The Gear language

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 complex you can also do:

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

or

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

helloworld()
```

or

```
func greetings(to name)
  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.

INDEX

Variables, constants and values

var and let

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'.

Examples are:

```
var planet := 'Neptune'
let Pi := 3.1415926
var radius := 2
var area := Pi * radius^2
var ready := False
let A := "A
var initial := Null
```

Each variable declaration needs to be preceded by the keyword 'var' or 'let'. 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! 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 A is a Char, while variable initial is Null.

After a constant gets a value other than Null, it becomes immutable and cannot be changed anymore. Variables can be changed, however, after their initialization (other than Null) 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. The declarations must be separated by a comma.

```
var x:=1, y:=2, z:=3, s:='Saturn'
let v:=4, w:=5, p:='Pluto'
```

Values

Next to constants and variables, there's a third variable type called "value", and the associated keyword is 'val'. A value variable 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 I 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:

```
let Pi := 3.1415926
var radius := 2
val area
  var result := Pi * radius^2
```

```
  return result
end

print(area)  // prints 12.5663704
radius := 1
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. Assigning new values to constants or calculated values is not allowed.

Gear supports the regular constant types number, string, char and boolean. The types are not explicitly available as such; they are internal to the interpreter. In principal values are not implicitly converted to another type, with one exception. You can add any value to a string, whereby the added value is converted to string.

Strings and characters

A string is text between single quotes. Here are some examples:

```
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
```

Strings can be added to each other.

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

Numbers and Booleans can be added to strings.

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

Next to strings there are characters, which are preceded by a double quote, like this:

"a is the character a

"1 is the character 1

Characters can also be added to strings:

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

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 socalled string interpolation.

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

The values must appear in parentheses and preceded by a backslash. 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.

```
use system
let chars := Array('0123456789abcdefghijklmnopqrstuvwxyz')

for each char in chars do
  print('Char \(char) has ascii code \(ord(char)).')
end
```

The function ord() belongs to the standard available functions of Gear. It returns the ordinal value of a character. The system library provides functionality on arrays and iteration, for instance.
Unicode strings and characters are also supported though you need to use single quotes, like for instance in this example:

```
let c := '🄯'
print(c)
```

Strings may contain '\n' and '\t' for newlines and tabs.

Numbers

The number type in Gear is a double precision floating point, which also handles all required integers. Examples of Gear numbers are:

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

Booleans

There are of course two Boolean values True and False. They start with a capital.

Null

Null is a value. You can assign Null to variables, with a capital N. By assigning Null 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. Null cannot be used in calculations. This will result in a Gear runtime error.

Unassigned

Comparable to Nil in some languages. It's usually the result of a variable that doesn't exist or a non initialized variable. You can test variables if they are assigned through function assigned(variable) or simpler with ?variable.

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. A tuple's first element has index 1. You access a tuple's element by putting the index aftera dot '.'. You can assign a new value to a tuple element by using its index.

```
let one := (1, 'one')
print(one)                // (1, 'one')

let other := one
print(other)              // (1, 'one')

let a := 6, b := 'ten', c := True

let some := (a,b,c)
print(some)               // (6, 'ten', True)

var s := some.2
print(s)                  // 'ten'

some.2 := 'eleven'
print(some)               // (6, 'eleven', True)
```

Expressions

Basic arithmetic

Gear features the basic arithmetic operators you know from other languages:

Addition: **a + b**

```
Number + Number -> Number      // 8+9->17
String + Number -> String      // 'Hello ' + 42 -> 'Hello 42'
String + String -> String      // 'Hello ' + 'world' -> 'Hello world'
String + Char -> String        // 'Hello world' + "! -> 'Hello world!'
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
```

Remainder: **a % b**

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

But also 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
```

And special array operators:

Concatenation: **a >< b**

```
Array1 >< Array2 -> Array1Array2   // [1,2,3] >< [4,5,6] -> [1,2,3,4,5,6]
```

Dot product: **a :: b**

```
Array1 :: Array2 -> Number         // [1,2,3] :: [4,5,6] -> 32
```

Dot-product: a1b1+a2b2+a3b3 => 1*2 + 2*5 + 3*6 = 32
More detailed info on array operations in the chapter Arrays.

Logical operators

not: !	!True -> False, !!True -> True
and: &	True & False -> False
or: |	True | False -> True
xor: ~	True ~ False -> True

Relational 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
is	a is b	a is instance of b

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 the latter case it's better to use the match-expression discussed hereafter.

Match-expression

A match or case expression is rarely seen in programming languages. E.g. consider the following function, which calculates the Fibonacci range: 1 1 2 3 5 8 13 21 34 55 etc. for a given number n.

```
func Fibonacci(n)
  if n = 0 then
    return 0
  elseif n = 1 then
    return 1
  else
    return Fibonacci(n-1) + Fibonacci(n-2)
  end
end
```

This can be rewritten using a match expression, which is not only shorter in code, but also more clear to read.

```
func Fibonacci(n) =>
  match n
    if 0,1 then n
    else Fibonacci(n-1) + Fibonacci(n-2)

func Factorial(n) =>
  match n
    if 0,1 then 1
    else n*Factorial(n-1)
```

A match expression consists of 1 or more if-then parts and a mandatory final else-clause. Here's an example with more if-then cases:

```
var someCharacter := 'u'

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

There is no limit on the number of if-then cases.

Arrays and dictionaries

Both arrays and dictionaries are created using square brackets: [] for arrays and [:] for dictionaries. Elements of both are accessed by using an index (arrays) or a key (dictionary) between the square brackets.

```
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.

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

There are two ways of adding new items to an array. One is by using the concat '><' operator, and the other is by using the .add(value:) function, which comes with the system 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 requires the use of the external parameter name 'value:'.

Adding multiple new values to an existing array can also be done using the concat operator or the .add(list:) function.

```
groceries ><= ['Milk', 'Bread', 'Flower']
groceries.add(list: ['Milk', 'Bread', 'Flower'])
```

or

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

If you insert an item at a specific index take care that for arrays the first index is 0.

```
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, remove() function or extract() function.

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

An empty array is created in either of the following ways:

```
var emptyArray := Array()
var anotherEmptyArray := []
```

An empty dictionary is created in a similar way:

```
var emptyDict := Dictionary()
var anotherEmptyDict := [:]
```

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
```

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

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

With the use statement you more or less 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 arrays
var a := [0,1,2,3,4,5,6,7,8,9].filter(x=>x%2=0)
print(a)  // [0, 2, 4, 6, 8]
```

In file arrays.gear the functionality for the use of the standard Array type is defined. For example the functions filter, map and reduce are defined in this file.

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.

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, repeat and for statements. A special case is the for-each loop, which is based on an iterator pattern. All loops have in common that the statement block starts with 'do' and ends with 'end'.

while

The while-statement is used to repeat zero or more statements while a certain condition in True. The 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
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.

for

Much has been written about the for-do 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.

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
```

repeat

The repeat loop is more or less copied from the Pascal language. It is a great statement for its simplicity and straightforwardness.

```
var a := 5
var b := 1
let c := 3.1415926
repeat
  a := a - 1
  print(a*b*c)
until b>a
```

The repeat loop repeats the statements until a condition is met.

```
var n := 10
repeat
  print('n=', n)
  n-=1
until n<=0
```

Contrary to the while and for loops, the repeat loop is executed at least once.

for each

As mentioned, a for-each loop is a special loop as it is based on iterators. For arrays, dictionaries and ranges these iterators are already predefined in libraries arrays.gear, dictionaries.gear and ranges.gear. Using the system library is sufficient to make all of them available at once.

```
use system
let scores := [8.3, 7.7, 9.0, 5.4]
var sum := 0
for each score in scores do
  sum += score
end
let average := sum / scores.count

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

The variable ‘score’ is declared inplicitly after the ‘each’ keyword and is of the same type as an element of the ‘scores’ array.

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.

```
use system
let scores := ['Math': 8.3, 'Science': 7.7, 'English': 9.0, 'Economy': 5.4]
var sum := 0
for each 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.
```

You iterate over a range by using either the Range(from, to) function or by using the dotted notation.

```
for each n in 1..10 do  // or use Range(1,10)
  print(n^2)
end
```

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

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

Value list.count is not included in the range.

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

```
use system

let list := [1,2,3,4,5,6,7,8,9,10]

for each item in list where item%2=0 do
  print(item, terminator: ' | ')
end
print()
```

The even numbers are printed, separated by vertical bars: 2 | 4 | 6 | 8 | 10 |

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.

It is also possible to declare the variable 'average' as part of the if-statement. The scope of the variable will 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.

```
use system
let scores := [8.3, 7.7, 9.0, 5.4]
var sum := 0
for each score in scores do
  sum += score
end
```

```
if let average := sum / scores.count where average > 6.0 then
  print('You passed!')
elseif (average>5.0) & (average<=6.0) then
  print('You need to work harder!')
else
  print('You failed!')
end

print('gimme a number: ')
if var x := toNum(readln()) where x <> Null then
  x := x^2
  print(x)
else
  print('That''s not a number')
end
```

switch

What the match expression is to expressions (which can be used for pattern matching), is the switch statement to statements, which can be used for quick branching in the code. 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 else clause.

```
let char := 'r'
switch char
  case 'a', 'e', 'i', 'o', 'u':
    print('\(char) is a vowel with ascii value \(ord(char)).')
  case 'b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm',
     'n', 'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'y', 'z':
    print('\(char) is a consonant with ascii value \(ord(char)).')
  else
    print('\(char) is not a vowel nor a consonant. Ascii value: \(ord(char)).')
end
// r is a consonant with ascii value 114.
```

The switch can be used to test against any value. You can even test if an instance of a class is of a certain class type, by using the 'case is' construct.

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

class First (Root)
end

class Second (Root)
end

class Third (Second)
end
class Fourth
end

var sec := Second()

switch sec
  case is Root: print('Root')
  case is First: print('First')
```

```
  case is Second: print('Second')
  case is Third: print('Third')
  else
    print('None of the above')
end
```

Note that after a case clause is executed, the switch statement is finished automatically.

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.

```
a := 1
b := a + 2

a := a + 1, which is the same as
a += 1
```

Gear supports the following assignment operators:

token	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
><=	a ><= b	a := a >< b	Concatenation

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.

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

while True do
  print('Guess a number (1 to \(max)): ', terminator: '')

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

  ensure (guess > 0) & (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.

The **readln()** function reads a string input from the command line. This can be converted to a number by using the toNum() function. If it cannot be converted the result will be Null.

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
```

However, a more convenient way to use break is making the condition part of the break statement, like this:

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

Both solutions are allowed in Gear. The first solution comes in handy if you need to add more complex actions in the if-then statement. The second option adds convenience.

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.

Functions

A function is declared with the 'func' keyword. It has a name and optional parameters (the parentheses are required). Then follow statements and a final 'end'.

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

The function is called by its name and the required arguments between parentheses.

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

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)
  print('Hello \(person), it''s \(day) today.')
end
```

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

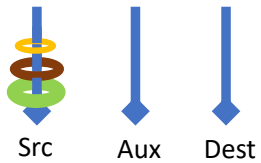
```
hello(name: 'Jerry', day: today())
```

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)
  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.

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, or use a tuple. But you can also return dictionaries, arrays, enums and all simple types such as strings, numbers and Booleans.

```
use system

func calculate(.scores)
  class Statistics
    var min := 0
    var max := 0
    var sum := 0
  end
  var statistics := Statistics()

  for each 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.

```
use system
func calculate(.scores)
  var min := 0, max := 0, sum := 0, avg := 0
  for each 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.1)')
print('max: \(statistics.2)')
print('sum: \(statistics.3)')
print('avg: \(statistics.4)')
```

The benefit is you don't have to declare an internal class, however you loose the variable names.

Another example:

```
func test()
  return (2, 'two', 'twelve')
end

let p := test()
print(p)                // (2, 'two', 'twelve')
p.3 := 'fifteen'
print(p)                // (2, 'two', 'fifteen')
```

A tuple may consist of function expressions.

```
var w := (x=>x^2, x=>x^0.5)
print(w.1(5))     // 25
print(w.2(16))    // 4
```

The respective functions can be called by using the index number between square brackets and the argument between parentheses.

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

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

function as first-class type

A function can return another function as its value. This implies that a Gear function is a first-class type.

```
func startAt(x)
  func incrementBy(y)
    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
```

arrow functions

Functions that return only a single expression can be written in a simpler manner.

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

can also be formulated through an arrow function.

```
func add(a,b) => a+b
func sub(a,b) => a-b
func mul(a,b) => a*b
func div(a,b) => a/b

print(add(35,7))    // 42
print(sub(50,8))    // 42
print(mul(21,2))    // 42
print(div(168,4))   // 42
```

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)
  return function(x,y)
end

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.
```

anonymous functions and closures

Gear supports anonymous functions. An anonymous function (function literal, lambda abstraction, or lambda expression) is a function definition that is not bound to an identifier. Anonymous functions 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)
  return function(x,y)
end

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

In fact, the keyword 'func' is not even required in such cases, so you can simplify even further.

```
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 can be omitted.

```
use system
func calc(times n, function)
  for each k in 1..n do
    print(function(k), terminator: ' ')
  end
  print()
end

calc(times: 5, x=>x^2)  // 1 4 9 16 25
calc(times: 5, x=>x^3)  // 1 8 27 64 125
calc(times: 5, x=>x^4)  // 1 16 81 256 625
```

Using anonymous 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)
  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 – 2*x + 4
2) f(x) = x^3
3) f(x) = x^2 + 4*x - 21

```
print(integral(func(x)=>x^2-2*x+4, from: 0, to: 1, steps: 10000))
print(integral(func(x)=>x^3, from: 0, to: 1, steps: 10000))
print(integral(func(x)=>x^2 + 4*x - 21, from: 0, to: 1, steps: 10000))

3.3333333325
0.24999999875
-18.6666666675
```

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

higher order functions

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

The **forEach** function applies an anonymous function or closure to each item in an array.

```
use system
var numbers := [1,2,3,4,5,6,7,8,9,10]
numbers.forEach(
  func(x)
    let squared := x^2
    print('Number \(x) squared is: \(squared).')
  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
```

functions as variables

In Gear variables can be declared as a function.

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

var helloWorld := func()
  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.

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

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
```

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
```

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 above defined functions the three identifiers are:

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

As an example, consider the following.

```
func stringOf(.number)
  return toStr(number)
end

func stringOf(.boolean)
  return toStr(boolean)
end

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

var a := stringOf(number: 3.14)
print(a)

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

var b := stringOf(boolean: !(True|False))
print(b)
```

standard 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
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
random()	random number between 0 and 1, 1 not included
randomLimit(n)	random integer number between 0 and the limit, limit not included
toNum(s)	tries to convert a string to a number, Null if not succesful
toStr(n)	tries to convert a number to a string, Null if not succesful
readln()	returns input from the keyboard, Null if not succesful
assigned(v); ?v	return True if v is not Unassigned
length(v)	returns length of string, array or dictionary

Classes

A class has a name, some fields and methods, and a constructor. A class declaration starts with the keyword 'class', followed by the class name.

```
class Car
  var name := ''
  func show()
    print(self.name)
  end
  init(name)
    self.name := name
  end
end

var car := Car('Volvo')
print(car.name)    // "Volvo"
car.show()         // "Volvo"
```

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

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
  var x:=0, y:=0
  init(.x, .y)
    self.x:=x
    self.y:=y
  end
  func distance(to point)
    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
```

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
  var field := 'field'
  init(field)
    self.field := field
  end
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.

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

```
class Car
  let brand := Null
  init(brand)
    self.brand := brand  // brand has a value now and becomes immutable
  end
end

car := Car('Volvo')  // brand has value 'Volvo'
car.brand := 'Merc'  // this is not allowed, since brand is a constant
```

classes inside classes

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

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

class Person
  var name := 'Person',
      birthDate := Date()
  init(name, .birthDate)
    self.name := name
    self.birthDate := birthDate
  end
  func toString()
    return self.name + ': ' + self.birthDate.toString()
  end
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 between parentheses after the class name. A class can have at most one parent.

```
class Vehicle
  ...
end

class Car (Vehicle)
  ...
end

class Train (Vehicle)
  ...
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
  func method()
    print('Parent Method')
  end
end

class Child(Parent)
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)
  func method()
    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
  var radius := 1
  val area := pi() * self.radius^2
  init(.radius)
    self.radius := radius
  end
end

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

- inherited init(params) or
- inherited(params), which is the short version.

Both are allowed and it only applies to init(). Inheriting other functions always require the name of the function to inherit.

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

class Apfelstrudel(ApplePie)
  func serve()
    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.
```

arrays with class instances

You can store class instances in an array. Class Person is defined in file person.gear in your 'current' folder.

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

```
    self.address := address
    self.age := age
    self.income := income
    self.cars := cars
  end
end
```

The main program makes use of this file and system.gear.

```
use person
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)
let totalIncome := people.map(person=>person.income).reduce(0, (x,y)=>x+y)
let allCars := people.flatMap(person=>person.cars)
```

The last one creates a flat list of all available 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.

```
class Math
  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.

Extensions

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.

```
extension Array
  func toString() => self.toString()
  val count := length(self)
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.

A func or val defined in an extension overwrites earlier defined ones.

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
  var field := Null
  init(field)
    self.field := field
  end
end

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

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

Look in arrays.gear and dictionaries.gear for extensions on both types.

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
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
  func instanceOf(Class)
    return self is Class
  end
end

class Number: Instance
  var code := 0
  init(code)
    self.code := code
  end
  func write()
    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
end

trait Two
end

trait Three: One, Two
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.

```
use system

trait ValidatesUsername

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

  // checks user name characters are valid
  func isUsernameValid(username)
    let chars := Array(username)
    for each char in chars do
      ensure self.validUNChars().contains(value: char) else
        return False
      end
    end
    return if chars.count < 8 then False else True
  end

end

trait ValidatesPassword

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

  // checks password characters are valid
  func isPasswordValid(password)
    let chars := Array(password)
    for each char in chars do
```

```
      ensure self.validPWChars().contains(value: char) else
        return False
      end
    end
    return if chars.count < 8 then False else True
  end

end

class Login: ValidatesUsername, ValidatesPassword
  var userName := ''
  var passWord := ''

  init(userName, passWord)
    self.userName := userName
    self.passWord := passWord
  end

  func loginEntered()
    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')
```

Arrays

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]
```

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

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

array definition

An array can be defined in several ways. An array can be defined like we did for class definition, creating an array-type so to speak. We can also declare an array variable directly. And for both ways, we can enforce type consistency, if we want to create an array of a specific defined type.

When you define an array type, it's also possible to define declarations such as functions and value properties. Variables, constants and other declarations such as classes are not allowed.

An instance of an array is called like a function:

```
array Numbers
  [0,1,2,3,4,5,6,7,8,9]

  func write()
    print(self)
  end
end

var numbers := Numbers()
numbers.write()
```

standard Array

Gear has a default array type, called 'Array'. A variable declared as

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

is implicitly defined as type Array. It is the same as using Array explicitly.

```
var a := Array(1,2,3,4,5)
```

Internally, the Interpreter has defined a standard array type called 'Array', like this:

```
  array Array [] end
```

It is an empty shell without any functionality. That has to be added. All array expressions use this type. So for example, in the following variable declaration:

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

variable 'a' is of type Array.

Of course you can create your own array type, and add funtionality to it. For example:

```
array Numbers [] end
var numbers := Numbers([1,2,3,4,5])
```

But be careful if you do this since all operations between arrays of type Number must be the same, meaning, e.g. that

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

returns a Boolean False, since [1,2,3,4,5] is of type Array and not of Numbers. Even concatenation of these two will not work, because they are not the same.

```
let other := Numbers(1,2,3,4,5)
compare := numbers = other
```

In the last case the comparison returns True.

array index

Like in any other programmig language, in Gear an array item is accessed through its 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.

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

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

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]
```

standard functions

The standard available type Array comes with a few standard functions.

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.

Standard function	Function in type Array	Example
length(array)	.count	n := numbers.count
listInit(array, size, fillChar)	.set(size:, fillChar:)	numbers.set(size: 10, fillChar: '0')
listAdd(array, value)	.add(value:)	numbers.add(value: 10)
listAddList(array, value)	.add(list:)	numbers.add(list: [1,2,3,4,5])
listInsert(array, index, value)	.insert(at:, value:)	numbers.insert(at: 1, value: 9)
listDelete(array, index)	.delete(index:)	numbers.delete(index: 10)
listRemove(array, value)	.delete(value:)	n := numbers.remove(value: 'a')
listExtract(array, value)	.delete(value:)	a := numbers.extract(value: 'a')
listContains(array, value)	.contains(value:)	b := numbers.contains(value: 'a')
listIndexOf(array, value)	.index(of:)	n := numbers.index(of: 9)
listRetrieve(array, index)	.retrieve(index:)	a := numbers.retrieve(index: 0)
listClear(array)	.clear()	numbers.clear()
listFirst(array)	.first	a := numbers.first
listLast(array)	.last	a := numbers.last

instantiation

When the argument list in the call to create a new array is empty, the elements as defined in the array definition become the elements of the instance. However, if there are any arguments, these are taken as the contents for the array, and the predefined ones are discarded.

Then, there is a difference between 1 and more arguments. If there's exactly one argument than this must be an array expression. If there are more, it will take all values and create an array expression.

```
array myArray
  []  // empty, no elements
  val count := length(self)
end

var a := myArray([1,2,3,4,5]) // call with array expression
print(a)          // [1,2,3,4,5]
print(a.count)    // 5

var b := myArray(["a, "b, "c]) // array expression
print(b)          // ['a', 'b', 'c']
print(b.count)    // 3
var c := myArray(b)  // create a copy

var d := myArray(pi(), 'Hello', "z, 42)  // call with multiple arguments
var e := myArray(["a, "b, "c], [1, 2, 3, [4, 5]], pi())  // multiple nested arguments
var f := myArray('Pretty cool!') // f is an array of characters
```

This way you can define your own master array type and include all functionality you want.

array math

Arrays come with built-in mathematics. Math on arrays support the following standard operations. Note that the types of the arrays must be the same.

- concatenation, e.g. [a, b, c] >< [d, e, f] results in [a, b, c, d, e, f]. Here we introduce a new operator '><'. In some languages the '+' operator is used for this, however that's not the right way.
- addition, e.g. [1, 2, 3] + 1 results in [2, 3, 4], and [1, 2, 3] + [3, 2, 1] results in [4, 4, 4].
- multiplication, e.g. [1, 2, 3] * 10 results in [10, 20, 30].
- subtraction and negation
- division
- dot product
- comparison of arrays
- check if an element is in an array

The Gear array system supports the creation of nested arrays, or multiple dimension arrays known as matrices.

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

var n := [[3,2,1],
          [6,5,4],
          [9,8,7]]

var x := 2      // a simple value
```

It is possible to do all kinds of math on arrays and matrices.
Array concatenation is defined by the >< operator.

m >< n = [[1,2,3],[4,5,6],[7,8,9],[3,2,1],[6,5,4],[9,8,7]]

When two arrays or matrices are added, subracted, multiplied or divided together, their sizes and types must be equal.

m + n = [[4,4,4],[10,10,10],[16,16,16]]

m + x = [[3,4,5],[6,7,8],[9,10,11]]

m * n = [[3,4,3],[24,25,24],[63,64,63]]

m * x = [[2,4,6],[8,10,12],[14,16,18]]

m - n = [[-2,0,2],[-2,0,2],[-2,0,2]]

m - x = [[-1,0,1],[2,3,4],[5,6,7]]

Division works the same way. Other operation that are allowed are negation, test for equal and the dot-product.

-m = [[-1,-2,-3],[-4,-5,-6],[-7,-8,-9]]

m = n = False
m <> n = True

m :: n = not allowed! Only non-nested arrays are supported.

[1,2,3,4,5,6,7,8,9] :: [3,2,1,6,5,4,9,8,7] = 273

By equal sizes, we mean the number of array elements in both arrays are the same. Given the following matrix and vector, the mentioned operations are allowed.

```
var A := [[1,2,3],
          [4,5,6],
          [7,8,9]]
var V := [10,20,30]
print(A*V)

// [[10,20,30],[80,100,120],[210,240,270]]
```

List comprehension-building lists

Gear supports list comprehension for single variables. The system library must be used for this to be available. The official mathematical notation for building a set though the set builder notation is:

result = { 2x | x ∈ 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 must be 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. InputSet can be an array or a range.

```
var result := { 2*x for x in 1..100 where x^2>3 }
```

The result is an array filled with numbers that satisfy the conditions.

In terms of Gear code this could be done using filter and map as well.

```
let list := [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
var result := list.filter(x=>x^2>3).map(x=>2*x)

print(result) // [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

It is possible to create a function that simulates a list comprehension:

```
use arrays

func listComp(apply transform, on input, when predicate)
  return input.filter(predicate).map(transform)
end

let list := [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

print(listComp(apply: x=>2*x, on: list, when: x=>x^2>3))
```

Alternatively, one can use the flatMap func (given the same list):

```
var evens := list.flatMap(n => if n%2=0 then [n] else [])
print(evens)

var evensSquared := list.flatMap(n => if n%2=0 then [n*n] else [])
print(evensSquared)
```

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 inputSet where x^2>3 }
```

As input set you can use arrays 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 Range(1,10) 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. In fact any iterable class object can be used as input.

You can use a dictionary as an input set, for example:

```
use system

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 has two standard fields: 'key' and 'value'. Another one:

```
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]
```

Ranges

Two examples of the use of ranges, that perform an operation, which produces exactly the same result: an array of squared numbers if the number is even.

```
use system

var list := []
for each n in 0..<1000 where n%2=0 do
  list.add(value: n^2)
end

var list := {n^2 for n in 0..<1000 where n%2=0}
```

Though the second one takes 3 to 4 milliseconds longer, it is much more concise and readable than the first one.

In Gear library 'ranges.gear', which is used in library 'system.gear', the classes Range and RangeIterator are defined:

```
class RangeIterator
  var index := 0
  var range := Null
  var stepBy := 1

  init(range)
    self.range := range
    self.index := range.from
    self.stepBy := range.stepBy
  end

  val hasNext := self.index <= self.range.to

  val next
    var result := self.index
    self.index += self.stepBy
    return result
  end
end

class Range
  var from := Null
  var to := Null
  var stepBy := 1

  init(from, to)
    self.from := from
    self.to := to
  end

  val iterator := RangeIterator(self)
end
```

You use it like this:

```
use system
let list := [0,1,2,3,4,5,6,7,8,9]
for each i in Range(0,list.count-1) do
  print(i)
end
```

or:

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

If you add the function step(by: number) to a range, iterating over the range will be in the steps as defined.

```
for each i in (0..20).step(by:2) do
  print(i)
end
```

This only prints the even numbers. The following generates all prime numbers from 3 up to 1000.

```
func isPrime(number)
  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

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

The step(by:2) makes sure we don't evaluate even numbers, since they cannot be prime numbers, except for 2 of course. Alternatively, you can generate primes using the list comprehension method in a one-liner.

```
let primes :=
  {n for n in (3..<1000).step(by:2) where
    {m for m in (1..sqrt(n)).step(by:2) where n%m=0}.count=1}
```

The first solution is 8 times faster though…

Dictionaries

Like an array, a dictionary can also be defined in two ways: as a type instance or as an expression. In the latter case the standard type Dictionary is applied. Here are a few examples:

```
dictionary Words
  [:]
  val count := length(self)
  func toString() => self
end

var words := Words()
words['pair'] := 'couple'
words['collection'] := 'group'
print(words.toString())

var newDict := [:]
var family := ['Harry': 37, 'Susan': 29, 'John': 8, 'Mary': 5]
```

In the latter two cases, the predefined type Dictionary will be used.

As you can see, when you define a dictionary type, it's also possible to define declarations such as functions and values inside a dictionary. Variables and other declarations such as enums and classes are not allowed. The dictionary elements can either be empty [:] or filled with expressions [expr:exper, expr:expr]. A dictionary expression will always be of type Dictionary, like we have type Array for array expressions.

Also, for dictionaries, a library file is available which holds extensions to type Dictionary.

```
extension Dictionary
  val count := length(self)
  func toString() => self

  func add(.key, .value) => dictAdd(self, key, value)
  func add(.list) => dictAddList(self, list)

  func delete(.key)
    dictDelete(self, key)
  end

  func contains(.key) => dictContainsKey(self, key)
  func contains(.value) => dictContainsValue(self, value)
  func value(of key) => dictValueOf(self, key)
  func key(of value) => dictKeyOf(self, value)

  func set(.sorted)
    dictSetSorted(self, sorted)
  end
  func sort()
    dictSort(self)
  end

  func clear()
    dictClear(self)
  end
end
```

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

Standard function	Function in Dictionary	Example
length(dict)	.count	words.count
dictAdd(self, key, value)	.add(key:, value:)	words.add(key: 1, value: 'One')
dictAdd(self, list)	.add(.list)	words.add([1: 'One', 2: 'Two'])
dictDelete(dict, key)	.delete(key:)	words.delete(key: 1)
dictContainsKey(self, key)	.contains(key:)	a := words.contains(key: 2)
dictContainsValue(self, value)	.contains(value:)	a := words.contains(value: 'Two')
dictValueOf(self, key)	.value(of:)	a := words.value(of: 2)
dictKeyOf(self, key)	.key(of:)	a := words.key(of: 'Two')
dictSetSorted(dict, sorted)	.set(.sorted)	words.set(sorted: True)
dictSort(self)	.sort()	words.sort()
dictClear(self)	.clear()	words.clear()

As an example, see this code:

```
use system
var numbers := [:]

numbers.add(key: 0, value: 'Zero')
numbers.add(key: 1, value: 'One')
numbers.add(key: 2, value: 'Two')
numbers.add(key: 3, value: 'Three')
numbers.add(key: 4, value: 'Four')
numbers.add(key: 5, value: 'Five')
numbers.add(key: 6, value: 'Six')
numbers.add(key: 7, value: 'Seven')
numbers.add(key: 8, value: 'Eight')
numbers.add(key: 9, value: 'Nine')
numbers.add(key: 10, value: 'Ten')

numbers.add(list: [11:'eleven', 12:'twelve', 13:'thirteen'])
numbers.set(sorted: True)
print(numbers)

print(numbers.value(of: 7))
print(numbers.key(of: 'Eight'))

numbers.delete(key: 10)
print(numbers)
print(numbers.contains(key:3))
print(numbers.contains(value:'Three'))
print(numbers.contains(value:'blabla'))
print(numbers[5])

numbers[11] := 'not eleven'
print(numbers)
print(numbers[11])
```

Enums

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

```
enum Color
  (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'
print(Color.count)     // prints '6'
```

As you see an enum will have a default field called 'name' that holds the string representation of an enum plus a field that holds the number of items: count.

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

```
enum TokenType
  (Plus='+', Min='-', Mul='*', Div='/', Rem='%')
end
var tokenType := TokenType.Mul
print(tokenType.name)    // prints 'Mul'
print(tokenType.value)   // prints '*'
```

Next to this you can create enum sets: use the keyword 'case' followed by the set-name, like this:

```
enum Color
  (None
   case Primary: Red, Yellow, Blue
   case Secondary: Orange, Green, Brown)
end

var myColor := Color.Blue
if myColor in Color.Primary then
  print('\(myColor.name) is a primary color.')  // "Blue is a primary color."
end
```

The 'Elements' field holds the list of all enum elements.

```
use system
for each color in Color.Elements where color in Color.Primary do
  print(color)
end
```

Color.Elements provides access to the total list of enum elements in Color.

```
print(Color.Elements)  // [None, Red, Yellow, Blue, Orange, Green, Brown]
```