Oforth Programming Language

Manual

Franck Bensusan

Table of contents

1 Introduction

1.1 What is Oforth ?

Oforth is a Forth dialect that implements an Object Oriented Programming model.

It is a stack based language : it uses a data stack to pass parameters to functions and to retrieve results. It uses a RPN notation : there is no instructions, just words that are executed one after the other.

It is an extensible language : from built-in words available at start-up, you create new words. The system makes no difference between built-in and user-defined words.

Oforth is an interactive language : there is no separate compilation phase and words are compiled as soon as they are sent to the interpreter. And they are available immediately to create new words or to test them.

Oforth is dynamically typed : all items on the stack are objects and each object has a type. Polymorphism is supported through messages that can be implemented for each class. Objective is to have an intuitive and simple OOP implementation ie being able to call messages exactly the same way you would call classic words. There is no special syntax, no current object : when a message is called, the inner interpreter checks the object on top of the stack and calls the method this object respond to : calling a method is no more complicated than calling a function and many classical Forth words are actually implemented as methods (like +, -, ...).

OOP model is as "pure-OO" as possible, which means that everything is an object, even OO meta-model (functions, messages, classes, ...) and all classes, regardless of class hierarchy, can implement any message.

Oforth comes with an incremental garbage collector (GC). It is not mandatory to handle objects de-allocation but it is necessary for various advanced features (closures, ...). If necessary, you can decide which objects will be handled by the GC and which will be handled manually.

Immutability is enforced : by default, objects are immutable, but you can choose to create mutable objects.

Oforth implements a task/channel model to handle multi-tasking : tasks are small objects with isolated memory that can communicate with other tasks using channels.

Security is enforced : arrays bounds and integer overflow are checked. Direct access to memory is possible but limited.

1.2 For forthers...

If you come from Forth, there are some differences that can surprise you.

Why some words have been renamed compared to Forth ?

When possible, Oforth uses the same names than Forth. But there are 3 topics that make impossible the use of some of these names : the support for arrays syntax, the support for json syntax and polymorphism.

Oforth interpreter can recognize arrays and 3 words are dedicated to this : [ ] and , . Corresponding Forth words have no correspondence in Oforth.

Interpreter can recognize jsons objects (if you load the json package) and some words are dedicated to support the json syntax : ' , { and } . Corresponding Forth words are renamed :

- ' (tick) is now used for characters only and is replaced by # (Forth # family words are not implemented in Oforth)
- { and } are replaced by (and) (and commentaries are only possible with \ and -- words)
- There is no Forth double word syntax (Oforth integers have arbitrary precision).

And last point, Oforth support polymorphism even for basic types, so many Forth words have been removed. For instance, + works for integers, floats, strings, ...

You can find a cross reference between ANS Forth words and Oforth words in the last chapter of this manual.

The interpreter

Oforth's interpreter is very similar to any Forth interpreter but you might be surprised by some behaviors :

- Names are case sensitive and you have to enter the right name (many words are lower case). For instance, Bye, bye and BYE are 3 different names.
- Redefinition of words is not allowed
- forget : name will just forget the word <name>, not all word until <name>.
- As the interpreter detects some characters as separators (see 2.1), some characters are not allowed as prefix of suffix for names.
- Embbeded comments ((...)) are not available, just comments to the end of line using \ or --. ( ) are used as local definer.
- The return stack is not accessible (R>, >R, ...) and replaced by locals.
-

1.3 Words naming conventions

Oforth built-in words follow some naming conventions. The main conventions used are :

Classes and properties begin with an uppercase :

```
Integer Array Comparable
```

Functions/messages names generally begin with a lowercase:

```
dup kindOf? name is? digit? >upper
```

Constants are often all uppercase. If related to a particular class, the name begins with this class and a dot :

```
File.READ System.VERSION CELLSIZE
```

Functions/methods that parse the input stream and/or need something after (which is not the natural RPN notation) have a name ending with ':' :

```
new: loop: : method: try: const: tvar: ifTrue: else:
```

Functions/methods can have a prefix or suffix to describe its purpose. For instance :

```
prefix ">" is used for a conversions (>float, >string, ... )
suffix "?" is used for words that return a boolean ( even? odd? is? ... )
```

Those rules are not mandatory, but give important information about a word's purpose.

1.4 Installation

Oforth 32bits runs on 32bits and 64bits operating systems. It comes as an archive file that contains everything necessary to run Oforth :

- Oforth binary : it is the only executable into the archive (no library, ...)
- oforth.of : the file loaded at startup.
- lang repository : all features loaded at startup.
- packs repository : optional packages.

The binary included into this archive supports the following features :

- Float
- TCP
- FFI
- Multi-thread.

Oforth requires the environment **OFORTH_PATH** variable to be created and set. You can find instructions on the web if you don't know how to create an environment variable on your specific operating system.

OFORTH_PATH value is a list of directories. Oforth will try to find sources and packages into those directories. It should at least include the directory where you have extracted the archive.

On Windows platforms, this value is a list of directories with ';'. haracter as separator. Characters

for directory names can be '/' or '\'. For instance :

```
OFORTH_PATH=\Home\Oforth;\Home\Oforth\test
```

On Linux and Mac OS platforms, this value is a list of directories with ':' character as separator and '/' character for directory names. For instance :

```
OFORTH_PATH=/users/oforth/oforth:/home/oforth
```

You can also have this variable set each time you launch a command prompt. Instructions can also be found on the web according to your system.

Finally, if you want to run Oforth from everywhere, you can also add your installation directory to your PATH variable.

Before running Oforth, check that OFORTH_PATH variable set correctly. On Windows, open a command prompt and run:

```
set
```

On Linux or Mac OS, open a command prompt and run :

```
env
```

The OFORTH_PATH variable must appears with the correct value to be able to run Oforth.

Currently, Oforth is a 32bits application so, if libraries are used, the 32bits versions must be installed (if not present).

1.5 Builting Oforth

Oforth is an open source project. You can build the Oforth binary with provided sources.

You just need to run "make all" to build Oforth. This will create the binary into the directory where the sources have been unzipped. You can now copy it where you want (and replace the executable provided in the archive if you want).

Into the makefile file, you can set some options for the image you want to build :

- Add Float support (default is yes)

- Add TCP support (default is yes)

- Add multi-core support (default is yes)

- Add FFI support (default is yes).

- Add debug support (default is no).

1.6 Invoking the interpreter

Oforth is invoked from a command line. Interpreter is launched using "--i" command line option (if you launch oforth without the --i option, nothing will happen as oforth will wait for input):

```
oforth --i
```

This command starts the interpreter mode : you can directly type commands and see results. You can leave the interpreter using **bye** command.

Various optional command line options can be set :

```
oforth [ options ] [file]

Launch options :
   --i    : Interpret mode
   --P"s" : Perform string s
   [file] : Perform code into file.

Behavior options :
   --t   : Tests are compiled and checked (default is not checked)
   --Wn  : Max number of workers (default is 1). If 0, use number of cores detected.
   --Sn  : Max size (in objects) for main data stack (default is 256 objects)
   --Mn  : Max memory to use by Oforth process (Ko)
    --C  : Removes some optimizations and adds checks (possible only if debug built).

Garbage Collector options :
   --XTn : Set number of milliseconds between 2 GC (default is 120 ms)
   --XMn : Set min allocated memory (Ko) for GC to run (default is 1024 Ko)
   --XGn : Set GC ticks by step during mark & sweep phase (defaut is 6000)
   --XAn : Set app ticks by step during GC (defaut is 300)
   --XVn : Set GC verbose level (0 to 3, default is 0)
```

Examples :

```
oforth
Error : if you don't launch the interpreter (--i), you must provide a file of a
command to execute.

oforth --i
Launch Oforth interpreter : you can execute commands until bye is typed.

oforth myfile.of
Launch Oforth, load myfile.of file and exit.

oforth --P"test" myfile.of
Launch Oforth, load myfile.of file, run "test" word and exit.

oforth --i --P"myfile.of load"
Launch Oforth interpreter, load myfile.of and prompt for commands.

oforth --i --P"import: date"
Launch Oforth intepreter, import date package and prompt for commands.
```

1.7 Running programs

As Oforth comes with an interpreter, you can test interactively your code. But, at one time, you will want to keep your work into a file and load it at startup.

By convention, Oforth sources are stored into .of files, but this is not mandatory.

If you want to load the "test.of" file and then run the #test function :

```
oforth --P"test" test.of
```

But, in order to test your code, it is often more convenient to load your file(s), and execute tests into the interpreter. To do this, you can load your file into the interpreter :

```
oforth --i
"test.of" load
```

If you update your source files, you must leave the interpreter and reload the file. To avoid loading your file(s) each time you launch the interpreter, you can load it using the command line :

```
oforth --i --P"\"test.of\" load"
```

Then, you will launch the interpreter with your file(s) already loaded, ready to test your code.

And, a time will come when you will have packages (see Packages chapter). You can do the same thing and load you package before launching the interpreter :

```
oforth --i --P"import: mypackage"
```

You can also use Oforth as a pipe or redirect input or output

```
oforth < source.of >result.txt
```

2 Interpreter and data stack

This chapter explains how the interpreter works and the concept of data stack. The best way to learn this is to run an Oforth interpreter (using oforth --i) and run all examples to see the results.

2.1 The outer interpreter

When you launch Oforth with --i option, the interpreter (called outer interpreter) is launched, waiting for input : you can play with the stack, create words, load files, ... : Oforth is an REPL system (Read Eval Print Loop).

Interpreter is very simple: after you type something and press the ENTER key, it reads the first name (by collecting all characters until a space is encountered) and executes it. After this name is executed, the interpreter reads the next name and execute it. And so on, until there is no more name to execute (or the word executed is **bye**). There is no instructions, just words separated by spaces.

A name is a sequence of characters terminated by a separator (or end of line). An important difference with most Forth interpreters is that Oforth declared various characters as separators (and not only space) to delimit them. Those characters are :

```
# @ { } , : ; ( ) ! $ [ ] | ' " // \ -> #[ => 0x 0b #! :=
```

For instance, if you type [[, the interpreter will detect 2 words ([and [ ) and not one word ([[ ).

When the interpreter has found a name, it checks if it is the name of a word (a built-in word or a user-defined word). For this, it searches this name into the **dictionary**, a place where all words are stored. If a word with this name is found, the interpreter executes it, then handles the next name. If the name is not found, the interpreter tries to detect if the name read is an integer or a float. If so, it pushes this number on the stack. If not, an exception is raised and the rest of the line is ignored.

Interpreter is case-sensitive : if you try **Bye** or **BYE**, interpreter will not recognize the word **bye**.

2.2 The data stack

Oforth is a stack based language. Parameters needed for a word are pushed on a stack before calling the word and those words will also push (if any) their return values on this stack. This stack is called the **data stack** (or ... the stack).

The data stack is a LIFO stack (Last In First Out) : the last pushed object will be the first popped. All words have access to this stack and each time you create an object or type a number, it is pushed on this data stack.

There is only one data stack (in fact, one by task in a multi-tasking context), whatever the kind of objects : integers, floats, strings, arrays, ... use the same data stack and each object uses only one slot on the data stack.

Each word interacts with the stack (by removing parameters and pushing results). For instance:

```
sqrt          \ f -- f'
```

means that the word **sqrt** is expecting a float on top of the stack, consumes it and, when it has done its job, pushes a float on the stack as its result (\ word is for declaring a comment till the end of the line).

So you can try :

```
12.3 sqrt
```

The interpreter pushes 12.3 on the stack and then call the sqrt word. 12.3 is consumed and the result (sqrt(12.3)) is now on top of the stack.

```
1.2 ln sqrt exp
```

Here, each word uses the result of the previous one as its parameter. When reading this sequence, nothing tells you what parameter(s) each word uses or what are the results of each word. The stack is the support for passing parameters between each word.

The data stack is one of the most important concepts in Oforth. If you don't use it correctly, your code will be longer and more complex than necessary. Mastering the usage of the data stack is an important step.

2.3 RPN notation

Oforth uses **RPN notation**. In this notation, there is no need for parenthesis : a word's parameters are pushed before calling the word. For instance, in order to calculate 1 + 2, you write:

```
1 2 +
```

+ consumes two parameters on stack, calculates the sum, and pushes the result on the stack :

```
+             \ n1 n2 -- n3
```

To see the objects currently on the stack, you can use **.s**

```
1 2 + .s
[1] (Integer) 3 ok
```

You can also consume and print the top of the stack using **.** :

```
1 2 3 + . .s
5 [1] (Integer) 1
```

You can clear the stack using **clr**

Finally, you can toggle interpreter to automatically print the stack after each command you enter using **.show** command . If so, a **.s** will run after each command executed. .show again will go back to the previous behavior (not show the stack).

```
.show         \ --        Toggle intepreter to show the stack after each command
clr           \ --        Clear the stack.
bye           \ --        Leaves the interpreter.
```

2.4 Word's stack effects

A word's stack effect describes how it interacts with the stack : it is a comment that documents this interaction :

```
\ stack before -- stack after
```

For instance :

```
foo           \ n1 n2 n3 -- n4 f
```

means that the word foo, when performed, consumes 3 objects (here 3 integers) on the stack and returns two objects (here an integer and a float). This also means that the action of foo does not affect items under n1.

With this notation, n3 is the top of the stack before foo is performed and f is the top of the stack after foo is performed : the top of the stack is always the rightmost element.

It is a good practice (and required for readability) to describe the stack effect of each word.

For instance, the stack effect of - arithmetic operation on integers is:

```
-             \  n1 n2 -- n3   with n3 = n1-n2
```

Conventions used to describe stack effects are:

- x, y, z An object, whatever its type is.
- n An signed integer
- u An unsigned integer
- f A float
- b A boolean **(true** or **false)**
- c A character (ie an integer representing the unicode value of a character).
- s A string
- [x] A collection of objects.
- r A runnable (ie something you can perform, like functions, methods, blocks, ...).
- cl A class
- ex An exception
- <name> A name read directly from the input buffer.

2.5 Data stack and objects

Oforth is a dynamically typed language : each data is typed and each data stack slot holds one typed object : an integer, a float, a string, a date, a collection of 1000 objects, ... uses one position on the stack.

Objects themselves are created on the heap (or in the dictionary) and the data stack holds references to those objects. Words that manipulate the stack items, manipulate references to objects (an exception to this rule is small integers, see Integer chapter).

When you type **.s**, the object's class (its type) is what appears between (), before the object value :

```
1 2.3 Integer "abcd" .s
[1] (String) abcd
[2] (Class) #Integer
```

```
[3] (Float) 2.3
[4] (Integer) 1
```

2.6 Manipulating the stack

Once objects are on the stack, various words allow manipulate them. Those manipulations are used to order items before calling functions and methods.

Remember that these words manipulate only items references. If you duplicate (dup) an object, you duplicate only its reference value, you don't create a new object.

The most important words that manipulate the stack are :

```
dup           \ x -- x x          Duplicates the top of the stack
drop          \ x --              Removes the top of the stack
swap          \ x y -- y x        Swap the two items on top of stack
over          \ x y -- x y x      Copy the second item on top of stack
rot           \ x y z -- y z x    Rotate 3 items on the stack
```

Other less used words are :

```
nip           \ x y -- y          Remove the second item
tuck          \ x y -- y x y      Copy the tos under the second (same as swap over)
-rot          \ x y z -- z x y    Rotate 3 items
2dup          \ x y -- x y x y    Duplicate 2 items
2drop         \ x y --            Remove 2 items
pick          \ x*i n -- x*i xn   Copy the nth item on top ( 1-based )
.depth        \ x*i -- x*i n      Return stack size
```

Examples :

```
1 2 3 swap           \ 1 3 2
1.1 2.3 over         \ 1.1 2.3 1.1
"aa" "bb" tuck       \ "bb" "aa" "bb"
12 14 2dup           \ 12 14 12 14
clr 1.3 2.3 .depth   \ 1.3 2.3 2
```

3 Arithmetic

Oforth implements arithmetic operations as methods : the same method name is used for all kind of numbers. For instance, the "+" method is defined for integers, floats, strings, arrays, ...

Methods defined for Integer and Float will handle automatic conversion to float if necessary.

```
+             \ x y -- x+y        Implemented for integer, floats, strings, arrays
-             \ x y -- x-y        Implemented for integer, floats
*             \ x y -- x*y        Implemented for integer, floats
/             \ x y -- x/y        Implemented for integer, floats
neg           \ x -- -x           Implemented for integer, floats
abs           \ x -- y            Implemented for integer, floats
sq            \ x -- x*x          Implemented for integer, floats
pow           \ x n -- x^n        Implemented for integer, floats
^             \ x n -- x^n        Same as pow.
powf          \ x f -- x^f        Implemented for integer, floats
ln            \ x -- f            Implemented for integer, floats
log           \ x -- f            Implemented for integer, floats
exp           \ x -- f            Implemented for integer, floats
sqrt          \ x -- f            Implemented for floats

mod           \ n m -- r          Return the remainder of n by m
/mod          \ n m -- r q        Return the remainder and quotient of n by m
even?         \ n -- b            Return true if n is an even integer.
odd?          \ n -- b            Return true if n is an odd integer.
```

Some words are shortcuts for usual operations:

```
1+            \ x -- x+1          Add 1 to x
1-            \ x -- x-1          Substract 1 to x
```

All arithmetic operators use the RPN notation so no parentheses are necessary :

```
1 2 + 3 *           \ Calculates   (1+2) * 3
1.2 ln 4 + exp      \ Calculates   exp((ln(1.2)+4))
```

4 Functions and instructions

Functions are named piece of code. Calling a function is done by typing its name. They are objects representing the classical Forth words.

4.1 Declaring a function

A function is declared using a colon (the word :). The word ; closes the function definition. Everything between : and ; is the function body (the current definition) : the instructions that will be performed when the function is called.

```
: hello       \ --
    "Hello, world!" . ;
```

This creates a word named **hello** into the dictionary. The stack effect shows that this function takes no parameter from the stack and returns no value, so whatever was on the stack before calling this function is not modified. **hello** instructions push a string on the stack, then print it.

Calling **hello** word is simple : you just type its name. It can be called directly from the interpreter (to test it, for instance), or called from another function :

```
hello
Hello, world! ok

: test        \ --
    hello hello hello ;

test
Hello, world! Hello, world! Hello, world! ok
```

A function is not only a named piece of code; it is also an object of type Function stored into the dictionary. This object can be manipulated as any other object : it can be pushed on the stack, used as parameter for other words, ... To push this object on the stack, word # is used before the function name (this is comparable to ' (tick) word in Forth). # reads the next string and retrieves the word which name is this string (no space is needed just after #).

```
#             \ "name" --

#hello .s
[1] (Function) #hello
```

After pushing a function on the stack, it can be used as any other object. In particular, #execute is a method that executes the top of the stack :

```
#hello execute
Hello, world! ok
```

```
#test #execute #execute execute
Hello, world! Hello, world! Hello, world! ok
```

Remark for Forthers

In Oforth, there is no xt (execution token) as in Forth : a function is an object that allows to retrieve all information about it. This object is close to a header token. The ‘ Forth word that allows to retrieve an execution token from a word is replaced by # word, which returns an object on the stack (' word is dedicated to characters).

Into the rest of this manual, a function (or method) will often be mentioned using #.

4.2 Flow control

A function body can use conditional structures to control the instructions flow. These structures consume a boolean on the stack and execute instructions according to its value.

Booleans don't have a dedicated type, they are implemented as integers. They are the constants **true** (value 1) or **false** (value 0). In any test, everything (even null) different from **false** (0) is **true**.

Various structures are available to condition instructions. They all work the same way : they consume a boolean on the stack and condition instructions according to its value.

```
: myabs       \ n1 -- n2
    dup 0 <= ifTrue: [ neg ] ;
```

#<= consumes two objects and returns a boolean on the stack. This boolean is consumed by ifTrue: . If this boolean is not false, all instructions between [and] are performed. Otherwise, the program jumps directly after].

Other words to test conditions are:

```
ifTrue: [ instr ]    \ b --       Instructions are performed only if b is not false
ifFalse: [ instr ]   \ b --       Instructions are performed only if b is false
ifZero: [ instr ]    \ x --       Instructions are performed only if x is 0
ifNull: [ instr ]    \ x --       Instructions are performed only if x is null
ifNotNull: [ instr ] \ x --       Instructions are performed only if x is not null
if=: [ instr ]       \ x y --     Instructions are performed only if x = y
```

All these blocks can be followed by an **else** block. If so, this block is performed only if the instructions are not performed.

```
: mysign            \ x -- n
    dup 0 < ifTrue: [ drop -1 ] else: [ 0 > ] ;
```

The classical **if/else/then** conditionals used in Forth are also declared :

```
: myabs             \ x -- y
    dup 0 <= if neg then ;
```

```
: mysign            \ x -- n
    dup 0 < if drop -1 else 0 > then ;
```

4.3 Comparisons

Two different objects can have the same value (two strings, two floats, ...). We must differentiate if we want to test objects values or objects references.

The word = checks if two objects are the same object (ie have the same reference on the stack) :

```
=                   \ x y -- b : Returns true if x and y are the same object

12 dup =            \ true
12 12 =             \ true
1.2 dup =           \ true
'a' 'a' =           \ true
1.2 1.2 =           \ false
"aaa" dup =         \ true
"aaa" "aaa" =       \ false
#dup #dup =         \ true
[ 1, 2 ] [ 1, 2 ] = \ false
```

In other words, #= tests equality of the values of the two cells on top of the stack, whatever the objects are and whatever their type. Of course, if two objects have the same reference, they have the same value.

On the other hand, the method == tests if two objects have the same value. By default, this method calls #= , but it can (and sometimes must) be redefined into classes to test objects values rather than object's references.

```
==                  \ x y -- b : Returns true if x and y have the same value

12 12 ==            \ true
"aaa" "aaa" =       \ false
"aaa" "aaa" ==      \ true
1.2 1.2 ==          \ true
[ 1, 2 ] [ 1, 2 ] = \ false
[ 1, 2 ] [ 1, 2 ] == \ true
```

For integers and floats, == will convert them if necessary :

```
12 12.0 ==          \ true
```

There is no word to test if two references are different but there is a word to test if two objects don't have the same value : #<>

```
<>                  \ x y -- b : Returns true if x y don't have the same value
```

4.4 General loops

Three structures allow to iterate the same instructions.

The first one is an infinite loop **begin/again** : all instructions between those words will be performed endlessly, unless an instruction orders to leave the loop (**return**, **break**, ...).

```
: test1       \ n --
    begin
        dup 100 = ifTrue: [ break ]
        1+ dup .
    again
    "Done" .
;
90 test1
91 92 93 94 95 96 97 98 99 100 Done ok
```

The second loop is the **while** loop. While a boolean value is true, instructions are performed :

```
while ( ... b ) [ instructions ]
```

Instructions between (and) must leave a boolean on the stack. This boolean is consumed. If it false, the program leaves the loop. Otherwise, instructions are performed and the program jumps back just after the while :

```
: mygcd       \ n1 n2 -- n3
    while ( dup ) [ tuck mod ] drop ;

120 32 mygcd .
8 ok
```

The last loop is the **begin/until**. Unlike while, instructions are always performed at least once :

```
begin instructions b until
```

Instructions are executed and must leave a boolean on the stack. This boolean is consumed and while it is false, instructions are executeed again, ie the loop ends when the boolean is true.

Some words allow modifying the flow into a loop : if **break** is encountered, the program leave immediately the current loop and if **continue** is encountered, the program restart immediately the current loop :

```
: test2       \ --
    10 while ( dup ) [
        1-
        dup 6 if=: [ continue ]
        dup 3 if=: [ break ]
        dup .
        ]
    drop
;
test2
9 8 7 5 4 ok
```

4.5 Return stack and locals

When a function calls another function, return addresses must be saved to be able to continue the

function body when the second function returns. This is done on a second stack, the **return stack**.

Each call to a function creates a frame on the return stack. And, when the function returns, its frame is removed. By default, this frame holds only the return address of the function, but it also allows to save information local to a function call. For instance, to avoid many data stack manipulation, it is possible to store a value on the return stack. As the entire frame is removed from the return stack when function exits, this value will be lost when the function returns.

Storing local information on the return stack is done by declaring a **local** into a function. Two kind of locals can be declared : parameters and local variables.

If a function declares a parameter, when this function is called, an object is removed from the data stack and stored on the return stack as the parameter value. Using the parameter name into the body will push this value on the data stack. To store another value (the top of the stack) into this parameter, #-> word is used.

Parameters are declared using ( ) after the function name. All names until) are declared parameters. #-- is the beginning of a commentary and used to describe stack effect while declaring parameters. Everything between -- and) is ignored. So it is possible to mix the parameters declaration and the stack effect description of a function, avoiding to create a separated commentary to describe the stack effect.

CAUTION (for Forthers)

In Oforth, parentheses () are not commentaries, they are used to actually declare parameters.

{ } are used for another purpose, to declare Json objects (if the json package is imported)

For instance :

```
: sumDigits( n -- m )
    0 while( n ) [ n 10 /mod ->n + ] ;

123 sumDigits .s
[1] (Integer) 6
```

This function declares one parameter (named n). Text between -- and) is a commentary that describe the stack effect of this word. When this function runs, it removes an object from the stack (here 123) and stores it on the return stack into the frame created when the function is called. Each time n is used into the function's body, the current value stored on the return stack is pushed on the stack. And #-> is used to change the value of n on the return stack (by removing an object from the stack and storing it on the return stack).

If more than one parameter is declared for a function, the last one will have the value of the top of the stack, then the previous one : parameters are declared in the same order than stack effects. For instance, #under+ can be declared to consume and add the top of the stack to the third item of the stack :

```
: under+ ( x a )      \ y x a -- y+a x
   a + x ;

1 2 3 under+ .s
[1] (Integer) 2
```

```
[2] (Integer) 4
```

A function can also declare local variables. While parameters values are initialized with objects removed from the stack, local variables values are initialized to null value on the return stack. This is the only difference : using a local variable name will push its value on the stack and #-> will set its value with an object on the stack. Local variables are declared after parameters and before the function body using #| word.

```
: mybench            \ r -- ...
| tick |
   System.tick ->tick
   execute
   System.tick tick - . ;
```

Usage of local variables is subject to discussions : if you use too much locals, you will miss the goal of factoring your words and you will miss the objective to master the data stack usage. But, sometimes, locals can avoid too many stack juggling (many swap, dup, ...). What you have to know about local is that they are fully optimized and there is no reason to not use them just for performance purposes. Just be careful to about factoring your code : Oforth is not C and function body are almost never more than 3 lines long. Locals could help you to maintain longer functions, but you can miss good factorizations. Using too much locals will block your learning of good use of the data stack and factorization.

4.6 Comments and data stack diagrams

Unlike Forth where parenthesis are comments, they are used to declare parameters. Comments are declared using :

- \ the rest of the line is a comment
- -- everything until) is encountered is a comment

For instance :

```
: test1              \ a b -- n
#test1 has no declared parameters. Everything after \ is a comment

: test2 ( a b -- n )
#test2 has two parameters. They will be automatically poped from the stack when the
function starts.

: test3 ( a b -- n )    a sq b sq + ;
Same as test2. Everything between -- and ) are comments. After ), the definition
starts.

: test4 ( -- n )           \ a b -- n
Same as #test1. Everything between -- and ) is a comment and everything after \ is a
comment.
```

4.7 Integer loops

Integer loops use an index and will run a loop for each value of this index between a range. All integer loops need a local variable to be declared into the function.

```
: fact        \ n -- n!
   | i | 1 swap loop: i [ i * ] ;

50 fact .
30414093201713378043612608166064768844377641568960512000000000000 ok
```

#loop: consumes an integer on the stack and runs instructions between [ ] for each value between 1 and this integer. Into the block, the local value is this current value.

#for: is similar to **#loop:** but consumes 2 integers on the stack : instructions will run for each value between those 2 integers (included).

```
: test        \ --
   | i | 10 20 for: i [ i . ] ;

test
10 11 12 13 14 15 16 17 18 19 20 ok
```

4.8 Recursion

Into a function's body, it is possible to directly call this function to implement a recursion :

```
: fact        \ u -- u!
   dup ifZero: [ drop 1 ] else: [ dup 1- fact * ] ;
```

Another example with Fibonacci sequence (and declaring one parameter) :

```
: fib ( n -- m )
   n 1 <= ifTrue: [ 1 ] else: [ n 1- fib  n 2 - fib  + ] ;

10 fib .
89 ok
```

4.9 Returning from a function

It is possible to return immediately from a function using **#return** word. The return value(s) is what is on the stack when #return is performed.

```
: test        \ n1 -- n2
    dup 3 if=: [ return ]
    4 + ;
```

The return stack is accessible only using locals, so there is no restriction to return from a function. Removing a function's frame from the return stack when the function returns is handled

automatically.

4.10 A (little) transgression to RPN notation

The interpreter always uses RPN notation. This notation is in the heart of the system (see Compilation chapter).

But, sometimes, when a function has many parameters or if parameter(s) are calculated, it can be interesting, for readability purposes, to have a notation to separate those parameters. In Oforth, there is a "sugar" notation for this : () allows to push parameters after the function call. Of course, this sugar notation is never mandatory.

```
: fib ( n -- m )
    n 1 <= ifTrue: [ 1 ] else: [ fib( n 1- ) fib( n 2 - ) + ] ;
```

Here, #fib has been rewritten to use this notation for the two inner calls : #fib parameters (and how they are calculated) are now clearly identified.

This notation has no runtime cost w :hen you write this version of #fib, the interpreter will translate it into the previous version.

This notation has nothing to do with how #fib function was declared (with or without declaring a parameter). As it is only sugar, it can be used for calling a function in either case.

This notation is also possible if a function takes more than one parameter :

```
: diag        \ a b -- x
    sq swap sq + sqrt ;

: test1       \ -- f
    diag( 2, diag ( 3, 4 ) ) ;

: test2       \ -- f
    2  3 4 diag  diag ;
```

#test1 and #test2 not only return the same value but the code generated is also the same : #test1 is translated to #test2 during the compilation.

Last point : it is possible to use this notation even at the interpreter level, ie you can type :

```
10 fib .
```

or

```
fib( 10 ) .
```

4.11 Factoring

Oforth programming is building your program by writing functions that have a contract with the data stack (parameters removed and returns values).

Functions should be very small (no more than 3 ou 4 lines) because otherwise, it becomes harder to follow what happens on the stack. Don't be afraid to write small words, even words that call no more than 3 or 4 words.

Oforth programming is all about finding the good, small, reusable, named factors, to define them as functions and call them into other functions.

You can test those factors very easily by calling them directly at the interpreter level. And you should test your factors as soon as you write them.

As immutability is enforced (see below) , most of the time your function will have a very important characteristic : every time you send them the same parameters, they will return the same value(s), without side effects. This means that, after you test your functions once, they will work forever.

The choice of function's names is very important. If you can't find a good name for a piece of code, it is probably not a good factor. Finding good names is much more important than in other languages : as the space is the word separator, good names make your code much more readable.

Factoring is a key concept to write good code. And the best possible factor is a small, simple, namable, with no side effect, function.

Last point : factoring is not something frozen that ends when you have finished to write your code. If you write some code, re-read your previously written functions. The code you just wrote today can make you think of new factors for the code you wrote a week ago.

5 Object Oriented Programming

5.1 Introduction

Until now, we have talked a lot about objects but almost never used OOP mechanisms, at least not explicitly. This can seem weird, but it is one of Oforth objectives : you can write code without "heavy" OO concepts and constraints that often comes with them. You can even write code without knowing a lot about OO programming : you just write your functions and use them. Each basic type comes with its interface and you use them to create your program.

At this point, the main difference with classical Forth is that polymorphism allows having the same name for different words. For instance, #+ is a method that can be used on integers, floats, strings, ... instead of having a different name for each operations.

OOP is about creating objects that will encapsulate their data. Those objects expose methods that will be used as its interface to use them. When you use a string or a float or big integer, you do not care about the data stored inside those objects : you just use them. You adds two strings, two lists, two big integers and you have a result. This is called **encapsulation**. If the internal storage of a big integer evolves, your program should not be impacted because your program uses a public interface that is quite stable : you push two big integer on the stack, use #+ to add them and retrieve the result.

Encapsulation is implemented by defining classes that will hold attributes and methods exposed. You can think about classes as C structures or records with fields declared. Those classes allow to create objects (like a factory) and each object will have its own values for the attributes declared for the class. In Oforth, classes are implemented as a new kind of words : **Class**

The second concept is **polymorphism** : a message with the same name can be implemented for different classes : #+ is implemented for floats, strings, arrays, ... Each implementation is different, but the method have the same name. The system decides which implementation to use according to the object that use the method (the top of the stack). In order to handle polymorphism, Oforth introduce another kind of word : **Message**. A message is like a function but can have an implementation for each class.

And the last concept is the class **hierarchy** : a class has a unique parent and will inherit all attributes and methods declared for its parent. For instance, an Array is a Collection so all methods defined at the collection level can be used by arrays objects. In Oforth, the class hierarchy is very limited : this is possible because it is not required for two classes to have a common parent to implement the same message : all classes can implement all messages, whatever the hierarchy is and whatever their parents are.

5.2 Classes

Like functions, a class is a kind of word in the dictionary. Creating a new class is asking to the class **Class** to create a new object :

```
new:          \ aParent Class <name> --
```

```
Object Class new: Person
```

This creates the class **Person** in the dictionary, with **Object** as its parent. Once a new class is created, it can be used as any other word and will be detected by the intepreter (corresponding action is to push the class on the stack).

```
Person .s
[1] (Class) #Person
```

Once a class is created, objects of this class can be created by sending the #new message to the class :

```
new           \ aClass -- aObject
Person new .s
[1] (Person) aPerson
```

#new creates objects on the heap, that will be handled by the garbage collector. It is possible to handle manually object allocation and de-allocation of objects using alloc/free (see Memory management for more information).

5.3 Messages

Messages are another kind of word. They are executable words (like Functions) that allow polymorphism : each class can give its own code to execute when a message is sent to its instances. When a message is executed, the code that will be performed is the one corresponding to the object present on top of the stack.

Messages themselves are not related to a particular class. It is possible (but seldom used) to create a new message without an implementation :

```
message: m
```

This creates a new message into the dictionary which name is m. Like other word, the message can be retrieved using # :

```
#m .s
[1] (Message) #m
```

5.4 Methods

Even if stored in the dictionary, methods are not words : they are objects representing an implementation of a message for a particular class.

To create a method, the word **#method:** is used :

```
aClass method: m [ ( param1 param2 ... paramn ) ]
[ | var1 var2 ... varm | ]
 instructions
```

```
;
```

This will create an implementation of message m for class aClass. If the message does not exist yet, it is first created before creating the method (that is why creating a message without implementation is seldom used).

Calling a message is almost the same as calling a function. The only difference is that the method to run is resolved at runtime according to the the top of the stack. If no method is found a "does not understand" exception is raised.

If a method is found, its code is launched. Before executing the method's body, **the top of the stack is removed and stored on the return stack** as an implicit parameter of the method : this parameter is called **the receiver** of the method or **self**. Into the method's body, this receiver can be pushed back on the stack using **self**.

For instance, to implement a #dup as a message, self must be pushed twice on the stack.

```
Object method: mydup \ x -- x x
    self dup ;
```

To implement a #drop as a method, there is nothing to do :

```
Object method: mydrop       \ x --
 ;
```

Remembering that the receiver is removed from the stack when you call a message is almost all what you have to know to write methods compared to functions. Just think about methods as function with an implicit parameter : self.

Those three words are performing the same computation (calculate inverse hyperbolic cosinus of a float):

```
: acosh1      \ f1 -- f2   : Calculate acosh(f1) = ln(f1 + sqrt(f1^2 - 1))
   dup sq 1.0 - sqrt  + ln ;

: acosh2 ( f -- f1 )
   f sq 1.0 - sqrt  f +  ln ;

Float method: acosh3 -- f
   self sq 1.0 - sqrt self + ln ;

1.5 acosh1 .
0.962423650119207 ok

1.5 acosh2 .
0.962423650119207 ok

1.5 acosh3 .
0.962423650119207 ok
```

Like functions, a method can declare parameters and local variables.

```
Object method: test( a b -- n )
   self b + a - ;

10 20 30 test .s
```

```
[1] (Integer) 40

30 test ( 10, 20 ) .s
[1] (Integer) 40
```

Here, the receiver is 30 (the receiver is always the top of the stack) so it is removed from the stack and stored as "self" parameter. Next, two parameters are declared so a value is 10 and b value is 20.

The “sugar” notation works also for methods but the receiver must remain on top of stack and is not part of the parameters. For instance :

```
put                 \ i x aArray  : Put x at index i of aArray
10 1.2 aArray put
aArray put( 10, 1.2 )
```

#execute is declared as a message and can be used to execute ... messages.

```
Integer method: test
    self 1+ ;

#test .s
[1] (Message) #test

10 #test execute .s
1] (Integer) 11
```

If you look at the examples in this chapter, you can see that classes are not closed : there is not a beginning and an end for a class definition. You create a class and you can add methods for this class whenever you want. You can even add new methods for built-in classes (Integer, Float, String, ...), like the #test method. If you create methods for the last class created into the dictionary, you can use **m:** instead of **method:** .

```
Object Class new: A
m: test1  self 1+ ;
A method: test2  self 2 + ;
m: test3  self 3 + ;
```

5.5 Attributes and data initialization.

It is possible to define attributes for a class when creating a new class : they will be the internal data for each object of this class. The general form of the #new: message for classes is :

```
new:          \ aParent Class <name> [ ( [ mutable ] <attname>, ... ) ] --
```

This allows to list attributes for a class. Default for attributes is to be immutable, but each attribute can be declared as mutable using the mutable keyword before the attribute name.

Attributes must be declared while creating a class : after that, there is no way to add other attributes to the class.

Into a method body, attributes values are pushed on the stack using word **@** and attributes values are set using word **:=**

```
@           \ <name> -- x      : push attribute value with name <name> on the stack
:=          \ x <name> --      : set x as value of attribute with name <name>
```

Let's change the Person declaration to add attributes :

```
Object Class new: Person( name , age )
```

Each object of type Person will have two internal data, one name and one age. But objects created from this class won't be very useful because attributes are initialized with null value and, as they are not declared as mutable, there is no way to update those values. In order to set attributes values for immutable attributes, we must do this during an object initialization. **#new** always calls an **#initialize** method with the new allocated object as its receiver.

```
Person method: initialize  \ string n aPerson --
 := age  := name ;
```

Now we can create a new Person by giving attributes value :

```
"John" 24 Person new
Person new ( "Marie", 22 )
```

We can also provide an implementation for #<< method, that is used by #.s to print an object :

```
Person method: <<          \ aStream aPerson -- aStream
 @name << " : " << @age << ;

"John" 24 Person new .s
[1] (Person) John : 24
```

Default behavior for attributes is immutability ie that, after initialization, their value can't be updated anymore. Immutability rules are checked at runtime. If you try :

```
Person method: setAge             \ n aPersonn --
 := age ;

"John" 24 Person new
25 over setAge
[console:1] #Exception : Immutable rule violation
```

An exception is raised because we are trying to update an immutable attribute after the object's initialization. If we want to update an attribute after initialization, we must define it as mutable when we create the class, using **mutable** keyword :

```
Object Class new: Person ( name, mutable age )
```

Now the attribute can be updated after initialization. But the drawback is that objects created from Person are now mutable objects. Mutable objects have restrictions that are also checked at runtime : for instance, they cannot be the value of a constant, they cannot be the value of an immutable attribute, they cant be shared between tasks, ... (see concurrent programming).

It is possible to freeze a mutable object using the #freeze word. If so, its attributes can't be updated anymore and the object has no more restrictions associated to mutable objects.

5.6 Class methods

You can also create class methods implementations for a message. A class method is a method for which the receiver is the class itself (and not an instance of the class). This is done using **classMethod:** instead of method:

```
Float classMethod: zero     \ Float -- 0.0
    0.0 ;

Float zero .s
[1] (Float) 0
```

5.7 Polymorphism

Polymorphism is the ability for messages to have different implementations (one by class).

Each class, whatever its position into the class hierarchy, can declare a method for a particular message :

```
Object Class new: A
m:  "I am a A object of class :" . self class . ;

Object Class new: B
m: m "I am a B object of class :" . self class . ;

B Class new: C

A new m
I am a A object of class : #A ok

B new m
I am a B object of class : #B ok

C new m
I am a B object of class : #C ok
```

It is also possible to override an implementation into a subclass, but only if the implementation is declared as **virtual** into the parent class :

```
Object Class new: A
A method: m "I respond to an A object " . ;

A Class new: B
B method: m "I respond to a B object " . ;
[console:1] #Exception : Can't redefine non virtual method <#m>
```

This raises an compilation error as m is not defined as virtual.

```
Object Class new: A
A virtual: m "I respond to an A object " . ;

A Class new: B
B method: m "I respond to a B object " . ;

A new m
I respond to an A object  ok

B new m
I respond to a B object  ok
```

Into an overloaded method, it is possible to call the implementation at the upper level using #**super**. #super is like #self but the implementation called is the one of the superclass :

```
Object Class new: A
A virtual: m "I respond to an A object" . ;

A Class new: B
B method: m  super m  "but it is a B" . ;

A new m
I respond to an A object ok

B new m
I respond to an A object but it is a B ok
```

Oforth implements single inheritance : each class has one and only one parent. So this link is very strict and implement a "IS-A" relation between two classes (an Array IS-A Collection).

Each attribute and each method declared at a parent level is available at the child level and, consequently, should be fully applicable at this level.

5.8 Properties

Oforth OO meta-model includes properties; they are words like classes ie they can implement methods and can have attributes, but :

- There is no hierarchy between properties
- You can't create objects from properties.

The comparable property could be implemented like this (see Comparable.of for the full version) :

```
Property new: Comparable

Comparable method: >       \ x y -- b
   self <= not ;
```

```
Comparable method: <( c )  \ x y -- b
   self c == not c self <= and ;

Comparable method: >=      \ x y -- b
   self < not ;

Comparable method: min     \ x y -- min(x,y)
   self over <= ifTrue: [ drop self ] ;

Comparable method: max     \ x y -- max(x,y)
   self over <= ifFalse: [ drop self ] ;

Comparable method: between( x y -- b )
   self y <=  x self <=  and ;
```

Now that this property is created, it is possible for classes to be of this property :

```
Integer is: Comparable
Float is: Comparable
Date is: Comparable
```

Once a class is Comparable, all objects of this class will automatically answer to all methods declared into the property.

If you identify a common pattern between various classes and if the class hierarchy is not relevant, probably a property is what you are looking for.

5.9 Dictionary and OO meta-model

The dictionary is the area where all words are stored. When a name is read from the input stream, the interpreter searches for the corresponding word into the dictionary.

We have already encountered some kind of word (Class, Property, Function, Messages, ...). Here, we list all the word types that define the OO meta-model. Words created into the dictionary inherit from Word class. After Oforth is launched, words created are :

```
Object
----- Word
--------- Class
--------- Property
--------- Function
------------- Dual
--------- Method
--------- Constant
--------- TVar
--------- Package
--------- DynLib
--------- DynProc
```

All words have a name and this name is unique into the system (well, into a package). A word name is a symbol. # allows to retrieve a word by its name into the dictionary :

```
#             \ "name" -- aWord | null : Read a name and retrieve corresponding word.
```

Methods implemented at the Word level are :

```
find          \ str Word -- aWord | null : Find a word in the dictionary
name          \ aWord -- aSymbol : Returns word's name.
alias:        \ aWord "name" --  : Creates an alias of aWord with "name"
forget:       \ "aWord" --
```

#forget makes a word no more findable in the dictionary (but definitions that use this word will continue to work).

5.10 Constants

A **Constant** is a word that returns a constant value. When a constant name is used (at the interpreter level or into a body), its value is pushed on the stack. To create a constant, the provided value must be immutable.

```
const:        \ x "name" -- : Creates a constant with "name" and value x

2.0 ln const: Ln2
Ln2 .
0.693147180559945 ok
```

It is not possible to create a constant whose value is a mutable object :

```
Array new const: A
[console:1] #Exception : Immutable rule violation
```

5.11 Task variables

A **TVar** is a global variable. When a TVar is created (using tvar:) , its value is null. Using a TVar name pushes the value on the stack. #to allows to modify a TVar value :

```
tvar:         \ "name" -- : Creates a new tvar initialized with null value.

tvar: myvar
myvar .
null ok

2.3 to myvar
myvar .
2.3 ok
```

You should not rely on Tvar too much. One important characteristic your functions or methods

should have it to answer the same return when called with the same parameters. This is not sure if you use a TVar. So each time it is possible, it is better to send the values needed by parameters on the stack.

Sometimes, you will need a global state and a TVar could be used.

tvar are global words, but each task has its own value (hence its name: a task variable). You can't use a tvar to share values between tasks (this is the same mechanism as USER variables in Forth).

There is no word to create a variable global to the whole system as this would break isolation between tasks.

6 Basic types

This section describes the basic types defined at startup. See "Word reference chapter" for all words defined for a particular class.

6.1 Object

Object is the top of the class hierarchy (superclass of Object is null).

Words implemented at this level are available for all objects, whatever their type. Many words are higher order functions and will be described into the dedicated chapter. Other words are :

```
yourself            \ x -- x : Returns the receiver
class               \ x -- aClass : Returns an object's class
null?               \ x -- b : Returns true if x is null.
is?                 \ cl x -- b : Returns true if x class is cl
==                  \ x y -- b : Checks if two objects have the same value (virtual)
<>                  \ x y -- b : Checks if two objects don't have the same value.

<<                  \ aStream x -- aStream : Send x to aStream
<<n                 \ aStream n x -- aStream : Send x to aStream n times
```

6.2 Null

Null is the class of the **null** object.

null object means "nothing". It is used :

- To initialize a newly allocated object attributes.
- To initialize local variables values.
- As the return value of some function when we want to express that the function returns nothing.

6.3 Integer

Integers have arbitrary precision.

On 32bits systems, Integers between [-1073741823, 1073741823] = [$-2^{30}+1$, $2^{30}-1$] don't allocate memory and are stored directly on the stack. Other integers are allocated on the heap and their reference is stored on the stack.

Integers overflow is checked and the system automatically switches to big integers (allocated on the heap) if necessary.

```
: fact        \ n -- n!
   | i | 1 swap loop: i [ i * ] ;

1000 fact .
```

Integers can be written in hexadecimal or binary using **#0x** and **#0b** words (there is no such words as BASE in Oforth) :

```
0xFFFF0000
0b01001111
```

In addition to number's operations and Integer operations listed in the Arithmetic chapter, Integer class declares the following words :

```
bitAnd        \ n1 n2 -- n3 : Do a bit and between n and m
bitOr         \ n1 n2 -- n3 : Do a bit or between n and m
bitXor        \ n1 n2 -- n3 : Do a bit xor between n and m
bitLeft       \ nb n -- m : Do a shift of n (nb bits to the left )
bitRight      \ nb n -- m : Do a shift of n (nb bits to the right )

each          \ r n -- : Perform r on integers between 1 and n.
```

For instance :

```
#. 10 each
1 2 3 4 5 6 7 8 9 10 ok
```

Other operations on integers are declared into the optional math package. To use them, you will have to import this package (see Package chapter) :

```
import: math
```

6.4 Boolean

There is no dedicated type for booleans. Booleans are implemented as integers.

Booleans **true** and **false** are constants respectively equals to **1** and **0** and any object different from **0** is considered as a true value.

Integer class implements those operations for booleans :

```
not                \ b1 -- b2
and                \ b1 b2 -- b
or                 \ b1 b2 -- b
xor                \ b1 b2 -- b
```

Those operations are not performing operations on bits. They return booleans. Operations on bits are performed using #bitAnd, #bitOr, #bitXor, ...

Boolean objects are small integers and are not allocated on the heap.

6.5 Character

There is no dedicated type for characters. Characters are implemented as integers, which represent their unicode code.

A char is entered using word #' :

```
'a' .
97 ok
```

Integer class implements those operations for characters :

```
space?              \ c -- b : Returns true if c is BL or HTAB
upper?              \ c -- b : Returns true if c is an upper case char
lower?              \ c -- b : Returns true if c is a lower case char
>upper              \ c -- v : Returns upper of c
>lower              \ c -- v : Returns lower of c
digit?              \ c -- b : Returns true if c is a digit
letter?             \ c -- b : Returns true if c is a letter

>digit              \ c -- n | null : Returns value of character c
>char               \ n -- c | null : Returns char value of n in base 10

16 'F' >digitOfBase .
15 ok

'8' >digit .
8 ok

6 >char .
54 ok
```

The word ' detects some special characters :

```
'\n'                \ New line
'\r'                \ Carriage return
'\t'                \ Horizontal tab
'\b'                \ Backspace
'\"'                \ Double quotation mark
'\''                \ Single quotation mark
'\\'                \ Backslash
'\uxxxx'            \ Character which unicode code is hex xxxx
'\Uxxxx'            \ Same a \u
```

Characters are small integers and are not allocated on the heap.

6.6 Float

Floats are 64 bits (even on 32bits versions). They have the following form :

```
n[.m] or n[.m]e[p]
```

The "Word reference" chapter lists all words implemented for Float.

On Oforth 32bits, floats are allocated on the heap.

6.7 Block and anonymous functions.

Blocks are anonymous functions.

They are created (at the interpreter level or into a function) using #[and] words.

```
#[ sq 1+ ] .
```

When created, the block is pushed on the stack. Like functions, blocks are performed using the **#execute** method.

Blocks can be used to write small pieces of code that we don't want to name as a factor. This is often used for parameters to higher order functions (see the dedicated chapter).

A block can be nested into another block.

```
10 #[ #[ 10 fib ] bench ] times
```

Blocks are allocated into the dictionary. If a block is a closure (see below), it is allocated on the heap and handled by the garbage collector.

6.8 Closures

Closures are not fully implemented in Oforth : what you can do is use function's parameters or locals into a block. For instance :

```
: compose ( f g -- bl )
    #[ g execute f execute ] ;
```

This function returns a block that, when performed, returns the composition of #f with #g.

```
#1+ #[ 2 * ] compose
5 swap execute .
11 ok
```

The values are copied at the moment the block is created :

```
: f( n -- aBlock )
    #[ n + ] ;

10 12 f execute .s
[1] (Integer) 22
```

This means that multiple closures into the same function don't share the same values : each closure will have its own value(s) at the moment the closure is created :

```
: g ( x y -- bl1 bl2 )
   #[ x y + . ]
   x 2 * -> x
   y 3 * -> y
   #[ x y + . ]
;
2 3 g execute perfom
13 5 ok
```

Apart from this restriction, Oforth's closures work as closures usually work. For instance, you can update the closure's slots :

```
: foo ( n -- q )   #[ n + dup ->n ] ;

: testfoo
| x |
   1 foo ->x
   2 x execute .
   4.5 x execute .
;

testfoo
3 7.5 ok
```

Here, each time you execute the closure, n is updated with the old value and the number on the stack and the result is also returned.

A closure can't declare its own parameters (perhap's in a future version...). If you want to do this, you can declare a local variable into the function that created the closure and use it as parameter(s). For instance, the block returned by the following function compute f(x+1) - f(x) with f given as parameter :

```
: test( f -- aBlock )
   | x | #[ ->x  x 1+ f execute  x f execute  - ] ;

10 #sq test execute .
21 ok
```

6.9 Symbol

A symbol is an identity string : only one version of a symbol exists in the system. Two symbols that have the same value are the same symbol, ie the same object (which is not true for strings).

A symbol is created (or just pushed on the stack if it already exists) using word $

```
$apple $apple = .
1 ok
```

All words name are symbols.

Symbols are often used to defines enumerations. For instance :

```
[ $apple, $banana, $orange ] const: Fruits
Fruits .
[apple, banana, orange] ok
```

Symbols are allocated into the dictionary.

7 Collection classes

This chapter describes all collections available at startup. They inherit from the Collection class :

7.1 Collection

Collections created at start-up are :

```
Object
--- Collection
------ Interval           Range of items ( from x to y with step s )
------ Pair               Array with 2 items [ x, y ]
------ Array              General immutable Array of item.
------ Buffer             Collection of bytes
--------- String          UTF8 strings
```

Methods implemented at the Collection level are :

```
size                \ x -- u : Returns nb of items. Must be redefined.
empty?              \ x -- b : Returns true if the collection is empty
<<                  \ aStream x -- aStream : Send x into aStream.
```

7.2 Pair

A pair is a collection with 2 elements. They can be used as elements of dictionaries [key, value]

```
1 2 Pair new.s
[1] (Pair) [1, 2]
```

If a collection is a collection of pairs, it is possible to retrieve the pair with a key or a value :

```
keyAt         \ x y -- aPair : Returns pair with key x into y
[ [ $a, 1], [ $b, "abc"], [ $c, [1, 2, 3] ] ]  keyAt( $b )   \ Returns [b, abc]

valueAt       \ x y -- z : Returns value of pair with key x into y
[ [ $a, 1], [ $b, "abc"], [ $c, [1, 2, 3] ] ]  valueAt( $b ) \ Returns "abc"
```

7.3 Interval

Interval implements a range of values. An interval is created with :

- The initial value
- The final value
- The step between values.

Methods/ functions of Interval class are :

```
new           \ init end step Interval -- aInterval

size          \ aInterval -- size : Returns number of values.
at            \ n aInterval -- x : Returns value at position n
seqFrom       \ n m -- aInterval : New interval between n and m (step is 1)
seq           \ n -- aInterval   : New interval between 1 and n (step is 1)

10 seq .
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ok
```

Interval are also used to implement #step: loop :

```
n m step step: o [ instructions ]
```

Into instructions, o takes all values between n and m with step.

```
: test        \ --
 | x | 0 20 2 step: x [ x . ] ;

test
0 2 4 6 8 10 12 14 16 18 20 ok
```

7.4 Buffer

A buffer is a collection of bytes and is the parent of String class. A buffer is created with #new or #newSize

In order to differentiate bytes from UTF8 characters, there a different methods :

```
byteSize      \ aBuffer -- n         : Returns number of bytes
size          \ aBuffer -- n         : For some classes, returns number of chars
byteAt        \ i aBuffer -- n       : Return byte at position i
at            \ i aBuffer -- n      : Return character at position i (unicode)
bytePut       \ i byte aBuffer --    : Put byte at i position
put           \ i byte aBuffer --    : Same as bytePut
```

7.5 String

Strings are a collection of UTF8 characters. String is a subclass of Buffer and is Comparable. String characters are accessed using #at method (1-based). In order to retrieve a byte from a string, you can use #byteAt (declared into the Buffer class) :

```
at                  \ i s -- c   : Returns UTF8 character at position i (1 based)
```

```
byteAt              \ i s -- n   : Returns byte at position i (1 based).
```

Word **#"** allows to create new constant immutable string. As " is detected by the interpreter, no space is required after it.

```
"Hello world!" .s
```

The world #" detects special characters :

```
\n                  \ New line
\r                  \ Carriage return
\t                  \ Horizontal tab
\b                  \ Backspace
\"                  \ Double quotation mark
\'                  \ Single quotation mark
\\                  \ Backslash
\uxxxx              \ Character which uncode code is hex xxxx
\U                  \ Same a \u
```

Mutable string are created with the following words :

```
new           \ String -- aString       : Creates a new string
newSize       \ n String -- aString   : Creates a new string and allocates n byte
newWith       \ n c String -- aString
init          \ n r String -- aString

10 #[ 26 rand 'A' + 1- ] String init .
XQLOAGBFYG ok
```

Main words defined for strings are (see Word reference for all words) :

```
size                \ s -- n           Returns number of UTF8 chars
at                  \ i s -- c          Returns UTF8 character at index i (or null)
==                  \ s1 s2 -- b        Return true is s1 and s2 have the same value
<=                  \ s1 s2 -- b        Returns true if s1 <= s2
empty               \ s --              Empty a mutable string.
addChar             \ c s --            Adds c to the string
add                 \ c s --            Same as addChar
addAll              \ x s --            Add all elements of x into a string
put                 \ i c s --          Put c at position i
hashValue           \ s -- n            Returns hash value of s
evaluate            \ s --              Evaluate the string as Oforth code.
load                \ s --              Load file which name is s.
```

8 Higher order functions and collections

A higher order function (HOF) is a function (or method) that takes a runnable as parameter and/or returns a block.

We have already encountered a higher order functions while describing closures :

```
: compose ( f g -- aBlock )
    #[ g execute f execute ] ;
```

This function takes two runnables as parameters and returns a block.

HOF allows to apply a runnable on collections, to create new collections, ...

8.1 #forEachNext method and #forEach: loop

The #forEachNext method is a virtual method. It allows to traverse items contained into an object in a generic way. Its stack effect is a little complicated :

```
forEachNext          \ x o -- y item true | false
```

o is the object we want to traverse and x is an object that allows to retrieve the next object into o.

This function should retrieve the next item into object o, using x value :

- If there is no more objects, this function just returns false.
- If x is null, this is the first time that forEachNext if called, so the first item is to be retrieved.
- If another item is found, this function should return y, this item and true. y is the object that will be sent back to the next call to #forEachNext as x parameter to retrieve another item.

So #forEachNext is an iterator : each call to #forEachNext retrieve and returns the next item of an object.

At the Object level, #forEachNext is :

```
Object virtual: forEachNext      \ x obj -- y o true | false
   ifNull: [ 1 self true return ] false ;
```

The first time #forEachNext is called, it returns the object itself. The next time, it returns false. So traversing an object, by default, is just returning this object one time.

This method is used by the **forEach:** loop. This immediate function creates a loop to traverse all items included into an object :

```
x forEach: o [ instructions ]
```

o must be a declared as a local variable. #forEach: removes x from the stack and generates a loop that calls #forEachNext on x; instructions will be executed for each item into x. Into instructions, o value is the value of the current item.

```
: test        \ --
```

```
   | o | [ 1, 2, 3, 4, 5 ] forEach: o [ o sq . ] ;

test
1 4 9 16 25 ok
```

#forEach: is the base structure for all high order functions. And, as #forEach: uses #forEachNext, new collections just need to overload #forEachNext method to be able to answer to all defined HOF.

8.2 Arrays

Collections are containers for items and the most common collection is the array : it is an container for items, which can be accessed by an index (1-based).

An array can be explicitly created using #[, #, and #] words. Those arrays are created as immutable. No space is needed before or after them :

```
[ 1, 2, 3, 4, 5 ] .s
[1] (Array) [1, 2, 3, 4, 5]
```

Items into an array don't have to be of the same type (and they can be arrays too) :

```
[ 1, 2.3, "abcd", 'a', [ 1, 2, 3 ] ]
```

If used into a definition, arrays created with [...] are created at runtime :

```
: foo ( x y -- list )
   [ x 1+ , y , 4 ] ;

2 3 foo .
[3, 3, 4] ok

10 5 foo .
[11, 5, 4] ok
```

An array can also be created with #new or #newSize

```
Array new           \ Creates a new mutable array with default allocation size.
n Array newSize     \ Creates a new mutable array with n as first allocation size.
```

If there is not enough room to hold items, the array is automatically reallocated. So #newSize is a hint to allocate enough space to limit re-allocation.

Once an array is created, items are handled with :

```
add                 \ x arr --    : add x at the end of the array
at                  \ i arr -- x  : return item at index i (1-based), null if none.
put                 \ i x arr --  : put x at index i.
```

(see Word reference chapter for all array methods).

8.3 Higher Order Functions

Object class implements many HOF. These functions (or methods) take an object as receiver and a runnable as parameter. The most basic one is **#apply** : it takes a runnable and an object on the stack and, for each item into this object, it pushes it on the stack, then execute the runnable on it.

```
apply          \ r x -- ... : execute r on each item of x
```

Examples :

```
#. 10 apply
10 ok

#. [ 1, 2, 3, 4, 5 ] apply
1 2 3 4 5 ok
```

#apply is implemented using #forEach: . Its code is very simple (from Object.of file) :

```
Object method: apply ( r -- ... )
   | o | self forEach: o [ o r execute ] ;
```

As the receiver and the runnable are removed from the data stack (they are stored on the return stack as self and r), it is ok for the runnable to use objects on the stack at the moment #apply is called :

```
0 #+ [ 1, 2, 3, 4, 5 ] apply .
15 ok

0 [ 1, 2, 3, 4, 5 ] apply( #+ ) .
15 ok

0 #[ sqrt + ] [ 1, 2, 3, 4, 5 ] apply .
8.38233234744176 ok

0 [ 1, 2, 3, 4, 5 ] apply( #[ sqrt + ] )  .
8.38233234744176 ok
```

The #+ will accumulate results using the 0 on the stack.

This is a general rule : all HOF have been written to allow the runnables to access objects on the data stack when they are executeed.

#applyIf applies a runnable only on items that respond true to a condition.

```
applyIf        \ cond r x -- ...

0 #even? #+ [ 1, 2, 3, 4, 5 ] applyIf .
6 ok

0 [ 1, 2, 3, 4, 5 ] applyIf( #even? , #+ )
6 ok
```

#reduce is like #apply, but the first item of the collection is pushed on the stack as initial value for accumulator before looping across items :

```
reduce        \ r x --

#+ [ 1, 2, 3, 4, 5 ] reduce .
15 ok

#+ [ "aaa", "bbb", "ccc" ] reduce
aaabbbccc ok

[ "aaa", "bbb", "ccc" ] reduce( #+ )
aaabbbccc ok
```

#reduceWith is a generalized version of reduce. Before applying the runnable, another runnable is applied on each item :

```
reduceWith          \ p r x -- : reduce x using r, but after applying p on each item

#sq #+ [ 1, 2, 3, 4, 5 ] reduceWith     \ Returns 55 = 1*1 + 2*2 +3*3 + 4*4+ 5*5
[ 1, 2, 3, 4, 5 ] reduceWith( #sq, #+ )  \ Returns 55
```

#reduce is defined using #reduceWith

```
Object method: reduce( r -- x )
    #yourself r self reduceWith ;
```

#detect returns the first element that answer a particular value (e) to a runnable (r) :

```
detect        \ r e x --

#>upper 'C' [ 'A', 'B', 'c', 'd' ] detect      \ returns 'c'
```

#include? checks if an element is included into a collection (using #==) :

```
include?            \ e x -- b : Returns true if e is included into x
```

#conform? checks if all items respond true to a condition :

```
conform?            \ rcond x -- b

#even? [ 1, 2, 3, 4 ] conform           \ Returns false
```

#2apply works on 2 collections : for each index, it pushes the 2 items of the collections , then execute a runnable. The loop stops when one of the collections has no more items.

```
2apply        \ y r x -- ...

0 [ 1, 2, 3 ] #[ * + ] [ 4, 5, 6 ] 2apply      \ Returns 1*4 + 2*5 + 3*6
0 [ 1, 2, 3 ] [ 4, 5, 6 ] 2apply( #[ * + ] )   \ Same...
```

#iapply allows to loop with the index : for each item, it pushes the item, its index, then call a runnable

```
iapply        \ r x --

0 #[ even? if +  else drop then ] [ 2, 3, 4 ] iapply   \ Sum of items at even indexes
```

#maxFor returns the element of a collection with max value when r is applied :

```
maxFor        \ r x -- y

#second [ [ 1, 2 ], [ 3, 7 ], [ 5, 6 ] ] maxFor .
[ 3, 7 ]
```

#minFor works like #maxFor but returns the element with min value.

```
#second [ [ 1, 2 ], [ 3, 7 ], [ 5, 6 ] ] minFor .
[ 1, 2 ]
```

#sum sums all elements of a collection :

```
[ 1, 2, 3, 4, 5 ] sum .
15 ok

[ "aaa", "bbb", "ccc", "ddd" ] sum .
aaabbbcccddd ok
```

8.4 Mapping

Mapping is like applying, but a new collection is created and returned as the result. All mapping words return collections created with #new, so desallocation is handled by the garbage collection (see memory management chapter).

Only the following 3 words are loaded at startup :

#+ is declared for collections and returns a new collection with the concatenation of all items

```
+             \ x y -- coll

[ 1, 2, 3 ] [ 4, 5, 6 ] +              \ Returns [ 1, 2, 3, 4, 5, 6 ]
"abc" "def" +                          \ Returns "abcdef"
```

#zip and **#zipWith** creates new collections with 2 collections :

```
zipWith       \ x r y -- aArray : Return Array of results : itemx r itemy
[ 1, 2, 3] #+ [4, 5, 6] zipWith          \ Returns [ 5, 7, 9]
[ 1, 2, 3] [4, 5, 6] zipWith( #+ )       \ Returns [ 5, 7, 9]

zip           \ x y -- aArray   : Return Array of pairs [ itemx, itemy ]
```

```
[ 1, 2, 3 ] [ 4, 5, 6 ] zip       \ Returns [ [1, 4], [ 2, 5], [ 3, 6] ]
```

For other following mapping methods, you need to import the **mapping** package to load them :

```
import: mapping
```

#map applies a runnable (or a list of runnables) on items of a collection and returns the results into a new immutable array :

```
map                                   \ r x -- aArray

#sq [ 1, 2, 3, 4, 5 ] map             \ Returns [ 1, 4, 9, 16, 25 ]
[ #sq, #1+ ] [ 1, 2, 3 ] map          \ Returns [ [ 1, 2 ], [ 4, 3 ], [ 9, 4 ] ]
```

If we need a mutable array as return, it is possible to use **#mapm** instead of **#map**

#mapIf works like #map but collects results only for items that respond true to a condition :

```
mapIf         \ cond r x -- aArray

#even? #sq [ 1, 2, 3, 4, 5 ] mapIf      \ Returns [ 4, 16 ]
[ 1, 2, 3, 4, 5 ] mapIf( #even?, #sq )  \ Returns [ 4, 16 ]
```

#filter returns a new collection with only items that respond true to a condition :

```
filter        \ rcond x -- aArray

#[ 3 <= ] [ 1, 2, 3, 4, 5 ] filter      \ Returns  [ 1, 2, 3 ]
"acbDEfgHI" filter( #upper? )           \ Returns "DEHI"
```

#- is declared for collections and removes all items included into another collection :

```
-             \ x y -- array : Removes items of y from x

[ 1, 2, 3, 4, 5, 5 ] [ 2, 5 ] -        \ Returns [ 1, 3, 4 ]
"abcdefgABCabcd" "bcd" -                \ Returns "aefgABCa"
```

#extract, **#left** and **#right** extract elements from collections.

```
extract       \ i j x -- array : Extract items from index i to index j
left          \ n x -- array   : Extract the first n items
right         \ n x -- array  : Extract the last n items.
```

Many other mapping functions exists into this package. Check Word reference chapter or mapping.of file to learn more.

9 Memory management

Memory management is an area with the most differences with Standard Forth. Objects are created to the heap and are either handled by a garbage collector or manually.

9.1 Memory areas

Memory addressed by an Oforth system is divided into various areas :

Code space is the area where definitions are compiled into native code. This area is handled automatically by words that compile definitions.

Data stack is the area that holds parameters. There is one data stack by task. Default data stack size can hold 256 slots ie 256 objects. This value can be changed at startup with the --S command line option. Unless the --C command line option is used, data stack underflow/overflow are not checked.

Return stack is the area that hold word frames (return adresses and locals). This area is handled automatically and, if possible, is implemented using the processor stack.

Heap is the area where objects are created by #new. The heap is automatically handled by the garbage collector. All objects implicitly created by the system are created with #new : closures, arrays, floats, ...

```
new           \ ( cl -- x )
Creates a new object of class cl on the heap and push it on the stack. The object's
desallocation will be handled by the garbage collector when the object is no more used
```

Dictionary is the area where words are created. It also holds objects created with #alloc. This area is never de-allocated, but memory used by #alloc can be retrieved with #free and can be re-allocated again by next #alloc.

```
alloc         \ ( cl -- x )
Creates a new object of class cl into the dictionary and pushes it on the stack. The
object's desallocation must be manually handled using #free. The GC does not see those
objects.


free          \ x --
Free an object allocated by alloc. An exception is raised if the object does not
belong to the manual heap area.
Trying to #free an object not created by #alloc will throw an exception.
```

When the object is allocated, #new and #alloc call #initialize method with this object as receiver. Parameters needed to initialize attributes can be retrieved on the stack.

9.2 Mixing objects handled by GC and objects handled manually.

With some restrictions, Oforth allows to mix objects handled by the GC and objects handled manually.

Restrictions are :

- An object created with #new can't be the value of an attribute of an object created by #alloc (because the GC could not see it).
- An object created with #new can't be an element of an array created by #alloc
- An object created with #new can't be free with #free.

Theses restrictions are checked at runtime.

Apart from that, you can mix objects from all memory areas.

Note that objects created implicitly by the system are created with #new and will be handled by the garbage collector. Those objects are :

- Floats constants (1.2 for instance).
- String constants at the interpreter level ("aaaa" for instance). But String constants into a definition are created into the dictionary.
- Array or Json constants ([1, 2, 3] for instance).
- Big integers.
- Closures.
- Collection created when mapping other collections (see mapping chapter).

9.3 The garbage collector

Oforth Garbage Collector is an incremental Mark and Sweep GC : tasks are allowed to run while the GC is running.

The garbage collector is responsible to automatically deallocate object created into GC Heap memory area (so created by #new) and that are no more used.

If interested by the GC's internals or GC parameters, you can find information here :

http://www.oforth.com/memory.html

9.4 Direct access to memory

It is possible to have direct access to memory to read or write to a specific addresses. This should be done only to read or write specific ports (GPIO, ...), as there is no method to retrieve an object's address.

Direct access is done using integers as addresses. Methods available are :

```
_byteAt       \ n -- byte         Return byte value (0 - 256 ) at address n
_bytePut      \ byte n --         Set byte value at address n
_wordAt       \ n -- word         Return word value (0 - 65536 ) at address n
_wordPut      \ word n --         Set word value at address n
_int32At      \ n -- m            Return 32bits integer value at address n
_int32Put     \ m n --            Set 32 bits integer value at address n
_intAt        \ n -- m            Return integer value at address n
_intPut       \ m n --            Set integer value m at address n
_>string      \ n -- s            Create a new string with buffer at address n
```

Those methods should be used very carefully as accessing a forbidden address will probably result in a core dump. Errors are not catched.

Examples :

```
0x40000000 _byteAt .
0xFF 0x40000000 _bytePut
0x40000000 _bytePut( 0xFF )
```

10 Compilation

Oforth is an interpreter : it reads words from the inupt stream and execute them. Some of those words (#:, #method: , ...), when performed, create new words into the dictionary and associate code to them.

There is no separated compilation phase : the interpreter generates native code on the flow. As soon as a definition is closed, native code has been generated and can be executed.

The code is generated by a one-pass compilation, while the interpreter read names from the input buffer. This chapter explains how all this works.

10.1 The current definition and the STATE variable

The current definition is the definition the interpreter is currently compiling :

- For functions, the current definition is everything between : and ;
- For methods, the current definition is everything between method: (or virtual:) and ;
- For block declared at the interpreter level, the current definition is everything between #[and].

Outside those words, the current definition does not exists.

The interpreter works with a variable named STATE. This variable tells if it is running in INTERPRET state (outside a definition) or in COMPILE state (inside a definition).

STATE can only be modified by specific words :

- #: function and #method: change the STATE value to COMPILE
- #; function changes the STATE value back to INTERPRET.

When running in INTERPRET state, the interpreter behavior is :

- Runnables (functions, methods, ...) are performed at once.
- Other words (classes, properties, ...) and literals (integers and floats) are pushed on the stack.

When running in COMPILE state, the interpreter behavior is :

- Runnables are not performed : they are compiled into the current definition ie the current definition, when executed, will call this word.
- Some runnables are special and are executed even if the interpreter is in COMPILE (see Dual words).
- Other words and literals are not pushed on the stack, they are compiled into the current definition : code is added to push them, at runtime, on the stack.

Please note that, with the introduction of dual words (see below), the use of STATE variable in user-defined words is no more necessary. This variable is kept for compatibility purpose, and will probably be removed on a future release. So, unless required, you should not use this variable

(even for reading its value).

10.2 Dual words

Some words have a special behavior when the interpreter is compiling definitions: they are called "dual" (and are instance of Dual class, subclass of Function) because they don't follow the default compilation behavior of classical functions : when encountered during compilation, they will execute some specific code attached to them.

A dual word is created using the **#dual:** word instead of #: . In addition to classic code, executed during runtime, you can add instructions that will be executed during compilation time, using the **#comp:** word. Those instructions will replace the default compilation semantic and will also be executed by the #compile message (and by the outer interpreter as it calls #compile when it encounters a word during compile time). For instance :

```
dual: test
    "Hi, I'am here and this is runtime time" .cr ;
comp:
    "Hi, I'am here and this is compile time" .cr ;

test
Hi, I'am here and this is runtime time
ok

: test1  12 13 test + test ;
Hi, Iam here and the is compile time
Hi, Iam here and the is compile time
ok
test1 .
25
```

#test compilation behavior is performed by the interpreter during #test1 compilation. Nothing related to #test is compiled into #test1. #test1 has been compiled as if it was :

```
: test1 12 13 + ;
```

Dual words are used to execute actions while the interpreter compiles a definition. They are used to compile control structures, loops, ... into a definition and to end a definition using ; word.

And creating your own dual words allows to extend the language, add new control structures, ...

Note for Forthers : immediate words are here and you can create those words but this is no more necessary with dual words : apart from the interpreter itself, there is no more STATE-smart words. Immediate words are just specific dual words that have the same interpretation and compilation actions.

10.3 Example : compiling a simple word

Dual words allow to generate native code for functions and methods in a one pass JIT compilation. For instance :

```
12 dup * .
```

If you type this command, the interpreter it is into INTERPRET state, so :

- It read a token (12) , detects a number and push it on the stack.
- It reads a token (dup), detects a function and performs it at once.
- It reads a token (*), detects a method and perform it at once.
- It reads a token (.), detects a function and perform it at once.

Now if you write :

```
: square     \ x -- x*x
    dup * ;
```

The interpreter begins in INTERPRET state, too :

- It reads a token (:), detects a function and executes it at once. This function reads a token (“square”) from the input stream and creates a function into the dictionary with name "square". Then it changes the STATE value to COMPILE.
- As the "square" token has been consumed by #: , the interpreter does not "see" it. It reads the next token ie "\" , and detects a dual word. So instead on compiling it into the current definition, it performs it at once. Compilation action of #\ is to read all token until an end of line is reached.
- The interpreter reads the next token ie "dup" , and detects a function. As STATE is COMPILE, it compiles a call to "dup" into the current definition.
- It reads a token (*) and detects a message. It compiles a call to this message into the current definition.
- The interpreter reads a token (;) and detects a dual word. So, instead of compiling it into the current definition, it performs its compilation actions at once. This word closes the current definition and set the STATE value back to INTEPRET.

#square function has now been compiled into native code in a one-pass compilation and STATE value is INTERPRET again : the interpreter is ready for new instructions or new compilation.

Of course, it is crucial for #; to be a dual word. Otherwise there would be no way for the interpreter to leave the COMPILE mode.

This way of compiling definitions fits perfectly with RPN notation; you don’t have to wait to read an entire instruction before compiling it. Each token is compiled into the current definition the moment it is read by the interpreter.

It is the text interpreter that, along with dual words, controls interpretation and compilation of definitions. From the text interpreter perspective, compilation is not separated from runtime; everything is done by executing words. It just happens that some words, when executed during compilation time, generate native code.

10.4 The Control Stack and control structures resolution

Another stack ? Yes, another stack...

This stack is used during compilation to save information that will permit to resolve control structures. As the compilation is done in one pass, everything must be handled during this pass.

Let's see an example with the begin ... again loop :

begin is, of course, a dual word : it pushes the current code address on the control stack without adding nothing to the current definition. **again** is also a dual word : it pops the code address pushed by **begin** from the control stack and generate a jump to this address into the current definition. That's all.

This mechanism allows to compile definitions on the flow as the interpret reads words from the input stream.

Let's see an example with the **if** word .

if is a dual word that will be performed by the interpreter during compilation. It :

- Generates code to remove the top of the stack and test this value with false.
- Generates an "empty" conditional jump.
- Pushes the current code address on the control stack.

An "empty" jump means that, at this point, the address where to jump is not already known (but the space for this address is allocated).

After a short time (or a long time...), the **then** word corresponding to this "if" is reached by the interpreter. This word is also a dual word. Its action is very simple : it removes the code address (pushed by if) from the control stack, calculate the jump value and update the "empty" jump with the correct value. Now, our **if** test is completely generated and ready to be performed.

All other control structures (loops, while, continue, break, ...) work exactly the same way (for more information, see the prelude.of and compiler.of files). When everything is resolved, the body has been compiled in a one-pass compilation.

The control stack is a LIFO stack, so this allows nesting various syntax : tests, loops, ... without limitation on the number of nested levels.

10.5 Dual words, optimization and inlining

Default behavior when functions (or messages) are compiled is to add code to generate a call to this word. As dual words have a special compilation behavior, they can also be used to optimize the code generated for calling functions.

For instance, let's take the word #2dup, that duplicates two items on the stack (x y -- x y x y). If we write :

```
: 2dup    \ x y -- x y x y
    over over ;

: t 2dup ;
```

The compilation of 2dup into the #t function is a call to #2dup code. If needed (and this is the case for #2dup), we can "inline" this code, by declaring #2dup as a dual word :

```
dual: 2dup    \ x y -- x y x y
    over over ;
comp:
    #over compile #over compile ;
```

Now, when compiling #2dup, two #over are compiled. And, as #over is also a dual word (that, when compiling, generates code into the current definition), everything is inlined into the function that calls #2dup.

10.6 Macros

In order to simplify the comp: part of a dual word, macros can be used. A macro is a piece of code between <M> and</M>. Into a macro, words will be compiled into the current definition and literal are added as literal. For instance, 2dup can be rewritten using a macro :

```
dual: 2dup
    over over ;
comp:
    <M> over over </M> ;
```

Macro instructions can include dual words to. If so, their compilation action will be executed when the macro is performed. For instance :

```
dual: this
    compileOnly ;
comp:
    <M> self </M> ;
```

10.7 Directives

Into source files, it is possible to use some directives to condition some parts of the files. Those directives are uppercase to differentiate them from words used into definitions :

```
IFTRUE:              \ b -- : Execute following block if b is non false
System.ISWIN IFTRUE: [ : test "Windows system" .cr ; ]

IFFALSE:             \ b -- : Execute following block if b is false
DEFINED:             \ "name" -- b : Returns true if "name" is a defined word.
```

10.8 The interpreter revisited

Oforth's meta model is implemented as instances of classes that inherit from the Word class : Class, Property, Function, Dual, Message, Constant, Tvar, Package, ... Two methods are important for those words because they define their behavior when the interpreter encounters them. When the interpreter detects a word, it will execute (on this word) the #execute method if interpreting, and the #compile method if compiling.

Defaults actions (defined at the Word class level) are to handle the word detected as a literal, ie push it on the stack when interpreting and add it as literal to the current definition when compiling. Some words change this default action.

Here are the methods implemented for the built-in word types. If not redefined, the parent behavior applies :

Class	execute	compile
Word	Leave the word on the stack	Add the word as a literal to the current def
Class		
Property		
Package		
Function	Execute the function	Add code to execute the function at runtime
Dual	Same as Function	If defined, execute the comp: part of the definition. Otherwise, behave like a function.
Message	Execute the message	Add code to execute the message.
Constant	Execute #value method	Add constant value as a literal
Tvar	Execute #at method	Add Tvar as literal and compile #at message

10.9 Words for findind, compiling and postponing

Here are the word related to compilation and execution :

```
execute      \ w -- ...
    Execute word w
```

Message #execute allows to execute a word and is used by the interpreter when it detects a word in INTERPRET state.

```
compile      \ w --
    Compile word w
```

Message #compile allows to execute the compilation actions of a word and is used by the interpreter when it detects a word in COMPILE state.

```
literal      \ x --
    Compile object x as literal into the current definition.
```

#literal compiles an object into the current definition. When the definition will be executed, this object will be pushed on the stack. #literal is typically used into a word that will be executed when compiling.

For instance:

```
Constant method: compile
    @value literal ;
```

Compiling a constant is compiling its value as a literal into the current definition. At runtime, this value will be pushed on the stack.

```
: >compile      \ x --
    literal #compile compile ;
```

```
: >execute      \ x --
    literal #execute compile ;
```

Those two words are used to add the compilation or runtime action of a word (dual or not) to the current definition. For instance, an immediate word is a word which compilation action is equal to its runtime action, so #immediate is declared like this :

```
: immediate --
   comp: LAST-WORD >execute #; compile ;
```

Finally, we can use #postpone (instead of #>compile) if the word we want to append the compilation semantic is to be read from the input stream :

```
dual: postpone      \ <name> --
   compileOnly ;
comp:
   parse-word >compile ;
```

11 Declaring new kind of words

In the previous chapter, we have seen how to extend the language adding dual words that will execute actions during compilation. Here, we will see how to add new kind of words to the metamodel.

Forth, with its CREATE .. DOES> structure, was a precursor language by allowing to create words with a specific behavior, a first step to Object Oriented Programming.

Oforth implements a full OOP meta model, so the Forth CREATE ... DOES> construction is replaced by classes.

11.1 New kinds of words

It is possible to create new kinds of words by declaring a new class that will inherit from the Word class. For instance :

```
Word Class new: Material
```

Once the class is created, words are created like any other objects, by sending the #new message to this class. This requires a string as parameter that will be the name of the new word in the dictionary :

```
"cement" Material new .
#cement ok
```

#new creates a new word into the dictionary, with name "cement". This name is now detected by the interpreter as a word and the default actions, defined at the Word level, are executed : #execute leaves the word on the stack and #compile compiles this word a a literal into the current definition.

```
cement .s
[1] (Material) #cement

: test
   cement . ;
test
#cement ok
```

Once a word is created, it is persistent : it can't be free and won't be removed by the garbage collector. It can, of course, be retrieved by its name using "Word find" or # word :

```
"cement" Word find .s
[1] (Material) #cement

#cement .s
[1] (Material) #cement
```

It is possible to redefine the interpretation and compilation behavior of a particular kind of words by redefining the #execute and #compile methods.

11.2 Words classes versus CREATE ... DOES structure

This chapter explains the differences between defining new kind of words using classes (in Oforth) and the CREATE ... DOES structure (in Forth). Even if you don't know about this structure, you should be able to follow this chapter and its examples.

In Oforth, new kind of words are created by declaring a subclass of the Word class. This replaces the Forth structure, where "code1" are instructions used to initialize the new word (typically allocate and set memory) and code2 are instructions that are executed when the new created word is executed :

```
CREATE code1 DOES> code2 ;
```

The first difference is that, in Oforth, memory is allocated by declaring attributes for the word's class. The memory used for the new words will be those attributes and will be allocated by the #new message. As the #initialize method will be executed, it is the perfect place to initialize each new word. All this (attributes and #initialize) replaces the "code1" instructions of the create.. does> structure.

For instance, let's create a new kind of word : deferred words. The behavior of these words can be updated. Fisrt, let's create a Defer class :

```
Word Class new: Defer ( mutable action )
m: action      @action ;
m: set         := action ;
m: initialize  #[ "No action defined" abort ] := action ;

: defer:       parse-token Defer new drop ;
```

Here, the Word's data (the "action" attribute) is allocated by #new and the word initialization code is handled by the #initialize method (that is automatically called by #new). Please note that the #new message does not behave like the CREATE word : it takes a string as its last parameter, that will be the name of the word to create.

```
"foo" Defer new
```

This will create a new word, #foo, into the dictionary and push it on the stack. If you want to take the word's name from the input stream, you have to retrieve it using #parse-token. This is what the #defer: word does :

```
defer: bar
```

This will create a new word, #bar, of class Defer, into the dictionary. This word has one attribute, action, initialized with the block declared into the #initialize method.

The runtime action of words created by CREATE (code2) is replaced by 2 actions, declared using #execute and #compile methods. The default actions for interpretation (execute) and compilation (compile) are to push the word on the stack and to compile the word as a literal into the current definition. Those actions are similar to the default action of Forth word CREATE (when there is

no "code2" instructions), which is to push the address of the word's body on the stack. The difference is that this address is replaced by the word itself in Oforth.

The deferred words created with #defer: don't work as intended with the default behaviors. In order to have the correct behavior, we have to redefine them :

```
m: execute     @action execute ;
m: compile     self >execute ;
```

Now, deferred words work as intended and the interpreter handles them correctly (when it encounters a word, it calls #execute or #compile according to the STATE value) :

```
defer: foo

#[ dup + ] #foo set
12 foo .
24 ok
: test1  foo ;
14 test1 .
28 ok

#[ dup * ] #foo set
: test2 foo foo ;
20 test2 .
160000 ok
10 test1 .
100 ok
```

12 I/O and formatting

Input/Output are handled by files and console (sockets are described into the tcp package).

A stream is an object that can receive objects after formatting them. Streams are :

- Files
- String

12.1 Formatting objects

Formatting objects is the action to send objects in a particular format to a stream. A stream can be a File or a String. The same words are used to format objects, whatever the stream is. The first word is #<< :

```
<<            \ aStream x -- aStream    Sends object x to a stream
<<c           \ aStream c -- aStream    Sends character c to a stream
```

#<< is used by #.s to print the stack. It leaves the stream on the stack in order to use consecutive calls :

```
System.Out "aaaa" << 12 << Integer << 1.3 << drop
aaaa12#Integer1.3ok

String new "aaaa" << 12 << Integer << 1.3 << .s
[1] (String) aaaa12#Integer1.3
ok
```

To define a specific format, other functions are available.

#<<w allows to define a width for the format. If the formatted output of the object is greater than this width, the parameter is ignored. Otherwise, the object will be formatted using this width with a default justification

```
<<w           \ aStream w x -- aStream

System.Out "abcd" <<w( 8 ) "cdef" <<w( 8 ) 1.3 <<w( 8 ) 1234567 <<w( 5 )
abcd    cdef         1.31234567ok
```

#<<wj allows to specify the width and to change the default value to justify the object :

```
<<wj          \ aStream w justif x -- aStream

System.Out "abcd" <<wj(10, JUSTIFY_RIGHT
      abcdok

System.Out 12 <<wj(10, JUSTIFY_LEFT)
12        ok
```

#<<wjp allows to also define a precision for the output format. This method is only available for

numbers (Integers and floats) :

```
<<wjp         \ aStream w justif precision aNumber -- aStream

System.Out  12 <<wjp(10, JUSTIFY_RIGHT, 5)     \ Output "     00012"
System.Out -12 <<wjp(10, JUSTIFY_RIGHT, 5)     \ Output "    -00012"
System.Out 1.2234 <<wjp(10, JUSTIFY_RIGHT, 2)  \ Output "       1.2"
```

These methods use #addFloatFormat, #addIntegerFormat, ... that must be defined for the stream used.

As an example, here is how a date is formatted :

```
Date virtual: <<
   self year   <<wjp(0, JUSTIFY_LEFT, 4) '-' <<c
   self month  <<wjp(0, JUSTIFY_LEFT, 2) '-' <<c
   self day    <<wjp(0, JUSTIFY_LEFT, 2) BL  <<c
   self hour   <<wjp(0, JUSTIFY_LEFT, 2) ':' <<c
   self minute <<wjp(0, JUSTIFY_LEFT, 2) ':' <<c
   self second <<wjp(0, JUSTIFY_LEFT, 2) ',' <<c
   self microSecond 1000 / <<wjp(0, JUSTIFY_LEFT, 3)
;

Date now .
2018-05-07 14:18:51,638 ok
```

12.2 Basic input/output

At startup, two files are created and open. They are affected to constants :

```
system.Out          Output defined when Oforth is launched.
System.Err          Error defined when Oforth is launched.
```

Basic output words use System.Out :

```
emit          \ n --       Print the character corresponding to unicode n
type          \ s --       Send string s to System.Out
.             \ x --       Print object x, then a space
.cr           \ x --       Print object x, then perform a carriage return
princr        \ --         Perform a carriage return
```

13 Multi-tasking and concurrent programming

Oforth implements a task/channel model : a task is a piece of code that can run concurrently with other tasks. Each task is isolated and can communicate with others only through channels.

A channel is a structure that allows tasks to send objects and for other tasks to receive them.

13.1 Tasks

Tasks are instructions that will run in parallel with other tasks. If the system has more than one core/processor (and if the --Wn option is set) , tasks will truly run in parallel. Otherwise, each task will use a part of the CPU. It is the Oforth system that will decide which task will run.

Unlike threads, tasks are very light objects, with its own data stack. The data stack is not shared by tasks.

Creating a task is done using #sched or #& word :

```
sched   \ r n --     : Create and schedule a new task that will run r in parallel
                       n is the data stack size to be created (n objects).

&       \ r --       : Creates and launch a new task that will run r in parallel.
                        with default data stack size (160 objects).
```

Launching a task does not mean that this task will run immediately. It is tagged as resumable and will run when a CPU is available. If a task stay some time in resumable state, the VM can decide (if possible) to create another worker.

For instance :

```
: helloWorld
    "Hello, World\n" . ;

#helloWorld &                    \ Launches function #helloWorld in parallel
#[ 10 #helloWorld times ] &      \ Launches a block in parallel (running 10 hw).

10 #[ #helloWorld & ] times       \ Launches 10 tasks each running #helloWorld
```

Sending parameters to a task is done by using a closure : a closure will keep values that will be used by the task. Of course, those values can't be mutable (if so an exception is raised). You can create a closure with mutable values, but, if so, you can't run it in parallel.

```
: hello             \ s --
    "Hello," . .cr ;

: test( s -- )      \ Launches a new task
    #[ s hello ] & ;
```

```
"Franck" test                    \ ok
String new "Franck" << test      \ ko : an exception is raised
```

The current task can #yield or #sleep. In this case, the system will resume another resumable task.

```
sleep         \ n --             Sleep current task for n milliseconds
yield         \ --               Allow another task to run.
```

#yield is not necessary for other tasks to run : the system will periodically execute automatic yields on the current task to allow other tasks to run.

13.2 Threads and workers

In Oforth, threads are not exposed to the programmer : they are handled automatically by the virtual machine.

The programmer creates tasks and the VM chooses (or creates if necessary) a thread to run this task. If a task is paused because it waits for an event or a resource (a channel, a socket, a console, ...), this does not block the thread the task was running on : the thread is automatically affected to run another resumable task.

Those threads are also called workers. At startup, only one worker is launched, running the interpreter. If tasks are resumable and no worker is available, the VM creates a new worker up to the maximum number of workers declared. By default, only one worker will run ie the maximum number of workers is 1. This can be changed used a command line option :

```
--Wn          \ Defines n as the maximum number of worker created by the VM.
```

13.3 Channels

A channel is a way to communicate between tasks. It is a structure dedicated for sending and receiving objects. Multiple tasks can send to the same channel and multiple tasks can receive from the same channel. Only immutable objects can be sent between tasks into a channel.

```
newSize       \ n Channel -- aChannel  : creates a new channel with size n
new           \ Channel -- aChannel    : creates a new channel with defaut size (200)
allocSize     \ n Channel -- aChannel  : creates a new channel with size n
alloc         \ Channel -- aChannel    : creates a new channel with defaut size (200)
```

After a channel is created, it is open and objects can be sent into or receive from a channel :

```
sendTimeout   \ x n ch -- true | err false : Send x to the channel with n as timeout.
```

#sendTimeout sends an immutable not null object into a channel. If the channel is full, the task will wait until there is room or timeout (n microseconds) is reached. If the send is successful, return is true. Otherwise, return error and false. Sending null as timeout value means "no

timeout", so this will be the same behavior as #send.

```
send          \ x aChannel -- b   : Send x into aChannel
```

#send sends an immutable not null object into a channel. Same as sendTimeout, but block (if necessary) until the channel is no more full.

```
receiveTimeout\ n aChannel -- x true | err false : Receives an object from a channel.
```

#receiveTimeout retrieves an object from a channel. If the channel is empty, the task waits until an object is present into the channel or the tiemout is reached (n microseconds) . Return the object and true. Return the error and false if timeout occured or the channel is closed AND empty.

```
receive       \ aChannel -- x | null : Receives an object from a channel.
```

Same as #receiveTimeout, but wait until an object is available into the channel. Return this object or null if the channel is closed and empty.

A task can receive objects from a closed channel while it is not empty (but a task can't send an object into a closed channel).

Channels themselves represent the transport of objects between tasks, not the objects themselves. So a channel is an immutable object and can be the value of a constant or sent as parameter to a task. For instance :

```
Channel new const: MyMailBox

: job                \ --
   1 2 + 4000 sleep MyMailBox send drop ;

#job &
 MyMailBox receive .
```

It is common to send a channel as parameter to another task. This allows to specify on which channel this task will send or receive objects. As channels are immutable objects, they can be values into a closure.

Here is a ping pong between two tasks where the channels used are created prior to running the tasks :

```
: pong( n  ch1  ch2 -- )
| i |
    n loop: i [
        ch1 receive
        "Pong : receiving" . dup .cr
        "Pong : sending back sqrt" .cr
        sqrt ch2 send drop
        ]
    "Pong : job done" .cr
;

: ping ( n -- )
| ch1 ch2 i |
     Channel new ->ch1
     Channel new ->ch2
     #[ n ch1 ch2 pong ] &
     n loop: i [
```

```
        "Ping : sending" . i .cr
        i ch1 send drop
        ch2 receive
        "Ping : receiving back " . .cr
      ] ;

10 ping
```

The #ping function creates 2 channels, then launches a new task running #pong in parallel. It uses a closure to send 3 parameters to #pong function : n, ch1 and ch2. Then it sends integers from 1 to n into channel ch1, waits for the answer from ch2 and print the object returned.

The #pong function loops n times : it waits until an object is available on channel ch1 and, when received, calculates #sqrt and sends the result to channel ch2.

A channel is the only object that allows to synchronize tasks.

A channel can be used for various purposes :

- Manage a log file where multiple tasks can write in parallel (see logger package).
- Manage responses to events (see emitter package).
- Manage servers (see tcp package)
- ...

13.4 Resources

A task can wait for a resource to be available. If so, the task is stopped until the resource is available and the worker can run other tasks. A task waiting for a resource consumes no CPU : the task is blocked (but not the thread).

Channels are resources, but they are not the only resources defined. Other resources are :

- Waiting for a duration (#sleep)
- The console input / output
- Sockets.

When a task is asking for a resource and this resource is not available (channel empty or full, no input, no data on the socket, ...), the task enters into a WAIT state and is no more resumable until the resource is available. When the resource is available, the VM detects the event and set the task back to a resumable state. The task will run when a thread is ready to run it.

The console is also a resource, so a task waiting for console input does not block any thread. See the console package chapter for more information/

Sockets are also resources. Reading or writing on a socket will block the task (but not the thread) if the socket is not ready for the operation. See the socket package for more information.

13.5 Immutability and task isolation

Objects have the property to be immutable or not. The rule si simple : if a class has no mutable attribute, all objects created from this class are immutable. On the other hand, if a class has at least one mutable attribute, all objects created from this class will be mutable.

There is no way to change an immutable object into a mutable object, but you can change a mutable object into an immutable object using **#freeze** :

```
freeze        \ x --
```

To be able to use **#freeze**, all object's attributes values must be immutable. Otherwise an exception is raised :

```
Object Class new: A ( mutable x )
m: setX    := x ;

A new dup setX ( 12 ) dup freeze
ok
A new dup setX ( Array new ) dup freeze
[console:1] #Exception : Immutable rule violation
```

Memory used by a task is isolated : other tasks can't see mutable objects created by a task. Only immutable objects can be visible. Two tasks can't update the same object at the same time, so a task is assured that its mutable objects are only visible by itself.

This is done by design :

- There are no global variables that could hold a mutable object visible by tasks.
- There are no class attributes.
- tvar values are "by tasks".
- Constant values can't be mutable (an exception is raised if you try of create a constant with a mutable value).
- Channels only accept immutable objects.
- An immutable attribute value can't be set with a mutable object.
- Attributes value of an immutable object (ie an object with only immutable attributes) can't change after the object is initialized.

All this is checked at runtime and an exception is raised if a problem occurs.

The first consequence of this model is that objects are never copied when sent to a channel. As this object is immutable, there is no way for a task receiving it to update it.

The second consequence is that there is no mechanism such as mutexes, semaphores, ... Those mechanisms are not necessary as there is no situation where a mutable object can be updated by another task, other than the one that created it.

14 Exceptions

Exceptions are objects that can be thrown when an exceptional event occurs.

Some built-in functions generate exceptions and a program can generate its own exceptions.

When an exception is thrown, the current execution stops and the program restart to a point where the exception is catched.

14.1 Catching exceptions

The text interpreter catches all exceptions; if an exception is not catched by the program, it will be catched by the text interpreter :

```
: test1 1.2 0 / ;
: test   test1 ;
test
[console:1] #Exception : Division by zero error
```

User defined exceptions, if not catched, will be catched by the interpreter too :

```
: test1 "This is my exception" abort ;
: test   test1 ;
test
[console:1] #Exception : This is my exception
```

A control structure try/when allows to catch an exception and decide what to do with it. A try/when structure needs a local variable to be declared. When the exception is catched, the local value is the catched exception into the when block :

```
try: e [ instructions1 ] when: [ instructions2 ]
```

If instructions1 throw an exception, the program stops and execute intructions2 into the when block.

If instructions1 don't throw an exception, instructions2 are not executed and the program continue after the when block.

There are three ways to handle a catched exception into the when block :

1) You can do some work (log, print, clean, ...) and throw the exception again for another try/when block (or the interpreter) :

```
: test1       1.2 0 / ;
: test | e | try: e [ test1 ] when: [ "I catched you, " . e . e throw ] ;
```

2) You can do some work and continue

```
: test1       "An exception" null Exception throw ;
: test
```

```
| i e |
 10 loop: i [
    try: e [ test1 ] when: [ "I catched you, but I don't care" .cr ]
         ]
;
```

3) If you want to catch only some kind of exception you can check the exception type (i#sA or #isKindOf) to decide what to do (handle or throw again).

14.2 Exception class

Exception class is the base class exceptions.

Methods implemented are :

```
throw          \ s Exception --     : Creates and throw an exception with message s.

throw          \ aException --      : Throw an exception
message        \ aException -- s    : Returns the exception's message.
log            \ aException --      : Log an exception on standard error.
```

All exceptions inherit from this class.

An exception is created using #new and takes 2 parameters : the exception message and an object that describe the exception (or null if it is a general exception).

```
"My exception" null Exception new
```

Once an exception is created, it can be raised using #throw.

Two shortcuts exist to directly throw an exception

```
"My exception" abort              \ Create an exception and throw it
"My exception" 1.2 abortWith      \ Create an exception with an aobject and throw it
```

15 Files

File objects represent OS files. They allow to read and write into the corresponding system files.

#newMode is used to create a new file :

```
newMode              \ filename mode File -- aFile
   filename is a string corresponding to system file name
   mode is :
        File.BINARY : open a binary file
        File.TEXT : open a text file
        File.UTF8 : open a file containing UTF8 characters.

"myfile" File.BINARY File newMode
File newMode( "myfile", File.BINARY )
```

To create a file with mode = File.UT8, #new can be used :

```
new                  \ filename File -- aFile

"myFile.txt" File new
"myFile.txt" File new
```

Creating a file does not open it. Some methods don't require the file to be open :

```
name           \ aFile -- s : returns file name

stats          \ aFile -- nc nm ns | null null null : Returns file stats
 nc is the number of microseconds for file creation
 nm is the number of microseconds for file modification
 ns is the file size
```

Those values are null if the file is not accessible or does not exists. They can be retrived directly using :

```
exists         \ aFile -- b
size           \ aFile -- ns
created        \ aFile -- nc
modified       \ aFile -- nm
```

In order to read or write, the file must be open :

```
open           \ access aFile --
open?          \ aFile -- b : Returns true is the file is open

aFile open(File.READ)
File.WRITE aFile open
```

#open opens a file with an access mode. This method throws an exception if the file can't be open. Access can be :

- File.READ : Open for reading

- File.WRITE : Create and open for writing
- File.APPEND : Open for writing at the end of the file.

Once the file is open :

```
close          \ aFile --          Closes the file. Do nothing if already closed.
position       \ aFile -- n        Returns current file position
reposition     \ origin offset aFile --  Set file posiition.
```

You can get the current file position using #position and, with some constraints set the file position using #setPosition. #setPosition will set the position according to an origin and an offset (the new position will be origin + offset).

- If the file is created with File.BINARY mode, origin can be File.BEGIN, File.CURRENT or File.END and offset is a number of bytes from this origin (it can be the value returned by a previous call to #position).
- If the file is created with File.TEXT or File.UTF8 mode, origin can only be File.BEGIN and offset can only be zero or a position returned by a previous call to #position.

Methods used to write into a file are :

```
add            \ n aFile --        Adds n to aFile
addChar        \ n aFile --        Same as #add
flush          \ aFile --          Flush pending data.
```

#add writes n to a file :

- If the file is not created as an UTF8 file, the byte n is written.
- If the file is created as an UTF8 file, the byte(s) corresponding to the UFT8 sequence of unicode code n is written. An exception is raised if n is not an unicode code.

Files are buffered and an effective write on disk will occurs when the buffer is full. #flush allows to flush immediately all pending data to write. #flush can't be used on files opened with File.READ (to flush standard input, use #flush on the console, see Console chapter).

A file is a stream, so formatting methods can be used to write to a file (see Formatting objects chapter for details on those methods) :

```
<<c            \ aFile c -- aFile              Write character c ot byte c to a file.
<<             \ aFile x -- aFile              Write object x to a file.
<<w            \ aFile width x -- aFile
<<wj           \ aFile width justif x -- aFile
<<wjp          \ aFile width justif precision aNumber -- aFile
```

Writing a buffer (aMemBuffer, aString, ...) into a file is done using #<<

```
aFile "abcdef" <<
```

In order to read from a file, methods used are :

```
end?           \ aFile -- b        Returns true if end of file is reached.
>>             \ ( aFile -- c )    Read a character from a file
```

#>> reads a file and returns an integer. The value returned depends on the file mode :

- For File.BINARY and File.TXT, a byte is returned.
- For File.UTF8, the unicode code of the next UTF8 encoded char is returned.

Multiple bytes or chars can be read at the same time :

```
readWith       \ n aMemBuffer aFile -- aMemBuffer
read           \ n aFile -- aMemBuffer

readCharsWith  \ n s aFile -- s : Store characters into a string
readChars      \ n aFile -- s

readLineWith   \ aBuffer aFile -- aBuffer | null
readLine       \ aFile -- aString | aBuffer | null
```

#readWith and #read are dedicated to read non UTF8 chars. They read n bytes and populate a MemBuffer (#read creates a new MemBuffer).

#readCharsWith and #readChars are dedicated to read UTF8 characters. They populates a string.

For all these methods, the number of bytes of characters read can less then than the number asked. The effective number is the size of the object returned.

#readLineWith and #readLine read a file line by line. and #forEach: and all higher order functions can be used on a file :

```
"myfile.txt" File new map(#[ words first ]) .
```

This will return an Array of the first word of each line of file "myfile.txt", null if none.

```
"myfile.txt" File new map(#yourself) const: LINES
```

This will create a constant LINES which value is the list of all lines of "myfile.txt".

16 Packages

16.1 Package word

During an Oforth session, two words can't have the same name : if you try to do this, you will raise an exception. This rule is strict and, if you load many features, name conflicts may appear.

In order to handle this, Oforth implements packages. A package is a word that create a namespace into the system.

Two words can have the same name if they are declared into two different packages. A word name can be prefixed with its package to avoid ambiguity when necessary :

```
12 dup .s
13 oforth:dup .s
```

oforth is the package where all built-in words are declared.

At any time, the system has a current package, where all new words are created. When the interpreter starts, the current package is **oforth**.

Package's themselves are words declared into the oforth package :

```
import: date
date .s
oforth:date .s
```

A package word is represented by a file on the disk, which name must be **package_name.pkg**

This file is an Oforth source file that will be loaded when the package is imported into the system. It can contains any Oforth code or directive.

So :

```
1 package = 1 word = 1 namespace = 1 package file
```

16.2 Package file and the files: directive

An package file is like any other Oforth source file. When the package is imported, the file will be loaded and interpreted.

The package file (which name is package_name.pkg) must be found. This file is searched :

- Into the current directory.
- Into directories listed into the OFORTH_PATH variable value.

- Into a "packs" directory into these directories.

The **file:** directive can be used to load other files from the package file. The string provided is a file name, relative to the directory where the package file has been found.

For instance, when importing the **date** package, if the file **date.pkg** has been found into "/home/oforth/packs" and if this file has this directive :

```
file: date/Date.of
```

Then the file /home/oforth/packs/date/Date.of will be loaded into the namespace **date**.

For more details, you can see how packages are defined into the **packs** directory provided into the Oforth archive.

16.3 Importing or using a package

A package is loaded using #import: or #use:

```
import:             \ "name" --         : Import the package named "name"
use:                \ "name" --         : Use the package named "name".

import: date
use: tcp
```

#use: loads the package which name is into the input stream :

- If this package is already loaded, #uses: does nothing and returns.
- It creates a new package word into the **oforth** namespace.
- It changes the current package to this package
- It loads the package file into this namespace (and so all files loaded with **file:** directive).
- It restore the current package to the previous value.

With #uses: , words created must be prefixed with the package name. Otherwise, they are not found :

```
use: date
Date .s             \ Not found, words must be prefixed
date:Date .s        \ ok
```

It is possible to create aliases to those words :

```
#date.Date alias: Date
```

A package can also be loaded using #import:

```
import: package_name
```

This function does the same thing than #use: but the package loaded is declared as an imported package of the current package. Into this current package, words imported can be used without prefixing them :

```
import: date
Date .s
```

Here, package **date** is imported into the current package (probably **oforth** package, unless imported from another package). So date package is declared as an imported package of oforth and its words can be accessed without prefixing them (you can prefix them if you want or to avoid ambiguity).

```
Date now .s
```

A package has a list of imported packages. When this package is the current package (ie when it is loaded into the system), each package loaded using "import:" is added to this list. All words declared into a imported package don't have to be prefixed with the package's name.

import: and use: can import multiple packages using the ',' separator. For instance :

```
import: date, mapping, collect
```

16.4 Search order for words

Search for words occurs when # is used (the name is read from the input stream) or using #find word (the name is into a string provided as parameter).

If the name is qualified (ie prefixed by its package), the search is done only into this package.

If the name is not qualified, the search is extended :

- To the current package.
- If not found, to imported packages of current package.
- If not found, to the oforth package.
- If not found, to imported packages of oforth package.

Consequences :

- Names created into a package can be used into this package without prefixing them.
- Importing a package just adds it to the list of the imported packages of the current package. If oforth package imports a tcp package and if tcp package imports logger package, words defined into logger must be prefixed into oforth (unless oforth itself imports logger package).
- There is no conflict error when a non qualified word is used and has a name declared more than once. The first word found will be used.

17 FFI

Oforth allows to define structures like C structures and call functions from dynamic libraries.

Currently, structures are not fully implemented.

In order to use FFI, you have to import the ffi package

```
import: ffi
```

17.1 Structures

To be described

17.2 Dynamic Libraries

Dynamic libraries are words. They are created and loaded using #extlib: or extwlib: according how parameters should be handled.

```
"kernel32"  extwlib: LIBKERNEL
"msvcrt"    extlib:  LIBC
"libc.so.6" extlib:  LIBC
```

This creates and loads a new dynamic library as an Oforth word.

17.3 Dynamic functions

Once a library is created, dynamic functions can be created using extern: directive

```
lib nbparam returnClass str extern: name
```

Where :

```
lib is the dynamic library that contains the procedure to call
nbparam is the number of parameters to send
returnClass is the object returned (null or Integer or String)
str is the name of the procedure into the library
name is the name of the word to create to represent this procedure.
```

Examples :

```
LIBC 1 Integer "printf" extern: _printf
LIBC 1 String  "getenv" extern: _getenv
LIBC 1 Integer "system" extern: _system
```

```
: cls
    "cls" _system drop ;

: dir
    "dir" _system drop ;
```

18 Environment

18.1 Environment constants

Here are the constants set at startup :

```
System.VERSION      \ -- s       Returns Oforth version.
System.WIN?         \ -- b       True if build for Windows OS.
System.LINUX?       \ -- b       True if build for Linux OS
System.MAC?         \ -- b       True if build for Max OS
System.FLOAT?       \ -- b       True if build support Float
System.TCP?         \ -- b       True if build support TCP
System.DYNLIB?      \ -- b       True if build support TCP
System.MTHREAD?     \ -- b       True if build support Mutli-Thread
System.DEBUG?       \ -- b       True if build support debug mode.
System.OPTIMIZE?    \ -- b       True if not --C option.
System.TEST?        \ -- b       True if --t option
System.CORES        \ -- n       Number of cores detected.
System.WORKERS      \ -- n       Max number of workers that the VM will create.
System.ARGS         \ -- aArray  Array of command line arguments (only with -).
System.PATHS        \ -- aArray  Array of paths retrieved from OFORTH_PATH env var.
System.Console      \ -- aConsole Oforth console (null if no console).

System.Out          \ -- aFile   Standard output file
System.Err          \ -- aFile   Standard error file
```

18.2 Time functions

Function **System.tick** retrieve a system tick and can be used to calculate elapsed time between two ticks.

```
System.tick         \ -- u       Return a tick in microsecond
```

To retrieve number of microseconds since 01/01/1970 :

```
System.time         ( -- u )
System.localTime    ( -- dst min u )
```

System.localTime returns :

- dst : boolean that say if daylight saving time.
- min : number of minutes between utc time and local time.
- u : same as System.time (number of microseconds since 01/01/1970).

19 Words reference

This section lists all defined words at startup.

Words in grey are primitives (built-in words), other words are defined in Oforth.

Primitives in italic are necessary to construct Oforth language but are forgotten after Oforth is loaded : they are not available for the user.

Words created during launch sequence but forgotten when Oforth is launched are not listed here (if interested, see compiler.of and prelude.of files).

19.1 Words available for all built-in options.

Constants

CELLSIZE	-- n	Return cell size in bytes (4 on 32 bits)
System.FLOAT?	-- b	true if the build support floating point numbers.
System.TCP?	-- b	true if the build support TCP
System.DYNLIB?	-- b	true if the build support dynamic libraries (FFI)
System.MTHREAD?	-- b	true if the build support multi-thread
System.WIN?	-- b	true if the build is on Windows OS
System.LINUX?	-- b	true if the build is on Linux OS
System.MAC?	-- b	true if the build is on Mac OS
System.TEST?	-- b	true if test blocks are performed (--t option)
System.OPTIMIZE?	-- b	true if optimize mode (not --C option)
System.WORKERS	-- n	Number of workers launched (--W option)
System.CORES	-- n	Number of cores detected
System.ARGS	-- [s]	Command line arguments
System.PATHS	-- [s]	Array of defined paths (OFORTH_PATH env variabl333
System.VERSION	-- s	Oforth version
System.Out	-- stm	Output stream
System.Err	-- stm	Error stream
null	-- null	null value
false	-- false	Constant with value 0
true	-- true	Constant with value 1
CELLBITS	-- n	Number of bits into a cell (32 on 32 bits)
JUSTIFY_LEFT	-- n	Used by formatting methods
JUSTIFY_RIGHT	-- n	Used by formatting methods

Variables (task variables)

_LIT	*-- aWord*	*Word used to handle literals*
_INTERPRET	*-- aWord*	*Word used to interpret a name*
CURRENT	-- aPackage	Current package
CODE	-- n	Code pointer into current definition
STATE	-- n	State value (0 = runtime, > 0 = compile)
System.Console	-- console	Console object (or null if no console detected)
SOURCE	-- x	Current source for input stream
SOURCE-TYPE	-- aClass	Type of current source (String or File)
SOURCE-NAME	-- s	Name of current source
SOURCE-LINE	-- n	Current line number of current source
SOURCE-INDEX	-- n	Current index into current line.
CS	-- []	Control stack
LAST-NAME	-- symbol	Last word name read on input stream
LAST-METHCLASS	-- cl	Last method class declared (null if function)
LAST-IMPLTYPE	-- Object | Class	Last method type declared

Compilation and interpretation

_BYTECODE	*n --*	*Add bytecode n to current definition code*
_CELLCODE	*n --*	*Add cell n to current definition code*
_RAWCODE	*n --*	*Add raw integer to current definition code*
_RET	*--*	*Add return code to current definition code*
_checkStack	*--*	*Check for stack underflow and overflow*
_;	*--*	*Basic end of current definition*
_method	*cl t "name" --*	*Basic beginning for a new method implementation*
_resolveContinue	*--*	*Resolve continue(s) for current block*
_resolveBreak	*--*	*Resolve break(s) for current block*
_closureVarUsed	*--*	*Tag current block as a closure*
_closureVarSet	*--*	*Set variable into a closure*
_setAtt	*--*	*Generate code to set attribute*
_setImmAtt	*--*	*Generate code to set immutable attribute*
_setLongJump	*--*	*Set long jump to "when" block if exception*
_strAsInteger	*--*	*Convert a string to an integer (or null if not)*
_lit	*--*	*Generate code for literals*
to	x "name" --	Push x to variable "name"
"		Read and push a string
'		Read and push a character
\		Comment
LAST-WORD	-- x	Return last word created into the dictionary
parse-token	-- s	Parse next token as string from input stream
parse-char	-- n	Parse next char from input stream
next-char	-- n	Return next char from input stream
immediate	--	Set last definition as immediate
immediate?	x --	Return true if x is an immediate function
forget:	"name" --	Forget word with name "name"
:	"name" --	Beginning of function definition
dual:	“name” --	Beginning of a dual word.

comp:	--	Compilation action of the last dual word.
Compile>	w --	Compilation action of dual word w
;		End of definition or the compilation action.
const:	x "name" --	Create a new constant with value x
tvar:	"name" --	Create a new task variable
alias:	"name" --	Create a new alias
message:	"name" --	Create a new message (method)
_method:	cl t "name" --	Beginning of method for class cl
method:	cl "name" --	Beginning of method for class cl
m:	"name" --	Beginning of method for last class created
virtual:	cl "name" --	Beginning of virtual implementation for class cl
classMethod:	cl "name" --	Beginning of class implementation for class cl
classVirtual:	cl "name" --	Beginning of virtual class implement for class cl
abortWith	s x --	Throw an exception with message s and object x
abort	s --	Throw an exception with message s
CS>	-- x	Pop an object from the control stack
>CS	x --	Push x on the control stack
<mark	C: --	Mark code location (on CS) for backward jump
<resolve	C: --	Resolve jump to location marked by <mark
>mark	C: --	Generate code for a forward jump
>resolve	C: addr --	Resolve forward jump generated by >mark
branch	C: --	Generate code for a unconditional jump
?branch	C: type --	Generate code for a conditional jump of type type
literal	C: x -- R: -- x	Generate code to push x on the stack
#	"name" -- w	Retrieve word with name "name" and push it
$	"name" -- sym	Push symbol "name" (create it if necessary)
--	C: -- R: --	Comment
#!	C: -- R: --	Comment
RS?	n --	Check if n is a valid index on return stack
RS@	C: n -- R: -- x	Push value stored on return stack at index n
RS!	C: n -- R: x --	Pop value and store it at index n on the RS
self	C: -- R: -- x	Push the receiver of current method
super	C: "name" --	Push the receiver and call method at upper level
interpreter	b s --	Final interpretation of string s with mode b
(	C: --	Parameter list declaration
)	C: --	End of parameter list declaration
|	C: --	Local variables list declaration
parse-local	"name" -- n	Parse a name and retrieve corresponding local
->	"name" --	Pop top of stack and store it into local "name"
<M>	--	Beginning of a macro (that will end with </M>
#[	--	Beginning of block or closure
parse-attribute	"name" -- index	Retrieve attribute corresponding to "name"
@	"name" --	Generate a push of value of attribute "name"
:=	"name" --	Pop stack and store the value at attribute "name"
[	--	Begin of an array declaration
,	--	Array or Json separator
]	--	End of block or array declaration
parse-until	C -- s	Parse input until character c or end of stream
parse-skip	C --	Parse and skip input until c or end of stream

assert	b --	Throw an exception "assertion failed" if b is false
DEFINED:	"name" --	Return true if "name" is a defined word
0x	"hexa" --	Return integer value of hexadecimal string read
0b	"binary" --	Return integer value of binary string read
use:	"name" --	Load package with name "name"
import:	"name" --	Load package with name "name"

Control structures

if	C: -- R: b --	generate test to check top of stack
then	C: --	Resolve previous if
else	C:	Jump and resolve a previous if
continue	C: --	Jump to the beginning of current loop
break	C: --	Jump to leave current loop
begin	C: --	Beginning of unconditional loop
again	C: --	End of unconditional loop
while		Beginning of a while structure
until	--	End of a begin ... until loop
return	C: -- R: --	Exit current definition
--	C: -- R: --	Comment
#!	C: -- R: --	Comment
ifTrue: [	C: -- R: b --	Generate test to check top of stack with true
ifFalse: [	C: -- R: b --	Generate test to check top of stack with false
ifNull: [	C: -- R: b --	Generate test to check top of stack with null
ifNotNull: [	C: -- R: b --	Generate test to check top of stack with null
]	C: --	End of test block
else: [	C: --	Generate jump and test
ifZero: [	C: -- R: b --	Generate test to check top of stack with zero
if=: [	C: -- R: x y --	Generate test of 2 elements on the stack
loop: var [	"name" [--	Beginning of an integer loop
for: var [	"name" [--	Beginning of an integer loop
step: var [	'name" [--	Beginning of a interval loop
forEach: var [	"name" [--	Beginning of a collection loop
try: var [	"name" [--	Beginning of a try block
when: [	[--	Beginning of a when block
IFTRUE: [	--	Beginning of an IFTRUE block
IFFALSE: [	--	Beginning of an IFFALSE block
test: [	--	Beginning of a test block

Other functions

dup	x -- xx	Duplicate x
drop	x --	Drop x
swap	x y -- y x	Swap 2 elements
over	x y -- x y x	Duplicate second element
rot	x y z -- y z x	Rotate 3 elements
pick	... n -- ... x	Duplicate nth element (1 based)

=	x y -- b	Return true if x = y (same reference)
.depth	-- n	Return data stack size
bye	--	Leave interpreter
mem	--	Print memory allocation
sleep	n --	Sleep current task for n milliseconds
yield	--	Yield current task
sched	x n --	Perform x asynchronously, into a separated task, with n as data stack size (number of objects).
System.lastError	-- n	Return last error detected.
System.tick	-- n	Return a system tick
System.localTime	-- dst min mic	Return local time
tuck	x y -- y x y	Copy the top of stack under the second element
nip	x y -- y	Remove the second element
-rot	x y z -- z x y	Rotate 3 elements
cells	n -- m	Return number of bytes corresponding to n cells
not	b -- b	Return true if false, false otherwise
and	b b -- b	Compute a and betwwen 2 booleans
or	b b -- b	Compute a or betwwen 2 booleans
xor	b b -- b	Compute a xor betwwen 2 booleans
2drop	x y --	Drop 2 elements from the stack
2dup	x y -- x y x y	Duplicate 2 elements
1+	x -- x+1	Add 1 to top of stack
1-	x -- x-1	Substract 1 from top of stack
under	.. y r -- .. y	Remove y, execute r and push back y on the stack
clr	... --	Clear the stack
subclassResponsability	m --	Throw an exception : class should redefine method m
&	x --	Perform x asynchronously, into a separate task.
cr	stream --	Send CR to a stream
print	x --	Print object x on System.Out
printcr	--	Print CR on System.Out
.	x --	Print object x, then blanck on System.Out
.cr	x --	Print object x, then CR on System.Out
.s	--	Show the stack

Object class (child of null) :

new	aClass -- x	Create a new object handled by GC
alloc	aClass -- x	Create a new object handled by the user.
free	X --	Free an object created by #alloc
initialize	X --	Initialize an object (default is to do nothing).
freeze	x --	Freeze an object. The object is no more updatable.
hashValue	x -- n	Return a hash value of x
yourself	x -- x	Return the receiver
class	x -- aClass	Return class of x
is?	aClass x -- b	Return true if x is of class aClass
null?	x -- b	Return true if x is null, false otherwise
==	x y -- b	Return true if x and y have the same value
<>	x y -- b	Return true if x and y have different values

compile	X --	Perform the compilation action of x
execute	X --	Perform the runtime action of x
>compile	X --	Add the compilation action of x to current definition
>execute	X --	Add the runtime action of x to the current definition
forEachNext	p x -- n o b	Return next object into x (using p)
apply	r x -- ...	Apply r on each element of x
size	x -- n	Return object size
empty?	x -- b	Return true if x is empty
applyIf	p r x -- ...	Apply r on each element of x that respond true to p
detect	r e x -- o	Return first element of x that answer e to r
include?	e x -- b	Return true if x include e
conform?	p x -- b	Return true if all elements of x answer true to p
reduceWith	p r x -- y	Reduce x, applying p on each element then r
reduce	r x -- y	Reduce x, pushing each element and applying r
iapply	r x -- ...	Apply r after pushing each elemnt and its index
2apply	y r x -- ...	Apply r after pushing each element of y and x
maxFor	r x -- y	Return item of x with max value when r is performed
minFor	r x -- y	Return item of x with min value when r is performed
sum	x -- y	Return sum of all elements of x
>array	x -- arr	Return a new array with elements of x
>string	x -- s	Return a new string representing x
doesNotUnderstand	m x --	Send an exception : x does not understand method m
<<	stm x -- stm	Send x to stream stm
<<n	stm n x -- stm	Send x n times to stream stm

Null class (child of Object) :

new	Null -- null	Return null
<<	stm null -- stm	send null to stream stm

Exception class (child of Object) :

throw	aException --	Throw an exception
initialize	s x ex --	Initialize an exception with s as messsage and x
throw	m x Exception	Create and throw an exception
message	ex -- s	Return exception's message
object	ex -- x	Return exception's object.
<<	stm ex -- stm	Send exception on stream stm
log	ex --	Log exception on System.Err

Comparable property :

<=	*x y -- b*	*Required by the property*
>	x y -- b	
<	x y -- b	
>=	x y -- b	
min	x y -- z	Use #<=

max	x y -- z	Use #<=
between	x y z -- b	Return true if x <= y <= z

Integer class (child of Object), Comparable :

==	x y -- b	Compare two integers values
<=	x y -- b	Compare two integers values
+	x y -- x+y	Add two integers
-	x y -- x-y	Substract two integers
*	x y -- x*y	Multiply two inetgers
/	x y -- x/y	Divide two integers
mod	x y -- x mod y	Return remainder of two integers
/mod	x y -- z t	Return quotient and remainder of two integers
even?	x -- b	Return true is x is even
bitAnd	x y -- z	Return bit and of two integers
bitOr	x y -- z	Return bit or of two integers
bitXor	x y -- z	Return bit xor of two integers
bitLeft	n x -- y	Shift to left
bitRight	n x -- y	Shift to right
>float	n -- f	Convert integer to float
>digit	n -- m	Convert integer to digit
>prt	n -- aPrt	Convert integer to pointer
new	Integer -- 0	Just return 0
sq	n -- n*n	
odd?	n -- b	Return true if n is odd
>integer	n -- n	Return the receiver
neg	n -- m	Return opposite of n
abs	n -- m	Return absolute value of n
inv	n -- f	Return inverse of n (as float)
pow	x m -- n^m	Return x pow m (only if build includes Float support)
sqrt	n -- f	Return sqr of n (as float)
rand	n -- n	Return random integer between 1 and n (only if Float support)
each	r n --	Perform r on each integer between 1 and n
space?	c --	Return true if c is space (ie ' ' or '\t')
upper?	c -- b	Return true if c is uppercase
lower?	c -- b	Return true if c is lowercase
digit?	c -- b	Return true if c is a digit
letter?	c -- b	Return true if c is a letter
>upper	c -- c	Return upper character of c
>lower	c -- c	Return lower character of c
>digitOfBase	b c -- n	Return digit value of c in base b
>digit	c -- n	Return digit value of c in base 10
>char	n -- c	return character corresponding to n
<<wjp	stm w j p n -- stm	Send formatted value of n into stream stm
<<wj	stm w j n -- stm	Send formatted value of n into stream stm
<<w	stm w n -- stm	Send formatted value of n into stream stm
<<	stm n -- stm	Send n into stream stm

<<c	stm c -- strm	Send character c to stream stm (c is unicode value)
emit	c --	Send receiver as character c to System.Out

Runnable property :

execute	*x -- ...*	*Required by the property*
runnable?	x -- b	Return true if x is runnable
curry	x r -- bl	Return a block that executes "x r execute"
times	n r -- ...	Execute r n times
bench	r -- ... n	Return elapsed time for r (microseconds)

Block class (child of Object), Runnable :

new	Block -- aBlock	Create a new block
compile	aBlock --	Compile a block into current definition
execute	aBlock --	Execute a block

Collection class (child of Object), Comparable :

at	n coll -- x	Return value at position n, null if none
first	coll -- x	Return value at 1, null if none
second	coll -- x	Return value at 2, null if none
last	coll -- x	Return last value, null if none
==	coll1 coll2 -- b	Compare elements of 2 collections
<=	coll1 coll2 -- b	Compare elements of 2 collection
ketAt	key coll -- pair	Return pair into a collection with key as key
valueAt	key coll -- x	Return value into a collection with key
addAll	x coll --	Add all elements of x into coll
+	c1 c2 -- c3	Return a new map with elements of c1 and c2
zipWith	c1 r c2 -- c3	Return an array of results of applying r on each items
zip	c1 c2 -- c3	Return an array of pairs with elements of c1 and c2
>string	coll -- s	Return a new string with coll elements as characters
<<	stm coll -- stm	Send collection coll into stream stm

Interval class (child of Collection) :

initialize	b e s itv --	Create a new interval [b, e] wuth step s
size	itv --	Return interval size
at	n itv -- x	Return value at n
forEachNext	x itv -- y o b	Traverse an interval
seqFrom	from to -- itv	Return a new interval of integers [from, to]
seq	to -- itv	Return a new interval of integers [1, to]

Pair class (child of Collection) :

initialize	x y pair --	Create a new pair [x, y]

size	pair -- 2	Return 2
first	pair -- x	Return x of [x, y]
second	pair -- y	Return y of [x, y]
at	n pair --	Return value a n.
forEachNext	x pair -- y o b	Traverse a pair

Array class (child of Collection) :

size	array -- n	Return array size
at	n array -- x	Return value at index n (1-based)
put	n x array --	Set value at index n
add	x array --	Add value at end of the array
last	array -- x	Return last value
pop	array -- x	Remove last value from the array and return it
removeAt	n array -- x	Remove value at index n and return it
empty	array --	Empty the array
forEachNext	x a -- y o b	Retrieve next element into an array
newSize	n Array -- arr	Create a new array with n as initial allocation
allocSize	n Array -- arr	Create a allocated array with n as initial allocation
new	Array -- arr	Create a new array with default initial allocation
alloc	Array -- arr	Create a allocated array with default initial allocation
newWith	n x Array -- arr	Create a new array with n times x as elements
init	n r Array -- arr	Create a new array with results of r performed n times
arrayWith	x1...xn n -- arr	Create a new arry with n values on the stack
array?	x -- b	Return true if x if an array

Symbol class (child of Object) :

size	aSymbol -- n	Return symbol size (in characters)
new	s Symbol -- sym	Create/return a symbol corresponding to string s
<<	stm sym -- stm	send symbol sym to stream stm

Buffer class (child of Collection) :

byteSize	buf -- n	Return buffer size (bytes)
byteAt	n buf -- x	Return byte at index n (1-based)
bytePut	n x buf --	Set value at index n
utf8At	n buf --	Return UTF8 value at index n
forEachNext	x buf -- y o b	Retrieve next byte into a buffer
evaluate	buf -- ...	Evaluate a buffer
==	buf1 buf2 -- b	Compare two buffers
<=	buf1 buf2 -- b	Compare two buffers
hashValue	buf -- n	Return hash value
newSize	n Buffer -- buf	Create a new buffer with n as initial allocation
new	Buffer -- buf	Create a new buffer with default initial allocation
allocSize	n Buffer -- buf	Create an allocated buffer with n as allocation

alloc	Buffer -- buf	Create an allocated buffer with default allocation
size	buf -- n	Return buffer size in bytes (same as byteSize)
at	i buf -- byte	Return byte at index i (1-based), same as byteAt
put	i byte buf --	set byte at index i (1-based), same as bytePut
>array	buf -- arr	Return an array of bytes corresponding to buf

String class (child of Buffer) :

add	c s --	Add character c at the end of the string
removeAt	n s -- c	Remove and return character at index n (1-based)
removeLast	s --	Remove ans return last character
empty	s --	Empty the string
addChar	c s --	Add character c at the end of the string
addSymbol	sym s --	Add symbol name
addIntegerFormat	n ... s --	Add integer with format options
addFloatFormat	f ... s --	Add float with format options
addBufferFormat	buf s --	Add buffer
forEachNext	x s -- y o b	Traverse a string, character by character
load	s --	Load a file with s as file name
>symbol	s -- sym	Create/return symbol corresponding to s
>float	s -- f	Return float value of s
newWith	n c String -- s	Create a new string, by adding c n times
init	n r String -- s	Create a new string with result of r n times
size	s -- n	Return number of UTF8 characters into the string
at	i n -- c	Return UFT8 character at position i (1 based).
>integerOfBase	b s -- n	Return integer represented by s in base b
>integer	s -- n | null	Convert a string to an integer
>number	s -- n | f | null	Convert a string to an integer or a float
>string	s -- s	Return a new mutable string with same value
<<wj	stm w j s -- stm	Send formatted string to a stream
<<w	stm w s -- stm	Send formatted string to a stream
<<	stm s -- stm	Send string s to stream stm
type	s --	Send string s to stream System.Out

Word class (child of Object)

find	s Word -- aWord	Return word which name is s, null if none
forget	aWord --	Forget aWord. It can't be found anymore.
name	aWord -- sym	Return symbol corresponding to the name of a word
<<	stm w -- stm	Send word w to stream stm
process	w -- ...	Compile or execute a word according to STATE
interpret	b w -- ...	Default interpretation : call literal

Function class (child of Word), Runnable

_compile	f --	Compile f into current def without optimizations
execute	f --	execute function f

compile	f --	Compile f into current def with optimizations
interpret	b f --	If immediate, execute, else handle params and process

Method class (child of Word), Runnable

_compileTOS	*m --*	*Compile m with receiver into TOS*
_compileSuper	*m --*	*Compile call to m at upper level*
_compile	m --	Compile m into current definition without optimization
execute	x m --	execute method m with x as the receiver
compile	m --	Compile m into current definition with optimizations
interpret	m --	Handle params if any and process the method

Constant class (child of Word)

new	x s Constant --	Create a new constant
value	aConstant -- x	Return constant value
interpret	b cst --	Interpret the constant

Variable class (child of Word)

new	s Variable --	Create a new task variable
at	aVariable -- x	Return variable value
put	x aVariable --	Set variable value
interpret	b var --	Process #at on the variable

Class class (child of Word)

_findAtt	*s cl -- n*	*Find class attribute*
_addAtt	*s cl --*	*Add attribute to class*
new	cl s Class --	Create a new class with parent cl
implement	m cl -- impl	Return implement of method m for class cl
classImplement	m cl -- impl	Return class implement of method m for class cl
is:	cl "name" --	Add property "name" to class cl
new:	cl Class "name" --	Create a new class, child of cl, with name "name"

Property class (child of Word)

_findAtt	*s pr -- n*	*Find property attribute*
_addAtt	*s pr --*	*Add attribute to property*
new	cl s Class --	Create a new class with parent cl
requires:	pr "name" --	Add method "name" as requirement for property pr
new:	Propery "name" --	Create a new property with name "name"

Alias class (child of Word)

new	x s Alias --	Create a new alias for x

Package class (child of Word)

_imported	pkg --	Set package pkg as imported (forgotten)
new	x s Alias --	Create a new alias for x
load	pack --	Load package pack (signature and all source files)

Resource class (child of Object)

open?	res -- b	Return true if the resource is open
close	res --	Close the resource

File class (child of Object) :

File.BINARY	-- n	Binary mode when opening
File.TEXT	-- n	Text mode when opening
File.UTF8	-- n	UFT8 mode when opening
File.READ	-- n	Read access when creating file
File.WRITE	-- n	Write access when creating file
File.APPEND	-- n	Append access when creating file
File.BEGIN	-- n	File position (beginning)
File.CURRENT	-- n	File position (current position)
File.END	-- n	File position (end of file)

stats	s File -- c m s	Return created/modified/size of a file with name s
open	access aFile --	Open a file with access (File.READ, File.WRITE, File.APPEND).
open?	aFile -- b	Return true if a file is open.
end?	aFile -- b	Return true if end of fiel is reached
close	aFile --	Close a file
position	aFile -- n	Return file position
reposition	org off aFile --	Set file position with origin and offset
flush	aFile --	Flush a file
readLineWith	buf aFile -- buf	Read a line from aFile and store it into buf
readWith	n buf aFile -- buf	Read n bytes from aFile and store them into buf
>>	aFile -- c	Read a byte or a char from aFile (according to mode)
addChar	c f --	Add character c to aFile
addSymbol	sym s --	Add symbol name
addIntegerFormat	n ... s --	Add integer with format options
addFloatFormat	f ... s --	Add float with format options
addBufferFormat	buf f --	Add buf to file f
exist?	s File -- b	Return true if file with name s exists
initialize	name mode f --	Init a file object with name and mode
newMode	name mod File -- f	Create a new file object with name and mode (File.TEXT, File.BIN or File.UTF8).

new	name File -- f	Create a new UTF8 file
name	f -- s	Return file name
size	s File -- n	Return file size with name s
size	f -- n	Return file size
created	s File -- n	Return timestamp
created	f -- n	Return timestamp
modified	s File -- n	Return timestamp
modified	f -- n	Return timestamp
read	n f -- buf	Read n bytes and return a new buffer
readCharsWith	n buf f -- buf	Read n characters and append them to buf
readChars	n f -- str	Read n characters and return them into a string
readLine	f -- str	Read a line from file f
forEachNext	f -- x str true	Return next line from a file
add	c f --	Add char c to file f

19.2 Optional Float words

Float numbers (and Float class) are available if Oforth is built with Float support (default).

Constants

PI	-- f	PI constant
E	-- f	exp(1) constant
LN2	-- f	ln(2) constant
LN10	-- f	ln(10) constant

Float class (child of Object), Comparable :

==	x y -- b	Compare two floats values
<=	x y -- b	Compare two floats values
+	x y -- x+y	Add two floats
-	x y -- x-y	Substract two floats
*	x y -- x*y	Multiply two floats
/	x y -- x/y	Divide two float
sqrt	f -- f	Square root
powf	f g -- h	Pow
ln	f -- g	Ln
cos	f -- g	Cos
sin	f -- g	sin
tan	f -- g	tan
acos	f -- g	Arccos
asin	f -- g	arcsin
atan	f -- g	arctan
>integer	f -- n	Convert float to integer
new	Float -- 0.0	Just return 0.0
>float	f -- f	Return the receiver
neg	f -- -f	

abs	f -- f	Return absolute value of f
inv	f -- f	Return inverse of f
exp	f -- f	Return exp(f)
log	f -- f	Return log(f)
rand	Float -- f	Return random number between 0 and 1 (excluded).
<<wjp	stm w j p f -- stm	Send formatted value of f into stream stm
<<wj	stm w j f -- stm	Send formatted value of f into stream stm
<<w	stm w f -- stm	Send formatted value of f into stream stm

19.3 Optional dynamic libraries/procedure words and ffi package

Dynamic libraries and procedures words are available if Oforth is built with dynamic library support (default).

This is required to load ffi package.

DynLib class (child of Word), available according to Oforth built options

new	DynLib -- aDynLib	Create a new dynamic librarie
<<	stm dynlib -- stm	Send a dynamic library to steam stm

DynProc class (child of Word), available according to Oforth built options

new	DynProc -- aDynProc	Create a new dynamic procedure
execute	aDynProc --	execute a dynamic procedure
compile	aDynProc --	Compile a dynamic procedure into current definition
interpret	aDynProc --	Interpret a dynamic procedure.
<<	stm dynproc -- stm	Send a dynamic procedure to steam stm

Functions added

extlib	DynLib "name" -- aDynLib	Create a new dynamic library with C params
extwlib	DynLib "name" -- aDunLib	Create a new dynamic library with pascal params
extern:	lib np ret sname "name" --	Create a new word with name "name" corresponding to fucntion sname into dynamic library lib

19.4 Optional TCP words

TCP words are available if Oforth is built with TCP support (default).

It is required to load tcp package :

```
import: tcp
```

Constants

TCPSocket.IPV4	-- n	Mode for TCPSocketServer creation
TCPSocket.IPV6	-- n	Mode for TCPSocketServer creation
TCPSocket.IPALL	-- n	Mode for TCPSocketServer creation

TCPSocket class (child of Object), available according to Oforth built options

initialize	TCPSocket -- sock	Create a new dynamic librarie
select	mode sock --	
close	sock --	Close the socket.
port	aSock -- n	Return socket's port

TCPSocketServer class (child of TCPSocket)

initialize	TCPSocketServer - s	Create a new socket for server side
_acceptTimeout	sc n sock --	Accept and populate new connection with timeout n
newBacklog	port backlog mode TCPSocketServer -- s	Create a new socket that will accept connexions on port with mode IPV4, IPV6 or IPALL
new	port TCPSocketServer -- s	Create a new socket that will accept connexions on port with mode IPALL and 10 as backlog
acceptTimeout	n sock -- sockclient	Accept a new connexion.
accept	sock -- s true | err false	Block until the next connexion. Return the socket and true (or error and false).
<<	stm sock-- stm	Send a socket to steam stm

TCPSocketClient (child of TCPSocket)

initialize	host port s	Initialize the socket for host and remote port
remotePort	sock -- port	Return the remote port
host	sock -- host	Return the host
connectMode	mode n sock -- true | errorfalse	Try to connect the socket with mode (IPV4, V6, ALL) and timeout. Return true or error and false.
connect	sock -- true | error false	Try to connect the socket with mode IPALL. Wait unitl connexion is ok or error.
receiveWithTimeout	s ns nt sock -- n true | err false	Receive ns bytes from remote host with timeout nt and append them to buffer or string s. Return number of bytes read and true or error and false.
receiveWith	s ns sock -- n true | err false	Same as receiveWithTimeout but block until read is ok or error is detected.
receiveTimeout	ns nt sock -- s	Same as receiveWithTiemout but returns a new

	true | err false	string with the bytes read.
receive	ns sock -- s true | err false	Block until receiving ns byte. Return a new string and true or err and false.
sendTimeout	b nt sock -- true | err false	Send buffer b to the socket with nt as timeout. Return true or err and false
snd	b sock | true | err false	Block until buffer b is sent to the socket. Return true or err and false
<<	stm sock -- stm	Send socket to stream stm

TCPConnection (child of Object)

See TCPConnection.of file into packs/tcp directory

TCPRequest (child of Object)

See TCPRequest.of file into packs/tcp directory

TCPServer (child of Object)

See TCPServer.of file into packs/tcp directory

20 Packages

Available packages when downloading Oforth are :

chars	Additional words that work with characters
collect	Additionnal collections : Cycle, Hash, Set, Stack
console	Console object (loaded automatically if --i command line option)
date	Date class and methods
emitter	Objects that handle events asynchronously
ffi	Foreign Function Interface : call C functions from Oforth.
json	Json objects
libc	Load standard library to LIBC according to current OS
logger	Asynchronous logging file
mapping	Higher order words to map collections
math	Additionnal math words
quicksort	quicksort
reflect	Adds some reflection words for OO metamodel
resource	Resources and Channels
tcp	TCP support and TCP server

For packages not described below yet, you can check files into the pack directory

21 Package console

21.1 Console class

This package implements console words. It is loaded automatically when Oforth is launched with --i command line option.

A console is a resource and a stream : objects can be sent to the console using << family words.

If standard streams are not redirected when Oforth is launched, a console object is created and available as a constant :

```
System.Console
```

The console allows to wrap standard input, output and error. Unlike standard streams, the console is a resource, so a task waiting for the console will enter in WAIT state until the console is ready (a key is available,). See the "Concurrent programming" chapter for more information.

Constants defined in the console package give a name to the values returned when an extended key is pressed: (K-CHAR-MASK K-CTLR-MASK K-ALT-MASK, ...)

Methods and functions defined for a console allow to read from and write to the console :

For instance, if you want to wait for a key during 2 seconds :

```
: wait2s              \ -- x | null
  2000000 System.Console receiveTimeout ;
```

If you want to read an integer (or null is the string is not an integer) :

```
System.Console accept >integer
```

Currently, cursor handling is not supported; this will be added in a later version.

21.2 Reference

Constants

K_CHAR-MASK	-- n	Mask for character (to apply to an extended key)
K_CTRL-MASK	-- n	Mask for ctrl key pressed (to apply to an extended key)
K_ALT-MASK	-- n	Mask for alt key pressed (to apply to an extended key)
K-PRIOR	-- n	Prior key
K-NEXT	-- n	Next key

K-END	-- n	End key
K-HOME	-- n	Home key
K-LEFT	-- n	Left key
K-UP	-- n	Up key
K-RIGHT	-- n	Right key
K-DOWN	-- n	Down key
K-ESCAPE	-- n	Esc key
K-INSERT	-- n	Insert key
K-DELETE	-- n	Del key
K-F1	-- n	F1 key
K-F2	-- n	F2 key
K-F3	-- n	F3 key
K-F4	-- n	F4 key
K-F5	-- n	F5 key
K-F6	-- n	F6 key
K-F7	-- n	F7 key
K-F8	-- n	F8 key
K-F9	-- n	F9 key
K-F10	-- n	F10 key
K-F11	-- n	F11 key
K-F12	-- n	F12 key

Variables

PROMPT	-- s	Word used to output prompt.
SHOWSTACK	-- b	Used by #.show to toggle showing stack after each cmd

Functions

.l	--	Print the stack current on one line
.show	--	Toggle stack printing after each command
defaultPrompt	--	Print "ok" if runtime more, "->" if compile mode
repl	--	Read Eval Print Loop used when --i command line option

ConsoleHistory class (child of Object)

cmd	ch -- s	Command
next	ch -- ch | null	Next console history
prev	ch -- ch | null	Previous console history

Console class (child of Resource)

select	mode cons -- b	Check if the console is ready for input or output
receiveTimeout	n cons -- exk	Wait for an extended key from the console.
new	Console -- cs	Return System.Console
cmd	cs -- s	Return command line currently edited
history	cs -- ch	Return first command history

addChar	c cs --	Console as a stream : add char
addIntegerFormat	n cs --	Console as a stream : add integer
addFloatFormat	f cs --	Console as a stream : add float
addBufferFormat	c cs --	Console as a stream : add buffer
ekey	cs -- n	Wait until an extended key is pressed and return it
key	cs -- n	Wait until a key is pressed and return it
ekey?	cs -- b	Return true if a key is available
flush	cs --	Flush the console input
accept	cs -- s	Read a string from console until end of line
fill	s cs --	Fill the console with string s.
cursorLeft	cs --	If possible, position cursor to the left
cursorRight	cs --	If possible, posotion cursor to the right
readCmd	cs --	Read a new command
repl		

22 Package mapping

mapping packing adds mapping features to various classes :

Object class :

newMap	n x -- coll	Create a new map corresponding to x
map	r x -- map	Create a map with results of applying r to x elts
mapIf	p r x -- map	Map with only elements of x that answer true to p
expandTo	arr x --	Add x to arr. If x is a collection add each item
expand	x -- arr	Recursively expand x to a new array
mapFlat	r x -- arr	Like map, but add all elements into the same array
mapParallel	r coll -- coll	execute r on each items of the collection and return results. Each computation is done in a separate task.

Collection class :

filter	p coll --	Return a new map with elements that answer true to p
-	c1 c2 -- c3	Return a new map with elements of c1 not in c2
zipAll	x ... x n -- arr	zip n collections and returns an array
groupWith	r x -- arr	Returns an array grouping adjacent items that return the same value
group	x -- arr	Group all identical adjacent items of x
extract	i j c -- c	Extract elements from i to j into a new map
del	i j x -- y	New collection after removing items from i to j
left	i c -- c	Extract i first elements of c into a new map
right	i c -- c	Extract i last elements of c into a new map
splitBy	n x -- y	Split a collection grouping elements by n
transpose	[[]] -- [[]]	Transpose an array of arrays.
indexOfFromTo	x i j y -- n	Return index of x into y between i and j indexes
indexOfFrom	x i y -- n	Return index of x into y between i and the end
indexOf	x y -- n	Return index of x into y
lastIndexOfFromTo	x i j y -- n	Same as indexOfFromTo but last index
lastIndexOfFrom	x i j y -- n	Same as indexOfFrom but last index
allAt?	x n y --	Return true if all items of y are at index n into x
indexOfAllFrom	x i y --	Return index of all items of x into y from i
indexOfAll	x i y --	Return index of all items of x into y

String class (child of Buffer) :

newMap	n s -- s	Return a new map for a string
split	c s -- [s]	Return array of s splited using character c
extractAndStrip	i j s -- s1	Extract and strip spaces of s between i and j

strip	s -- s1	Remove leading and trailing whitespaces from s
wordsWith	c s -- [s]	Return array of words separated by c into s
words	s -- [s]	Return array of words separated by space into s
unwordWith	c [s] -- s	Return a string with all strings separated by c
unwords	[s] -- s	Return a string with all strings separated by space

23 Package json

json package loads implementation of Json objects.

23.1 Json class

Json is a Array's subclass. It implements json objects.

#{, #: and #} words allow to create jsons.

```
{             \ --          : Beginning of JSON object
:             \ --          : Member separator
}             \ -- aJson    : Creates the JSON.
```

With those words, the Oforth interpreter can parse JSON objects from the input stream (or compile them into a definition) as any other object.

For instance :

```
{ "abcd" : 12, "cde" : { $f : [ 1, 2, 3 ], $g : null }, "fgh" : 1.2 } .s
```

As those words can parse JSON, a string containing a JSON can by parsed as a JSON :

```
{ "abcd" : 12, "cde" : { $f : [ 1, 2, 3 ], $g : null }, "fgh" : 1.2 }
dup >string execute .s == .s
```

23.2 Words added by this package

Compilation and interpretation

{	--	Beginning of a Json object
}	--	End of Json object

Object class

<<json	stm x -- stm	Send a Json representation of x to stream stm

Symbol class

<<json	stm x -- stm	Send a Json representation of x to stream stm

Collection class

<<json	stm x -- stm	Send a Json representation of x to stream stm

String class

<<json	stm x -- stm	Send a Json representation of x to stream stm

Json class (child of Array) :

<<	stm j -- stm	Send a Json to stream stm
<<json	stm x -- stm	Send a Json representation of x to stream stm
>string	json -- str	Convert a json to a string

24 Ans Forth / Oforth cross reference

This chapter lists most Ans Forth words and Oforth counterpart, with commentaries if necessary.

Creating words and compilation

:	:	
	method:	Creates a method implementation
	virtual:	Creates a virtual method implementation
	classMethod:	Creates a class method implementation
	classVirtual:	Creates a class virtual implementation
;	;	
IMMEDIATE	immediate	
{ } {: :}	( )	Oforth uses ( ) to declare local parameters { : , and } are used for Json objects syntax {: and :} are ... not defined
|	|	Local variables are declared between two |
[		Not defined in Oforth. [and] are used for arrays
]		Not defined in Oforth. [and] are used for arrays
[: :NONAME	#[	#[...] can be used inside or outside definitions.
;]	]	
VARIABLE 2VARIABLE FVARIABLE		Not defined. Global variables are not permitted in Oforth.
USER	tvar:	Value of a tvar is by task.
VALUE 2VALUE	tvar:	Value of a tvar is by task.
TO	to ->	to is used for setting tvars -> is used for setting locals
CONSTANT 2CONSTANT FCONSTANT	const:	
ALIAS	alias:	
CREATE DOES> >BODY		Not defined. Replaced by classes definition.
' [']	#	Return an object on the stack, not an execution token. # can be used inside or outside definitions.
FIND FIND-NAME	Word find	
NAME>STRING	name	
EXECUTE	execute	Oforth has no xt, so execute is not defined
COMPILE,	compile	
EVALUATE	evaluate	
QUIT ABORT		Not defined
ABORT"	abort	

THROW	throw	throw need an Exception object
CATCH		Not defined
TRY ... ENDTRY	try: [...] when: [...]	
DEFER		Not defined
DEFER@ ACTION-OF		Not defined
IS DEFER!		Not defined
POSTPONE	postpone	
LITERAL 2LITERAL FLITERAL SLITERAL	literal	literal is not immediate.
STATE	STATE	
SOURCE >IN	SOURCE SOURCE-TYPE SOURCE-INDEX	Not implemented
PARSE PARSE-NAME	parse-token	
REFILL	fill	Fill console input buffer with a string value.
INCLUDE	load	
 REQUIRED	import: use:	
SEE	see	
BYE	bye	
 FORTH	oforth	oforth is the default package. Unlike forth, its name just pushes it on the stack.
ENVIRONMENT ?		Not defined

Stack manipulation :

DUP FDUP	dup	
DROP FDROP	drop	
OVER FOVER	over	
SWAP FSWAP	swap	
ROT FROT	rot	
-ROT	-rot	
NIP FNIP	nip	
TUCK	tuck	
2DUP	2dup	
2DROP	2drop	
PICK FPICK	pick	
DEPTH FDEPTH	.depth	
.S	.s	
ROLL		Not defined
2NIP		Not defined
2OVER		Not defined

2SWAP		Not defined
2ROT		Not defined
?DUP		Not defined

Arithmetic :

+ F+ D+ M+	+	
- F- D-	-	
* F* M*	*	
/ F/	/	
MOD	mod	
/MOD UM/MOD FM/MOD	/mod	
NEGATE FNEGATE DNEGATE	neg	
ABS DABS FABS	abs	
SQRT FSQRT	sqrt	
MAX DMAX FMAX	max	
MIN DMIN FMIN	min	
1+	1+	
1-	1-	
2* D2* F2* 2/ D2/ F2/		Not defined.
S>D D>S		Not defined. Oforth integers have arbitrary precision
D>F S>F	>float	
F>D F>S	>integer	
F**	powf	
FEXP	exp	
FLN	ln	
FLOG	log	
1/F	inv	
FSIN	sin	
FCOS	cos	
FTAN	tan	
FASIN	asin	
FACOS	acos	
FATAN	atan	
FSINH	sinh	Need import: math
FCOSH	cosh	Need import: math
FTANH	tanh	Need import: math
FASINH	asinh	Need import: math
FACOSH	acosh	Need import: math
FATANH	atanh	Need import: math
F~	==	

Conditions :

TRUE	true	
FALSE	false	
AND	and bitAnd	
OR	or bitOr	
XOR	xor bitXor	
INVERT NOT	not	not returns true if false and false otherwise
LSHIFT	bitLeft	Be careful, in Oforth stack effect is inverted (method of integer)
RSHIFT	bitRight	Be careful, in Oforth stack effect is inverted (method of integer)
= D=	=	
F~ COMPARE	==	Test objects values
< U< D< DU< F<	<	
<= F<=	<=	
<>	<>	Carefull, Oforth compare by value.
> U> F>	>	
>= F>=	>=	
0= 0<> 0> 0< 0<> D0< D0= F0< F0=		Not defined

Control flow :

\ ( ) --	\ --	In Oforth, () are used to declare parameters
IF	if ifTrue: [ifFalse: [ifZero: [ifNull: [ifNotNull:[if=: [	
THEN REPEAT UNTIL	then]	
ELSE	else else: [	if .... else ... then ifTrue: [...] else: [...]
BEGIN AGAIN	begin ... again	
BEGIN ... WHILE ... REPEAT	while (...) [...]	
BEGIN ... UNTIL	begin ... until	
AHEAD	ahead	
CS-PICK CS-ROLL	CS> >CS	
CASE OF ENDCASE		Not defined

DO ?DO +DO U+DO -DO LOOP +LOOP	loop: i [...] for: i [...]	
I J		Not defined. Oforth loops require a local declaration.
RECURSE		Not defined, use function name to recurse
LEAVE	break	
	continue	
EXIT	return	
>R R@ >R RDROP 2>R 2R> 2R@ N>R 2RDROP	->	Not defined. Locals handles values on the return stack.
[IF]	IFTRUE: [IFFALSE: [	
[ENDIF]	]	
[DEFINED]	DEFINED:	

I/O :

. U. ." .ID D. UD. F.	.	. applies to all objects
EMIT	emit	
EKEY	ekey	ekey is done on a console object
KEY	key	key is done on a console object
BASE HEX DECIMAL	0x 0b	base hex decimal are not defined
.R		
<# <<# # #S #> #>> HOLD	<< <<w <<wj <<wjp	
BL	BL	
SPACE		Not defined.
CR	printcr	
ACCEPT	accept	accept is done on a console object
MS	sleep	
TIME&DATE	System.local Time System.time	

Characters and Strings :

CHAR [CHAR]	'	No need to have a space after '
S"	"	No need to have a space after "
C, C@	at put	
TYPE	type	
COMPARE	==	
SEARCH	indexOfAll	
FILL	<<cn	
-TRAILING	strip	

S>NUMBER? S>UNUMBER? >NUMBER	asInteger	
>FLOAT	asFloat	
COUNT		Not defined
SUBSTITUTE	replaceAll	

Memory :

@ ! , F@ F! +! C@ C! 2@ 2! SF@ SF! DF@ DF!		Not defined : you can't make direct access to memory
	@ :=	Access to an object attributes
	at: put:	Access to a structure fields
CHARS CHAR+		Not defined
CELLS FLOATS	cells	
CELL+ FLOAT+		Not defined
ALLOT		Not defined
ALLOCATE	alloc	Create a new object manually handled
FREE	free	Free an object created by alloc
	new	Create a new object handled by the garbage collector
RESIZE		Not defined : if resize is possible (Array, ...) resize automatically when needed
ALIGN ALIGNED FALIGNED SFALIGN DFALIGN		Not defined. In Oforth all objects are aligned.
HERE C, F, , 2,		Not defined. Either new or alloc.
ADDRESS-UNIT-BITS	CELLSIZE	
MOVE ERASE CMOVE CMOVE>		Not defined
BUFFER:		Not defined. Can't define a mutable word. Use : MemBuffer newSize to create a buffer on the heap
BEGIN-STRUCTURE	struct	
END-STRUCTURE	;	
FIELD:		Not defined : structure fields are declared without field:

Files :

R/O	File.READ	
W/O	File.WRITE	
R/W	File.APPEND	
BIN	File.BINARY File.TEXT	

	File.UTF8	
CREATE-FILE	newMode new	
OPEN-FILE	open	
CLOSE-FILE	close	
READ-FILE	>> readWith read readCharsWith readChars	
READ-LINE	readLineWith readLine	
WRITE-LINE	<<	
WRITE-FILE	add addChar << <<w <<wj <<wjp	
FLUSH-FILE	flush	
FILE-STATUS	exists	
FILE-POSITION	position	
REPOSITION-FILE	setPosition	
FILE-SIZE	size	
STDIN	System.Console	
STDOUT	System.Out	
STDERR	System.Err	

Vocabularies and word lists :

Word lists words are not used in Oforth. Oforth implements packages instead and they are too different from lists or vocabularies to have a cross reference.