daScript Reference Manual

Release 0.1 alpha

Anton Yudintsev

Nov 16, 2019

CONTENTS

CHAPTER

ONE

INTRODUCTION

daScript is high-performance statically strong typed scripting language, designed to be high-performance as embeddable "scripting" language for real-time applications (like games).

daScript offers a wide range of features like strong static typing, generic programming with iterative type inference, Ruby-like blocks, semantic indenting, native machine types, ahead-of-time "compilation" to C++ and fast and simplified bindings to C++ program.

It's philsophy is build around modified Zen of Python.

- *Performance counts.*
- *But not at the cost of safety.*
- *Unless is explicitly unsafe to be performant.*
- *Readability counts.*
- *Explicit is better than implicit.*
- *Simple is better than complex.*
- *Complex is better than complicated.*
- *Flat is better than nested.*

daScript is supposed to work as "host data processor". While it is technically possible to mantain persistent state within script context (with a certain option set), daScript is designed to transform your host (C++) data/implement scripted behaviours.

In a certain sense, it is pure functional - i.e. all persistent state is out of scope of scripting context, and script's state is temporal by it's nature. So, the memory model and managment of persistent state becoming are or responsibility of application, and leads to extremely simple and fast memory model in the daScript itself.

1.1 Performance.

In a real world scenarios, it's interpretation is 10+ times faster than LuaJIT without JIT (and can be even faster than LuaJIT with JIT). Which is probably even more important for embedded scripting languages, it's (both-ways) interop with C++ is extremely fast, order of magnitude faster than such in most popular scripting languages. Fast "call from C++ to daScript" allows to use daScript for simple stored procedures, and makes it's ECS/Data Oriented Design friendly language. Fast call C++ from daScript allows to write performant scripts which are processing host (C++) data, or relies on bound host (C++) functions.

It also allows Ahead-of-Time compilation, which is not only possible on all platforms (unlike JIT), but also always faster/not-slower (JIT is known to sometimes slow down scripts).

daScript already has implemented AoT (C++ transpiler) which produces code more or less similar with C++11 performance.

Table with performance comparisons on a syntetic samples.

1.2 How it looks?

Mandatory fibonacci samples

```
def fibR(n)
   if (n < 2)
       return n
   else
       return fibR(n - 1) + fibR(n - 2)

def fibI(n)
   var last = 0
   var cur = 1
   for i in range(0, n - 1)
       let tmp = cur
       cur += last
       last = tmp
   return cur
```

1.3 Generic programming and type system

Although above sample seem to be dynamically typed, it is actually a generic programming. The actual instance of fibI/fibR function is strong typed and basically is just accepting and returning int. This is similar to templates in C++ (although C++ is not atrong-typed language) or ML. Generic programming in daScript allows very powerful compile-time type reflection mechanisms, significally simplifying writing optimal and clear code. Unlike C++ with it's SFINAE, you can use comon conditions (if) in order to change instance of function depending on type info of arguments. Consider the following example:

```
def setSomeField(var obj; val)
    if typeinfo(has_field<someField> obj)
          obj.someField = val
```

this function will set someField in provided argument *if* it is a struct with someField member.

For more info see *Generic programming*).

1.4 Features

It's (not)full list of features includes:

- strong typing
- Ruby-like blocks
- tables
- arrays
- string-builder

- native (C++ friendly) interop
- generics
- semantic indenting
- ECS-friendly interop
- easy-to-extend type system
- etc.

CHAPTER

TWO

THE LANGUAGE

2.1 Lexical Structure

2.1.1 Identifiers

Identifiers start with an alphabetic character (and not the symbol '_') followed by any number of alphabetic characters, '_' or digits ([0-9]). daScript is a case sensitive language meaning that the lowercase and uppercase representation of the same alphabetic character are considered different characters. For instance, "foo", "Foo" and "fOo" are treated as 3 distinct identifiers.

2.1.2 Keywords

The following words are reserved and cannot be used as identifiers:

struct	def	recover	panic	let	var
continue	break	finally	new	delete	enum
enum	for	while	if	else	elif
options	null	deref	return	typeinfo	this
cast	upcast	reinterpret	operator	in	scope
require	true	false	typedef	with	override
type	expect	auto	include	where	addr
assert	invoke				

Keywords and types are covered in detail later in this document.

2.1.3 Operators

daScript recognizes the following operators:

`!`	`:=`	`??`	`==`	`&`	`>=`	`<=`	`>`
`->`	`+`	`+=`	`-`	`-=`	`/`	`/=`	`*`
`*=`	`|>`	`<|`	`++`	`--`	`<-`	`=`	`&`
`^`	`|`	`~`	`>>`	`.`	`?.`	`??`	

2.1.4 Other tokens

Other significant tokens are:

{	}	[	]	.	:
::	'	;	"	]]	[[

2.1.5 Literals

daScript accepts integer numbers, unsigned integers, floating and double point numbers and string literals.

`34`	Integer number(base 10)
`0xFF00A120`	Unsigned Integer number(base 16)
`0753`	Integer number(base 8)
`'a'`	Integer number
`1.52`	Floating point number
`1.e2`	Floating point number
`1.e-2`	Floating point number
`1.52d`	Double point number
`1.e2d`	Double point number
`1.e-2d`	Double point number
`"I'm a string"`	String
`@" I'm a multiline verbatim string "`	String

Pesudo BNF

```
IntegerLiteral          ::=   [1-9][0-9]* | '0x' [0-9A-Fa-f]+ | ''' [.]+ ''' | 0[0-7]+
FloatLiteral            ::=   [0-9]+ '.' [0-9]+
FloatLiteral            ::=   [0-9]+ '.' 'e'|'E' '+'|'-' [0-9]+
StringLiteral           ::=   '"'[.]* '"'
VerbatimStringLiteral   ::=   '@'"'[.]* '"'
```

2.1.6 Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to embed annotations in the code. The compiler treats them as white space.

A comment can be `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C:

```
/*
This is
a multiline comment.
This lines will be ignored by the compiler.
*/
```

A comment can also be `//` (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. It is commonly called a *"single-line comment."*:

```
// This is a single line comment. This line will be ignored by the compiler.
```

2.1.7 Semantic indenting

daScript follows semantic indenting (much like Python). That means, that logical blocks are arranged with a same indenting, and if control statement requires nesting of block (such as body of function, block, if, for, etc.) it have to be indented one step more. Indenting step is part of options of program, so it is either 2, 4 or 8, but always the same for whole file. Default indenting is 4, and can be globally overriden per project.

2.2 Values and Data types

daScript is a strong statically typed language and all variables do have a type. daScript's basic POD (plain old data) data types are:

```
int, uint, float, bool, double, int64, uint64
int2, int3, int4, uint2, uint3, uint4, float2, float3, float4
```

All PODs are represented with machine register/word. All PODs will be passed to function argument by value.

daScript's storage types (which can't be manipulated with, but can be used as storage type within structs or otherwise):

```
int8, uint8, int16, uint16 - 8/16-bits signed and unsigned integers
```

daScript's other types are:

```
string, das_string, struct, pointers, references, block, lambda, function pointer,␣
↪array, table
```

all daScript's types are initialized with zero memory by default.

2.2.1 Integer

An Integer represents a 32-bit (un)signed number:

```
let a = 123    // decimal, integer
let u = 123u   // decimal, unsigned integer
let h = 0x0012 // hexadecimal, unsigned integer
let o = 075    // octal, unsigned integer

let a = int2(123, 124)    // two integers type
let u = uint2(123u, 124u) // two unsigned integer type
```

2.2.2 Float

A float represents a 32-bit floating point number:

```
let a = 1.0
let b = 0.234
let a = float2(1.0, 2.0)
```

2.2.3 Bool

Bool is a double-valued (Boolean) data type. Its literals are `true` and `false`. A bool value expresses the validity of a condition (tells whether the condition is true or false):

```
let a = true
let b = false
```

All conditions (if, elif, while) works only on bool type.

2.2.4 String

Strings are an immutable sequence of characters. In order to modify a string is it necessary create a new one.

daScript's strings are similar to strings in C or C++. They are delimited by quotation marks(") and can contain escape sequences (\t, \a, \b, \n, \r, \v, \f, \\, \", \', \0, \x<hh>, \u<hhhh> and \U<hhhhhhhh>):

```
let a = "I'm a string\n"
```

Strings type can be thought of as 'pointer to actual string' type, like 'const char *' in C language. As such they will be passed to function argument by value (but this value is just reference to immutable string in memory).

das_string - is mutable string, which content can be changed. It is simply builtin handled type, i.e., std string bound to daScript. As such, it passed as reference.

2.2.5 Table

Tables are associative containers implemented as a set of key/value pairs:

```
var tab: table<string; int>
tab["10"] = 10
tab["20"] = 20
tab["some"] = 10
tab["some"] = 20 // replaces the value for 'some' key
```

(see *Tables*).

2.2.6 Array

Arrays are simple sequence of objects. There are static arrays (fixed size), and dynamic array (container, size is dynamic) and index always starts from 0:

```
var a = [[int[4] 1; 2; 3; 4]] // fixed size of array is 4, and content is [1, 2, 3, 4]
var b: array<string>          // empty dynamic array
push(b,"some")                // now it is 1 element of "some"
```

(see *Arrays*).

2.2.7 Struct

Structs are record of data of other types (including structs), similair to C language. All structs (as well as other non-POD types, except strings) will be passed by reference

(see *Structs*).

2.2.8 Tuple

Tuple are anonimous record of data of other types (including structs), similair to C++ std::tuple. All tuples (as well as other non-POD types, except strings) will be passed by reference

(see *Tuples*).

2.2.9 Function

Functions are similar to those in most other languages:

```
def twice(a: int): int
    return a + a
```

However, there are generic (templated) functions, which will be 'instantiated' during compilation of call to them.

```
def twice(a)
    return a + a

let f = twice(1.0) // 2.0 float
let i = twice(1)   // 2 int
```

(see *Functions*).

2.2.10 Reference

References are types that 'references' (points) some other data.

```
def twice(a: int&)
    a = a + a
var a = 1
twice(a) // a value is now 2
```

All structs are always passed to functions arguments as references.

2.2.11 Pointers

Pointers are types that 'references' (points) some other data, but can be null (points to nothing). In order to work with actual value, one need to dereference using deref builtin function them or use safe navigation operators. deref will panic, if null pointer is passed to it. Pointers can be created using new operator, or with C++ environment.

```
def twice(a: int&)
    a = a + a
def twicePointer(a: int?)
    twice(deref(a))

struct Foo
    x: int

def getX(foo: Foo?)  // it returns either foo.x or -1, if foo is null
   return foo?.x ?? -1
```

All structs are always passed to functions arguments as references.

2.2.12 Iterators

Iterator is an entity which can be travered, and associated data retrived. It is similar to C++ iterators.

(see *Iterators*).

2.3 Statements

A daScript program is a simple sequence of statements:

```
stats := stat [';'|'\n'] stats
```

Statements in daScript are comparable to the C-Family languages (C/C++, Java, C#, etc.): assignment, function calls, program flow control structures etc. plus some custom statement like block, struct and array initializers (will be covered in detail later in this document). Statements can be separated with a new line or ';'.

2.3.1 Visibility Block

```
visibility_block := indent (stat)* unindent
visibility_block := '{' (stat)* '}'
```

A sequence of statements delimited by indenting or curly brackets ({ }) is called visibility_block.

2.3.2 Control Flow Statements

daScript implements the most common control flow statements: `if, while, for`

true and false

daScript has a strong boolean type (bool). Only boolean type expression can be part of condition in control statement.

if/elif/else statement

```
stat:= 'if' exp '\n' visibility_block (['elif' exp '\n' visibility_block])*  ['else'
↪'\n' visibility_block]
```

Conditionally execute a statement depending on the result of an expression:

```
if a > b
    a = b
elif a < b
    b = a
else
    print("equal")
```

while statement

```
stat:= 'while' exp '\n' indent stat
```

Executes a statement while the condition is true:

```
while(true)
{
    if (a<0)
        break
    a--
}
```

2.3.3 Ranged Loops

for

```
stat := 'for' iterator 'in' [rangeexp] '\n' visibility_block
```

Executes a loop body statement for every element/iterator in expression, in sequenced order:

```
for i in range(0, 10)
    print("{i}")       // will print numbers from 0 to 9

// or

let arr: array<int>
resize(arr, 4)
for i in arr
    print("{i}")       // will print content of array from first element to last

// or

var a: array<int>
var b: int[10]
resize(a, 4)
for l, r in a, b
    print("{l}=={r}")  // will print content of a array and first 4 elements of array␣
↪b

// or

var a: table<string; int>
for k, v in keys(tab), values(tab)
    print("{k}:{v}")   // will print content of table, in form key:value
```

You can implement your own iteratable types, by implementing iterator.

2.3.4 break

```
stat := 'break'
```

The break statement terminates the execution of a loop (for or while);

2.3.5 continue

```
stat := 'continue'
```

The continue operator jumps to the next iteration of the loop skipping the execution of the following statements.

2.3.6 return

```
stat := return [exp]
stat := return <- exp
```

The return statement terminates the execution of the current function and optionally returns the result of an expression. If the expression is omitted the function will return nothing, return types is assumed to be void. You can't return mismatching types from same function (i.e., all returns should return value of same type), and if function return type is explicit, return expression should return that same type. Example:

```
def foo(a: bool)
    if a
      return 1
    else
      return 0.f  // error, different return type

def bar(a: bool): int
    if a
      return 1
    else
      return 0.f  // error, mismatching return type

def foobar(a)
    return a  // return type will be same as argument type
```

'return <- exp' syntax is for move-on-return

```
def make_array
    var a: array<int>
    a.resize(10)  // fill with something
    return <- a   // return will return

let a <- make_array() //create array filled with make_array
```

2.3.7 Finally statement

```
stat := finally visibility-block
```

Finally declares a block which will be executed once for any block (including control statements). Finally block can't contain break/continue/return statements. This is to require some expression to be run after 'all done'. Consider

```
def test(a: array<int>; b: int)
    for x in a
        if x == b
            return 10
     return -1
finally
```

(continues on next page)

(continued from previous page)

```
    print("print anyway")

def test(a: array<int>; b: int)
    for x in a
        if x == b
            print("we found {x}")
            break
    finally
        print("we print this anyway")
```

Finally can be, for example used for resource de-allocation.

2.3.8 Local variables declaration

```
initz := id [:type] [= exp]
ro_stat := 'let' initz
rw_stat := 'var' initz
```

Local variables can be declared at any point in the function; they exist between their declaration to the end of the visibility block where they have been declared. 'let' declares read only variable, 'var' declares mutable (read-writer) variable.

2.3.9 Function declaration

```
stat := 'def' id ['(' args ')'] [':' type ] visibility_block

arg_decl = [var] id (',' id)* [':' type]
args := (arg_decl)*
```

declares a new function. Examples:

```
def hello
    print("hello")

def hello(): bool
    print("hello")
    return false

def printVar(i: int)
    print("{i}")

def printVarRef(i: int&)
    print("{i}")

def setVar(var i: int&)
    i = i + 2
```

2.3.10 try/recover

```
stat := 'try' stat 'recover' visibility-block
```

The try statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a panoc statement. The catch clause provides the exception-handling code.

It is important to understand, that try/recover is not a correct error handling code. Much like in GO lang, this is really invalid situation which should not happen in production environemnt normally. Examples of potential exceptions are: dereferencing null pointer, indexing array out of bounds, etc.

2.3.11 panic

```
stat := 'panic' '(' [string-exp] ')'
```

Panics in runtime. String expression will be output to log.

2.3.12 global variables

```
stat := 'let' '\n' indent id '=' expression
```

Declares a constant global variable. This variable will be inited once during initialization of script (or each time when script init is manually called).

2.3.13 enum

```
enumerations := ( 'id' ) '\n'
stat := 'enum' id indent enumerations unindent
```

Declares an enumeration (see *Constants & Enumerations*).

2.3.14 Expression statement

```
stat := exp
```

In daScript every expression is also allowed as statement, if so, the result of the expression is thrown away.

2.4 Expressions

2.4.1 Assignment

```
exp := exp '=' exp
exp := exp '<-' exp
exp := exp ':=' exp
```

daScript implements 3 kind of assignment: the normal assignment(=):

```
a = 10
```

“move” assignment

```
var b = new Foo
var a: Foo?
a <- b
```

move assignment nullifies source (b) It's main purpose is to correctly move ownership, and optimize copying if you don't need source for heay types (such as arrays, tables). Some external handled types can be non assignable, but still moveable;

var

"clone" assignment

```
a := b
```

Clone assignment is syntactic sugar for calling clone(var a: auto&; b: auto&) if it exists or basic assignment for POD types. It is also implemented for das_string, array and table types, and creates 'deep' copy.

Some external handled types can be non assignable, but still copyable;

2.4.2 Operators

?: Operator

```
exp := exp_cond '?' exp1 ':' exp2
```

conditionally evaluate an expression depending on the result of an expression.

?? Null-coalescing operator

```
exp := exp1 '??' exp2
```

Conditionally evaluate an expression2 depending on the result of an expression1. Given code is equivalent to:

```
exp := (exp1 '!=' null) '?' deref(exp1) ':' exp2
```

It evaluates expressions until the first non-null value (just like || operators for the first 'true' one).

Operator precendence is also follows C# design, so that ?? has lower priority than ||

?. - Null-propagation operator

```
exp := value '?.' key
```

If value is not null exists, return deref of field 'key' for struct, else return null.

```
struct TestObjectFooNative
    fooData : int

struct TestObjectBarNative
    fooPtr: TestObjectFooNative?
    barData: float

def test
```

(continues on next page)

(continued from previous page)

```
    var a: TestObjectFooNative?
    var b: TestObjectBarNative?
    var idummy: int
    var fdummy: float
    a?.fooData ?? idummy = 1 // will return reference to idummy, since a is null
    assert(idummy == 1)

    a = new TestObjectFooNative
    a?.fooData ?? idummy = 2 // will return reference to a.fooData, since a is now␣
↪not null
    assert(a.fooData == 2 & idummy == 1)

    b = new TestObjectBarNative
    b?.fooPtr?.fooData ?? idummy = 3 // will return reference to idummy, since while␣
↪b is not null, but b.?barData is still null
    assert(idummy == 3)

    b.fooPtr <- a
    b?.fooPtr?.fooData ?? idummy = 4 // will return reference to b.fooPtr.fooData
    assert(b.fooPtr.fooData == 4 & idummy == 3)
```

Arithmetic

```
exp:= 'exp' op 'exp'
```

daScript supports the standard arithmetic operators `+, -, *, / and %`. Other than that is also supports compact operators (`+=, -=, *=, /=, %=, ^=, <<=, >>=`) and increment and decrement operators(++ and –);:

```
a += 2
// is the same as writing
a = a + 2
x++
// is the same as writing
x = x + 1
```

All operators work normally with (u)int* and float* and double.

Relational

```
exp:= 'exp' op 'exp'
```

Relational operators in daScript are : `==, <, <=, <, <=, !=`

These operators return true if the expression is false and a value different than true if the expression is true.

Logical

```
exp := exp op exp
exp := '!' exp
```

Logical operators in daScript are : `&, |, !`

The operator `&` (logical and) returns false if its first argument is false, otherwise returns its second argument. The operator `|` (logical or) returns its first argument if is different than false, otherwise returns the second argument.

It is important to understand, that it won't nesessariy 'evaluates' all arguments.

The '!' operator will return false if the given value to negate was true or false otherwise.

Bitwise Operators

```
exp:= 'exp' op 'exp'
exp := '~' exp
```

daScript supports the standard C-like bitwise operators `&, |, ^, ~, <<, >>`. Those operators only work on (unsigned) integer values.

Pipe Operators

```
exp:= 'exp' |> 'exp'
exp:= 'exp' <| 'exp'
```

daScript supports pipe operators. Pipe operator is similar to 'call' expression with other expresion is first argument.

```
def addX(a, b)
    assert(b == 2 | b == 3)
    return a + b
def test
    let t = 12 |> addX(2) |> addX(3)
    assert(t == 17)
    return true
```

```
def addOne(a)
    return a + 1

def test
    let t =  addOne <| 2
    assert(t == 3)
```

Operators precedence

post++ post-- . -> ?.	highest
|> <|	
- + ~ ! ++ --	
??	
/ * %	
+ -	
<< >> <<< >>>	
< <= > >=	
== !=	
&	
^	
|	
? :	
+= = -= /= *= %= &= |= ^= <<= >>= <- <<<= >>>=	...
',' comma	lowest

2.4.3 Struct Constructor

```
struct Foo
  x: int = 1
  y: int = 2

let fExplicit = [[Foo x = 13, y = 11]] // x = 13, y = 11
let fPartial  = [[Foo x = 13]]         // x = 13, y = 0
```

(see *Structs*).

2.4.4 Array Contructor

```
exp := '[['type[] [explist] ']]'
```

Creates a new fixed size array:

```
let a = [[int[] 1, 2]]     // creates array of two elements
let a = [[int[2] 1, 2]]    // creates array of two elements
let a = [[int[2] 1, 2, 3]] // error, too many initializers
```

Arrays can be also created with array comprehensions:

```
let q <- [[ for x in range(0, 10); x * x ]]
```

2.5 Tables

Tables are associative containers implemented as a set of key/value pairs:

```
var tab: table<string; int>
tab["10"] = 10
tab["20"] = 20
tab["some"] = 10
tab["some"] = 20  // replaces the value for 'some' key
```

List of relevant builtin functions: clear, key_exists, find, erase. for safety, find doesn't return anything. Instead it works with block as last argument. It can be worked with rbpipe operator

```
var tab: table<string; int>
tab["some"] = 10
find(tab,"some") <| $(var pValue: int?)
    if pValue != null
        assert(deref(pValue) == 10)
```

It is done so, because otherwise find would have to return pointer to value, which would continue to point 'somewhere', even if data is deleted. Consider this hypothetical find and the following example

```
var tab: table<string; int>
tab["some"] = 10
var v: int? = find(tab,"some")
assert(v)       // not null!
tab |> clear()
deref(v) = 10  // where we will write this 10? UB and segfault!
```

So, if you just want to check for existence of key in table, use key_exists(table, key).

Tables (as well as arrays, structs, and handled types) will be passed to functions by reference only.

Tables can not be assigned, only cloned or moved.

```
def clone_table(var a, b: table<string, int>)
  a := b      // a is not deep copy of b
  clone(a, b) // same as above
  a = b       // error

def move_table(var a, b: table<string, int>)
  a <- b  //a is no points to same data as b, and b is empty.
```

Table key can be not only string, but any other 'workhorse' type as well.

Tables can be constructed inline

```
let tab = {{ "one"=>1; "two"=>2 }}
```

Which is syntax sugar for

```
let tab : table<string;int> = to_table_move([[tuple<string;int>[2] "one"=>1; "two"=>
↪2]])
```

2.6 Arrays

An array is a sequence of values indexed by a integer number from 0 to the size of the array minus 1. Arrays elements can be obtained through their index.

```
var a = [[int[4] 1; 2; 3; 4]] // fixed size of array is 4, and content is [1, 2, 3, 4]
assert(a[0] == 1)

var b: array<int>
b.push(1)
assert(b[0] == 1)
```

There are static arrays (of fixed size, allocated on stack), and dynamic array (size is dynamic, allocated on heap)

```
var a = [[int[4] 1; 2; 3; 4]] // fixed size of array is 4, and content is [1, 2, 3, 4]
var b: array<string>          // empty dynamic array
push(b, "some")               // now it is 1 element of "some"
b |> push("some")             // same as above line
```

Resizing, insertion, deletion of dynamic arrays and arrays elements is done through a set of standard functions (see *built-in functions*).

List of relevant builtin functions: push, emplace, resize, erase, length, clear, capacity.

Arrays (as well as tables, structs, and handled types) will be passed to functions by reference only.

Arrays can not be assigned, only cloned or moved.

```
def clone_array(var a, b: array<string>)
  a := b      // a is not a deep copy of b
  clone(a, b) // same as above

def move_array(var a, b: array<string>)
  a <- b  // a is no points to same data as b, and b is empty.
```

Arrays can be constructed inline

```
let arr = [[auto 1.; 2.; 3.; 4.5]]
```

Which infers to

```
let arr : float[4] = [[float[4] 1.; 2.; 3.; 4.5]]
```

Dynamic arrays can also be constructed inline:

```
let arr = [{auto "one"; "two"; "three"}]
```

Which is syntatic equivalent to:

```
let arr : array<string> = to_array_move([[string[4] "one"; "two"; "three"]])
```

If only one elmeent is specified, local data construction is that element:

```
let i1 = [[int 1]]      // same is the line bellow
let i2 = 1
```

To create an array of unspecified type, use [] syntax:

```
let ai1 = [[int[] 1]]   // actually [[int[1] 1]]
let ai2 = [[autop[] 1]  // same as above
```

2.7 Functions

Functions Pointers are first class values like integer or strings and can be stored in table slots, local variables, arrays and passed as function parameters. Functions themselves are declarations (much like in C++ language).

2.7.1 Function declaration

Functions are similar to those in most other typed languages:

```
def twice(a: int): int
    return a+a
```

Function calls

You can call function by using it's name and passing all arguments (with a possible ommision of default arguments)

```
def foo(a, b: int)
    return a + b

def bar
    foo(1, 2) // a = 1, b = 2
```

Named Arguments Function call

You can also call function by using it's name and passing all arguments with explicit names (with a possible ommision of default arguments)

```
def foo(a, b: int)
    return a + b

def bar
    foo([a = 1, b = 2])  // same as foo(1, 2)
```

Named arguments should be still in same order, i.e., this is error

```
def bar
    foo([b = 1, a = 2])  // error
```

Named arguments call is to increase readibility of callee code and ensure correctness of refactor of existing functions. It also allows defaulting not just last arguments.

Function pointer

Function pointer to function can be obtained using @ and called using invoke builtin:

```
let fn = @twice
let t = invoke(fn, 1)  // t = 2
```

Generic functions

However, there are generic (templated) functions, which will be 'instantiated' during compilation of call to them:

```
def twice(a)
    return a + a

let f = twice(1.0)  // 2.0 float
let i = twice(1)    // 2 int
```

Generic functions allows writing in daScript like it is dynamic-type language, similar to Python or Lua, but still enjoy the performances and robustness of Strong-static typing.

Generic function address can not be obtained.

Function overloading

Function names can be overloaded, if function argument types are different

```
def twice(a: int)
    print("int")
    return a + a
def twice(a: float)
    print("float")
    return a + a

let i = twice(1)    // prints "int"
let f = twice(1.0)  // prints "float"
```

Declaring function with a same list of arguments is compile time error.

Function arguments of same type can be delcared as follows

```
def foo(a, b: int)
    return a + b

//same as above
def foo(a: int; b: int)
    return a + b
```

Completely empty function (without arguments) can be also declared as

```
def foo
    print("foo")

//same as above
def foo()
    print("foo")
```

Default Paramaters

daScript's functions can have default parameters.

A function with default parameters is declared as follows:

```
def test(a, b: int; c: int = 1; d: int = 1)
    return a + b + c + d
```

when the function *test* is invoked and the parameter c or d are not specified, the compiler will generate call with default value to the unspecified parameter. A default parameter can be any valid compile-time const daScript expression. The expression is evaluated at compile-time.

it is valid to declare function with not only last arguments with default values

```
def test(c: int = 1; d: int = 1; a, b: int) // valid!
    return a + b + c + d
```

Such syntax can only be used with named arguments call

```
test(2, 3)           // invalid call, a,b parameters are missing
test([a = 2, b = 3]) // valid call
```

And can still be combined with overloading

```
def test(c: int = 1; d: int = 1; a, b: int)
    return a + b + c + d
def test(a, b: int) // now test(2, 3) is valid call
    return test([a = a, b = b])
```

2.7.2 OOP-style calls

There are no methods or function member of structs in daScript. However, code can be easily written "OOP style" by using rpipe operator '|>':

```
struct Foo
    x, y: int = 0

def setXY(var this: Foo; x, y: int)
    this.x = x
    this.y = y
...
var foo:Foo
foo |> setXY(10, 11)    // this is syntactic sugar for setXY(foo, 10, 11)
setXY(foo, 10, 11)      // exactly same as above line
```

(see *Structs*).

2.7.3 Lambda Functions

. . . to be written. . .

2.7.4 Tail Recursion

Tail recursion is a method for partially transforming a recursion in a program into an iteration: it applies when the recursive calls in a function are the last executed statements in that function (just before the return).

Currently daScript doesn't support tail recursion.

2.8 Structs

daScript implements a struct mechanism similar to languages like C/C++, Java, C#, etc. however because it differs in several aspects. Structs are first class objects like integer or strings and can be stored in table slots, structs and local variables, arrays and passed as function parameters.

2.8.1 Struct Declaration

A struct object is created through the keyword 'struct' .

```
struct Foo
    x, y: int
    xf: float
```

Structs instances are created through a 'new expression' or a 'variable declatation statement':

```
let foo: Foo
let foo: Foo? = new Foo
```

There are intentionally no 'member functions', only data members, as it is data-type itself. However, struct can handle function typed member as data (meaning it's pointer can be changed during execution). There are kinda 'constructors', allowing to simplify writing structure initialization with complex time. Basically function with same name as struct itself will work as a constructor. Compiler will generate 'default' constructor if you have some of members with initializator:

```
struct Foo
    x: int = 1
    y: int = 2
```

Structs are also initialized as zero by default, regardless of 'initializators' for members, unless you specfically call constructor:

```
let fZero:Foo       // no constructor is called, x, y = 0
let fInited = Foo() // constructor is called, x = 1, y = 2
```

Structure field types are infered, where possible:

```
stryct Foo
        x = 1   // infered as integer
        y = 2.  // infered as float
```

Explicit structure initialization during creation will left all un-inited members zeroed:

```
let fExplicit = [[Foo x=13]]  // x = 13, y = 0
```

the previous code example is a syntactic sugar for:

```
let fExplicit: Foo
fExplicit.x = 13
```

Post construction initialization only needs to specify overwritten fields:

```
let fPostConstruction = [[Foo() x=13]]  // x = 13, y = 2
```

the previous code example is a syntactic sugar for:

```
let fPostConstruction: Foo
fPostConstruction.x = 13
fPostConstruction.y = 2
```

2.8.2 Struct Function Members

daScript doesn't have specific struct member functions, virtual (that can be overriden in derived structs) or non-virtual. For ease of Objected Oriented Programming, non-virtual member functions can be easily emulated with pipe operator '|>':

```
struct Foo
    x, y: int = 0

def setXY(var this: Foo; x, y: int)
    this.x = x
    this.y = y
...
var foo: Foo
foo |> setXY(10, 11)    // this is syntactic sugar for setXY(foo, 10, 11)
setXY(foo, 10, 11)      // exactly same thing as above line
```

Since function pointer is a thing, one can emulate 'virtual' functions by storing function pointers as member:

```
struct Foo
    x, y: int = 0
    set = setXY

def setXY(var this: Foo; x, y: int)
    this.x = x
    this.y = y
...
var foo: Foo = Foo()
foo->set(1, 2)  // this one can call something else, if overriden in derived class.
                // It is also just syntactic sugar for function pointer call
invoke(foo.set, foo, 1, 2)  // exactly same thing as above
```

This makes explicit difference between virtual and non-virtuall calls in OOP paradigm.

2.8.3 Inheritance

daScript's struct support single inheritance by adding the ' : ', followed by parent struct name, in the struct declaration. The syntax for a derived struct is the following

```
struct Bar: Foo
    yf: float
```

When a derived struct is declared, daScript first copies all base's members in the new struct then proceeds with evaluating the rest of the declaration.

A derived struct has all members of it's base. It is just a sytax sugar for copying all members manually first.

OOP. It's possible to override method of the base class by override syntax. Here an example:

```
struct Foo
    x, y: int = 0
    set = @Foo_setXY

def Foo_setXY(var this: Foo; x, y: int)
    this.x = x
    this.y = y

struct Foo3D: Foo
    z: int = 3
    override set = cast<auto> @Foo3D_setXY

def Foo3D_setXY(var this: Foo3D; x, y: int)
    this.x = x
    this.y = y
    this.z = -1
```

It is safe to use 'cast' keyword to cast derived struct instance to reference to it's parent type:

```
var f3d: Foo3D = Foo3D()
(cast<Foo> f3d).y = 5
```

It is unsafe to 'cast' to cast base struct to it's dereived

```
var f3d: Foo3D = Foo3D()
def foo(foo: Foo)
  (cast<Foo3d> foo).z = 5  // error, won't compile
```

if needed, the upcast can be used with [unsafe] annotation

```
struct Foo
  x: int
struct Foo2
  y: int
[unsafe]
def setY(foo: Foo; y: int)  // Warning! Can make awful things to your app if not-
↪really Foo2 is passed!
  (upcast<Foo3d> foo).y = y
```

As the example above is very dangerours, and in order to make it safer, you can modify it to following:

```
struct Foo
  x: int
  typeTag: uint = hash("Foo")

struct Foo2
  y: int
  typeTag: uint = hash("Foo2")

[unsafe]
def setY(foo: Foo; y: int)  // this won't do anything really bad, but will panic on␣
↪wrong reference
    if foo.typeTag == hash("Foo2")
        (cast<Foo3d> foo).y = y
    else
        assert(0, "Not Foo2 type references was passed")
```

2.9 Tuples

todo: to be written

2.10 Iterators

todo: to be written

2.11 Constants & Enumerations

daScript allows to bind constant values to a global variable identifier, and all constant global variables will be evaulated compile time if possible. There is also Enumerations, which are strong type constants (similar to enum class in C++).

2.11.1 Constants

Constants bind a specific value to an identifier. Constants are exactly global variables, their value cannot be changed.

are declared with the following syntax:

```
let
  foobar = 100
let
  floatbar = 1.2
let
  stringbar = "I'm a constant string"
```

constants are always globally scoped, from the moment they are declared, any following code can reference them.

You can not change such global variables. Mutable global variables are defined as:

```
var
  foobar = 100
```

and their usage can be switched of per-project basis.

2.11.2 Enumerations

As Enumerations bind a specific value to a name. Enumerations are also evaluated at compile time and their value cannot be changed.

An enum declaration introduces a new enumeration into the program. Enumeration values can only be integers. No expression are allowed. It is not required to assign specific value to enum:

```
enum Numbers
    zero     // will be 0
    one      // will be 1
    two      // will be 2
    ten = 10 // will be 10, as written
```

An enum name itself is strong type, and all enum values are of this type. An enum value can be addressed as 'enum name' followed by exact enumeration

```
let one: Numbers = Numbers one
```

An enum value can be converted to integer type with explicit cast

```
let one: Numbers = Numbers one
assert(int(one) == 1)
```

2.12 Built-in Functions

2.12.1 Misc

panic()
will cause panic. The program will be determinated if there is no recover. Panic is not a error handling mechanism and can not be used as such. It is indeed panic, fatal error. It is not supposed that program can completely correctly recover from panic, recover construction is provided so program can try to correcly shut-down or report fatal error. If there is no recover withing script, it will be called in calling eval (in C++ callee code).

print(*x*)
print prints any provided argument *x*, provided that type has DataWalker 'to string' (all PODs do have it).

stackwalk()
stackwalk prints call stack and local variables values

terminate()
terminates program execution

breakpoint()
breakpoint will call os_debugbreakpoint, which is link-time unresolved dependency. It's supposed to call breakpoint in debugger tool, as sample implementation does.

heap_bytes_allocated()
heap_bytes_allocated will return bytes allocated on heap (i.e. really used, not reserved)

assert(*x*, *str*)
assert will cause application defined assert if *x* argument is false. assert can and probably will be removed from release builds. That's why assert will not compile, if *x* argument has side effect (for example, calling function with side effects).

verify(*x*, *str*)
verify will cause application defined assert if *x* argument is false. verify check can be removed from release builds, but execution of *x* argument staus. That's why verify, unlike assert can have side effects in evaluating *x*

static_assert(*x*, *str*)
static_assert will cause compiler to stop compilation if *x* argument is false. That's why *x* has to be compile-time known constant. static_assert will be removed from compiled program.

debug(*x*, *str*)
debug will print string str and value of x (like print). However, debug also returns value of x, which makes it suitable for debugging expressions:

```
let mad = debug(x, "x") * debug(y, "y") + debug(z, "z") // x*y + z
```

2.12.2 Arrays

push (*array_arg*, *value*[, *at*])
push will push to dynamic array *array_arg* the content of *value*. *value* has to be of the same type (or const reference to same type) as array values. if *at* is provided *value* will be pushed at index *at*, otherwise to the end of array. The *content* of value will be copied (assigned) to it.

emplace (*array_arg*, *value*[, *at*])
emplace will push to dynamic array *array_arg* the content of *value*. *value* has to be of the same type (or const reference to same type) as array values. if *at* is provided *value* will be pushed at index *at*, otherwise to the end of array. The *content* of value will be moved (<-) to it.

resize (*array_arg*, *new_size*)
Resize will resize *array_arg* array to a new size of *new_size*. If new_size is bigger than current, new elements will be zeroed.

erase (*array_arg*, *at*)
erase will erase *at* index element in *array_arg* array.

length (*array_arg*)
length will return current size of array *array_arg*.

clear (*array_arg*)
clear will clear whole array *array_arg*. The size of *array_arg* after clear is 0.

capacity (*array_arg*)
capacity will return current capacity of array *array_arg*. Capacity is the count of elements, allocating (or pushing) until that size won't cause reallocating dynamic heap.

to_array (*arg*)
will convert argument (static array, iterator, another dynamic array) to an array. argument elements will be cloned

to_array_move (*arg*)
will convert argument (static array, iterator, another dynamic array) to an array. argument elements will be copied or moved

(see *Arrays*).

2.12.3 Tables

clear (*table_arg*)
clear will clear whole table *table_arg*. The size of *table_arg* after clear is 0.

capacity (*table_arg*)
capacity will return current capacity of table *table_arg*. Capacity is the count of elements, allocating (or pushing) until that size won't cause reallocating dynamic heap.

erase (*table_arg*, *at*)
erase will erase *at* key element in *table_arg* table.

length (*table_arg*)
length will return current size of table *table_arg*.

key_exists (*table_arg*, *key*)
will return true if element *key* exists in table *table_arg*.

find (*table_arg*, *key*, *block_arg*)
will execute *block_arg* with argument pointer-to-value in *table_arg* pointing to value indexed by *key*, or null if *key* doesn't exist in *table_arg*.

to_table (*arg*)
will convert an array of key-value tuples into a table<key;value> type. arguments will be cloned

to_table_move (*arg*)
will convert an array of key-value tuples into a table<key;value> type. arguments will be copied or moved

keys (*arg*)
returns iterator to all keys of the table

values (*arg*)
returns iterator to all values of the table

(see *Tables*).

2.12.4 Functions

invoke (*block_or_function*, *arguments*)
invoke will call block or pointer to function (*block_or_function*) with provided list of arguments

(see *Functions*).

2.12.5 Iterators

each (*obj*)
returns iterator, which iterates though each element of the object. object can be range, static or dynamic array, another iterator.

(see *Iterators*).

2.13 String Builder

Instead of formatting strings with variant arguments count function (like printf), daScript provides String builder functionality out-of-box. It is both more readable, more compact and more robust than printf-like syntax. All strings in daScript can be either string literals, or *built strings*. Both are written with “”, but string builder strings also contains any expresion in curly brackets ‘{}’:

```
let str1 = "String Literal"
let str2 = "str1={str1}"  // str2 will be "str1=String Literal"
```

In the example above, str2 will actually be compile-time defined, as expression in {} is compile-time computable. But generally, that can be run-time compiled as well. Expression in {} can be of any type, including handled extern type, provided that said type implements DataWalker. All PODs in daScript do have DataWalker ‘to string’ implementation.

In order to make string with {} inside, one has to escape it with ‘\’

```
print("Curly brackets=\{\}")  // prints Curly brackets={}
```

2.14 Generic Programming

daScript allows ommision of types in statements, functions, and function declaration, making writing in it similar to dynamically typed languages, such as Python or Lua. Said functions will be *instantiated* for specific types of arguments on first call.

There are also ways to inspect types of provided arguments, in order to change behaviour of function, or provided reasonable meaningful errors during compilation phase. Most of these ways are achieved with s

Unlike C++ with it's SFINAE, you can use comon conditions (if) in order to change instance of function depending on type info of arguments. Consider the following example

```
def setSomeField(var obj; val)
    if typeinfo(has_field<someField> obj)
        obj.someField = val
```

this function will set someField in provided argument *if* it is a struct with someField member.

We can do even more, for example

```
def setSomeField(var obj; val: auto(valT))
    if typeinfo(has_field<someField> obj)
        if typeinfo(typename obj.someField) == typeinfo(typename type valT delete␣
↪const)
            obj.someField = val
```

this function will set someField in provided argument *if* it is a struct with someField member, and more over, only if someField is of the same type as val!

2.14.1 typeinfo

Most of type reflection mechanisms are achieved with typeinfo operator. There are:

- typeinfo(typename object) // returns typename of object
- typeinfo(fulltypename object) // returns full typename of object, with contracts (like !const, or !&)
- typeinfo(sizeof object) // returns sizeof
- typeinfo(is_pod object) // returns true if object is POD type
- typeinfo(is_raw object) // returns true if object is raw data, i.e., can be copied with memcpy
- typeinfo(is_struct object) // returns true if object is struct
- typeinfo(has_field<name_of_field> object) // returns true if object is struct with field name_of_field
- typeinfo(is_ref object) // returns true if object is reference to something
- typeinfo(is_ref_type object) // returns true if object is of reference type (such as array, table, das_string or other handled reference types)
- typeinfo(is_const object) // returns true if object is of const type (i.e., can't be modified)
- typeinfo(is_pointer object) // returns true if object is of pointer type, i.e., int?

All typeinfo can wotk with types not objects, with type keyword, i.e.

```
* typeinfo(typename type int) // returns "int"
```

2.14.2 auto and auto(named)

Instead of ommision of type name in generic, it is possible to use explicit 'auto' type or 'auto(name)' type it.

```
def fn(a: auto): auto
    return a
```

or

```
def fn(a: auto(some_name)): some_name
    return a
```

same as

```
def fn(a)
    return a
```

this is very helpful, if function accept numerous arguments, and some of them has to be of the same type

```
def fn(a, b) // a and b can be of different types
    return a + b
```

Not the same as:

```
def fn(a, b: auto) // a and b are one type
    return a + b
```

Also, consider the following:

```
def set0(a, b; index: int) // a is only supposed to be of array type, of same type as␣
↪b
    return a[index] = b
```

If you cann this function with array of floats and int, you would get not obvious compiler error message:

```
def set0(a: array<auto(some>); b: some; index: int) // a is of array type, of same␣
↪type as b
    return a[index] = b
```

Usage of named auto with typeinfo

```
def fn(a: auto(some))
    print(typeinfo(typename type some))

fn(1) // print "const int"
```

you can also modify type with delete syntax:

```
def fn(a: auto(some))
    print(typeinfo(typename type some delete const))

fn(1) // print "int"
```