a', 'e', 'i', 'o', 'u' : ' is a vowel'
  if in {'b', 'c', 'd', 'f', 'g', 'h', 'j',
         'k', 'l', 'm', 'n', 'p', 'q', 'r',
         's', 't', 'v', 'w', 'x', 'y', 'z'} : ' is a consonant'
  else ' is not a vowel nor a consonant') // Prints "u is a vowel"
```

or alternatively:

```
let alphabet := (ord("a")..ord("z")).toSet().map(c=>chr(c)) //set "a".."z"
let vowels := {"a", "e", "i", "o", "u"}
let otherCharacter := "t"
print(otherCharacter, match otherCharacter
    if in alphabet - vowels: ' is a consonant'
    if in vowels: ' is a vowel'
    else ' is not a vowel nor a consonant'
)
```

First, we created the alphabet as a set of "a" to "z". The function ord(c) returns the ordinal value of a character. We can transform a range to a set, but since it contains numbers, we map each one back to a character again. Yes, quite cumbersome, I know ☺.

In the if-branch of the match expression we test whether the alphabet minus the vowels contains the value someCharacter.

It is also possible to test on classses. Here's a preview on classes, where the less than token '<' denotes that a class inherits from a parent class.

```
class Root is
  var name := 'Root'
end

class First < Root is
end

class Second < Root is
end

class Third < Second is
end

class Fourth is
end

var object := Second()

print(match object
  if is Root:   'Root'
  if is First:  'First'
  if is Second: 'Second'
  if is Third:  'Third'
  else 'None of the above'
)
```

Note 1. If two if-expressions have an overlap, such as the value 10 in

```
x := match 10
```

```
  if in 0..10: expr1
  if in 10..20: expr2
```

the compiler randomly picks an if-expression. This is because internally they are represented as an unordered dictionary, so it doesn't necessarily pick the first one in the program. Make sure not to overlap!

Note 2. An important thing to remember, is that if you want to test if a value is part of a set, a range or an array, the keyword 'in' must be used. In case of class instance checking the keyword 'is' is required. If the respective keywords are omitted then the comparison between value and expression will be based on '=', which will return false in such cases.

## Checking for nil, nil coalescing and nil safety

There are a number of ways to check whether a value is nil. The simplest one is using the '?' operator. Put it in front of a variable and if the value is nil, a Boolean true is returned, otherwise false.

```
if ?value then… // returns true if value is nil
```

Alternatively, the standard native function assigned() checks if a value is non nil.

```
if assigned(value) then… // returns true if value is non nil
```

Next, we have the nil coalescing operator ??, which checks if a value is nil, in which case a default value is returned.

```
var newValue := value ?? defaultValue
```

In fact this is short hand for

```
var newValue := if assigned(value) then value else defaultValue
```

However, in the latter case the variable value is evaluated twice, in contrast to using the ?? operator, where the variable is evaluated only once.

```
record Window is
  var caption := nil
  init(.caption) for self.caption
end

var newWindow := Window(caption:'Hello world')
print( newWindow.caption ?? 'Default caption' )  // Hello world

newWindow := Window()
print( newWindow.caption ?? 'Default caption' )  // Default caption
```

There's also the possibility that variable newWindow is nil. What will happen in such case?

```
newWindow := nil
print(newWindow.caption)
```

This will definitely result in a runtime error. In order to avoid a runtime error it is possible to use the so-called safety operator '?.', a question mark follwed by a dot. That will prevent the code from crashing, and it will instead return the value nil.

```
newWindow := nil
print(newWindow?.caption)  // nil
print(newWindow?.caption ?? 'Default caption')  // Default caption
```

Lastly, it is always possible to check if a variable is equal to nil.

```
if newWindow <> nil then
  if newWindow.caption <> nil then
    print(newWindow.caption)
  else
    print('Default caption')
  end
end
```

But you see the point here. Using the special operators is easier to read, and generates faster code.

# FLOW OF CONTROL

Control flow (or flow of control) is the order in which individual statements, instructions or function calls of a program are executed or evaluated. For Boolean expression evaluation Gear offers the if-then, ensure and switch statements. Loops can be created using while and for statements. A special case is the for-in loop, which is based on an iterator pattern. All loops have in common that the statement block starts with ‘do’ and ends with ‘end’.

## LOOP STATEMENTS

### While

The while-statement is used to repeat zero or more statements while a certain condition is true. This simple while loop prints the squares of numbers 1 to 10.

```
var x:=1
while x<=10 do
  print('x^2= ', x^2)
  x+=1 // the same as x := x + 1
end
print(x) // 11
```

Note that after the loop has finished, the variable 'x' is still available and has value 11. Alternatively, you can declare the variable 'x' as part of the loop, so that it's scope ends when the loop ends.

```
while var x:=1 where x<=10 do
  print('x^2= ', x^2)
  x+=1
end
print(x) // error variable “x” unknown
```

Variable 'x' is declared here as a local variable inside the while loop, and doesn't exist outside the loop anymore. Also note the condition where variable x has to adhere to. In this case the 'where' keyword is required.

Sometimes you want at least one loop to be completed. In such a case it's better to use the do-while loop.

```
var x := 9
do
  print(x, terminator: '|')
  x-=1
end while x>=0
print()
print('x=',x)
```

Result:

```
9|8|7|6|5|4|3|2|1|0|
x=-1
```

It's important to note here that the 'while' condition is outside the scope of the do-end block, which means that any variable declared inside the block cannot be used in the while-condition. So, this will not work:

```
do
  var x := 9    // variable dclared inside block
  print(x, terminator: '|')
  x-=1
end while x>=0  // Error: Undeclared variable "x".
```

## For

Much has been written about the for-statement, and there are many forms available. In fact it's actually another form of writing a while statement. In the Gear for-statements it is required to define a new loop variable that is only in scope during the for-loop. The declared variable adheres to a condition and is incremented or decremented in the iterator.

```
for var Identifier := value where Condition, Iterator do
  // do something
end
```

The for-statement is recognizable for C-type language programmers, though without the curly braces.

```
for var i := 0 where i < 100, i+=1 do
  print(i^3)
end
```

This is actually the same as the following while loop:

```
while var i := 0 where i < 100 do
  print(i^3)
  i+=1
end
```

## For in

As mentioned, a for-in loop is a special loop as it is based on iterators. For arrays, dictionaries, sets and ranges these iterators are already predefined in the compiler. They are available as standard functionality.

```
for var Identifier in Sequence [where Condition] do
  // do something
end
```

A sequence must be an iterable expression, such as an array. The where-condition is optional.

```
let scores := [8.3, 7.7, 9.0, 5.4]
var sum := 0

for var score in scores do
  sum += score
end
let average := sum/scores.count

if average>6.0 then
  print('You passed!')
elseif (average>5.0) and (average<=6.0) then
  print('You need to work harder!')
else
  print('You failed!')
end
```

The variable ‘score’ is declared inplicitly after the ‘for var’ keywords and is of the same type as an element of the ‘scores’ array. Outside the loop variable ‘score’ doesn’t exist anymore. It’s scope is the loop.

Iterating over a dictionary works more or less the same. The score variable contains both the key and value of the dictionary element. The key is available via score.key and the value is available via score.value. In fact, in this case 'score' is a tuple of (.key, .value).

```
let scores := ['Math': 8.3, 'Science': 7.7, 'English': 9.0, 'Economy': 5.4]
var sum := 0
for var score in scores do
  sum += score.value
  print('Your $(score.key) result was $(score.value).')
end
let average := sum/scores.count
print('The average is $(average).')
```

```
Your Math result was 8.3.
Your Science result was 7.7.
Your English result was 9.
Your Economy result was 5.4.
The average is 7.6.
```

Iterating over a range is done by using the dotted notation.

```
for var n in 1..10 do
  print(n^2)
end
```

Or if you don't want to include the right side of the range, use the '..<' symbol.

```
let list := [1,2,3,4,5,6,7,8,9,10]
for var n in 0..<list.count do
  print(list[n]^2)
end
```

Value list.count is not included in the range.

It's good to know that the sequence expression is only evaluated once!

### Using the where condition

You can use a where-clause in a for-in statement.

```
let list := [1,2,3,4,5,6,7,8,9,10]
for var item in list where item%2=0 do
  print(item, terminator: ' | ')
end
print()
```

The even numbers are printed: 2 | 4 | 6 | 8 | 10 |

## Loop with exit

The final loop structure, based on the loop from Ada, provides the possibilty for multiple exits, in a structured way. Of course there’s the break statement to exit early from any given while-loop or for-loop, however Gear is intentionally a language that is readable and thus its structures must be easily understandable. The basic structure of the loop with exit is as follows:

```
‘loop’ [VarDecls] ‘do’
  Block
  ‘exit when’ Condition
  Block
  ‘exit when’ Condition
  Block
  ...
‘end'
```

The first exit-when is mandatory. So, there must at least be 1 exit, otherwise there’s no escape from an infinite loop. Between the keywords ‘loop’ and ‘do’ you can declare variables that can be used inside the loop. Variables declared inside the Blocks will only have that respective block scope.

```
randomize()
loop var x := 0, y := 0 do
  x := random(10)
  print('$(x) squared', terminator: ': ')
  exit when x = 0
  y := x^2
  print(y)
  exit when y > 70
end
```

Above loop produces random squares and stops either when the random number is equal to zero or when the square is greater than 70.

```
func factorial(n) do
  var result := 1
  loop var counter := n do
    result *= counter
    counter -=1
    exit when counter=0
  end
  return result
end
print(factorial(70))
```

The block after the exit-when is optional.

## CONDITIONAL STATEMENTS

### If

The if-statement has a Boolean condition, followed by the 'then' keyword, zero or more elseif clauses, and an optional else clause. The 'end' keyword is mandatory.

```
let value := getValue()
if value<=10 then
  print('Action required')
elseif (value>10) and (value<=20) then
  print('Activity in progress')
else
  print('Activity stopped')
end
```

It is also possible to declare the variable 'value' as part of the if-statement. The scope of the variable will then be the if-statement. The 'where' clause is mandatory in such a case. You can define an if-variable using 'let' if you don't need to change it anymore or using 'var' if its value changes over time.

```
if let value:= getValue() where value<=10 then
  print('Action required')
elseif (value>10) and (value<=20) then
  print('Activity in progress')
else
  print('Activity stopped')
end
```

The scope of variable 'value' is the whole if-statement.

In the next example variable 'x' is declared as var, so its value can be changed.

```
print('gimme a number: ')
if var x := number(of: readln()) where x <> nil then
  x := x^2
  print(x)
else
  print('$(x) is not a number')
end
```

Standard native function readln() reads a string from the terminal input. Function number(of:) converts it to a numeric value. If the conversion doesn't succeed, the value nil is returned.

## Switch

Almost every programming language has a switch statement in some form. In some languages it’s called a case statement.

A switch statement can have multiple cases and requires a final default: clause. The general form of the Gear switch statement is:

```
switch expression
  case [‘in’ | ‘is’] value [, value]: statement(s)
  case [‘in’ | ‘is’] value [, value]: statement(s)
  default: statement(s)
end
```

Note that after a case clause is executed, the switch statement is finished automatically. There is no ‘break’ needed, nor can you fall through to the next case.

```
let value := 5
switch value
  case 1:
    print('one')                  // no break required!
  case 2:
    print('two')
  case 3:
    print('three')                // multiple statements
    print('right?')               // in a case
  case 4,5:                       // multiple comma separated values
    print('four or five')
  default:                        // mandatory default clause
    print('it''s the default value')
end
```

This is the standard usage we have seen in many other languages, whether it is called ‘switch’ or ‘case of’ in Pascal. What is not so common, is that it supports further pattern matching, e.g. checking if a class instance belongs to a certain class type by using ‘is’ right after ‘case’.

```
switch instance
  case is RootClass:
    print('Root')
  case is FirstClass:
    print('First')
  case is SecondClass:
    print('Second')
    print('is winning')
  case is ThirdClass, FourthClass:
    print('Third or Fourth')
  default:
    print('None of the above')
end
```

On top of this, it is also allowed to do pattern matching on arrays, sets and ranges.

```
let daughter := 15
let father := 48
let mother := 39

switch mother
  case in 0..<10: print('child')
  case in 10..<20: print('teenager')
  case in 20..<40: print('adult')
  case in 40..60: print('middle aged')
  default: print('pensionado')
end

switch father
  case in {1,2,3,4}: print('not aged')
  case in [55,56,57,58]: print('dry aged')
  case in [60,62,67]: print('old aged')
  default: print('unable to mention age')
end

switch daughter
  case in {n for n in 0..<10}: print('child')      // set
  case in 10..13: print('teenager')                // range
  case in [n for n in 14..17]: print('puberty')    // array
  default: print('thinks she knows everything')
end
```

You can check whether a value is in a certain range, part of an array or part of a set. You can even combine those in one switch statement.

## Ensure

Many languages have some form of assertion or guarding that certain conditions are met. If a condition is not met, usually an error is generated, or at least an early escape from a function is possible. We want to ensure that a certain condition is met with the **ensure** statement.

The general form of the ensure statement is:

```
ensure Boolean expression else
  statements
end
```

It’s also possible to declare a variable in the ensure statement.

```
ensure let variable := value where Boolean expression else
  statements
end
```

This number guessing game show the usage of the ensure statement.

```
let max := 100
randomize()
var number := random(max) + 1

while true do
  print('Guess a number (1 to $(max)): ', terminator: '')

  ensure var guess := number(of: readln()) where guess <> nil else
    print('That''s not a number!')
    continue
  end

  ensure guess>0 and guess<=max else
    print('That number is not in range 1 to ', max)
    continue
  end

  if guess < number then
    print('Too low.')
  elseif guess = number then
    print('You win!')
    break
  else
    print('Too high.')
  end
end
```

Within the ‘ensure’ statement you can declare a variable or a constant and add a where-clause. If the condition is met, basically nothing happens, and we continue the rest of the statements. If the condition fails, the statements in the ‘else’ block are executed. Though there is no prescription on what should be contained in the else-block, the most practical is to use a return-statement, so that early escape from the function is possible, or like in the above example a ‘continue’ statement to stop this iteration and continue with the next one. The ‘break’ statement can be used to leave a loop completely.

The variable that is declared inside the ensure statement is part of the surrounding scope; in the shown example, this is the ‘while’ scope.

Note that the variable declaration and where-clause are optional. You can also just ensure a conditon, as shown in the second ensure statement.

### Break and continue

When using a count-controlled loop to search through a table, it might be desirable to stop searching as soon as the required item is found. Some programming languages provide a statement such as break, which effect is to terminate the current loop immediately, and transfer control to the statement immediately after that loop.

Though there's always the possibility to use return from a statement, this in practice means you immediately return from the function. Also, since we support loop statements outside functions, a way to break early from a loop is welcome in some cases. Usually the break statement is part of an if-then statement, like in the following:

```
for var i:=0 where i<20, i+=1 do
  if i=10 then
    break
  end
  print(i)
end
```

**Continue** is used in order to stop the current iteration and continue with the next one. Contrary to break, where you leave the iteration, with continue you perform the next iteration. See example under the ensure statement above.

Note: break and continue do not work from the do-end-while loop.

## COMMON STATEMENTS

### Print

Gear's print statement lets you print all kinds of values, separated by comma's.

```
print('Hello world!')
print('The answer is ', 42, "!")  // separated by commas
print('The answer is $(42)!')     // interpolated string
print('This line ends with a newline ', terminator: '\n')
print('This line does not ', terminator: '')
print() // default new line
print(8*8, terminator: '!!!!\n')
```

The print statement takes expressions as arguments, separated by comma's and default it will always print a new line. It carries a parameter ‘terminator’, which accepts an expression. The default value of terminator is '\n', which is the newline character.

## Assignment

An assignment statement sets and/or re-sets the value stored in the storage location(s) denoted by a variable name; in other words, it copies a value into the variable.

Gear uses the ‘:=’ operator for plain assignments, and an assignment will be a statement, instead of an expression. An assignment can only be done to an existing variable. You cannot assign a new value to a constant.

```
a := 1
b := a + 2
a := a + 1 // which is the same as
a += 1
```

Gear supports the following assignment operators:

| operator | usage | alternative | meaning |
|---|---|---|---|
| := | a := b | | Normal assignment |
| += | a += b | a := a + b | Addition |
| -= | a -= b | a := a – b | Subtraction |
| *= | a *= b | a := a * b | Multiplication |
| /= | a /= b | a := a / b | Division |
| %= | a %= b | a := a % b | Remainder |

# FUNCTIONS LAMBDA’S AND CLOSURES

A function is declared with the ‘func’ keyword. It has a name and optional parameters (the parentheses are required). Then follow ‘do’, the statements and a final ‘end’. The general form of a function is:

```
func name(parameter, parameter, …) do
  declarations and/or statements
end
```

The defined function is called using its name and a number of arguments corresponding to the defined parameters.

```
name(argument1, argument2, …)
```

or if a return statement is defined in the function:

```
var result := name(argument1, argument2, …)
```

Note that function names are case-sensitive.

```
func hello(person, day) do
  print('Hello $(person), it''s $(day) today.')
end

hello('Jerry', today())
// Hello Jerry, it's Monday today.
```

## FEATURES OF FUNCTIONS

### NewLines

To improve readability, at certain places in the function declaration it’s required to use a NewLine. For example it’s not allowed to write the following:

```
func add(a, b) do return a + b end
```

Though it looks simple enough here, in more complex cases, the code may become unreadable. In general as a rule, after the ‘do’ a NewLine is required, as well after each statement or declaration inside a function. The correct way of writing the function is:

```
func add(a, b) do
  return a + b
end
```

The only difference is hitting the enter/return key two times on your keyboard!

### Named parameters

Sometimes it is more readable if you use parameter labels in the call. You can provide a new label or use the parameter name as the label if it has a dot (.) in front of it.

```
func hello(name person, .day) do
  print('Hello $(person), it''s $(day) today.')
end
```

Then, the call to hello() requires the labels followed by a colon.

```
hello(name: 'Jerry', day: today())
// Hello Jerry, it's Monday today.
```

Using name labels makes function identifiers more meaningful. It also plays a role in function overloading.

```
let value := numberOf('123')
```

versus

```
let value := number(of: '123')
```

## Function overloading

Function overloading is the ability to create multiple functions of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context.

In Gear the only way to overload functions is by using different alternative names (the context!) for its parameters. The way this is handled is to include the named parameters into the function identifier. So for below defined functions the three identifiers are:

- *stringOf#number*
- *stringOf#boolean*
- *stringOf#char#length*

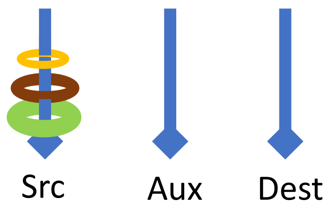
As an example, consider the following.

```
func stringOf(.number) do
  return '$(number)'
end

func stringOf(.boolean) do
  return '$(boolean)'
end

func stringOf(.char, .length) do
  var result := ''
  for var i := 0 where i < length, i+=1 do
    result += char
  end
  return result
end

var a := stringOf(number: pi())
print(a)

var c := stringOf(char: "=", length: 10)
print(c)

var b := stringOf(boolean: not(true or false))
print(b)
```

## Recursion

Functions can be recursive. This means inside the function it can call itself. The following function solves the famous Tower of Hanoi problem.

```
func tower(diskNumbers, source, auxilary, destination) do
  if diskNumbers = 1 then
    print('$(source) \t-> $(destination)')
  else
    tower(diskNumbers-1, source, destination, auxilary)
    print('$(source) \t-> $(destination)')
    tower(diskNumbers-1, auxilary, source, destination)
  end
end

tower(3, 'src', 'aux', 'dest')
```



```
src     -> dest
src     -> aux
dest    -> aux
src     -> dest
aux     -> src
aux     -> dest
src     -> dest
```

All disks have to be moved from source to destination, but you may use the auxilary as intermediate.

## Nesting

A function can be nested inside another function. The inner function then has access to variables and constants declared in the outer function.

```
func greet(name) do
  var result := name
  func makeGreeting() do
    result := 'Hello ' + name
  end
  makeGreeting()
  return result
end

print(greet('Emanuelle'))
```

## RETURNING VALUES

### Return statement

A return statement causes execution to leave the current function and to continue execution with the first statement after the call to the function. The general form of the return statement is:

```
return [expression]
```

The value of the optional expression is reported back from the function. Any possible Gear expression can be returned. If the return statement is used outside a function, a compile time error is generated.

If the expression is omitted then the value 'nil' is returned.

### Return types

A function can return any declared type. If you wish to return multiple values, it's possible to 'pack' them in a class/record, or use a tuple. But you can also return dictionaries, arrays, enums and all simple types such as strings, numbers and booleans. It's even possible to return functions.

```
func calculate(.scores) do
  record Statistics is
    var min := 0, max := 0, sum := 0
  end
  var statistics := Statistics()

  for var score in scores do
    if score > statistics.max then
      statistics.max := score
    elseif score < statistics.max then
      statistics.min := score
    end
    statistics.sum += score
  end
  return statistics
end

var statistics := calculate(scores: [6, 7, 8, 9, 5, 4, 10, 3])
print(statistics.min)
print(statistics.max)
print(statistics.sum)
```

## Returning multiple values

Functions may return multiple values using tuples. A tuple expression is returned by writing the elements between parentheses and separated by commas.

```
func calculate(.scores) do
  var min := 0, max := 0, sum := 0, avg := 0
  for var score in scores do
    if score > max then
      max := score
    elseif score < max then
      min := score
    end
    sum += score
  end
  avg := sum / scores.count
  return (.min, .max, .sum, .avg)
end

let statistics := calculate(scores: [6, 7, 8, 9, 5, 4, 10, 3])
print('min: $(statistics.min)')
print('max: $(statistics.max)')
print('sum: $(statistics.sum)')
print('avg: $(statistics.avg)')
```

The same function 'calculate' as before, however now a tuple expression is returned instead of a record. Note that the variables min, max, sum and avg have a dot (.) in front of them in the returned tuple expression. The variable names become the tuple names, which then can be used via the dot-notation, e.g. statistics.min.

## Arrow functions

Functions that return only a single expression can be written in a simpler manner using the arrow notation. The general form is:

```
func name(parameters) => expression
```

There is no need to use 'do' and 'end' in this case.

```
func add(a,b) do
  return a+b
end
```

can be rewritten as

```
func add(a,b) => a+b
```

## Defer actions

The Gear defer keyword is used to execute a block of code at a later point in time, which is just before the end of execution of the current function. The syntax to use the defer keyword is

```
defer Statement
```

or

```
defer do
  Statement
  Statement
  ...
end
```

If there are multiple defer statements in the current function scope, these defer blocks will execute in reversed order of definition.

```
func greetings() do
  print('Hello World')
  defer print('Bye now!')
  print('Welcome dude!')
  defer do
    print('Let''s meet later')
    print('Or call me')
  end
  print('Please, sit down.')
end
greetings()
```

Prints the following:

```
Hello World
Welcome dude!
Please, sit down.
Let's meet later
Or call me
Bye now!
```

So, first in, last out. Where defer really comes in handy is in cases when you have to clean up resources at the end of a function. In the following example a file is opened, processed and closed again, first without defer, then after that the solution with defer.

```
func process(.fileName) do
  let inputFile := fileOpen(fileName, forReading)
  ensure inputFile >= 0 else
    print('Error opening file "$(fileName)".')
    fileClose(inputFile)
    return nil
  end
  let fileSize := fileSeek(inputFile, 0, fromEnd)
  fileSeek(inputFile, 0, fromBeginning)
  let bytesRead := fileRead(inputFile)
  if length(bytesRead) < fileSize then
    print('Error reading file "$(fileName)".')
    fileClose(inputFile)
    return nil
  end
  fileClose(inputFile)
  return (.bytesRead, .fileSize)
end
```

You would need 3 places to call the fileClose() function. The defer statement provides a clean way to handle these types of a situation by declaring a block of statements that will be executed when execution leaves the current function.

```
func process(.fileName) do
  let inputFile := fileOpen(fileName, forReading)
  defer fileClose(inputFile)
  ensure inputFile >= 0 else
    print('Error opening file "$(fileName)".')
    return nil
  end
  let fileSize := fileSeek(inputFile, 0, fromEnd)
  fileSeek(inputFile, 0, fromBeginning)
  let bytesRead := fileRead(inputFile)
  if length(bytesRead) < fileSize then
    print('Error reading file "$(fileName)".')
    return nil
  end
  return (.bytesRead, .fileSize)
end
```

By using defer, we make sure that on returning from the function, the deferred statement is executed. It’s comparable to try-finally statements in other languages. And you only need to declare it once.

There are a number of limitations for the usage of defer. You cannot use the following statements inside a defer block: break, continue, return, use. Also, you cannot use defer inside a common block, such as a for-statement or an if-statement. It must always be part of a function scope.

## FUNCTION AS FIRST-CLASS TYPE

Gear treats functions as first-class citizens. This means Gear supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures. Gear also supports anonymous functions, within Gear known as lambda functions.

First-class functions are a necessity for the functional programming style, in which the use of higher-order functions, such as map, filter and reduce, is a standard practice.

### Returning a function

A function can return another function as its value.

```
func startAt(x) do
  func incrementBy(y) do
    return x + y
  end
  return incrementBy
end

var adder1 := startAt(1)
var adder2 := startAt(5)

print(adder1(3))    // 4
print(adder2(3))    // 8
print(startAt(7)(9)) // 16
```

### Function as argument

You can pass functions as arguments to other functions.

```
func add2Numbers(a,b) => a+b
func mul2Numbers(a,b) => a*b

func calc(x, y, function) => function(x,y)

var sum := calc(20, 22, add2Numbers)
var product := calc(2, 21, mul2Numbers)
print('Sum is: $(sum) and product is: $(product).')
// Sum is: 42 and product is: 42.
```

### Lambda functions and closures

Gear supports lambda or anonymous functions. An anonymous function (function literal, lambda abstraction, or lambda expression) is a function definition that is not bound to an identifier. In Gear they are called lambda functions and are often:

- *arguments being passed to higher-order functions, or*
- *used for constructing the result of a higher-order function that needs to return a function.*

Given this the above example can be rewritten as:

```
func calc(x, y, function) do
  return function(x,y)
end

var sum := calc(20, 22, lambda(a,b) => a+b)
var product := calc(2, 21, lambda(a,b) => a*b)
```

It's not needed to define the functions to add and multiply with a name.

In fact, the keyword 'lambda' is not even required in such cases, so you can simplify even further. In Greek, the lambda letter is represented as /\, so you can write:

```
var sum := calc(20, 22, /\(a,b) => a+b)
var product := calc(2, 21, /\(a,b) => a*b)
```

Where there's only 1 parameter in the anonymous function, the parentheses and lambda can be omitted.

```
func calc(times n, function) do
  for var k in 1..n do
    print(function(k)<~(width:3), terminator: ' ')
  end
  print()
end
calc(times: 5, x=>x^2)  // lambda or /\ and () can be omitted
calc(times: 5, x=>x^3)
calc(times: 5, x=>x^4)

  1   4   9  16  25
  1   8  27  64 125
  1  16  81 256 625
```

Using lambda functions can be handy in situations where you need to perform multiple repetitive calculations for instance. You don't have to create the function to calculate first, but you can immediately declare it in the calling function.

The following function calculates the integral over a mathematical function.

```
func integral(f, from a, to b, steps n) do
  var sum := 0
  let dt := (b-a)/n
  for var i := 0 where i<n, i+=1 do
    sum += f(a + (i + 0.5) * dt)
  end
  return sum*dt
end
```

Let us now calculate the integral over the distance 0..1 in 10,000 steps for the following functions:

1. *$f(x) = x^2 - 2x + 4$*
2. *$f(x) = x^3$*
3. *$f(x) = sqrt(1 - x^2)$, half circle*

```
print(integral(/\(x)=>x^2-2*x+4, from: 0, to: 1, steps: 10000))
print(integral(/\(x)=>x^3, from: 0, to: 1, steps: 10000))
print(integral(x=>sqrt(1-x^2), from: -1, to: 1, steps: 100000))
3.3333333325
0.24999999875
1.57079634219855, = ½pi
```

The more steps you define, the more accurate the answers will be.

## Functions as variables

In Gear variables can be declared as a function.

```
var square := /\x => x^2
print(square(16)) // 256

var helloWorld := lambda() do
  print('Hello world!')
end
helloWorld()  // Hello World!

var z := (x=>2^x)(9)  //parentheses are required here!
print(z)  // 512
```

The last one immediately executed the function, because of the argument in parens (9) behind it. Note the parenthesis around the function.

Lastly, an alternative (other than a loop) way of calculating factorials.

```
var factorial := /\n => (1..n).reduce(1, /\(x,y)=>x*y)
print(factorial(70))
```

## Lambda calculus and currying

Lambda calculus uses functions of 1 input. An ordinary function that requires two inputs, for instance the addition function x+y can be altered in such a way that it accepts 1 input and as output creates another function, that in turn accepts a single input. As an example, consider:

/\(x,y)=> x+y,

which can be rewritten as /\x=>/\y=>x+y, or even x=>y=>x+y.

This method, known as currying, transforms a function that takes multiple arguments into a chain of functions each with a single argument. See example:

```
var add := x=>y=>x+y
print(add(7)(9)) // 16
```

Note that the arguments are separated within their own parenthesis.

And here is the example from a previous paragraph but now as anonymous curried function.

```
func startAt(x) => y => x + y

var adder1 := startAt(1)
var adder2 := startAt(5)

print(adder1(3))   // 4
print(adder2(3))   // 8
print(startAt(7)(9)) // 16
```

## Higher order functions

The standard array type Array supports higher order functions, such as each, map, filter and reduce. The code of these functions can be examined in library ‘arrays.gear’.

The **each** function applies a lambda function or closure to each item in an array.

```
var numbers := [1,2,3,4,5,6,7,8,9,10]
numbers.each(
  lambda(x) do
    let squared := x^2
    print('Number $(x) squared is: $(squared).')
  end)
```

Also, the range and set types contain an each function.

```
(1..10).each(/\(x) do
    print('Number $(x) squared is: $(x^2).')
  end)
```

The higher order **map** function transforms all items in an array and returns a new array with the transformed items. The argument must be a function.

```
let squaredNumbers := numbers.map(x=>x^2)
print(squaredNumbers)
// [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The **filter** function returns a new array with items that have passed the boolean filter.

```
let evenNumbers := numbers.filter(x=>x%2=0)
print(evenNumbers)
// [2, 4, 6, 8, 10]
```

Each number when divided by 2 and which has a remainder of 0 is an even number.

The **reduce** function accepts an initial value (start value), and applies a two-parameter function on the array items, so that exactly one value is returned. Internally, the items of the array are processed recursively, and accumulated after each step.

```
let sum := numbers.reduce(0, /\(x,y)=>x+y)
print(sum) // 55
let factorial := numbers.reduce(1, /\(x,y)=>x*y)
print(factorial) // 3628800
```

## STANDARD NATIVE FUNCTIONS

Gear has predefined a number of standard functions.

| | |
|---|---|
| pi() | the constant Pi |
| sqrt(x) | square root of x, the same as x^0.5 |
| cbrt(x) | cubic root of x, the same as x^(1/3), or exp(ln(x)/3) |
| sqr(x) | square of x, the same as x^2 |
| trunc(x) | truncate a floating point value, return the integer part of x |
| round(x) | round floating point value to nearest integer number |
| abs(x) | the absolute value |
| arctan(x) | the inverse tangent of x |
| sin(x) | the sine of angle x |
| cos(x) | the co sine of angle x |
| exp(x) | the exponent of X, i.e. the number e to the power X |
| ln(x) | the natural logarithm of X. X must be positive |
| frac(x) | the fractional part of floating point value, the non integer part |
| ceil(x) | the lowest integer number greater than or equal to x |
| floor(x) | the largest integer smaller than or equal to x |
| ord(x) | the Ordinal value of a ordinal-type variable X |

| | |
|---|---|
| chr(x) | the character which has ASCII value X |
| milliseconds() | number of milliseconds since midnight 0:00 |
| date() | string with current date |
| time() | string with current time in hh:mm:ss |
| now() | string with current time in hh:mm:ss:ms |
| today() | string with current day |
| randomize() | create new random seed |
| random() | random float number between 0 and 1 |
| random(n) | random integer number between 0 and the limit n, limit not included |
| numberOf(s) | tries to convert a string to a any number, nil if not succesful. Alternative number(of:) |
| integerOf(x) | Tries to convert a float number or a string to an integer number. Alt integer(of:) |
| stringOf(n) | tries to convert a number to a string, nil if not succesful. Alt string(of:) |
| readln() | returns input from the keyboard, nil if not succesful |
| length(v) | returns length of string, array or dictionary |
| error() | generates error and halts program |
| getMethod(obj,name) | returns ‘pointer’ to class method by method’s string name (nil if not found) |
| clrScr() | clear screen in command mode |
| windowXY(x1,y1,x2,y2) | create window in command mode |

| | |
|---|---|
| gotoXY(x,y) | go to screen location in command mode |
| assigned(v) | returns true if variable is non nil |
| ?v | returns true if variable is nil |
| typeOf(v) | returns number representing the type of a variable. |
| type(of: v) | returns enum of Type representing possible value types (module typeinfo) |

The typeinfo module contains the enum Type and the function type(of:).

```
enum Type is
  Number=0,
  Boolean=1,
  String=2,
  Class=3,
  Array=4,
  Dictionary=5,
  Set=6,
  Enum=7,
  Func=8,
  Nil=9
end

func type(of value) do
  switch typeOf(value)
    case 0: return Type.Number
    case 1: return Type.Boolean
    case 2: return Type.String
    case 3: return Type.Class
    case 4: return Type.Array
    case 5: return Type.Dictionary
    case 6: return Type.Set
    case 7: return Type.Enum
    case 8: return Type.Func
    default: return Type.Nil
  end
end
```

And it is to be used as follows:

```
print(type(of: pi()))
print(type(of: 'abc'))
print(type(of: true))
if type(of: 4) = Type.Number then
  print('Yessss')
end
```

```
if type(of: false) <> Type.Number then
  print('wrong')
end

let value := false
if type(of: value) = Type.Boolean then
  print('Yes indeed')
end

let str := 'Hello'
if type(of: str) = Type.String then
  print('It sure is')
end
```

## USING MODULES

With the **use** statement you import the code from another file. The use statement should be used before the usage of the imported code. You can use multiple files and there’s no fixed location required, meaning you can put the use statement anywhere in the code, as long as it’s before code that uses it.

As an example consider:

```
print('Hello world!')

use typeinfo
print(type(of: pi()))  // Number
```

In file typeinfo.gear the functionality for the use of the type(of:) function is defined.

The parser looks in two locations for files: first it searches in the current folder, and if not found it searches in the folder /gearlib/. If also not found there, an error is generated.

**Note:** It’s not needed to import via the use clause arrays, sets and ranges, as they are automatically included in the compiler. The functionality in these libraries is thus always available! However, you can create your own libraries with extensions on the respective types.

# COLLECTIONS ARRAYS, SETS, DICTIONARIES

In Gear, an array is an ordered list of elements. Arrays are created using square brackets: [element, element, element, …]. Array elements can be retrieved by using an integer index.

A dictionary is an unordered list of key-value pairs. Dictionaries are created using a colon ‘:’ as separator between keys and values, and its elements can be retrieved using the key as the index.

[key:value, key:value, key:value, …]

A set is an ordered list of elements, denoted by curly braces { and }. Its elements are not indexable, and duplicates are automatically removed. {element1, element2, element3, …}

## ARRAYS

### The basics

```
var groceries := ['Potatoes', 'Cucumber', 'Tomatoes', 'Olive-oil',
  'Peanuts']
print(groceries[3])      // Olive-oil
groceries[1] := 'Salad'
```

The first index of an array is 0. By using the system library additional functionality is available, such as the first and the last item.

```
print(groceries.first)     // Potatoes
print(groceries.last)      // Peanuts

let slicedList := groceries[1..3] // returns array from index 1 to 3
print(slicedList)     // [Cucumber, Tomatoes, Olive-oil]
```

There are two ways of adding new items to an array. One is by using the concat '+' operator (defined via operator overloading of the + operator), and the other is by using the .add(value:) function, which comes with the arrays library.

```
groceries := groceries + ['French fries']
groceries.add(value: 'Mushrooms')
```

Since concatenation works with arrays, the 'French fries' must be between brackets, so it is seen as an array of 1 item. The concatenation can also be written as:

```
groceries += ['French fries']
```

The add() function can be used either with or without the use of the external parameter name 'value:'.

Adding multiple new values to an existing array can also be done using the .add(array:) function. Now, the parameter name 'array:' is mandatory.

```
groceries.add(array: ['Milk', 'Bread', 'Flower'])
```

or

```
let moreShopping := ['Oranges', 'Apples', 'Pears', 'Bananas', 'Melon']
groceries.add(array: moreShopping)
```

If you insert an item at a specific index take care that for arrays the first index is 0. The attributes at: and value: may be omitted, but are recommended to increase readabilty.

```
groceries.insert(at: 2, value: 'Mango')
```

To see if a list contains a certain item, you use the .contains(value:) function.

```
let tomatoesAvailable := groceries.contains(value: 'Tomatoes')     // true
```

Removing items from the list can be done through the delete() function or remove() function.

```
groceries.delete(index: 3)
var index := groceries.remove(value: 'Cucumber')
// returns the index of removed item
```

If the value to be removed cannot be found, the resulting index will be -1.

You can retrieve the index of a value using function indexOf(value) or index(of: value). If the value can't be found the index returned is -1.

```
var index := groceries.index(of: 'Mushrooms')
// returns the index or -1 if not found
```

An empty array is created in the following way:

```
var emptyArray := []
```

To check if an array is filled or empty is done via the isEmpty value property.

```
let checkIfEmpty := emptyArray.isEmpty
```

To clear an array use the clear() function.

```
groceries.clear()
```

## Array elements are expressions

In Gear, an Array is extremely flexible and it's not limited to just one element type. So, as an example, an array can be:

```
["+", "-", "*", "/", "%"]
[1,2,3,4,5]
['Hello', 'world', "!", 3.1415, true]
```

In fact you can put anything inside an array, as long as it's an expression. That means, even these are arrays:

```
[x=>x+5, x=>x-5, x=>x*5, x=>x/5, x=>x%5] or
[/\(x,y)=>x+y, /\(x,y)=>x-y, /\(x,y)=>x*y, /\(x,y)=>x/y, /\(x,y)=>x%y]
```

## Array declaration and representation

Gear has a default array type, called 'Array', which internally is represented as a record. A variable declared as

```
var a := [1,2,3,4,5]
```

is implicitly defined as a record of type Array. You can only create extensions on Array to enhance its functionality. For example, if you want to call Array() instead of [] then you need to add an initializer in an extension. As an example consider an initializer, which takes an argument that could either be an array of elements or a size. In the latter case the array is filled with nil.

```
extension Array is
  init(input) do
    if value is Array then
      return [].add(array: input)
    else
      return [].add(count: input, value: nil)
    end
  end
end

var a := Array([1,2,3,4,5])
print(a)  // [1, 2, 3, 4, 5]

var b := Array(5)
print(b)  // [nil, nil, nil, nil, nil]
```

Note that this is not part of the standard library.

If you want to limit an array to only contain a certain type of values, say numbers, it is possible to extend Array with an ‘add’ function that only accepts numbers.

```
use typeinfo // contains type information

extension Array is
  func add(number value) do
    ensure type(of: value) = Type.Number else
      error('Only numbers allowed.')
    end
    self.add(value)
  end
end

var numbers := []
numbers.add(number: 99)
numbers.add(number: 100)
print(numbers) // [99, 100]
numbers.add(number: 'hello') // Runtime error: Only numbers allowed.
```

It would still be possible to add other value types such as strings or booleans using the standard add(value:) function.

```
numbers.add(value: 'hello')
print(numbers) // [99, 100, hello]
```

In order to check if the array only contains numbers it is easy to extend the array functionality with a checking function.

```
extension Array is
  func all(.satisfy) do
    for var item in self where not satisfy(item) do
      return false
    end
    return true
  end
end

let check := numbers.all(satisfy: value=>type(of: value) = Type.Number)
```

If there is one value not adhering to the Number type, the result is false.

## Array index

Like in any other programmig language, in Gear an array is accessed through an integer index. Given the array a := [1,2,3,4,5], then its first index is 0, whereas the last index is 4.

So a[0] returns the value 1, a[1] returns 2, etc.

```
let a := [1,2,3,4,5]
for var i:=0 where i < length(a), i+=1 do
  print(a[i])
end
```

Items in multi dimensional arrays can be found by repeated indexes, for eample in array:

```
m := [[1,2],[3,4],[5,6]]
m[0][0] returns 1; m[1][0] returns 3; m[2][1] returns 6.

let m := [[1,2],[3,4],[5,6]]
print(m[1][1]) //4
```

You can also assign values to array expressions. For example:

```
var a := [1,2,3]
print(a[1]) // 2
a[1] := 6
print(a) // [1,6,3]
```

## Array slicing

Sometimes you are interested in only part of an array. In that case you may use the slice operator, which in fact is the range operator: ‘..’ or ‘..<’.

```
let a := [1,2,3,4,5,6,7,8,9,10]
print(a[0..4])          // [1,2,3,4,5]
print(a[0..<5])         // [1,2,3,4,5]
let s := a[2..5]
print(s)                // [3,4,5,6]
print(a[5..<a.count])   // [6,7,8,9,10]
```

The result of the slice operation is yet another array.

## Array operators

The arrays.gear library contains an operator overload. The standard '+' operator is overloaded for arrays. This means if you want to concatenate two or more arrays, you can simply use the + operator.

```
var numbers := [1,2,3,4,5]
numbers := numbers + [6,7,8,9]
print([0] + numbers) // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The operators '+' and '=' are already implemented internally in the compiler.

Though it is possible to overwrite the operators, and create your own version, be aware that the new version immediately applies to all uses of e.g. [array] + [array], which may result in unexpected results or even runtime errors.

```
extension Array is
  // concatenate 2 arrays
  infix +(other) do
    print('you are using your own + operator')
    return [].add(array: self).add(array: other)
  end
end
```

It takes an empty array and first adds 'self' and then the other array.

So, when wisely used it is possible to create powerfull operator functions. As an example, consider the following operator overloads, which are not available from the standard library.

Suppose you wish to create an operator that gives you the internal (dot) product of two same size number arrays. First create an operator to multiply the elements of two number arrays.

Note, there is no checking if the sizes are the same, and no checking on whether the values are numeric. That could be done by using the statement if type(of: value) = Type.Number then…

```
extension Array is

  // calculate the sum of all elements using recursive tail calls
  func summa() =>
    match self.count
      if 0: 0
      else self.head + self.tail.summa()

  //multiply 2 arrays
  infix *(other) do
    let result := []
    for var i in 0..<self.count do
      result.add(self[i]*other[i])
    end
    return result
  end
  // dot product using ~ tilde operator
  infix ~(other) do
    // using * operator to multiply arrays
    let multiplied := self * other
    return multiplied.summa()
  end

  //negate the elements of the array
  prefix -() do
    let result := []
    for var item in self do
      result.add(value: -item)
    end
    return result
  end
end
```

Additionally, a prefix operator is shown that negates all elements of the array, in case they are numeric.

The summa() function uses a technique called recursive tail calls. You can split an array in a head (i.e. the first item) and a tail (the rest of the items). The properties array.head and array.tail are predefined. With the above operators it's now possible to do the following:

```
let a := [1,2,3,4]
let b := [5,6,7,8]
print(a*b) // [5,12,21,32]
print(a~b) // dot product = 70
print(-a)  // [-1,-2,-3,-4]
```

## Standard array functions

The standard available type Array comes with a few standard functions, defined in the library.

The following table shows the standard function and the way they are implemented in the default type Array as an extension. See arrays.gear for details.

| Description | Function in type Array | Example |
|---|---|---|
| The number of items in array | .count | n := numbers.count |
| Check if an array is empty | .isEmpty | b := numbers.isEmpty |
| The first item in the array | .first or .head | a := numbers.first/numbers.head |
| All items in array minus first | .tail | a := numbers.tail |
| The last item in the array | .last | a := numbers.last |
| Add an item to an array | .add(value:) | numbers.add(value: 10) |
| Add an array to an array | .add(array:) | numbers.add(array: [1,2,3,4,5]) |
| Create array with count elem | .add(count:,value:) | Number.add(count: 5, value: 0) |
| Insert an item to an array | .insert(at:, value:) | numbers.insert(at: 1, value: 9) |
| Delete an item from an array | .delete(index:) | numbers.delete(index: 10) |
| Remove an item from an array | .remove(value:) | n := numbers.remove(value: 'a') |
| Check if array contains item | .contains(value:) | b := numbers.contains(value: 'a') |
| Get index of an item | .index(of:) | i := numbers.index(of: 9) |
| Clear the array | .clear() | numbers.clear() |

Higher order functions on array, defined in the array library.

| Function in type Array | Example |
|---|---|
| .reduce(initialValue, lambda(x,y)) | sum := numbers.reduce(0, /\(x,y)=>x+y) |
| .filter(lambda(x)) | evens := numbers.filter(x=>x%2=0) |
| .map(lambda(x)) | squares := numbers.map(x=>x^2) |
| .flatMap(lambda(x)) | allCars := people.flatMap(person=>person.cars) |
| .each(lambda(x)) | numbers.each(/\(x) do print(x) end) |
| .reversed() | list := numbers.reversed() |
| .sum() | sum := numbers.sum() |

Operators for arrays, defined in the library.

| Operator | Meaning | Example A=[1,2,3,4,5], B=[4,5,6,7,8] | |
|---|---|---|---|
| + | concatenation | s := A + B | [1,2,3,4,5,4,5,6,7,8] |
| = | comparison | b := A = B | false |
| <> | comparison | b := A <> B | true |
| in | element of | 4 in A | true |

# DICTIONARIES

## The basics

An empty dictionary is created in a similar way as an ampty array, but requires a colon between [ and ].

```
var emptyDictionary := [:]
```

Filling a dictionary is almost similar, except that the add() function requires a key: and value: pair.

```
use dictionaries
var smileys := ['smile': '😀', 'saint': '😇']
smileys.add(key: 'lol', value: '🤣')
smileys.add(key: 'cool', value: '😎')
smileys.add(key: 'cry', value: '😭')
print(smileys)
```

If you want to know the number of items in an array or dictionary, you use the field count.

```
let numberOfItems := smileys.count
```

To see if a dictionary contains a certain item, you use .contains(key:) or .contains(value:).

```
let smileyAvailable := smileys.contains(key: ‘lol')             // true
let keyAvailable := smileys.contains(value: '😭')               // true
```

The .contains() function is overloaded by using either the key: or value: label.

Removing items from a dictionary is done through the delete(key:) function.

```
smileys.delete(key: 'lol')
```

Clearing a dictionary is done the same way as clearing an array.

```
smileys.clear()
```

And testing for emptyness:

```
let result := smileys.isEmpty  // true if empty, false otherwise
```

## Declaration and representation

Gear has a default dictionary type, called ‘Dictionary’, which internally is represented as a record. A variable declared as

```
let numbers := [1:'One', 2:'Two', 3:'Three', 4:'Four']
```

is implicitly defined as a record of type Dictionary. You can only create extensions on Dictionary to enhance its functionality.

Like arrays, dictionaries can also hold all kinds of expressions, from constants to functions or classes.

## Dictionary access

Dictionary items are accessed via keys.

Retrieving an item:

```
let item := dictionary[key]
```

Setting a new value to an item:

```
dictionary[key] := item
```

## Iterating over dictionaries

Dictionaries have a predefined iterator, available from the standard library. Just like for arrays you can traverse through the dictionary elements.

```
for var element in dictionary do
  print(element.key, element.value)
end
```

Each dictionary element is returned as a tuple of (.key, .value).

And here’s an array of dictionaries. We loop through the array, and inside the array loop we have a second loop were we traverse through the elements of the respective dictionaries. The numbers in the dictionaries can be accessed via ‘type.value’.

```
let interestingNumbers :=
  ['Prime': [1,2,3,5,7,11,13,17,19,23],
   'Fibonacci': [1,1,2,3,5,8,13,21,34],
   'Square': [1,4,9,16,25]
  ]
```

```
var largest := 0
for var type in interestingNumbers do  // loop over dictionaries
  for var number in type.value do      // loop over array in dictionary
    if number > largest then
      largest := number
    end
  end
end
print('Largest number is $(largest)')
```

And here's another example: iteration over a dictionary of functions.

```
let table := [
  '+': /\(x,y)=>x+y,
  '-': /\(x,y)=>x-y,
  '*': /\(x,y)=>x*y,
  '/': /\(x,y)=>x/y,
  '%': /\(x,y)=>x%y]

func calc(x,y) do
  for var item in table do
    print('$(x) $(item.key) $(y) = $(item.value(x,y))')
  end
end
calc(16,6)
```

## Standard dictionary routines

The following table shows the standard functions and the way it is implemented in the default type Dictionary as an extension.

| Description | Function in Dictionary | Example |
|---|---|---|
| Number of items in dictionary | .count | n :=dictionary.count |
| Check if dictionary is empty | .isEmpty | b := dictionary.isEmpty |
| Add a key:value pair | .add(key:, value:) | dictionary.add(key: 1, value: 'One') |
| Delete an item via the key | .delete(key:) | dictionary.delete(key: 1) |
| Check if dict contains key | .contains(key:) | b := dictionary.contains(key: 2) |
| Check if dict contains value | .contains(value:) | b := dictionary.contains(value: 'Two') |
| Clear the dictionary | .clear() | dictionary.clear() |
| A list of the keys | .keys | a := dictionary.keys |
| A list of the values | .values | a := dictionary.values |

## SETS

### The basics

A set is defined using the curly braces { and }.

```
let emptySet := {}
let aSet := {1,1,2,3,5,5,8,13,21,8}
print(aSet)  // {5, 3, 1, 13, 8, 21, 2}
```

A set is ordered, and duplicates are automatically removed when creating a new set or when adding to the set. Use the count field to retrieve the number of elements in a set, and use isEmpty to check for emptyness.

```
print(aSet.count)      // 7
print(aSet.isEmpty)    // false
```

Adding a new value to a set is done by using function add(value:), though the attribute value: may be omitted.

```
aSet.add(value: 34)
```

Checking if a value belongs to the set happens through function contains(value:).

```
print(aSet.contains(value: 7))    // false
```

Alternatively, it is possible to use the keyword 'in' to check if a value is part of a set (or an array).

```
print(8 in aSet)    // true
```

Values can be removed from a set using function remove(value:) or by completely clearing the set.

```
let success := aSet.remove(value:21) // true
aSet.clear()
```

### Set representation

In Gear, a Set is an abstract data type than can store unique values, without any particular order. The mathematical set notation is used, '{' for opening and '}' for closing a set. Sets are dynamic, in the sense that you can add or delete elements from it. Internally, a set is represented as a record. Like for arrays and dictionaries it is possible to extend the functionality of sets.

## Standard set functions and operators

The standard available type Set comes with a few standard functions, defined in the library.

The following table shows the standard functions and how they are used.

| Description | Function in type Set | Example |
|---|---|---|
| Number of items in set | .count | n := set.count |
| Check if set is empty | .isEmpty | b := set.isEmpty |
| Add a value to the set | .add(value:) | set.add(value: 10) |
| Add a set to the set | .add(set:) | set.add(set: {1,2,3,4,5}) |
| Remove value from set | .remove(value:) | b := set.remove(value: 'a') //true/false |
| Check if set has value | .contains(value:) | b := set.contains(value: 'a') |
| Clear the set | .clear() | set.clear() |
| Convert set to array | .toArray() | a := set.toArray() |
| The first item in set | .first or .head | a := set.first/set.head |
| All items in set min first | .tail | a := set.tail |
| Last item in a set | .last | A := set.last |

Higher order functions on set, defined in the sets.gear library.

| Function in type Set | Example |
|---|---|
| .reduce(initialValue, lambda(x,y)) | sum := numbers.reduce(0, /\(x,y)=>x+y) |
| .filter(lambda(x)) | evens := numbers.filter(x=>x%2=0) |
| .map(lambda(x)) | squares := numbers.map(x=>x^2) |
| .flatMap(lambda(x)) | allCars := people.flatMap(person=>person.cars) |
| .each(lambda(x)) | numbers.each(/\(x) do print(x) end) |

Operations on set, defined in the sets.gear library.

| Function in type Set | Example A={1,2,3,4,5}, B={4,5,6,7,8} | |
|---|---|---|
| .union(with set) | s := A.union(with: B) | {1,2,3,4,5,6,7,8} |
| .intersect(with set) | s := A.intersect(with: B) | {4,5} |
| .except(with set) | s := A.except(with: B) | {1,2,3} |
| .symmetricExcept(with set) | s := A.symmetricExcept(with: B) | {1,2,3,6,7,8} |

Operators for sets, defined in the sets.gear library.

| Operator | Meaning | Example A={1,2,3,4,5}, B={4,5,6,7,8} | |
|---|---|---|---|
| + | union | s := A + B | {1,2,3,4,5,6,7,8} |
| * | intersection | s := A * B | {4,5} |
| - | complement (exception) | s := A – B | {1,2,3} |
| ~ | symmetric difference | s := A ~ B | {1,2,3,6,7,8} |
| in | element of | 4 in A | true |

## LOOPS VS HIGHER ORDER FUNCTIONS

Suppose you want do loop over an array, and print the square of the even numbers. In a normal loop that would be as follows.

```
for var n in 1..10 do
  if n%2=0 then
    print(n^2)
  end
end
```

With a higher order function you can remove the loop.

```
 (1..10).filter(n=>n%2=0).map(n=>n^2).each(/\n do
   print(n)
 end)
```

or

```
[n^2 for n in 1..10 where n%2=0].each(/\(n) do
  print(n)
end)
```

# CLASSES RECORDS TRAITS ENUMS EXTENSIONS

Gear classes partly support class oriented programming as seen in many other languages. A class has data and functions and can inherit from a parent class. You will not find interfaces, however traits are supported. You can extend existing classes and records, and you can define private variables or functions. Static functions are supported. You will not see virtual methods, abstract classes, polymorfy, so in that respect Gear doesn't support full OOP functionality.

## CLASSES

A basic class has a name, some fields and methods, and a constructor. A class declaration starts with the keyword 'class', followed by the class name. Its general form is:

```
class Name [:traits] is
  variables
  methods
end
```

Consider the below example.

```
class Car is
  var name := ''
  func show() do
    print(self.name)
  end
  init(name) do
    self.name := name
  end
end
var car := Car('Volvo')
print(car.name)    // “Volvo”
car.show()         // “Volvo”
print(car.className) // “Car” property className is standard available
```

The name of the class usually starts with a capital letter. Note that Gear is case-sensitive.

The variable (or property) name is declared just like it’s done for any other local or global variable. This also counts for functions (methods). However, if a method refers to a class-local variable, the keyword **self** must precede the variable name, separated with a dot.

## Class members

A class can have the following members:

- *a constructor init() // currently only 1 init is allowed, next to the implicit init*
- *(static) methods / functions*
- *variables*
- *constants*
- *calculated properties or values*
- *a subscript*

Every field (variable or constant), method, or value declared inside a class is called a member of that class. All members must be known at compile time, as it becomes very clear to the programmer what’s in the class and saves him/her from errors in this respect.

## Self

In order to use members inside other members it is required to use the keyword ‘self’.

```
class Point is
  var x:=0, y:=0
  init(.x, .y) do
    self.x:=x
    self.y:=y
  end
  func distance(to point) do
    return sqrt((point.x-self.x)^2 + (point.y-self.y)^2)
  end
  func toString() => 'x=$(self.x),y=$(self.y)'
end
var p0 := Point(), p1 := Point(x:1,y:1), p2 := Point(x:3,y:4)
print(p1.distance(to: p0))  // 1.4142135623731
print(p2.distance(to: p0))  // 5
```

If the self keyword is missing the runtime can not find the variable in the class instance and generates a runtime error.

### Using the @-sign for self

Sometimes it not only is tiresome to write a lot of ‘self’, but it disturbs the readabilty as well. For this reason, the @-sign is introduced. Putting ‘@’ in front of a variable replaces the ‘self’ keyword. The above defined class Point now becomes:

```
class Point is
  var x:=0, y:=0
  init(.x, .y) do
    @x:=x
    @y:=y
  end
  func distance(to point) do
    return sqrt((point.x-@x)^2 + (point.y-@y)^2)
  end
  func toString() => 'x=$(@x),y=$(@y)'
end
```

So, when ‘@’ is used it always refers to a variable or member that is defined inside the class.

## Class construction

In class-based programming, objects are created from classes by subroutines called constructors. In Gear, class construction is performed in two ways: through the init() method, or with the implicit construction through the call to Class() without any parameters.

An object is an instance of a class, and may be called a class instance or class object; instantiation is then also known as construction.

```
class Class is
  var field := 'field'
  init(field) do
    self.field := field
  end
end
```

If the initialization only contains an assignment to a field, it can be written simpler as:

```
init(field) for self.field
```

Since the compiler knows that always class fields are initialized, the self keyword can be omitted.

```
init(field) for field
```

in which case the class declaration becomes:

```
class Class is
  var field := 'field'
  init(field) for field
end
```

An instance of a class is created by using the following structure.

```
var instance := Class()           // the value of field is ‘field’
var instance := Class('Hello')    // the value of field is 'Hello'
```

Notice that it looks like a function call. In the first case, the init() is not called, in the second case it is called. In both cases the result is an instance of the class.

Multiple fields, separated by comma’s, can be assigned in one statement, using this construction.

```
class Vehicle is
  var brand := '', type := ''
  init(brand, type) for brand, type // the order is important!
end
```

It is required that the class field is declared using a var or let statement. A error occurs if the field is not defined.

The fields in a class can be variable or constant. If a constant (let) is declared, it can receive a one-time new value. This is especially handy in class initializers, for example:

```
class Vehicle is
  var brand := 'brand'
  let type := 'type'
  init(brand, type) for brand, type    // type gets a one time new value
end

var vehicle := Vehicle('Volvo', 'V90')
vehicle.brand := 'Mercedes'
vehicle.type := 'S class'     // error: Cannot assign value to constant
```

## Private members, getter and setter

Fields and functions can be defined as private members, such that they cannot be accessed from outside the class. A private member starts with an undersore '_'. You can access it via value properties as getter and functions as setter for instance.

```
class Building is
  var _type := 'Flat'
  val type := _type          // getter
  func set(.type) do         // setter
    _type := type
  end
  init(.type) for _type
end
var building := Building(type: 'Appartment')
print(building.type)          // use getter
building.set(type: 'Office')  // use setter
print(building.type)
print(building._type)         // error
```

The variable _type is only available inside the class. Trying to access it from outside the class results in a compile time error.

Note that if a field is private, you don't need to use 'self' or @ in front of it. Since private fields only reside inside a class, the compiler knows how to handle them.

## Default fields

A default field is used to easier show the contents of a class field.

```
class Car is
  default type := 'car'
  init(.type) for self.type
end

var car := Car(type: 'Volvo')
print(car)  // Volvo
```

The field ‘type’ is declared as default and is per definition a variable (not a constant), which means it can be changed. If the contents of an instance of class Car are printed then in this case the contents of variable ‘type’ are printed instead. You don’t have to create a toString() function for this. If a class doesn’t have a default field, it would print ‘Car instance’.

Only one default field can be declared in a class.

Obviously, a default field can also be private.

```
class String is
  default _value := ''
  val value := _value
  init(value) for _value
end

var chars := String('0123456789abcdefghijklmnopqrstuvwxyz')
chars := '0123456789'
var str := String()
str := '0123456789abcdefghijklmnopqrstuvwxyz'
print(str)
print(str.value)
```

As you can see, using a default field has another nice benefit: you can directly assign values to the class variable where the assigned expression has the same type as the default variable. In above example a string of characters is directly assigned to a variable of type String.

## Class field can be a class

Fields of classes can be classes themselves. However, it is not allowed to define a class inside a class.

```
class Date is
  var day := 1, month := 'January', year := 1980
  init(.day, .month, .year) for day, month, year
  func toString() do
    return 'Date: $(self.day)-$(self.month)-$(self.year)'
  end
end

class Person is
  var name := 'Person',
      birthDate := Date()
  init(name, .birthDate) do
    self.name := name
    self.birthDate := birthDate
  end
  func toString() => self.name + ': ' + self.birthDate.toString()
```

```
end

var Harry := Person('Harry', birthDate:
  Date(day: 23, month: 'May', year: 1987))
var Sally := Person('Sally', birthDate:
  Date(day: 18, month: 'July', year: 1992))
print(Harry.toString())
print(Sally.toString())
print(Person().toString())
```

## Inheritance

Classes can inherit from each other. You can declare a base class, which can be the parent of child classes. The parent class is written after the class name followed by a '<' token. A class can have at most one parent.

```
class Vehicle is
  ...
end

class Car < Vehicle is
  ...
end

class Train < Vehicle is
  ...
end
```

This means that both class Car and Train inherit from (or are subclass of) class Vehicle. Vehicle is the parent class. With inheritance in its simplest form, the methods at parent class level become available at child class level. Or in other words, the child class can use the methods of the parent class. If the respective called method can't be found in the child class, we'll try to find it in the parent class.

```
class Parent is
  func method() do
    print('Parent Method')
  end
end

class Child<Parent is
end

var child := Child()
child.method()          // "Parent Method"
```

If you define a method with the same name in a child class, it overrides the method of the parent class automatically. You can reuse the method of the parent class by using the ‘inherited’ statement.

```
class Child<Parent is
  func method() do
    inherited method()
    print('Child Method')
  end
end
var child := Child()
child.method()
// Parent Method
// Child Method
```

Especially in init() methods it can be helpful to call the parent’s init(). You do this by using:

- *inherited init() or easier:*
- *inherited()*

```
class Circle is
  var radius := 1
  val area := pi() * self.radius^2
  init(.radius) do
    self.radius := radius
  end
end

class Cylinder < Circle do
  var height := 0
  val volume := self.area * self.height
  init(.radius, .height) do
    inherited(radius)
    self.height := height
  end
end

var circle := Circle(radius: 1)
var cylinder := Cylinder(radius: 2, height: 10)
```

The two ways of using inherited initialization are with or without the keyword ‘init’. Both are allowed and it only applies to init(). Inheriting other functions always require the name of the function to inherit.

```
class ApplePie is
  func serve() do
    print('Serve warm apple pie', terminator: '')
  end
end
```

```
class Apfelstrudel(ApplePie) is
  func serve() do
    inherited serve()
    print(' on a hot plate with vanilla saus.')
  end
end

Apfelstrudel().serve()
// Serve warm apple pie on a hot plate with vanilla saus.
```

## Class instances in arrays

You can store class instances in an array.

```
class Person is
  var name := '', address := ''
  var age := 0, income := 0
  var cars := []
  init(.name, .address, .age, .income, .cars) for
    name, address, age, income, cars
end

use arrays
var people := [
  Person(name: 'Jerry', address: 'Milano, Italy', age: 39, income: 73000,
    cars: ['Opel Astra', 'Citroen C1']),
  Person(name: 'Cathy', address: 'Berlin, Germany', age: 34, income: 75000,
    cars: ['Audi A3']),
  Person(name: 'Bill', address: 'Brussels, Belgium', age: 48, income: 89000,
    cars: ['Volco XC60', 'Mercedes B', 'Smart 4x4']),
  Person(name: 'Francois', address: 'Lille, France', age: 56,
    income: 112000, cars: ['Volco XC90', 'Jaguar X-type']),
  Person(name: 'Joshua', address: 'Madrid, Spain', age: 43, income: 42000,
    cars: [])
]

let names := people.map(person=>person.name)
// [Jerry, Cathy, Bill, Francois, Joshua]
let totalIncome := people.map(person=>person.income).reduce(0, (x,y)=>x+y)
// 391000
let allCars := people.flatMap(person=>person.cars)
// [Opel Astra, Citroen C1, Audi A3, Volco XC60, Mercedes B, Smart 4x4,
  Volco XC90, Jaguar X-type] flat list of all cars
```

## Static methods

Static methods are meant to be relevant to all the instances of a class rather than to any specific instance. A static method can be invoked even if no instances of the class exist yet. Stratic methods are defined by putting the keyword ‘static’ in front of the function.

```
class Math is
  static func sqr(x)=>x^2
  static func sqrt(x)=>x^0.5
end

var y := Math.sqr(10)
print(y)
y := Math.sqrt(y)
print(y)
```

A static method is called by using the class name followed by a dot '.' and then the method name.

## Subscripts

A subscript allows an instance of a class or record to be indexed as an array. The user can define a subscript for a class, after which the class can behave as a virtual array. You can then use the array access operator [] to access items of the class and you don't have to access the class members directly.

The general structure of a subscript is

```
subscript[indexVar] for Expression
```

for a 'getter' only, and if we wish to include a 'setter' the structure is:

```
subscript[indexVar](value) for Expression, Assignment
```

Consider the example.

```
class List is
  var _items := []
  val count := _items.count
  func add(item) do
    _items.add(item)
  end
  // getter + setter
  subscript[i](value) for _items[i], _items[i] := value
end

var list := List()
list.add(3)
list.add(5)
list.add(7)
list.add(11)
let z := list[2] // getter called
print(z)
list[3] := 9  // setter called
print(list.items)
```

If you would only allow getter access then the subscript would look like this:

```
  subscript[i] for _items[i] // getter only
```

The parameter (value) is omitted in this case.

There is one restriction to the use of indexers: a class or record can have only one subscript! Defining multiple subscripts results in an error.

## RECORDS

A record is a class without inheritance. If you don’t want a type to be inherited from, use a record instead. All other functionality from classes stays the same.

Static methods:

```
record Math is
  static func sqr(x)=>x^2
  static func sqrt(x)=>x^0.5
end

var y := Math.sqr(10)
print(y)
y := Math.sqrt(y)
print(y)
```

Initialization:

```
record Person is
  var name := '', address := ''
  var age := 0, income := 0
  var cars := []
  init(.name, .address, .age, .income, .cars) for
    name, address, age, income, cars // self can be omitted in this case
end
var person := Person(name: 'Dave', address: 'Boston', age: 28,
  income: 20000, cars: [])
```

The use of ‘self’:

```
record Point is
  var x:=0, y:=0
  init(.x, .y) do
    self.x:=x
    self.y:=y
  end
```

```
func distance(to point) do
    return sqrt((point.x-self.x)^2 + (point.y-self.y)^2)
  end
end

var p0 := Point(), p1 := Point(x:1,y:1), p2 := Point(x:3,y:4)

print(p1.distance(to: p0))  // 1.4142135623731
print(p2.distance(to: p0))  // 5
```

Subscripts:

```
use ranges
record TimesTable is
  var multiplier := 1
  init(.multiplier) for self.multiplier
  subscript[i] for i*self.multiplier
end

let table := TimesTable(multiplier: 7)
print('Table of $(table.multiplier):')
for var index in 1..10 do
  print(table[index])
end

// prints the table of 7
```

Default and private fields:

```
record String is
  default _value := ''
  init(value) for _value
  subscript[i](char) for _value[i], _value[i] := char
end

let chars := String('0123456789abcdefghijklmnopqrstuvwxyz')
print(chars)
```

## TRAITS

A trait is a concept used in object-oriented programming, which represents a set of methods that can be used to extend the functionality of a class. Traits both provide a set of methods that implement behaviour to a class, and require that the class implement a set of methods that parameterize the provided behaviour. For inter-object communication, traits are somewhat between an object-oriented protocol (interface) and a mixin. An interface may define one or more behaviors via method signatures, while a trait defines behaviors via full method definitions: i.e., it includes the body of the methods. In contrast, mixins include full method definitions and may also carry state through member variable, while traits usually don't.

The traits in Gear are as described above, and they follow these set of rules:

- *they have reusable functions,*
- *a class may contain zero or many traits,*
- *traits can use other traits, so that the functions defined in a trait become part of the new trait,*
- *redefined functions result in a collission.*

In a class traits are defined right after the class header. Use a colon ':' to start the trait definitions.

```
class Car < Vehicle: Stringable, Drivable is
  members
end
```

In many languages you can test whether an instance belongs to a certain class, for example in Java, the function instanceOf() returns True if such is the case. In Object Pascal the operator 'is' is used for this.

We will use the keyword 'is' as well, for example:

```
trait Instance is
  func instanceOf(Class) do
    return self is Class
  end
end

record Number: Instance is
  var code := 0
  init(code) do
    self.code := code
  end
  func write() do
    print(self.code)
  end
end

var number := Number(10)

if number.instanceOf(Number) then
  print(true)
end
```

This example also shows how a trait can be used. For it to be reusable, it should have few, if none, dependencies with the classes that are using it. You can use ‘self’ in a trait, which always points to the class instance that implements the trait. Traits themselves as well as classes can have multiple traits, separated by commas.

```
trait One is
end

trait Two is
end

trait Three: One, Two is
end
```

Trait Three will receive all functions defined in the other traits. Functions defined in traits that use or refer to each other must be preceded by ‘self’.

The following example shows the use of traits in a class that checks if the user name and password contains correct characters.

```
trait ValidatesUsername is

  // builds valid characters for user name
  func validUNChars() do
    var result := []
    for ascii in Range(48, 57) do
      result.add(value: chr(ascii))
    end
    for ascii in Range(65, 90) do
      result.add(value: chr(ascii))
    end
    for ascii in Range(97, 122) do
      result.add(value: chr(ascii))
    end
    return result
  end

  // checks user name characters are valid
  func isUsernameValid(username) do
    let len := length(username)
    return if len < 8 then false else true

    for var i := 0 where i < len, i+=1 do
      ensure self.validUNChars().contains(username[i]) else
        return false
      end
    end
  end
end
```

```
trait ValidatesPassword is

  // builds valid characters for password
  func validPWChars() do
    var result := []
    for char in Range(34, 125) do
      result.add(value: chr(char))
    end
    return result
  end

  // checks password characters are valid
  func isPasswordValid(password) do
    let len := length(password)
    return if len < 8 then false else true

    for var i := 0 where i < len, i+=1 do
      ensure self.validPWChars().contains(password[i]) else
        return false
      end
    end
  end
end

record Login: ValidatesUsername, ValidatesPassword is
  var userName := ''
  var passWord := ''

  init(userName, passWord) for self.userName, self.passWord

  func loginEntered() do
    ensure self.isUsernameValid(self.userName) else
      print('Wrong user name')
      return false
    end
    ensure self.isPasswordValid(self.passWord) else
      print('Wrong password')
      return false
    end
    return true
  end
end

print('user name: ', terminator: '')
let userName := readln()
print('password : ', terminator: '')
let passWord := readln()

var login := Login(userName, passWord)

print(if login.loginEntered() then
  'Login correct'
else
  'Login incorrect')
```

# EXTENSIONS

## The basics

Extensions can add functionality to existing types. You can create extensions for all user declared types. An extension is defined using the keyword 'extension', followed by the name of the type that is extended. The general form is:

```
extension TypeName is
  members
end
```

Functionality defined in an extension immediately becomes available for all variables that were declared for the respective type, e.g. Array. Only func's and val's can be used in an extension.

```
extension Array is
  func toString() => '$(self)'
  val total := length(self)
end
```

A func or val defined in an extension overwrites earlier defined ones. Note that the keyword 'self' is available in extensions.

As mentioned an extension can only have function and/or value declarations. If any other declaration type is defined, an error is generated. If you want to add a new field for example, the usual way is to create a subclass from the original class and then add the field.

Multiple extensions can be added to a type.

```
class Class is
  var field := nil
  init(field) for self.field
end

extension Class is
  func method() do
    print('something')
  end
end

extension Class is
  func otherMethod() do
    print('something else')
  end
end
```

Look in arrays.gear, dictionaries.gear, sets.gear or ranges.gear for extensions on all these types. You can find these files in folder \gearlib\

## Operator overloading

Extensions can also be used to add operators on classes/records and their instances. The following operators can be overloaded (separated by spaces):

```
+  -  *  /  %  =  <>  >  >=  <  <=  ^  <<  >>  <~  !  &  |  ~
```

The following is an example showing how to extend functionality and operators for complex numbers.

```
record Complex is
  var re := 0, im := 0
  default _value := ''
  init(re, im) do
    self.re := re
    self.im := im
    self._value := self.toString()
  end
  func toString() do
    let sign := if self.im < 0 then '' else '+'
    return '$(self.re)$(sign)$(self.im)i'
  end
end

extension Complex is
  infix +(other) => Complex(self.re + other.re, self.im + other.im)
  infix >(other) => sqrt(self.re * self.im) > sqrt(other.re * other.im)
  infix <(other) => sqrt(self.re * self.im) < sqrt(other.re * other.im)
  infix =(other) => self.re = other.re and self.im = other.im
  infix <>(other) => not (self = other)
  prefix -() => Complex(-self.re, -self.im)
end
```

It can now be used as follows:

```
let c := Complex(3,4)
let d := Complex(5,3)
let x := c+d
let g := -c+d
print(c<d)
print(c=d)
print(x)
```

Note that operator overloading will also work on other types such as arrays, dictionaries, sets and enums. If an operator for a type is not defined, a runtime error occurs:

```
Infix operator "<=" for these types undefined.
[line 52] in script
```

From the example above you notice that for defining an operator either of the keywords ‘infix’ or ‘prefix’ must be used. As operators are always defined in an extension to a class or record, the following rules apply:

- *infix operators require exactly 1 argument. This argument always represents the right hand side of a binary expression. The left hand side is always ‘self’.*
- *Prefix operators require exactly 0 (zero) arguments. The operator always applies to ‘self’.*

Other than this, the operator definition follows the parsing rules of functions. If you just need to return a single expression, like in above example, the arrow notation may be used.

## ENUMS

### The basics

In Gear an enumeration is declared as starting with the keyword ‘enum’. A few examples of its declaration are shown below.

```
enum Color is
  Red, Blue, Yellow, Green, Orange, Purple
end
```

This is the simplest version whereby a simple enumeration of elements is given. A variable that uses this enum is declared like this:

```
var myColor := Color.Blue
print(myColor.name)    // prints ‘Blue’
```

As you see an enum will have a default field called ‘name’ that holds the string representation of an enum.

You can also define an enum with values, for example if you want to print different values than the name, like this:

```
enum TokenType is
  Plus='+', Min='-', Mul='*', Div='/', Rem='%'
end
var tokenType := TokenType.Mul
```

```
print(tokenType.name)    // prints 'Mul'
print(tokenType.value)   // prints '*'
```

## Complex functionality with Enums

Next to enum values in an enum definition, it is possible to add additional functionality.

```
enum TokenType is
  Plus='+', Min='-', Mul='*', Div='/', Rem='%'
  func toString() => '$(self.name)=$(self.value)'
end
```

Since the value of an enum can be any expression, nice functionality can be created.

```
use math // for the dec2hex function

record RGB is
  var red := 0, green := 0, blue := 0
  init(r,g,b) for self.red, self.green, self.blue
  func toString() => '$(self.red)-$(self.green)-$(self.blue)'
  val asHex do
    return dec2hex(self.red) + dec2hex(self.green) + dec2hex(self.blue)
  end
end

enum Color is
  Black  = RGB(0,0,0),
  White  = RGB(255,255,255),
  Red    = RGB(255,0,0),
  Green  = RGB(0,255,0),
  Blue   = RGB(0,0,255),
  Yellow = RGB(255,255,0),
  Orange = RGB(255,128,0),
  Purple = RGB(127,0,255)

  func toString() => '$(self.name)'
  func valueToString() => self.value.toString()
end

var color := Color.Orange
print(color)
print('color name: $(color.name)')
print('color.value: $(color.value)')
print('color.value as hex: $(color.value.asHex)')

switch color
  case .White: print('White: $(color.value.asHex)')
  case .Red: print('Red: $(color.value.asHex)')
  case .Green: print('Green: $(color.value.asHex)')
  case .Blue: print('Blue: $(color.value.asHex)')
  case .Yellow: print('Yellow: $(color.value.asHex)')
  case .Orange: print('Orange: $(color.value.asHex)')
  case .Purple: print('Purple: $(color.value.asHex)')
  default: print('no color')
end
```

Note that in a switch statement it is allowed to omit the enum name, so instead of Color.White also .White without Color can be used.

## Extending an enum

The following is an example of an enum extension, which solves the ternary logic problem. Ternary (or trinary) logic consists of an additional value Maybe to the existing values True and False. More information can be found on RosettaCode: https://rosettacode.org/wiki/Ternary_logic

The challenge as described on this page is solved below. It starts with an enum defining the three values True, Maybe and False. Note we use the values with a capital first letter, in order not to mix them with the predefined Gear values 'true' and 'false'.

```
enum Trits is
  True='True ', Maybe='Maybe', False='False'
end

extension Trits is
  prefix !() => match self            // simulates not
    if Trits.True: Trits.False
    if Trits.False: Trits.True
    else Trits.Maybe

  infix &(other) => match self        // simulates and
    if Trits.True: other
    if Trits.Maybe: if other=Trits.False then Trits.False else Trits.Maybe
    else Trits.False

  infix |(other) => match self        // simulates or
    if Trits.False: other
    if Trits.Maybe: if other=Trits.True then Trits.True else Trits.Maybe
    else Trits.True

  infix ~(other) => match self        // simulates xor
    if Trits.False: other
    if Trits.True: !other
    else Trits.Maybe

  infix *(other) => !(self ~ other)   // simulates equivalence
  infix >>(other) => !self | other    // simulates imp =>  >>
end
```

Now that we have defined the operator overloads in the extension, we can set them to work. The following code creates tables for the above defined operations. First, create an array with the 3 different Trits values.

```
use arrays
var ta := [Trits.True, Trits.Maybe, Trits.False]

print(' not !')
print(' -------------')
for var t in ta do
  print(' $(t.value) | $(!t)')
end
print()

print(' and & | True   Maybe  False')
print(' ----------------------------')
for var t in ta do
  print(' $(t.value) |', terminator: '')
  for var tt in ta do
    print(' $((t & tt).value)', terminator: ' ')
  end
  print()
end
print()

print(' or |   | True   Maybe  False')
print(' ----------------------------')
for var t in ta do
  print(' $(t.value) |', terminator: '')
  for var tt in ta do
    print(' $((t | tt).value)', terminator: ' ')
  end
  print()
end
print()

print(' xor ~ | True   Maybe  False')
print(' ----------------------------')
for var t in ta do
  print(' $(t.value) |', terminator: '')
  for var tt in ta do
    print(' $((t ~ tt).value)', terminator: ' ')
  end
  print()
end
print()

print(' eqv * | True   Maybe  False')
print(' ----------------------------')
for var t in ta do
  print(' $(t.value) |', terminator: '')
  for var tt in ta do
    print(' $((t * tt).value)', terminator: ' ')
  end
  print()
end
print()

print(' imp >>| True   Maybe  False')
print(' ----------------------------')
for var t in ta do
  print(' $(t.value) |', terminator: '')
  for var tt in ta do
    print(' $((t >> tt).value)', terminator: ' ')
  end
  print()
end
```

This results in the following.

```
not !
-------------
True  | False
Maybe | Maybe
False | True

and & | True   Maybe  False
----------------------------
True  | True   Maybe  False
Maybe | Maybe  Maybe  False
False | False  False  False

or |  | True   Maybe  False
----------------------------
True  | True   True   True
Maybe | True   Maybe  Maybe
False | True   Maybe  False

xor ~ | True   Maybe  False
----------------------------
True  | False  Maybe  True
Maybe | Maybe  Maybe  Maybe
False | True   Maybe  False

eqv * | True   Maybe  False
----------------------------
True  | True   Maybe  False
Maybe | Maybe  Maybe  Maybe
False | False  Maybe  True

imp >>| True   Maybe  False
----------------------------
True  | True   Maybe  False
Maybe | True   Maybe  Maybe
False | True   True   True
```

# ITERATORS RANGES LIST CREATION

## ITERATOR BASICS

An iterator is a class or record (instance) that enables traversing a container, in particular lists. Iteration can be done implicitly by using e.g. for-in constructs, or explicitly by creating an explicit iterator.

An example of implicit iteration is:

```
var list := [0,1,2,3,4,5,6,7,8,9]
for var item in list do
  doSomethingWith(item)
end
```

### For in

The biggest advantages of an implicit iteration are its readability and ease of use. The for-in statement is designed to traverse lists. For the user this means that very readable code can be created.

After the keyword ‘for’ and ‘var’, the variable works like a variable declaration, and the type of the variable will be determined by the type of the list elements. This all happens automatically, under the hood, so to speak.

In principle a for-in statement can be translated to a normal while loop, and you'll get an explicit iteration, like this:

```
var list := [0,1,2,3,4,5,6,7,8,9]
while var index:=0 where index<length(list) do
  var item := list[index]
  print(item)
  index += 1
end
```

Though a clear and good solution, it doesn't fit in all cases. E.g. try to create a similar solution for traversing enums or other types of collections. Another solution to this is to add iterator classes and an iterator function to collection classes. An example of such an explicit iterator is for example:

```
var list := [0,1,2,3,4,5,6,7,8,9]
while var iterator := list.iterator where iterator.moveNext do
  var item := iterator.current
  print(item)
end
```

## Iterator pattern

For this to properly work we have to create an iterator object and an extension to Array. So, if you do that for all collection types in the same way, you have a generic solution, which we call the iterator pattern.

```
record ArrayIterator is // class can also be used
  code for 'current'
  and 'moveNext'
end

extension Array is
  val iterator := ArrayIterator(self)
end
```

That seems like a lot of work for just creating an explicit iterator, but the trick is to use the implicit iterator of course, but in such a generic way, that it can be used for all collection types that adhere to the iterator pattern.

This means for each collection type you create an iterator class or record, that can be initialized and contains the value properties 'moveNext' and ‘current’. Then, create an extension to the collection type that contains the iterator value, which returns the iterator object.

For example the same iterator can also be used on an array of string:

```
var animals := ['Giraffe', 'Lion', 'Elephant', 'Monkey', 'Dolphin']
for var animal in animals do
  print(animal)
end
```

This becomes under the hood of the compiler:

```
while var iterator := animals.iterator where iterator.moveNext do
  var animal := iterator.current
  print(animal)
end
```

So what should an iterator look like? First you define a record or class containing the moveNext and current properties. It needs some additional initialized fields like a counter or index and preferably the number of elements. It must have an init() that takes the respective list as an argument. The start index must be -1.

Here’s an example iterator for a ficive Array type (fictive because type Array has an internal iterator, and doesn’t need to be created separately):

```
record ArrayIterator is
  let list := []
  var index := -1
  let count := 0

  init(list) do
    self.list := list
    self.count := length(list)
  end

  val moveNext do
    self.index +=1
    return self.index < self.count
  end

  val current := self.list[self.index]
end

extension Array is
  val iterator := ArrayIterator(self)
end
```

The count variable is set to the number of items, the moveNext property is a Boolean that returns true while the index is less than the count of items. The current property returns the current item in the list.

Finally, you define an extension to Array, in where you declare a new value property that returns a new ArrayIterator, with the array itself as the argument.

As an example, you can directly access the array iterator, should you wish to, but it's not actually necessary.

Compare both loops:

```
var list := [0,1,2,3,4,5,6,7,8,9]

for var item in list do
  print(item)
end

while var it := list.iterator where it.moveNext do
  var item := it.current
  print(item)
end
```

Both loops are in principle the same. The first loop is just easier to read and has the possibility to filter as well. Printing only the even numbers is easy:

```
For var item in list where item%2=0 do
  print(item)
end
```

## ITERATING RANGES

### The basics

A range shows the limits of a series of integer points, from low to high. The general form of a range is: lowest..highest. The following are ranges.

```
0..99
a..b
0..length(string)
```

Sometimes, the right hand side of a range should not be included. In these case, the operator '<' can be used.

```
0..<100  // 100 is not included
0..<length(string)
```

## Range representation

The compiler represents Range as a record with just 3 number fields: 'from', 'to' and 'step'. Declare a range as

```
let range := from..to                    // stepsize is 1
let range := (from..to).step(by: value). // stepsize is value
```

The default value of step is 1.

```
let r := (0..20).step(by:2)
```

Creates the following range of values for variable 'r':

0,2,4,6,8,10,12,14,16,18,20

## Iterating over a range

Next, we would like to iterate over a range. Consider the following example:

```
let list := [0,1,2,3,4,5,6,7,8,9]

for var i in 0..list.count-1 do
  print(i)
end
```

or of course using the dotted + < notation

```
for var i in 0..<list.count do
  print(i)
end
```

The iterator is defined in implicity in the compiler.

And by adding the step(by:) function the Range iterator becomes complete.

```
For var i in (0..30).step(by:3) do
  print(i)
end
```

This prints the multiplication table of 3. Do note the parenthesis around the range (0..30). This is needed in this case.

Another example. The following generates an array of all prime numbers from 3 up to 10000.

The full power of iterators…

```
func isPrime(number) do
  let sqrtNum := sqrt(number)
  for i in (3..sqrtNum).step(by:2) where number%i = 0 do
    return false
  end
  return true
end

var primes := (3..<10000).step(by:2).filter(n=>isPrime(n))
print(primes)
```

The step(by:2) makes sure we don't evaluate even numbers, since they cannot be prime numbers, except for 2 of course.

For the types Array, Dictionary, Set and Range the iterators are defined in the compiler.

## Creating your own iterator

For the basic type String there exists no predefined iterator. So you cannot do something like this:

```
for var char in 'abcdefghijklmnopqrstuvwxyz' do // not allowed (yet)
  print('Char $(char) has ascii code $(ord(char)).')
end
```

However, sometimes you might want to iterate over a string. You could use the below alternative solution.

```
let chars := '0123456789abcdefghijklmnopqrstuvwxyz'

for var i in 0..<length(chars) do
  print('Char $(chars[i]) has ascii code $(ord(chars[i])).')
end
```

Luckily, there is a way to create an iterator for a string, though it is not 100% exactly what we want, but it comes close. You define the iterator one time and you can reuse it any time you like.

First create a String type, with some basic member functionality:

```
record String is
  default _value := ''
  val value := _value
  init(value) for _value
  subscript[i](char) for _value[i], _value[i] := char
end
```

A variable declaration of type String looks like this:

```
let chars := String('0123456789abcdefghijklmnopqrstuvwxyz')
```

The subscript was added, so you can use an index on the variable chars:

```
print(chars[10]) // a
chars[10] := "a"
```

Now that chars is of type String and contains a default value of string-type, you can assign directly string values to chars.

```
chars := 'abcdefghijklmnopqrstuvwxyz'
```

Next, we will create the iterator.

```
record StringIterator is
  let _chars := ''
  var _index := -1
  let count := 0

  init(string) do
    _chars := string.value
    _count := length(_chars)
  end

  val moveNext do
    _index += 1
    return _index < _count
  end

  val current := _chars[_index]
end

extension String is
  val iterator := StringIterator(self)
end
```

And we are ready to use the for-in loop on variables of type String.

```
For var char in chars do
  print('Char $(char) has ascii code $(ord(char)).')
end
```

However, if this seems like too much overhead, which is imaginable, there’s one more alternative. Just create an array from the string.

```
let chars := array(of: '0123456789abcdefghijklmnopqrstuvwxyz')

for var char in chars do
  print('Char $(char) has ascii code $(ord(char)).')
end
```

The standard native function array(of: 'string') creates an array of characters from the string.

Here's another example: a Fibonacci number generator. The idea is to create a generator for Fibonacci numbers, such that you can write:

```
for var i in Fibonacci(15) do
  print(n, terminator: ' | ')
end
```

With the following result:

```
1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | 610 |
```

```
record Fibonacci is
  var max := 0
  init(max) for max
end

record FibonacciIterator is
  var _numbers := []
  var _index := -1

  init(max) do
    if max >= 1 then
      _numbers.add(1)
    end
    if max >= 2 then
      _numbers.add(1)
    end
    for var i in 2..<max do
      _numbers.add(_numbers[i-1] + _numbers[i-2])
    end
  end

  val moveNext do
    _index +=1
    return _index < _numbers.count
  end

  val current := _numbers[_index]
end

extension Fibonacci is
  val iterator := FibonacciIterator(self.max)
end
```

# LIST COMPREHENSION

## The basics

Gear supports list and set comprehension for single variables. The official mathematical notation for building a set though the set builder notation is:

result = $\{ 2x \mid x \in N, x^2 > 3 \}$

The result is the set (or list) of all numbers 2 times x for which x is an element of the natural numbers and where the predicate x squared is greater than 3 must be satisfied.

In Gear the input set or sequence expression must be represented by a closed set, e.g. 1..100, or [1,2,3,4,5,6], as the list will be generated immediately.

The representation of the set builder notation in Gear is more based on existing keywords. The input set can be an array, set, dictionary, range or any iterable expression. Gear implements a form of the set builder notation. Instead of using '|' as separator we use the keyword 'for', and for '∈' we use 'in' and for the comma, we'll use 'where'. Like this:

```
var result := { 2*x for x in 1..100 where x^2>3 } // creates an ordered set
```

As input set you can use arrays, sets and ranges, e.g.:

```
var result := [ 2*x for x in [1,2,3,4,5,6,7,8,9,10] where x^2>3 ]
var result := [ 2*x for x in {1,2,3,4,5,6,7,8,9,10} where x^2>3 ]
var result := [ 2*x for x in 1..<10 where x^2>3 ]
```

In all cases the result is the same: an array. If you need a set as result, use the left and right curly braces {}.

```
var result := { 2*x for x in 1..<10 where x^2>3 } // produces a set
```

In fact any iterable class object can be used as input. You can use a dictionary as an input set, for example:

```
let dict := [1:'One', 2:'Two', 3:'Three', 4:'Four']

var x := [ x.value for x in dict where x.key%2=0 ]
print(x)  // [Two, Four]
```

Remember that a dictionary element has two standard fields: 'key' and 'value'.

Another example:

```
let words := [
  'pair':'couple',
  'oma':'grandmother',
  'man':'male',
  'woman':'female']

let w := {word.value for word in words where word.key > 'oma' }
print(w)

// {couple, female}
```

## Advanced topics

First one more primes calculation, now using list comprehension.

```
func isPrime(number) do
  let sqrtNum := sqrt(number)
  for var i:=3 where i<=sqrtNum, i+=2 do
    if number%i = 0 then
      return false
    end
  end
  return true
end

let primes := [n for n in (3..<10000).step(by:2) where isPrime(n)]
print(primes)
```

This is still comparable to the one shown before. However, we can also create a one-liner using list comprehension.

```
let primes := [n for n in (3..<10000).step(by:2) where
  lambda(m) do
    for var i in (3..sqrt(m)).step(by:2) where m%i = 0 do
      return false
    end
    return true
  end(n)]
```

Try to figure out what happens here. The where-clause contains a lambda expression.

Another example that shows the use of the where-clause. Suppose you want to create a list of numbers that are divisible by both 2 and 5, you'll get this solution.

```
let  numbers := [n for n in 0..100 where n%2=0 and n%5=0]
print(numbers)
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Now let’s look at the expression before the ‘for’ in the list comprehension. The following solution produces an array where the even numbers become ‘even’ and the odd numbers become ‘odd’.

```
let toggle := [if i%2=0 then 'even' else 'odd' for i in 0..10]
print(toggle)

[even, odd, even, odd, even, odd, even, odd, even, odd, even]
```

Here the if-expression is used as the resulting expression.

You can also use this to solve the famous fizz-buzz problem. If a number is divisible by 3 then print ‘fizz’, if by 5 then print ‘buzz’, if by both 3 and 5 then print ‘fizzBuzz’.

```
let fb := [if n%3=0 and n%5=0 then 'fizzBuzz' else
           if n%3=0 then 'fizz' else
           if n%5=0 then 'buzz' else n
           for n in 1..100]
print(fb)

[1,2,fizz,4,buzz,fizz,7,8,fizz,buzz,11,fizz,13,14,fizzBuzz,16,17,etc]
```

Another nice application is to use it for transposing matrices.

```
let matrix := [[1, 2], [3,4], [5,6], [7,8]]
let transposed := [[row[i] for row in matrix] for i in 0..1]
print(transposed)
[[1, 3, 5, 7], [2, 4, 6, 8]]
```

Obviously, we need to prove it works the other way around as well.

```
let matrix := [[1, 3, 5, 7], [2, 4, 6, 8]]
let transpose := [[row[i] for row in matrix] for i in 0..3]
print(transpose)
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

# FILE HANDLING

A language must be able to read from and write to files in order to communicate with the outside world.

The file handling functions described below, are more or less copied from the ones available in Free Pascal. They are:

- *fileOpen, to open a file and return a file handle to the opened file;*
- *fileRead, to read an array of bytes from the file;*
- *fileClose, to close the file, defined by a file handle;*
- *fileWrite, to write an array of bytes to the file;*
- *fileSeek, to seek a position in the file;*
- *fileCreate, to create a new file;*

Finally, there is function ‘readFile(fileName)’ which combines opening, reading and closing a file with the given file name.

Function ‘fileOpen’ takes two arguments: the file path/name and the file mode. For the file mode three predefined constants can be used: ‘forReading’, ‘forWriting’ and ‘forReadingWriting’. The function returns the file handle, which is to be used by the other file functions.

If its value is -1, something went wrong, e.g. the file name is incorrect.

```
let file := fileOpen('/Users/test.txt', forReading)
if file <> -1 then
  input := fileRead(file)
else
  print('Path/file combination does not exist.')
end
fileClose(file)
```

The file handle is used to identify the file and after reading all bytes, variable 'input' contains the read bytes.

After reading the bytes from the file, they have to be copied to a Gear array or string, before it can be used. In this case, variable 'input' is a so-called buffer variable.

```
let bytes := array(of: input)
let string := string(of: input)
```

Creating a new file goes as follows:

```
let newFile := fileCreate('/Users/test.txt')
let output := 'This is for real.'

let i := fileWrite(newFile, bytes(of: output))

if i = -1 then
  print('error writing to file')
end

fileClose(newFile)
```

Writing to a file works the opposite way. Writing a string to a file requires to convert the string's characters to bytes. Then it's possible to write them to the file. The fileWrite function always expects a byte buffer as its 2nd argument, and the file handle as its first argument.

Using function fileSeek it is possible to go to a certain position in the file. Function 'fileSeek' takes three arguments: the file handle, an offset and an origin. The file pointer will be set on position Offset, starting from Origin. Origin can be one of the following values:

- *fromBeginning, where Offset is relative to the first byte of the file (position 0);*
- *fromCurrent, where Offset is relative to the current position;*
- *fromEnd, where Offset is relative to the end of the file. (Offset is zero or negative).*

The new file position is returned upon success, otherwise -1 is returned.

```
func process(.fileName) do
  let inputFile := fileOpen(fileName, forReading)

  if inputFile = -1 then
    print('Error opening file "$(fileName)".')
    return nil
  end
  let fileSize := fileSeek(inputFile, 0, fromEnd)

  fileSeek(inputFile, 0, fromBeginning)

  let bytesRead := fileRead(inputFile)
  if length(bytesRead) < fileSize then
    print('Error reading file "$(fileName)".')
    return nil
  end

  fileClose(inputFile)

  return (.bytesRead, .fileSize)
end

print('Enter file name: ')
let file := readln()
let contents := process(fileName: file)
print(contents.bytesRead)

let size := contents.fileSize
print('size: $(size)')
```

Note that the result of the fileRead function is an array of bytes that need conversion to characters or string etc.

Function readFile does the trick in one go: open, read and close again.

```
let fileName := '/Users/test.txt'
let string := string(of: readFile(fileName))
print(string)
```

# APPENDIX

## GEAR KEYWORDS

and, break, case, class, continue, default, defer, do, else,

elseif, end, ensure, enum, exit, extension, false, for, func,

if, in, infix, inherited, is, lambda, let, loop, match,

nil, not, or, prefix, print, record, return, self, static, subscript,

switch, then, trait, true, use, val, var, when, where, while intentionally left blank