The Eliza Programming Language

Language Specification and Reference

Version 0.11 (Rel. 2008-02-04)

Copyright © 2000 - 2008 Matthias Huerlemann

Eliza is released under the GNU General Public Licence v3:

Eliza Interpreter V0.11 (Rel. 2008-02-04)
Copyright (C) 2000-2008 Matthias Huerlemann

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

Table of Contents

1. Introduction, Motivation....4
  1.1 Audience....4
  1.2 Introduction....4
  1.3 Motivation....4
  1.4 Design Principles....4
  1.5 History and Influences....5
  1.6 Current Status....5
  1.7 Plan....5
  1.8 Website, Download, Installation....5
  1.9 Compilation....5
  1.10 Typing conventions....6
2. First Steps....7
  2.1 The first Eliza Program....7
  2.2 Usage of Eliza....7
3. Grammar....8
  3.1 Grammar supported in V0.11....8
  3.2 Full Grammar....9
4. GCTF Engine....10
  4.1 Core Concepts of Eliza....11
  4.2 Function Call Syntax and Operators....12
  4.3 Backtracker....12
  4.4 Function Types....13
  4.5 Connectors....14
  4.6 Argument Passing....16
  4.7 Patterns....17
  4.8 Sample Programs....20
5. Data Types....22
  5.1 Type Hierarchy....22
  5.2 Currently Supported Types....22
  5.3 All Types....22
6. Function Reference....24
  6.1 Supported Functions in V0.11....24
  6.2 Eliza Errors....25
  6.3 Library Structure (Work in Progress)....25
7. Dynamic Library API (Concept)....26
8. Analysis & Design with Eliza....26
9. Programming Style....26
10. Open Questions and Concepts....26

1. Introduction, Motivation

1.1 Audience

Developers interested in a new programming model and a new way of thinking. Eliza requires a different approach in problem solving. If this weren't the case, it would be superfluous. Developers interested in contributing code.

A *rich person* who would like to support the development of an exciting new programming language.

A *university professor* who is interested in academic research around Eliza. Maybe some *students* could investigate the areas „grammar“ or „basic library structure“ and maybe even contribute some code.

Of course I am interested in feedback and suggestions for improvement.

`info at eliza.ch`.

I will provide my IBAN number anytime upon request.

1.2 Introduction

I developed the Eliza language because I wanted a very good programming language for AI (artificial intelligence). And because none of the existing ones convinced me, I started thinking about a new one.

1.3 Motivation

At the beginning there was a bet (in 1991 or so): I bet that I could write a computer program that passes the Turing Test. Easier sad than done. But it was the start of an odyssey through all basic principles and concepts of programming languages. And in the end I had the best programming language in the world.

1.4 Design Principles

During the development at some point I defined four major principles as guiding rules for design decisions.

Simplicity: The language, the grammar, programming itself, documentation, educational material, tools around Eliza, basic concepts – must all be easy to understand. Simplicity prevents errors because we understand what we do.

Small size: Code, documentation, grammar, manual and language specification must be small and the core library clearly laid out.

Speed: The interpreter / compiler must be very fast at compile time and at runtime. A lot of problems don't even arise if a tool performs from the beginning.

Smartness: Functions must be smart, i.e. powerful, flexible, balanced. The engine must take control of boring jobs like `for` loops as in imperative programming languages. The interpreter / compiler must issue smart warnings and errors to easily identify bugs in the code. The paradigm would presumably be declarative.

1.5 History and Influences

The grammar of Eliza developed over about 8 years. Some aspects changed quite late and the behaviour is now in version 0.11 definitively defined for the first time.

Eliza has been influenced by many programming languages:

Prolog: Declarative programming and backtracking. „And“, „Or“ and Prologs way of catching errors.

Lisp, Scheme: Syntax. Eliza uses Lisps prefix notation. Eliza doesn't have operators. Interesting concepts like `map` or `reduce`. First-order-functions. Imperative parts in a declarative program. Symbols.

Icon: With Icon I realised that backtracking is not limited to Prolog. Even a procedural language can backtrack.

Bash: Pipes are very cool. Some small command line „programs“ do a lot in a very elegant way like (with the Eliza sources):

```
$ls | grep .cc | wc -l
5
```

To count all files with ending *.cc in a directory.

Curl: A nice possible successor of the ugly HTML, CSS, JavaScript, XML, Java etc. web programming mix (or mess). Eliza is influenced by its syntax.

C: C is a small language with only few keywords. It is quite consistent.

1.6 Current Status

V0.11 supports a subset of the whole grammar of Eliza. Also the library is small (10 functions). Currently only integers are supported to show how Eliza works. Strings can be printed.

The core of Eliza is defined. The engine works.

1.7 Plan

Next steps include:

- Implement other parts of the grammar,
- Implement other types and
- Functions to use them.

1.8 Website, Download, Installation

The Eliza interpreter, this document and the GPL v3 can be downloaded at:

`http://www.eliza.ch/`

1.9 Compilation

If you want to compile Eliza on your operating system the included makefile should work on an average Unix system. `cocor` is not required if you don't make

changes in the `eliza.atg` or `eliza.frm` files.

If you want to compile it on Windows, you need to install MinGW first. An `eliza.exe` can be downloaded.

`cocor` for C/C++ can be downloaded at: `http://www.scifac.ru.ac.za/coco/`

Coco/R is a compiler compiler developed at ETH Zurich by Prof. Mössenböck. It generates a recursive descent parser from an attributed EBNF grammar.

The implementation used, has been adapted to C/C++ by Frankie Arzu and extended by Pat Terry.

1.10 Typing conventions

Example code is always in courier font:

```
:call_a 55 99 | println
```

Placeholders in the code – parts you have to supply yourself are in courier italic:

```
:call_a arg1 arg2 | println
```

If you have to enter a filename or command line options they are in courier as well and between < > brackets:

```
$eliza <options> <filename>
```

Commands to type on the command line / shell start with a „$“.

Program names and file names are also written in courier.

2. First Steps

2.1 The first Eliza Program

Eliza is a very simple programming language. The very well known first program „Hello World!“ (for lack of ideas) looks in Eliza like:

```
:println „Hello World!“
```

Type it in a text file and run it.

On Unix:

```
$eliza hello.eliza
```

On Windows:

```
$eliza.exe hello.eliza
```

2.2 Usage of Eliza

If you just type „eliza“ you get the usage screen:

```
$eliza
Eliza interpreter V0.11 (Rel. 2008-02-04)
Copyright (C) 2000-2008 Matthias Huerlemann

Usage  : ./eliza <flags> <file>
Example: ./eliza -P test.eliza

Options:
d:  Print the documentation of all builtin library functions
D:  Turn debugging on (minimal)
P:  Turn profiling on (only elapsed time so far)
V:  Print the version information
L:  Print the licence (GPL v3)
```

On Windows, flags can also be used like `/P`.

Later there will be more flags:

```
Q:  A complexity metric for the program with call count, call
graph etc.
T:  Testing mode which disables printout on the command shell etc.
D:  Enhancement: Show internal flow charts and argument passing
d:  Enhancement:
    -d _all  prints all functions in short form, while
    -d <function_name> prints the detail for that function
    -d _doc  prints the whole program in LaTeX, HTML or so
P:  Enhancement: full profiling for library functions and / or
user functions
```

3. Grammar

3.1 Grammar supported in V0.11

The Eliza grammar follows the declaration rules of Coco/R. First there are character sets, then tokens, comments and characters to ignore. After that follows the grammar (PRODUCTIONS) – in EBNF (Extended Backus Naur Form), developed by Niklaus Wirth.

```
COMPILER Eliza

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit = "0123456789".
  hexdigit = digit + "abcdefABCDEF".
  cr = CHR(13).
  lf = CHR(10).
  tab = CHR(9).
  notquote = ANY - '"'.
  notapo   = ANY - "'" - lf -cr.
  under = CHR(95).
  ques  = CHR(63).
  spec  = "<>=?+$@%!".

TOKENS
  ident  = "/" | "*" | "-" | "*" "=" | "/" "=" | "-" ">" | ( ( letter | under
letter | spec ) { letter | digit | under | spec } ).
  param = under digit { digit }.
  number = digit {digit}.
  qstring = '"' { notquote } '"'.
  sregex  = "'" { notapo | "\'" } "'".

IGNORE cr + lf + tab

COMMENTS FROM "//" TO lf
COMMENTS FROM "#" TO lf

PRODUCTIONS
Eliza = { ":" Sequence }.

Sequence = FunCall { ("," | ";" | "|" | "&") FunCall }.

FunCall = ident { Expression }.

Expression = [ "-" ] number | param | qstring | ident.

END Eliza.
```

3.2 Full Grammar

Next extensions:

```
FunCall = ident { Expression } | "(" Sequence ")".

Expression = [ "-" ] number | param | qstring | ident | Array.

Array = "[" Expression { "," Expression } "]".
```

Function call sequences can be nested with „(„ and „)“. The „Array“ production shows how compound types can be parsed. To have all the compound types be parsed correctly a change in the grammar is necessary:

ToDo!!!

Future Eliza programs should look something like this. Used concepts:
Function declaration and function definition in a namespace.

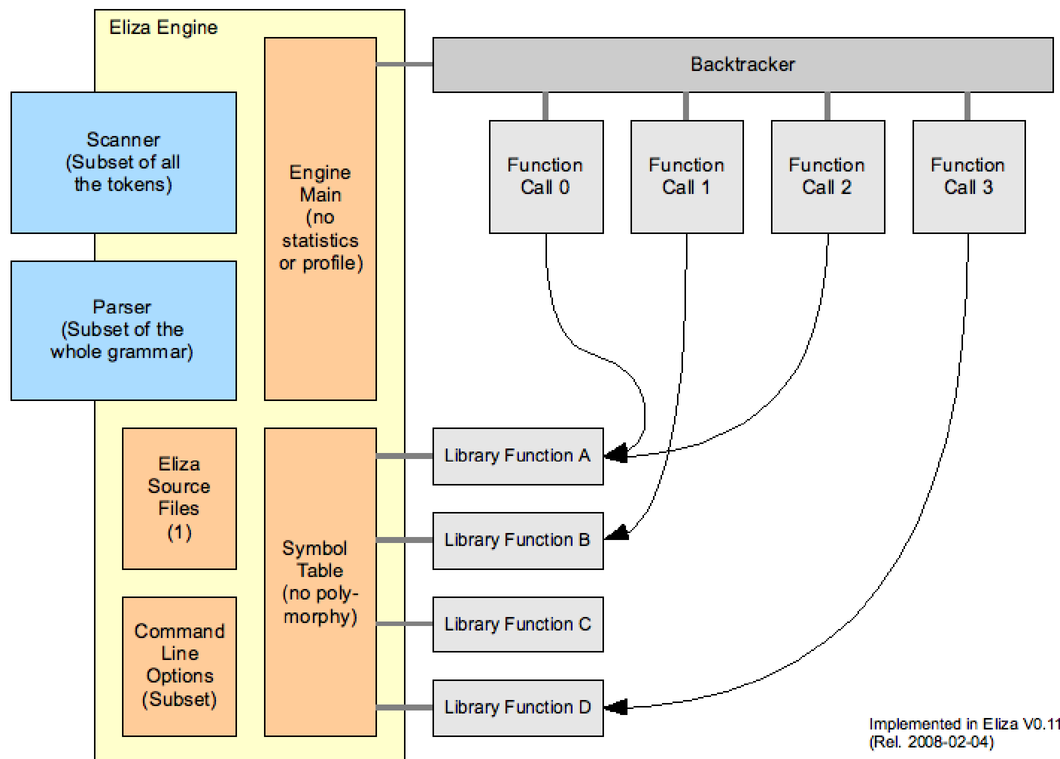
```
:func math.odd? int -> int <F>

{ ::math
  { odd? int a =
    % a 2 | == _1 1, -> a
  }
}

:math.odd? 47 | println _1 „ is even, that's odd!“
```

4. GCTF Engine

GCTF Engine means „four function type – engine“:

Generator (<G>), **C**oncentrator (<C>), **T**ransformer (<T>), **F**ilter (<F>) – engine. These four function types are described in chapter 4.4.

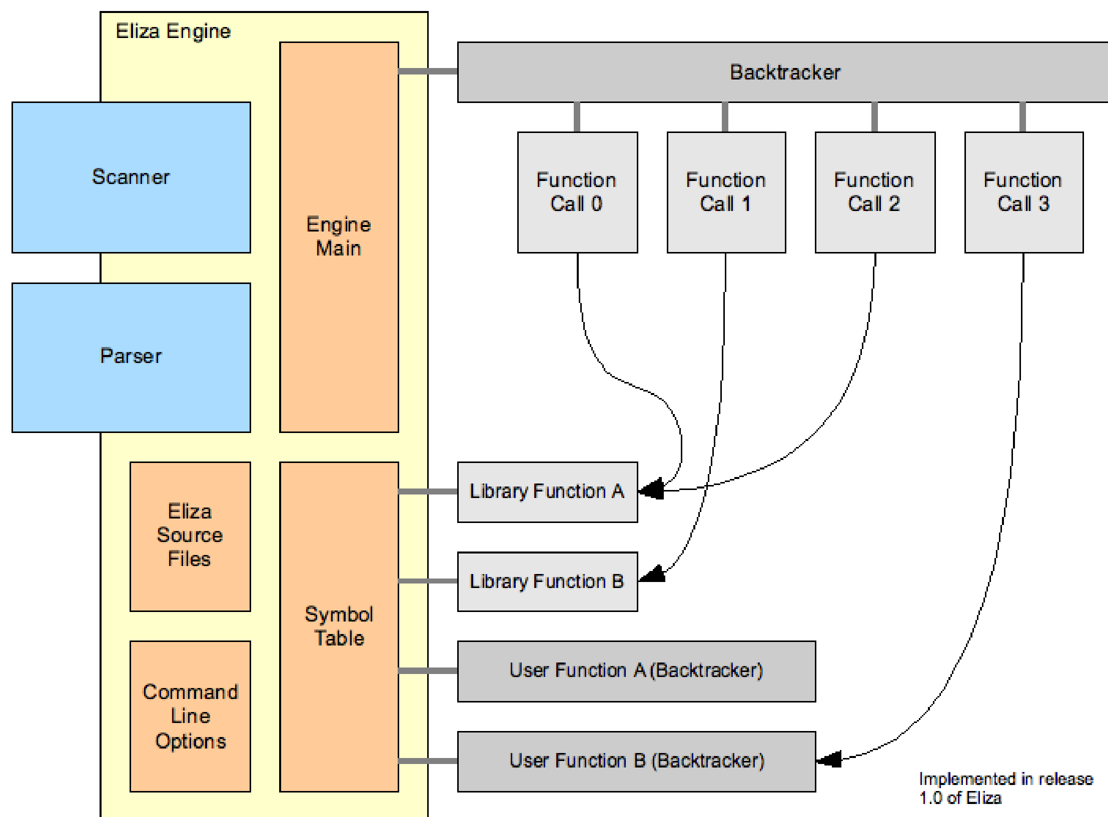
The engine consists of two main parts: The main structure and the symbol table. Scanner and parser are of course important, but not special to Eliza. Currently the symbol table contains only builtin library functions. These are called from a backtracker with linked function call objects.



The function call objects make the library functions reentrant because any „static“ content is stored in the „Function Call x“ object.

Execution order is always from left to right.

Arguments passed to functions can be evaluated in any order.



Later user functions will be supported (of course) and they will be linked into the symbol table and called by the function calls.

4.1 Core Concepts of Eliza

Eliza implements the producer-consumer pattern like in command line pipes as shown in the example in chapter 1.5.

Eliza uses backtracking as a replacement for all `for`, `while` and `until` loop constructs.

Eliza will use other concepts as well:

Concept	Description

Function definition	Functions are defined first and then used. Allows for cross usage of functions. Like:
	`:func * int * int -> int <T>`
	Explanation: * takes two int as input and produces one int as output. It is a Transformer.
	`:func println (type) -> (type) <T>`
	Explanation: println takes none, one or many values of any kind and prints them with a newline at the end. (type) means: Can be repeated.

Namespaces	Functions are grouped in namespaces for structuring purposes and to prevent name clashes.
Include	Eliza source file include.
Import	Binary library import. C/C++ libraries for dynamic linking.

Bytecode	A future version of Eliza should compile to portable bytecode. Maybe another environment is reused to have access to more libraries available like: Perl6.
Alias	Give a variable a second name.
Aspect	Provide features for functional aspect oriented programming.
Eliza libraries	A means for the compilation of libraries in Eliza must be provided like a bytecode compiler.
Recursion	Maybe recursion will not be allowed.

4.2 Function Call Syntax and Operators

Eliza functions are all called with prefix notation like in Lisp and Scheme.

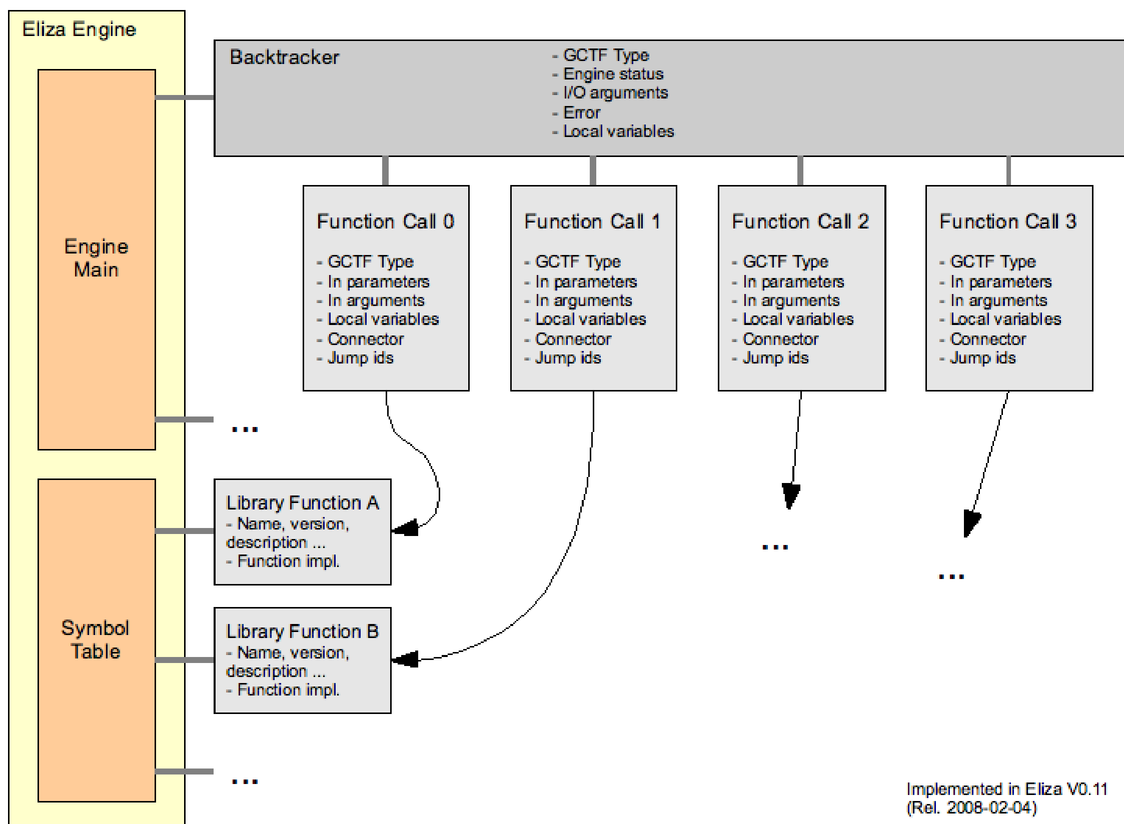
```
:* 8 9 | println
```

There are no operators in Eliza. Everything is a function. Even:

```
:+ a 3     // same as: a + 3
:< 45 46   // same as: 45 < 46
```

4.3 Backtracker

The backtracker is the core component for program flow.



The backtracker manages input to function calls and output from them, the

engine status, error handling and the program flow from one function call to the other. The backtracker has control over the initialisation of coroutines (<G> and <C> function types) and argument passing (chapter 4.6) according to the call sequence separated by connectors (chapter 4.5). The engine status is evaluated *after* a function has been executed.

The Eliza engine can have 4 different statuses:

Status	Description
OK	Execution of a function was successful. Usually the next function is executed. If a „;“ follows, execution continues after the next „|“.
BACK	An <F> or a <C> have induced backtracking. This means that the program flow jumps back to the first <G> in the sequence. If there is no <G> the program reaches its end. If there follows a „;“ after an <F>, execution continues after the „;“. Another special case are two <C> & <C>. If the first <C> induces backtracking by setting the status to BACK, the engine changes it to OK so that the second <C> can be executed also. After the last <C> in such a row, backtracking takes place.
NIL	If a <G> can't produce any more values it changes the engine status to NIL. In this case the engine jumps to the next <C> which in turn does not backtrack but changes the status to OK and produces a value. Only <G> can set NIL. Only <C> react on it.
ERR	If any error occurs in a function the engine changes in ERR status. If there follows a „;“ before the next „|“, the function after the „;“ is executed. If there is no „;“, execution reaches the end of the program.

4.4 Function Types

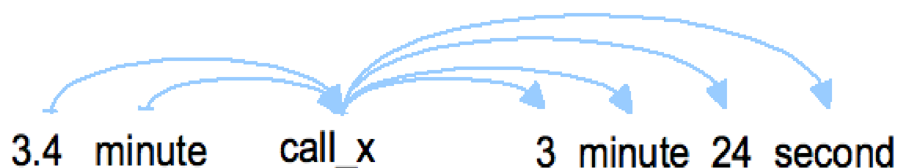
Programs consist of four function types. They behave differently in the execution flow of a progam with regard to the internal state of the engine.

Function Type	Description
<G>	A Generator is a stateful function, that takes arguments the first time it is called. After that it produces new values every time it is called again. When is it called again? When a <F> or a <C> induces backtracking and the program flow jumps back to this <G> to deliver a new value.
<C>	A Concentrator is a stateful function as well. It is like the opposite of a <G>. A concentrator usually collects or concentrates values until it gets a signal from the generator that there are no more values to collect. Then the <C> produces a result and passes it to the next function.

<T>	A Transformer is a function like in most programming languages. It takes input, transforms it and delivers an output to the next function. It is stateless.
<F>	A filter is a function that validates input. If the input conforms to the rules, it is passed as is to the next function. If not, backtracking is initiated and the function flow goes back to the first <G> in the function sequence. A new value is produced and validated in the <F>. <F> can be stateful as a special case. It must keep a state for example if it only passes every third value to the next function. Usually <F> are stateless.

It seems that these four types are sufficient to model any program. Tell me if I am wrong.

A function in Eliza can take several input values and produce several output values.



In this example minutes are translated from decimal form into minutes and seconds.

4.5 Connectors

There are four types of connectors between function calls: pipe („|“), and („**,**“), or („**;**“) and parallel and („**&**“).

Values (also called arguments) are passed from one function to the next. What to the connectors do?

<table>
<tr><th>Connector</th><th>Description</th></tr>
<tr><td>| (Pipe)</td><td>A pipe passes all the values it receives as output from the prior function – as input to the next function.
<code>:func_a 17 | func_b</code>
<code>func_a</code> takes 17 as input and produces, say 99 as output. 99 is passed to <code>func_b</code>.
Values are passed to the next function only through pipes.</td></tr>
<tr><td>, (And)</td><td>And does not pass any value to the next function. And is used if no values must be passed, for example if they are passed anyway. Function connected with „,“ are a sequence of independent function calls.</td></tr>
<tr><td>; (Or)</td><td>An or offers alternatives. If a function call before the „;“ would induce backtracking or causes an error, the alternative behind the</td></tr>
</table>

	„;“ is called. An „;“ works like an `else` in other programming languages. But it also catches errors (`catch` in e.g. C++ or Java).

& (Parallel And)	An „&“ allows you to have several functions in sequence which all can pass their output to the next function.
	`:func_a 17, func_b 19 | func_c`
	With an „,“ only `func_b` passes a value to `func_c`. With Parallel And also `func_a` can pass its output to `func_c`:
	`:func_a 17 & func_b 19 | func_c`
	`func_c` receives the values in order: first the output from `func_a`, then the output from `func_b`. Let's assume `func_a` produces 99 and `func_b` „Hey there“, then `func_c` receives two values: 99 and „Hey there“ as input.

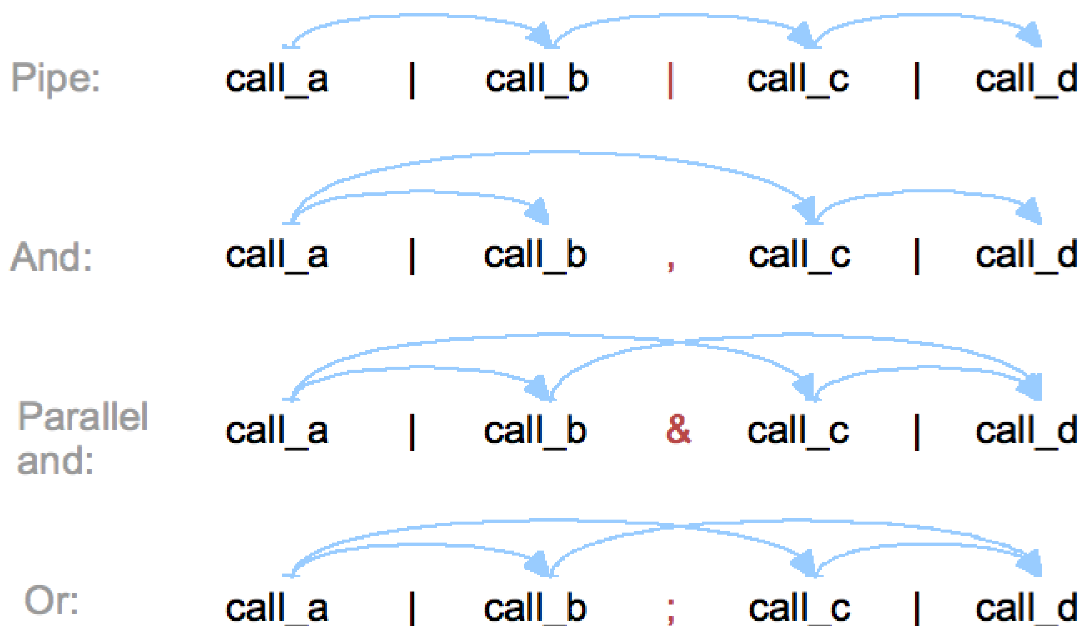
In V0.11 you can try progams that show you the effects of the connectors. Some running sample programs:

```
:-> 3 8 | * | println
:square 9 , println
:-> 7 | * | println
```

The „`:`“ means: Here starts a function call sequence.

Important note: If values are passed through a pipe, they are passed to all functions up to the next pipe.

Summary:



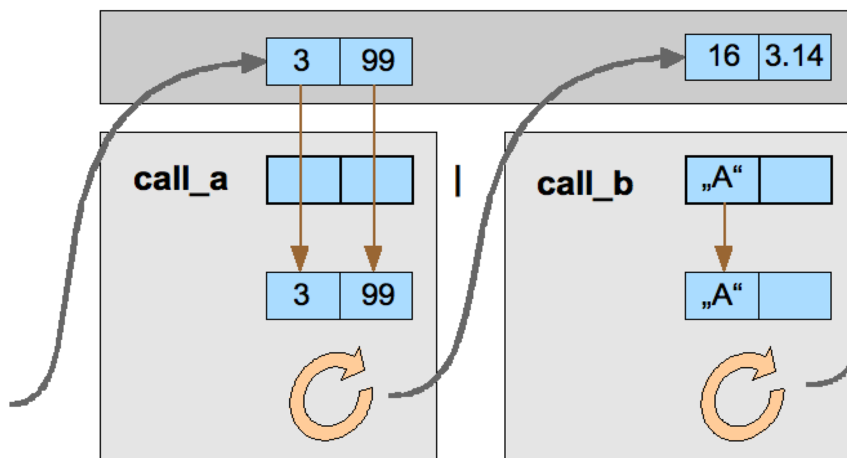
In the „`;`“ case either `call_b` or `call_c` will pass its values to `call_d`. If `call_b` is true, it passes its arguments to `call_d`.

4.6 Argument Passing

This is a very important chapter. It shows how values are passed from one function to the next.

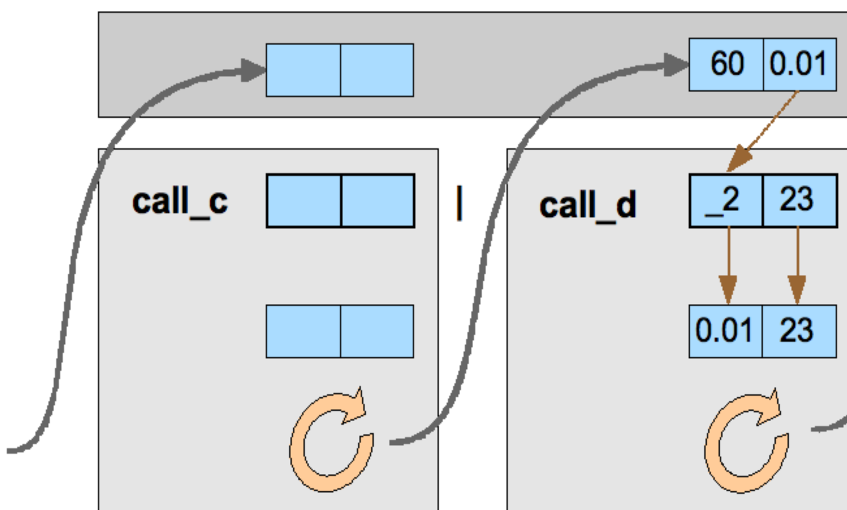
Function calls can define no input values, specific values (e.g. 17 or 5) or a position in the output values received from the prior function.

```
:-> 17 | println      // 1) 17
:-> 17 | println 5    // 2) 5
:-> 17 | println _1   // 3) 17
:-> 17 | println _3   // 4) error: only one value is passed
```



`call_a` receives everything from the prior function call because it doesn't specify its own input values (see example 1) above).

`call_b` defines one value. The passed output values from `call_a` are not used (see example 2) above).



`call_c` doesn't receive or define any values. Nothing is passed to the function.

`call_d` defines a position (_2) and a value (23). The position directive is to take the second value (0.01) of the passed arguments coming from `call_c` (see example 3) above).

4.7 Patterns

The function types make sense in certain combinations. Certain combinations are meaningless or cause erroneous behaviour.

1. Sequences

<T> | <T>
<T> | <F> , <T> ; <T>
<F> | <F>
<T> & <T> | <T>

2. Generator, Concentrator „Ping-pong“

<G> | <T> | <F> | <C>
<G> | <F> , <T> | <C>
<G> | <F> , <F> | <C>
<G> | <T> | <F> | <C>
<G> | <T> | <F> | <C>
<G> & <G> | <T> | <C> & <C>

3. Erroneous patterns

<G> , <T> | < C> // A <G> can't pass a value through an „,“ - error
<G> | <G> | <T> | <C> // Backtracking can't tell to which <G> to go
<G> | <T> | <C> | <C> // Endless loop because of two <C> inducing
// backtracking
<T> | <C> | <T> // No value is produced

Most functions in the library will be <T>. There will be many <F> also but much fewer <G> and <C>.

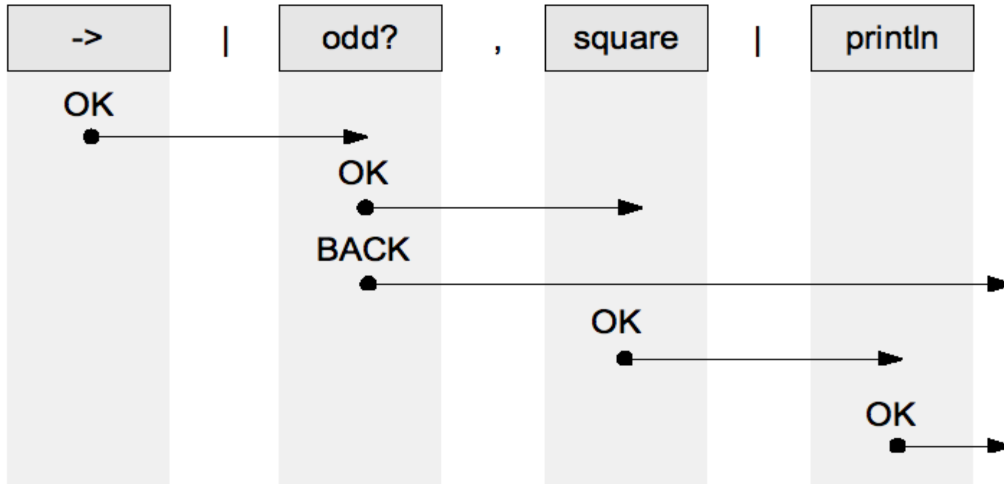
To understand the behaviour of a program it is helpful to know how execution flows.

Example program 1:

Description: If 7 is odd, square it and print it (with a newline at the end).

```
:-> 7 | odd?, square _1 | println
```

The call graph shows a relatively simple execution flow.

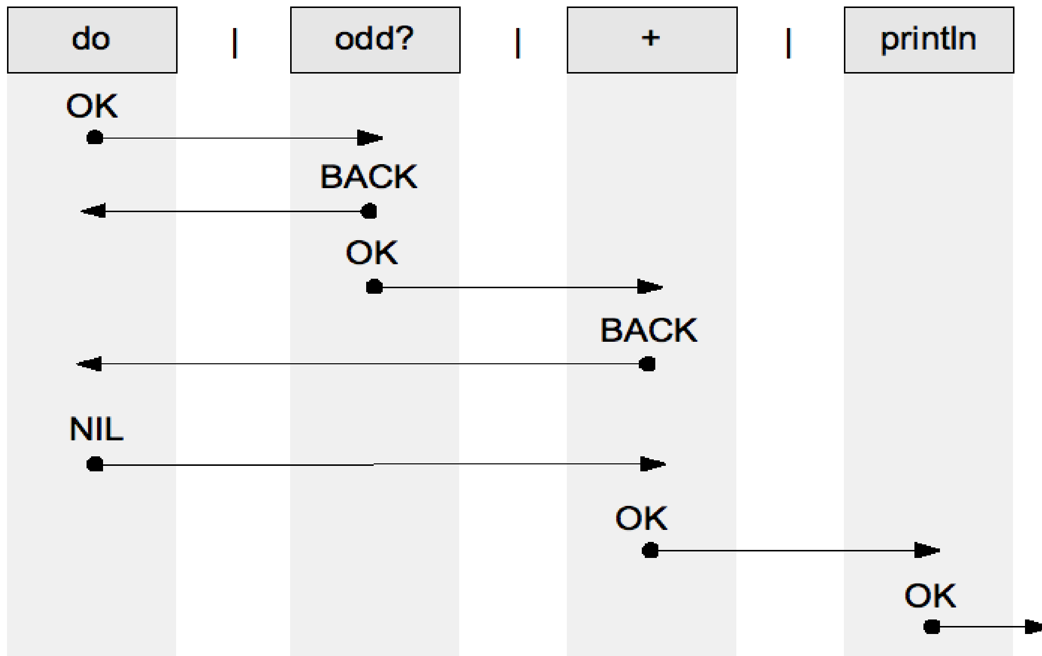


A slightly more complicated program: **Example 2**:

Description: Take all numbers between 1 and 10 (including) and sum them up if they are odd. Print the result.

```
:do 1 10 | odd? | + | println
```

And the according call graph:

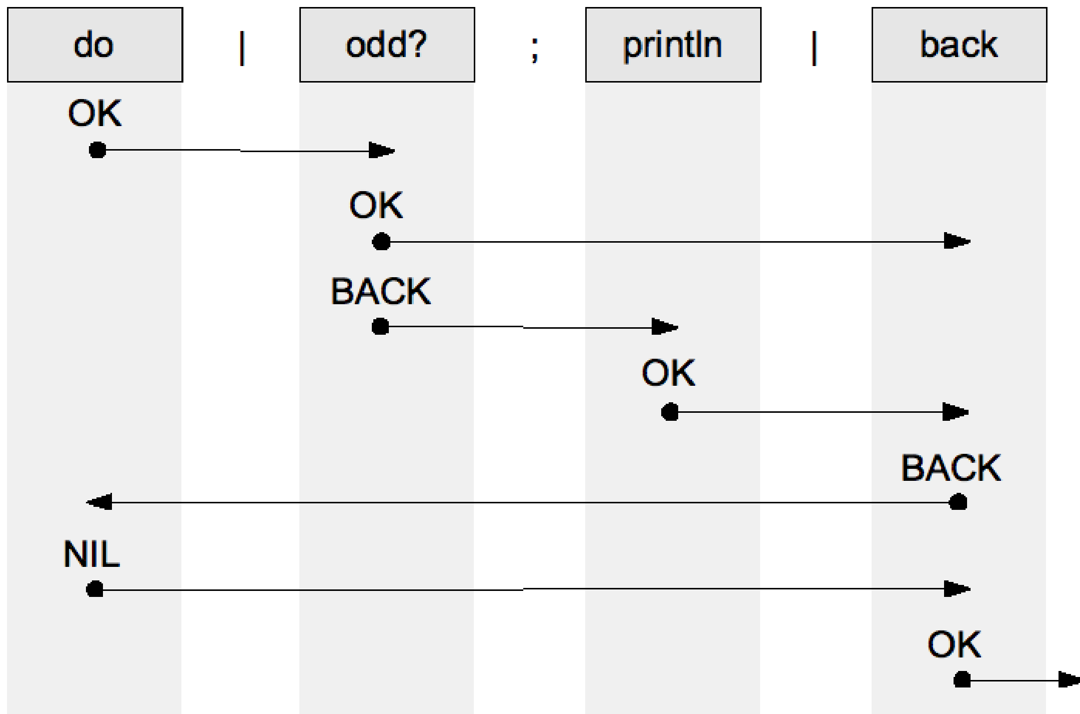


Example program 3 is similar, but with a different execution flow:

Description: Take alle numbers between 1 and 10 (including) and print them if they are odd.

```
:do 1 10 | odd? | println | back
```

The according call graph shows the different ways the execution flow takes:



4.8 Sample Programs

For testing purposes I have written a bunch of short programs. I add them here that you can see how the engine works and what can go wrong in Eliza programs. These examples use all the currently builtin functions, they are correct Eliza programs and the produce the expected result (according to the language specification).

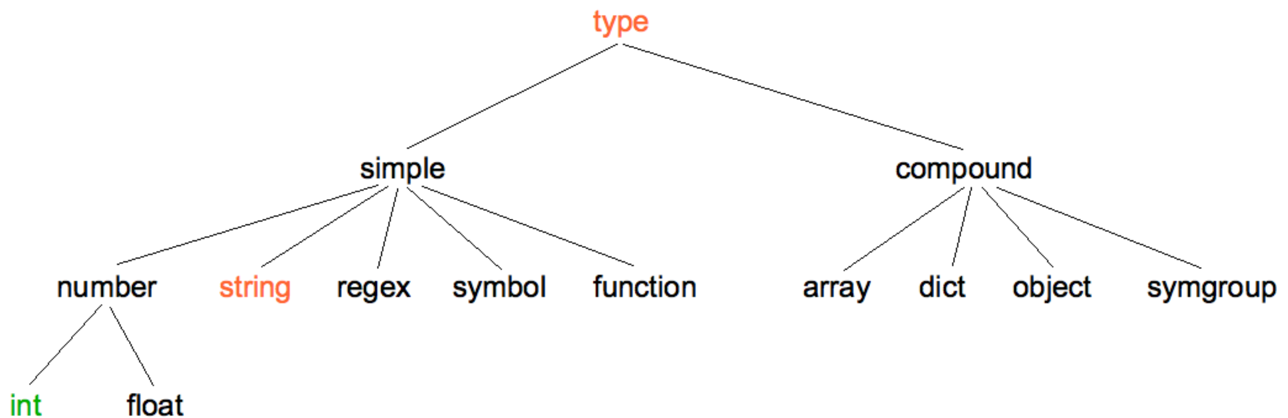
```
// Print 5 * 9
:* 5 9 | println

// Print the square of 5 * 9
:* 5 9 | square | println

// The same with positions
:* 5 9 | square _1 | println _1

// Wrong position for square
:* 5 9 | square _2 | println _1

// Some minimal formatting
:* 5 9 | square | println "-->" _1 "<--"

// 4 * 9 is even
:* 4 9 | odd? | println

// Nothing passed through a „,“
:* 4 9 , println

// Chose the passed argument
:square 17 & square 5 | println _1

// Print all there is
:square 17 & square 5 | println

// square of square
:square 4 | * _1 _1 | println

// odd vs. even
:square 58 | odd?, println ; println 1 " " _1 " " 2
:square 59 | odd?, println ; println 1 " " _1 " " 2

// Wrong arg count
:do 120 | println | back

// Some loops
:do 120 122 | println | back
:do 120 190 | count | println
:do 120 190 | + | println "sum [120, .. ,190]: " _1

// -> function
:-> 5 8 9 155 | println
```

```
// Argument passing
:println 5 | -> 6 | println

// Or
:odd? 99 ; -> 98 | println
:odd? 100 ; -> 98 | println

// Long program!
:* 5 7 & square 12 | odd? _2, println _2 ; * | println

// <G> , ...
:do 994 1000 , odd?, square, println , + , println

// The effects of „,“ vs. „|“
:do 994 1000 | odd?, square, println , + , println
:do 994 1000 | odd?, square, println , + | println
:do 994 1000 | odd?, square, println | + | println
:do 994 1000 | odd?, square | println | + | println
:do 994 1000 | odd? | square | println | + | println


// count
:do 65 59 | odd? | count | println

// A concentrator can't take fixed arguments
// because the count changes from one or more to
// none in the engine status NIL!
:do 101 103 | count _1 | println


// Change the arg count with ->
:do 5 9 | println | * 13 _1 & * 8 _1 | -> _1 | + & count | println

// Some calculation
:do 5 9 | println | * _1 13 & * _1 8 | * | + & count | println

// &
:println 17 | square _1 & square 12 | println _2 " " _1
```

5. Data Types

5.1 Type Hierarchy

Eliza functions can be polymorphic. For reasonable polymorphy checks a type hierarchy is required. Elizas type hierarchy is simple.



The colours designate:

– green: Used in most provided library functions
– orange: Used in some of the library functions
– black: So far not used

5.2 Currently Supported Types

Type	Description	Target implementation
type	Means „any“. Top of the hierarchy. Similar to Java Object or Delphi TObject.	–
int	Integer number. C++ unsigned int.	Arbitrary precision GMP mpz_class.
string	C++ string.	Unicode string.

5.3 All Types

Type	Description	Target implementation
simple	Supertype for all simple types. Will not be used much.	–
number	Number type for int and float. Could be a combined type for int and float,	??

	and maybe fractions.	
float	C++ double. Like: 0.0314e2	Arbitrary precision GMP mpf_class.
regex	Regular expression. Like: '[a-zA-Z]{1,5}[0-9]{4}'	Unicode capable regular expression.
symbol	A text constant like in Scheme or like enums in C/C++.	–
function	A name describing a sort of function that should be accepted by a function as a parameter. To have predictable behaviour the sort of the function must be restrictable.	Maybe the „functiontype“ format like: <namespace>::<parameters>::<function type> example: math::int*int->int::<T>
compound	Compound types that contain one or more simple types or simple values.	–
array	A dynamic array like a deque in C++. Like: [47, 199, 212, 68]	–
dict	A hashtable like a map in C++. Like: [a => 0, b => 1, c => 99]	–
object	Like a C++ struct with public data members. No member functions. Maybe private data members, accessible only from inside of the same namespace. Like: [address: string _name, string _prename, string _street, int _street_nr, int _plz, string _city]	–
symgroup	A group of symbols belonging together like: { time: second, minute, hour, day, week, month, year } To be used in a form like: time:day	

6. Function Reference

6.1 Supported Functions in V0.11

```
$eliza -d
Eliza interpreter V0.11 (Rel. 2008-02-04)
Copyright (C) 2000-2008 Matthias Huerlemann

...
```

Presented here in structured form. All the functions are in version 0.1 in status test and written by the author.

Function	Description
`* int * int -> int <T>`	Multiplies two integers. It produces the result as one integer.
`+ int -> int <C>`	Concentrates one integer at a time. Produces the sum when the engine status is NIL.
`-> (type) -> (type) <T>`	Passes all the arguments it gets.
`back - -> - <C>`	`back` is a special concentrator. It backtracks when the engine status is OK. If the status is NIL it continues. `back` never takes a value or produces one.
`count - -> int <C>`	It counts the number of OKs it gets. At NIL it produces the count as an integer.
`do int * int -> int <G>`	Generates integers from integer a to integer b (including). Ascending (like: `do 1 10`) or descending (like: `do 5 2`).
`odd? int -> int <F>`	Execution passes through an integer a if it is odd (a % 2 = 1). If a is even, the function backtracks.
`print (type) -> (type) <T>`	Prints any type(s) to the console in one line. Arguments are passed as-is to the next function.
`println (type) -> (type) <T>`	Prints any type(s) to the console and adds a newline at the end. Arguments are passed as-is to the next function.
`square int -> int <T>`	Produces the square of an integer a.

6.2 Eliza Errors

<table>
<tr><th>Error message</th><th>Problem</th></tr>
<tr><td>...: Wrong arg count</td><td>The function expects for example 2 arguments, but gets only one.</td></tr>
<tr><td>...: can't resolve parm</td><td>A parameter number is greater than the number of available arguments.
Example (trying to acces argument 4):
<code>:-> 3 6 9 | println _4</code></td></tr>
<tr><td>...: <G> , ... is not allowed</td><td>Since a <G> is there to produce values, but can't pass them to the next function via a „,“, such a program is useless. Example (nothing would be printed):
<code>:do 900 910 , println, back</code></td></tr>
<tr><td>...: 1st parameter has wrong type</td><td>A function receives e.g. a string instead of an int.</td></tr>
<tr><td>...: 2nd parameter has wrong type</td><td>Ditto.</td></tr>
<tr><td>...: parameter has wrong type</td><td>Ditto.</td></tr>
<tr><td>...</td><td></td></tr>
</table>

There will be some more.

6.3 Library Structure (Work in Progress)

The final Eliza builtin library must contain about these groups of functions:

Group	Description
–	The core functions are not in a namespace like println.
math	Math functions in arbitrary precision.
GUI	A simple and powerful user interface like FLTK.
net	All the net functions like send and receive, http, mail, webservices, rpc, maybe CORBA etc.
string	Special string matching and replacement functions.
db	Database access functions. A special API for access to SQLite should be provided including a Eliza-RDB-Mapping. Should include associative databases and knowledge networks.
fuzzy	All kinds of AI functions like fuzzy logic, search heuristics etc.
document	A general type for file handling. It is more general than a FILE type because it can be attributed with metadata and maybe versioned

	and can know about the type of a file content.
datetime	Functions for date and time calculations.

measure	Functions for correct calculations of measures and conversions.
	Something like:
	How deep does a body fall in 4 seconds?
	`:* 9.81 m/s^2 4 s`
	Should produce something like:
	`156.96 m`

graph	All kinds of graphs including search, optimisation and traversal.
rules	A rule framework for rule based solutions.
workflow	Workflow components for segmented applications that are easy to rewire.

7. Dynamic Library API (Concept)

ToDo.

8. Analysis & Design with Eliza

ToDo.

9. Programming Style

ToDo.

10. Open Questions and Concepts

There are a lot of open questions. Some of them are collected here. More to come.

Question	Discussion

Should the libraries be grouped in an object focused way or in a functionally polymorphic way?	Should there be a polymorphic
	`do [ 4, 20, 23478 ]`
	that produces different output according to the type of the input or a specific
	`array.do [ 4, 20, 23478 ]`
	in its own namespace for better searchability?

Should there be private data members to objects?	
Should there be some minimal kind of inheritance?	Eliza should not be an OO-language. But what can Eliza take profit of?
Should there be private objects to a	Should certain objects be hidden in a namespace and be only accessible via functions that are in the same

namespace?	namespace?
Should recursion be supported?	Every problem can be solved iteratively. Eliza should be an interative, sequential declarative language. Is that possible?
Are there problems that can't be solved with Eliza but with other languages?	Is Eliza universal?
What must be in the Eliza core library?	What belongs to a „core“? How bis shall Eliza become. My guess is: not bigger than about 1.5 MB. Is that realistic?
Where could be the main area of usage for Eliza?	AI, testing, configurations, scripting, highly secure applications for aerospace, firewall rule management, data flow analysis, education, ...?
How should Eliza grow now?	
What platforms must be supported and how portable must the code be?	Currently Eliza runs on Linux (SuSE 10.1), Mac OS X (10.5.1, Intel) and Windows XP.
Should there be a bytecode compiler?	Yes.
Should Eliza be set on top of another platform like .Net, Perl6 or Java?	Maybe Perl6?
Are there enough core datatypes?	
What functions should be provided for the core datatypes?	
How are available values not passed to a function that does not want to take any?	`:* 7 9 | println` prints 63. But if I don't want to print anything? Possible solution: `:* 7 9 | println _0` should just print a newline and take none of the provided arguments. Doesn't work right now.
What testing functionality should be provided?	Some function like: `:test „First test:“ [* 4 8], [32]` or `:test „Error: „ [* 9], [„*: wrong arg count“]`
How should multi-threading be i?	With a key word like this? `:threads 4 <function> [<arguments per thread>]`