GOALS

1) Take best part of imperative and functional languages and destil it down to the most important mechanisms.

2) Make it execute extremely quickly.

3) Easy to learn for new programmers, easy to enjoy for experienced programmers.

4) Have explict and sold way to do each thing - no need to reinvent basics or invent policies.

5) Robustness.

6) Composable: build huge programs (millions of lines of code).

Floyd Script Reference

Floyd Script is a fast and modern C-like program language that makes writing correct programs simpler and faster than any other programming language.

It's focus is composability, minimalism and proven programming techniques, like pure functions and immutability.

It as a unique concept to handle mutation of data, control of the outside world (files, network, UI) and controlling time.

Functions and classes are pure. A pure function can only call pure functions. Mutable data can exist locally inside a function, but never leak out of the function.

This delegates all mutation / time to the top levels of your program. Here you need to call non-pure functions to affect the world around the program. You must tag your functions with "nonpure" to be able to call other nonpure functions.

```
But programming is NOT math!
```

This makes those functions a risk. Have as little nonpure code as possible. Try to not have any conditional code inside nonpure code - this makes testing easier. Put all logic into the pure functions.

Here is hello world, as expected:

```
print("Hello, World!");
```

Trying the floyd REPL:

```
marcus$ ./floyd
Floyd 0.3 MIT.
Type "help", "copyright" or "license" for more informations!
>>>print("Hello, world!");
Hello, world!
```

Executing a floyd program:

```
marcus$ floyd my_program.floyd file1.txt file2.txt
```

This will runt the program my_program.floyd and call it's main(string args) with the arguments ["file1.txt", "file2.text"].

COMPOSABILITY

Features that break composabiity is limited and carefully controlled in the language. Things like threading and synchronisation, error responses, file handling, optimization choices. Normal Floyd code lives free, outside of time and the real world and can be used in many contexts.

To easy composability, the basic wiring between libraries and subsystems is built-in and standardized - this like common datatypes, error propagation between libraries, logging and asserts, memory handling and streaming data. All built-in and composable.

BASIC TYPES

These are the primitive data types built into the language itself. The goals is that all the basics you need are already there in the language. This makes it easy to start making useful programs, you don't need to chose or build the basics. It allows composability since all libraries can rely on these types and communicate bewteen themselves using them. Reduces need for custom types and glue code.

- **int** Same as int64
- **bool true** or **false**
- **string** built-in string type. 8bit pure (supports embedded nulls). Use for machine strings, basic UI. Not localizable.
- **float** 32-bit floating point number

- **function** A function value. Functions can be Floyd functions or C functions. They are callable.

COMPOSITE TYPES

These are composites and collections of other types. - **struct** like C struct or class or tuple. - **vector** an a continous array of elements addressed via indexes.

CORE TYPE FEATURES

These are features built into every type: integer, string, struct, collections etc.

- **a = b** This true-deep copies the value b to the new name a.
- **a == b** Compares the two values, deeply.
- **a != b** Derivated of a == b.
- **a < b** Tests all member data in the order they appear in the struct.
- **a <= b**, **a > b**, **a >= b** These are derivated of a < b

TRUE DEEP

True-deep is a Floyd term that means that all values and sub-values are always considered in operations in any type of nesting of structs and values and collections. This includes equality checks or assignment, for example.

The order of the members inside the struct (or collection) is important for sorting since those are done member by member from top to bottom.

VALUES, VARIABLES AND CONSTANTS

All "variables" aka values are immutable.

- Function arguments
- Local function variables
- Member variables of structs etc.

Floyd is statically typed, which means every variable only supports a specific type of value.

When defining a variable you can often skip telling which type it is, since the type can be deduced by the Floyd compiler.

Explicit

```
    int x = 10;
```

Implicit

```
    y = 11;
```

Example:

```
int main(){
    a = "hello";
    a = "goodbye";  //  Runtime error - you cannot change variable a.
    return 3;
}
```

You can use "mutable" to make a local variable changeable.

```
int main(){
    mutable a = "hello";
    a = "goodbye";  //  Changes variable a to "goodbye".
    return 3;
}
```

GLOBAL SCOPE

Here you normally define functions, structs and global constants. The global scope can have almost any statement and they execute at program start, before main() is called. You don't even need a main function if you don't want it.

main() This function is called by the host when program starts. You get the input arguments from the outside world (command line arguments etc) and you can return an integer to the outside.

FUNCTIONS

Functions in Floyd are by default pure, or referential transparent. This means they can only read their input arguments and constants, never read or modify anything else that can change. It's not possible to call a function that returns different values at different times, like get_time().

While a function executes, it percieves the outside world to stand still.

Functions always return exactly one value. Use a struct to return more values.

A function without return value usually makes no sense since function cannot have side effects. Possible uses would be logging, asserting or throwing exceptions.

Example function definitions:

```
    int f1(string x){
        return 3;
    }

    int f2(int a, int b){
        return 5
    }

    int f3(){
        return 100;
    }

    string f4(string x, bool y){
        return "<" + x + ">";
    }
```

Function types:

```
    bool (string, float)
```

This is a function that takes a function value as argument:

```
    int f5(bool (string, string))
```

This is a function that returns a function value:

```
    bool (string, string) f5(int x)
```

EXPRESSIONS

Reference: http://www.tutorialspoint.com/cprogramming/c_operators.htm Comparisons are true-deep - they consider all members and also member structs and collections.

Arithmetic Operators

How to add and combine values:

```
+   Addition - adds two operands: "a = b + c", "a = b + c + d"
-   Subtracts second operand from the first. "a = b - c", "a = b - c - d"
*   Multiplies both operands: "a = b * c", "a = b * c * d"
/   Divides numerator by de-numerator: "a = b / c", "a = b / c / d"
%   Modulus Operator and remainder of after an integer division: "a = b / c", "a =
 b / c / d"
```

Relational Operators

Used to compare two values. The result is true or false:

```
**a == b**              true if a and b have the same value
    a != b              true if a and b have different values
    a > b               true if the value of a is greater than the value of b
    a < b               true if the value of a is smaller than the value of b
    a >= b
    a <= b
```

Logical Operators

Used to compare two values. The result is true or false:

```
    a && b
    a || b
```

Conditional Operator

```
    condition ? a : b
```

When the condition is true, this entire expression has the value of a. Else it has the value of b. Condition, a and b can all be complex expressions, with function calls etc.

```
    bool is_polite(string x){
        return x == "hello" ? "polite" : "rude"
    }
    assert(is_polity("hiya!") == false);
    assert(is_polity("hello") == true);
```

IF - THEN - ELSE -- STATEMENT

This is a normal if-elseif-else feature, like in most languages. Brackets are required always.

```
        if (s == "one"){
            return 1;
        }
```

You can add an else body like this:

```
        if(s == "one"){
            return 1;
        }
        else{
            return -1;
        }
```

Else-if lets you avoid big nested if-else statements and do this:

```
        if(s == "one"){
            return 1;
        }
        else if(s == "two"){
            return 2;
        }
        else{
            return -1;
        }
```

In each body you can write any statements. There is no "break" keyword.

FOR LOOPS

For-loops are used to evaluate its body many times, with a range of input values. The entire condition expression is evaluated *before* the first time the body is called. This means the program already decided the number of loops to run before running the first time.

Closed range that starts with 1 and ends with 5.:

```
    for (index in 1...5) {
        print(index);
    }
```

Open range that starts with 1 and ends with 59:

```
    for (tickMark in 0..<60) {
    }
```

You can use expressions for range:

```
    for (tickMark in a..<string.size()) {
    }
```

Above snippet simulates the for loop of the C language but it works a little differently. There is always

exactly ONE loop variable and it is defined and inited in the first section, checked in the condition section and incremented / updated in the third section. It must be the same symbol. The result is the equivalent to

```
    b = 3
    { a = 0; print(a + b); }
    { a = 1; print(a + b); }
    { a = 2; print(a + b); }
```

The loop is expanded before the first time the body is called. There is no way to have any other kind of condition expression, that relies on the result of the body etc.

- init: must be an assignment statement for variable X.
- condition: Must be a bool expression with only variable X. Is executed before each time body is executed.
- advance-statement: must be an assignement statement to X.
- body: this is required and must have brackets. Brackets can be empty, like "{}".

WHILE LOOPS

```
    while (my_array[a] != 3){
    }
```

- condition: executed each time before body is executed. If the condition is false initially, then zero loops will run.

STRING

This is a pure 8-bit string type. It is immutable. You can compare with other strings, copy it using = and so on. There is a small kit of functions for changing and processing strings.

The encoding of the characters in the string is undefined. You can put 7-bit ASCII in them. Or UTF-8 etc. You can also use them as fast arrays of bytes.

This is the strings contents you enter into the script code. They look like "Hello". You cannot use any escape characters, like in the C-language.

```
a = "Hello, world!";
```

You can copy string using =. All comparison expressions work, like a == b, a < b, a >= b, a != b etc.

You can access a random character in the string, using its integer position.

```
a = "Hello"[1];
assert(a == "e")
```

Notice 1: You cannot modify the string using [], only read. Use update() to change a character. Notice 2: Floyd returns the character in a new string.

You can append to strings together using the + operation.

```
a = "Hello" + ", world!"
assert(a == "Hello, world!"
```

CORE FUNCTIONS

- **print()**: prints a string to the default output of the app.
- **update()**: changes one character of the string and returns a new string.
- **size()**: returns the number of characters in the string, as an integer.
- **find()**: searches from left to right after a substring and returns its index or -1
- **push_back()**: appends a character or string to the right side of the string.
- **subset**: extracts a range of characters from the string, as specified by start and end indexes. aka substr()
- **replace()**: replaces a range of a string with another string. Can also be used to erase or insert.

VECTOR

A vector is a collection of values where you lookup the values using an index between 0 and (vector size - 1). The positions in the vector are called "elements". The elements are ordered. Finding an element at a position uses constant time. In other languages vectors are called "arrays" or even "lists".

You can make a new vector and specify its elements directly, like this:

```
a = [ 1, 2, 3];
```

You can also calculate elements:

```
a = [ calc_pi(4), 2.1, calc_bounds() ];
```

You can put ANY type of value into a vector: integers, floats, strings, structs, other vectors etc. But all elements must be the same type inside a specific vector.

You can copy vectors using =. All comparison expressions work, like a == b, a < b, a >= b, a != b etc. Comparisons will compare each element of the two vectors.

This lets you access a random element in the vector, using its integer position.

```
a = [10, 20, 30][1];
assert(a == 20)
```

Notice: You cannot modify the vector using [], only read. Use update() to change an element.

You can append to vector together using the + operation.

```
a = [ 10, 20, 30 ] + [ 40, 50 ];
assert(a == [ 10, 20, 30, 40, 50 ]);
```

- **print()**: prints a vector to the default output of the app.
- **update()**: changes one element of the vector and returns a new vector.
- **size()**: returns the number of elements in the vector, as an integer.
- **find()**: searches from left to right after a subvector and returns its index or -1
- **push_back()**: appends an element to the right side of the vector.
- **subset**: extracts a range of elements from the vector, as specified by start and end indexes.
- **replace()**: replaces a range of a vector with another vector. Can also be used to erase or insert.

DICTIONARY

A collection that maps a key to a value. Unsorted. Like a C++ map.

You make a new dictionary and specify its elements like this:

```
[string: int] a = ["red": 0, "blue": 100,"green": 255];
```

or shorter:

```
b = ["red": 0, "blue": 100,"green": 255];
```

Dictionaries always use string-keys. When you specify the type of dictionary you must always include "string".

```
struct test {
    [string: int] _my_dict;
}
```

You can put any type of value into the dictionary (but not mix inside the same dictionary).

Use [] to look up elements using a key. It throws an exception is the key not found. Use exists() first.

You copy dictionaries using = and all comparison expressions work.

- **print()**: prints a vector to the default output of the app.

- **update()**: changes one element of the dictionary and returns a new dictionary
- **size()**: returns the number of elements in the dictionary, as an integer.
- **exists()**: checks to see if the dictionary holds a specific key
- **erase()**: erase a specific key from the dictionar and returns a new dictionary

STRUCTs - THE BASICS

Structs are the central building block for composing data in Floyd. They are used in place of structs and classes in other programming languages. Structs are always values and immutable. They are still fast and compact: behind the curtains copied structs shares state between them, even when partially modified.

Automatic features of every struct:

- constructor -- this is the only function that can create a value of the struct. It always requires every struct member, in the order they are listed in the struct definition. Make explicit function that makes making values more convenient.
- destructor -- will destroy the value including member values, when no longer needed. There are no custom destructors.
- Comparison operators: == != < > <= >= (this allows sorting too)
- Reading member values.
- Modify member values

There is no concept of pointers or references or shared structs so there are no problems with aliasing or side effects caused by several clients modifying the same struct.

This all makes simple structs extremely simple to create and use.

Not possible:

- You cannot make constructors. There is only *one* way to initialize the members, via the constructor - which always takes *all* members
- There is no way to directly initialize a member when defining the struct.
- There is no way to have several different constructors, instead create explicit functions like make_square().
- If you want a default constructor, implement one yourself: `rect make_zero_rect(){ return rect(0, 0); }`.
- There is no aliasing of structs -- changing a struct is always invisible to all other code that has copies of that struct.

Example:

```
    //  Make simple, ready-for use struct.
    struct rect {
        float width;
        float height;
    };

    //  Try the new struct:

    a = rect(0, 3);
    assert(a.width == 0);
    assert(a.height == 3);

    b = rect(0, 3);
    c = rect(1, 3);

    asset(a == a)
    asset(a == b)
    asset(a != c)
    asset(c > a)
```

A simple struct works almost like a collection with fixed number of named elements. It is only possible to make new instances by specifying every member or copying / modifying an existing one.

UPDATE()

b = update(a, member, value);

```
    //  Make simple, ready-for use struct.
    struct rect {
        float width;
        float height;
    };

    a = rect(0, 3);

    //  Nothing happens! Setting width to 100 returns us a new rect but we we don'
t keep it.
    update(a,"width", 100)
    assert(a.width == 0)

    //  Modifying a member creates a new instance, we assign it to b
    b = update(a,"width", 100)

    //  Now we have the original, unmodified a and the new, updated b.
    assert(a.width == 0)
    assert(b.width == 100)
```

This works with nested values too:

```
    //  Define an image-struct that holds some stuff, including a pixel struct.
    struct image { string name; rect size; };

    a = image("Cat image.png", rect(512, 256));

    assert(a.size.width == 512);

    //  Update the width-member inside the image's size-member. The result is a br
and new image, b!
    b = update(a, "size.width", 100);
    assert(a.size.width == 512);
    assert(b.size.width == 100);
```

COMMENTS AND DOCUMENTATION

Use comments to write documentation, notes or explanations in the code. Comments are not executed or compiled -- they are only for humans. You often use the comment features to disable / hide code from the compiler.

Two types of comments:

You can wrap many lines with "/.../" to make a big section of documentation or disable many lines of code. You can nest comments, for example wrap a lot of code with existing comments with /* ... */ to disable it.

```
/*  This is a comment */
```

Everything between // and newline is a comment:

```
//  This is an end-of line comment
a = "hello"; // This is an end of line comment.
```

FUNCTION TOOLBOX

These are built in primitves you can always rely on being available.

print()

This outputs one line of text to the default output of the application. It can print any type of value. If you want to compose output of many parts you need to convert them to strings and add them. Also works with types, like a struct-type.

```
print(any)
```

Example	Result
print(3)	3
print("shark")	shark
print("Number four: " + to_string(4))	Number four: 4
print(int)	int
print([int])	[int]
print([string: float])	[string:float]
print(["a": 1])	string:int

assert()

Used this to check your code for programming errors, and check the inputs of your function for miss use by its callers.

```
assert(bool)
```

If the expression evaluates to false, the program will log to the output, then be aborted via an exception.

to_string()

Converts its input to a string. This works with all types of values. It also works with types, which is useful for debugging.

```
string to_string(any)
```

You often use this function to convert numbers to strings.

get*time*of_day()

Returns the computer's realtime clock, expressed in the number of milliseconds since system start. Useful to measure program execution. Sample get*time*of_day() before and after execution and compare them to see duration.

```
int get_time_of_day()
```

update()

This is how you modify a field of a struct, an element in a vector or string or a dictionary. It replaces the value of the specified key and returns a completely new object. The original object (struct, vector etc) is unchanged.

```
obj_b = update(obj_a, key, new_value);
```

Type	Example	Result
string	update("hello", 3, "x")	"helxo"
vector	update([1,2,3,4], 2, 33]	[1,2,33,4]
dict	update(["a": 1, "b": 2, "c": 3], "a", 11]	["a":11,"b":2,"c":3]
struct	update(pixel,"red", 123)	pixel(123,---,---)

For dictionaries it can be used to add completely new elements too.

Type	Example	Result
dict	update(["a": 1], "b", 2]	["a":1,"b":2]

size()

Returns the size of a collection -- the number of elements.

```
int size(obj)
```

Type	Example	Result
string	size("hello")	5
vector	size([1,2,3,4])	4
dict	size(["a": 1, "b": 2])	2
struct		

find()

Searched for a value in a colletion and returns its index or -1.

```
int find(obj, value)
```

Type	Example	Result
string	find("hello", "o")	4
string	find("hello", "x")	-1
vector	find([10,20,30,40], 30)	2
dict		
struct		

exists()

Checks if the dictionary has an element with this key. Returns true or false.

```
bool exists(dict, string key)
```

erase()

Erase an element in a dictionary, as specified using its key.

```
erase(dict, string key)
```

push_back()

Appends an element to the end of a collection. A new collecton is returned, the original unaffected.

Type	Example	Result
string	push_back("hello", "3")	hello3
string	push_back("hello", "!?")	hello!?
vector	push_back([1,2,3], 7)	[1,2,3,7]
dict		
struct		

subset()

This returns a range of elements from the collection.

```
string subset(string a, int start, int end)
vector subset(vector a, int start, int end)
```

start: 0 or larger. If it is larger than the collection, it will be clipped to the size. end: 0 or larger. If it is larger than the collection, it will be clipped to the size.

Type	Example	Result
string	subset("hello", 2, 4)	ll
vector	subset([10,20,30,40, 1, 3)	[20,30]
dict		
struct		

replace()

Replaces a range of a collection with the contents of another collection.

```
string replace(string a, int start, int end, string new)
vector replace(vector a, int start, int end, vector new)
```

Type	Example	Result
string	replace("hello", 0, 2, "bori")	borillo
vector	replace([1,2,3,4,5], 1,3, 8)	[1,8,4,5]
dict		
struct		

Notice: if you use an empty collection for *new*, you will actually erase the range. Notice: by specifying the same index in *start* and *length* you will **insert** the new collection into the existing collection.

ENCODING

Floyd script files are always utf-8 files with no BOM.