forward

A stack-based programming language.

Bruno Arias

Version 0.0, Fall 2021: Work in Progress

Table of Contents

[asciidoctor-ghpages] [GitHub repo size] [forward?category=lines] [forward?category=blanks] [forward?category=code] [forward?category=comments] [forward?category=files]

A stack-based programming language. Powerful Enough, Simple, Correct.

Links: - Repo - Website - Ebook - Development Blog - Knowledge Base - Reading list

Edit this page on GitHub

Mission	To create a programming language that is simple enough to be understood fully under a week, but with **powerful enough mental models** to allow for its effective use and application within different problem domains.
Vision	That due to its simplicity it sees widespread use whithin its problem domains, remaining in the background, out of the way, ready to be used. Ubiquitous like the shell, but as effective as a scalpel, not a chainsaw!
Values	• "Correct" mental mapping of models to the problem domains. (Once learnt,) it should stay out of the way, you should focus on solving your problem, not fixing your tools. • Simplicity, through the removal of unnecessities. Being simplex, by choosing to untangle and compose concepts rather than to interleave them. • "Simply Correct", easy to use tools to formally specify and verfiy implementations • Fast (enough), no one likes a blunt, dull tool. Make it sharp, make it fast.

Quotes

From a French writer, poet and aviator:

> In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away, when a body has been stripped down to its nakedness.
>
> — Antoine de Saint Exupéry, Terre des Hommes (1939)

From an abstract expressionist painter:

> The ability to simplify means to eliminate the unnecessary so that the necessary may speak.
>
> — Hans Hofmann

From a computer science professor:

> If you cannot grok the overall structure of a program while taking a shower, you are not ready to code it.
>
> — Richard E. Pattis

From a pioneer in computer science:

> The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> — Edsger W. Dijkstra, The Humble Programmer (1972)

From a software engineer:

> Walking on water and developing software from a specification are easy if both are frozen.
>
> — Edward V. Berard, Essays on object-oriented software engineering (1993)

Chapter 1. Goals

- Literate programming
- Debugging
  - Read Eval Print Loop
- Denotational semantics
- Axiomatic semantics
  - Hoare triples: Pre- and Post-conditions. Possibly also while-conditions aka invariants.
  - Dependent types
- Stupidly Simple Grammar
- Pattern Matching
  - Macros
  - Intuitive Grammar (on layer 2, after macros are applied)
  - Code as Data, but not Data as Code.

1.1. Inspiration sources

Currently being inspired mainly by:

- UI/UX of factor, pharo and especially r4
  - Holistic approach of Lisp Machines, Oberon, Smalltalk, Forth
- Numerical semantics of APL
- Textual semantics of the unix shell.

1.2. Features: minimalism

> At the moment I'm partially trolling here, but not by much...

Expected feature set:

- doesn't have `goto` statements
- doesn't have `if` or `case` statements, implemented by pattern matching instead
- doesn't have `for`,`while`,`until` loops, implemented by recursion instead
- doesn't have objects, implemented by dictionaries and anonymous functions instead
- doesn't have classes, implemented by types instead

(I lied)

- doesn't have pattern matching, implemented by dictionaries instead
- doesn't have recursion, implemention is vectorized already
- doesn't have types, implemented by compile-time pre- and post-conditions instead

Consequences:

- does have a point-free style of programming, leading to concatetative programs that focus on composition
- does have a powerful parsing vocabulary based on the pattern matching abilities
- does have in-built design by contract, using Hoare-triples
- does have hyperoperators, which due to careful design allows reversible computing

Chapter 2. Waterfall process

1. Integration Requirements
2. Feature Design
3. Properties
4. Specification
5. Implementation
6. Verification
7. Maintenance

We are applying the above "Waterfall model" process to each Domain Specific Language, in order to somewhat avoid analysis paralysis and to have something to show. This means that forward as a programming language will grow incrementally, as new vocabularies are added.

Trying to have one Milestone for each step in the process

Chapter 3. Semantics

Some general properties we want are:

- decidability,
- invertability (bijections),
- tractable problem

3.1. Informal denotational definitions

From Maths

- Sets : For SQL databases
- Formal Grammars : For tokenizing, parsing, compiling, transpiling
- Matrixes : For parallelism

From Computer Science

- Dictionary
- Atoms/Elements:
  - Booleans
  - Characters
  - Numbers
- Arrays
  - Bitarrays
  - Strings
  - Vectors
  - Matrixes

3.2. Formal definitions

3.2.1. Grammar

WARNING	work in progress

Whether stack-based/concatenative languages have grammars is a bit unclear, due to 2 main reasons: 1) most, if not all, of the program is made of space-delimited words/tokens 2) the parsing of stack-based languages can often be extended at runtime. So supposedly, due to the aforementioned reasons, forth doesn't really have a grammar, as there is no syntax. [1] [2] [3] [4] On the other hand, factor has clear documentation discussing its syntax. [5] [6]

Ironically, the same claims are made about LISP. [7] Although, LISP very clearly has a syntax. It is probably the first thing people notice. The Common Lisp HyperSpec defines a standard syntax, [8]

and supposedly uses BNF to describe the syntax of its macro forms and special forms. [9] Unfortunately the Common Lisp HyperSpec doesn't seem to be consistent with its use of BNF, as (it seems like) its only the syntax for numeric tokens that is defined with BNF. [10] Luckily, there are other LISPs, such as Shen, which does have a specification of its syntax in BNF. [11] [12] [13]

Properties

We want proof of termination for parsing of the grammar. [14] We also want a non-ambigous grammar. Parsing Expression Grammars, for example, have an ordered choice operator that results in an unambiguous grammar. Last but not least, we want composability of grammars, as each DSL might require its own syntax, and should be able to do so without interupting the surrounding source code.

```
S -> a
```

Specification

Informal defintion: Backus–Naur form

forward's level 1 grammar, (using the BNF-playground based on Nearly):

```
/* limiting the syntax to the ASCII charset */

/* one or more words */
/* one or more characters */
<words>  ::= <word> | <word> <words>
<lowers> ::= <lower> | <lower> <lowers>

/* words have to end with a space */
/* regular words just have lowercase alphabetical characters */
<word>  ::= <lowers> " "

<lower> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" |
"m"
         | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
"z"
```

NOTE	the level 1 grammar doesn't have support for string literals, or lists/arrays, which means writing higher-order functions at this level is non-intuitive.

Formal definition: Formal Grammar

```
Alphabet = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
         | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
"z"
         | " "
```

Executable Specification

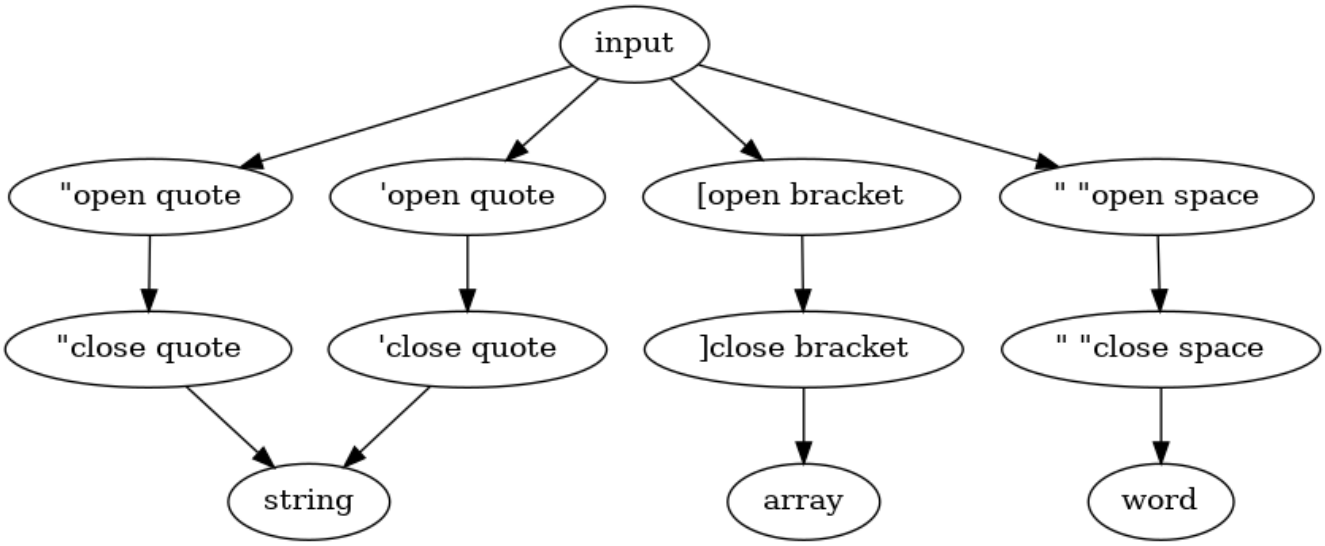
forward's level 1 grammar, (using the scratchpad of jscoq based on coq):

Tools

BNF-playground	https://bnfplayground.pauliankline.com/
Nearly	https://nearley.js.org/ two cool things about nearley: 1. it can output railroad diagrams for documentation, 2. and it can be inverted to form generators which output random strings that match a grammar
scratchpad	https://jscoq.github.io/scratchpad.html
jscoq	https://github.com/jscoq/jscoq
coq	https://coq.inria.fr/

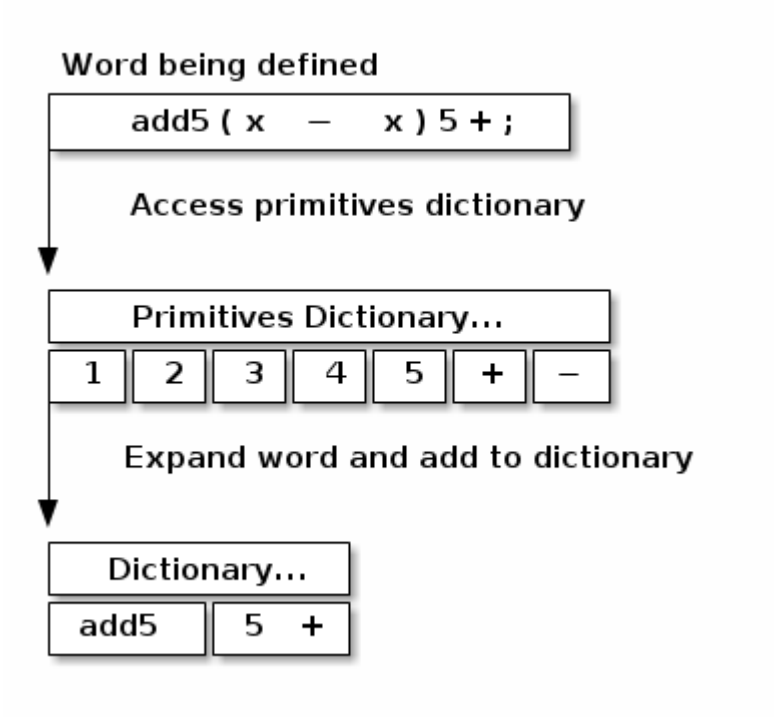
3.2.2. Parser

WARNING	work in progress



3.2.3. Evaluation

WARNING	work in progress



3.2.4. Arithmetic

WARNING	work in progress

5 Whys:

1. *Why postfix?*

   Its easier to parse than infix, and easier evaluate than prefix since all the arguments have been evaluated already.

2. *Why a formal grammar?*

   ...

3. *Why commutative operators?*

   reduces stack shuffling

4. *Why focus on inverses?*

   makes it easier to implement reversible computing

Table 1. binary operations

symbol	**name of binary operation**	**commutative?** [15]
+	addition	yes
×	multiplication	yes

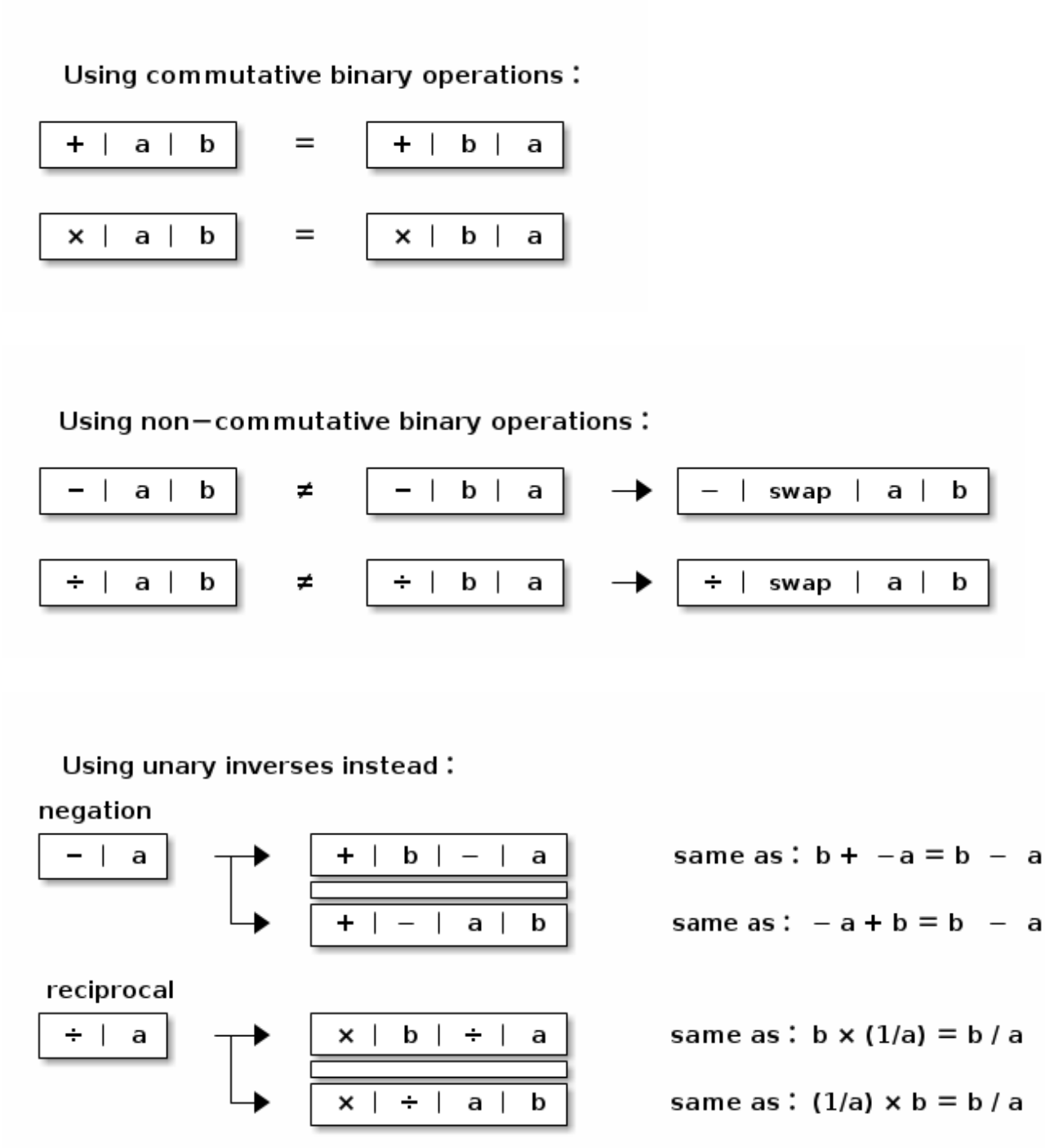
Table 2. unary inverses

symbol	**name of binary operation**	**commutative?** [16]	**name of unary inverse**
–	substraction	no	negation [17]

symbol	name of binary operation	commutative? [16]	name of unary inverse
÷	division	no	reciprocal [18]

Subtraction and division as binary operations aren't commutative, ie `a - b ≠ b - a` and `a ÷ b ≠ b ÷ a`. That's why forward uses the unary operations (which are inverses); so that we can't preserve the commutative property.

Why do we want to preserve the commutative property? Because in stack-based languages, one thing that often bothers me is having to manipulate the stack so that things are in the right order. By having preserving the commutative property, we don't have to worry as much in what order things are on the stack.



Interesting property of inverses:

https://en.wikipedia.org/wiki/List_of_types_of_numbers

- negation "creates" the negative numbers from the Natural numbers □ {1,2,3,4,...}, while
- reciprocal "creates" fractions from the Natural numbers

In other words, these two inverses break the closure property of addition and multiplication (respectively).

Also, multiplication can be thought of as the repetitive application of addition.

```
5 x 3 = 15
0 + 5 + 5 + 5 = 15
```

```
8 x 2 = 16
0 + 8 + 8 = 16
```

The same cannot be said of subtraction and division, ie division is not the repetitive application of subtraction.

```
15 ÷ 5 = 3
15 + 5 + 5 + 5 = 0
```

```
16 ÷ 8 = 2
16 - 8 - 8 = 0
```

However it does correspond to modulo and remainder

[1] https://softwareengineering.stackexchange.com/questions/370518/why-does-forths-flexibility-make-a-grammar-inappropriate-for-it

[2] https://groups.google.com/g/comp.lang.forth/c/nbVrIzbafKM

[3] https://skilldrick.github.io/easyforth/#defining-words

[4] https://users.ece.cmu.edu/~koopman/forth/hopl.html

[5] https://docs.factorcode.org/content/article-parser-algorithm.html

[6] https://docs.factorcode.org/content/article-syntax-literals.html

[7] https://groups.google.com/g/comp.lang.lisp/c/_JYqG712WvU

[8] http://www.lispworks.com/documentation/HyperSpec/Body/02_a.htm

[9] http://www.lispworks.com/documentation/HyperSpec/Body/01_dab.htm

[10] http://www.lispworks.com/documentation/HyperSpec/Body/02_ca.htm

[11] http://shenlanguage.org/osmanual.htm#12%20The%20Syntax%20of%20Shen

[12] http://shenlanguage.org/shendoc.htm#The%20Syntax%20of%20Symbols

[13] http://shenlanguage.org/shendoc.htm#Numbers

[14] https://github.com/coq/coq/wiki/CoqTerminationDiscussion Coq has a termination checker

[15] https://en.wikipedia.org/wiki/Commutative_property

[16] https://en.wikipedia.org/wiki/Commutative_property

[17] https://en.wikipedia.org/wiki/Additive_inverse

[18] https://en.wikipedia.org/wiki/Multiplicative_inverse

Appendix A: References

Hoare triples	https://en.wikipedia.org/wiki/Hoare_logic#Hoare_triple
Waterfall model	https://en.wikipedia.org/wiki/Waterfall_model#Model
Dependent types	https://en.wikipedia.org/wiki/Dependent_type
factor	https://github.com/factor/factor/
APL	https://tryapl.org/
pharo	https://github.com/pharo-project/pharo
r4	https://github.com/phreda4/r4
chainsaw	The Unix shell is sometimes referred to as a chainsaw (see `rm -rf *`)
simplex	https://www.infoq.com/presentations/Simple-Made-Easy/
analysis paralysis	knowledge-base.pdf

Colophon