Building Forth from Spherepop Primitives: Stacks, Continuations, and Recursive Control

Flyxion

August 2026

Abstract

This essay demonstrates that a small Forth compiler can be derived from a compact Spherepop machine rather than implemented as an unrelated collection of parsing and runtime tricks. Starting from only a few structural operations—introduce a value, remove a value, duplicate a reachable value, exchange adjacent positions, compose transformations, and preserve a continuation for later return—the familiar Forth data stack, return stack, dictionary, reverse Polish notation, colon definitions, conditionals, and nested loops emerge as progressively richer organizations of the same primitive state transitions. The central claim is that Forth exposes precisely the machinery Spherepop is designed to make explicit: values are not merely stored but moved through a structured field of stack positions, words are local transformations of that field, and control flow is constructed from continuation, repetition, and return rather than hidden inside a conventional syntax tree.

1 Forth as a Language of Explicit State Transformation

Forth is unusually compatible with Spherepop because it already operates at the level of explicit state transformations. In an ordinary expression language, the source text

$$(2 + 3) * 4$$

contains a syntax tree whose evaluation order must be recovered by the compiler. In Forth, the equivalent expression

$$2\ 3\ +\ 4\ *$$

already specifies a sequence of transformations on a stack:

$$[] \longrightarrow [2] \longrightarrow [2, 3] \longrightarrow [5] \longrightarrow [5, 4] \longrightarrow [20].$$

A Forth word can therefore be represented by a stack effect

$$w : \Sigma \to \Sigma,$$

where Σ is the machine state. More precisely, once multiple stacks are introduced,

$$\Sigma = (D, R, C, M),$$

with D the data stack, R the return stack, C a control or continuation stack, and M the dictionary or memory environment.

The Spherepop interpretation is that a word does not primarily denote a value. It denotes an admissible reconfiguration of reachable positions.

2 The Minimal Spherepop Machine

The Forth stack effects introduced above are not primitive in Spherepop. They are surface notation for compositions of the four primitives the specification treats as final and irreducible:

$$\mathcal{P} = \{\text{Pop}, \text{Refuse}, \text{Bind}, \text{Collapse}\}.$$

Pop commits to a specific option, foreclosing the branches where a different continuation was taken. Refuse documents that an option is no longer admissible without erasing the historical record of it. Bind couples two elements as dependent without identifying them. Collapse observes the accumulated history under a chosen rule, projecting it onto that rule's quotient space. Every other construct in the calculus, and every Forth stack word below, is a fixed composition of these four.

A terminological collision has to be named before it causes confusion: Forth's stack `pop` (remove the top cell and return it) and Spherepop's primitive Pop (commit to one branch of possibility) denote unrelated operations that happen to share a name. Throughout, the capitalized Pop is the Spherepop primitive; lowercase `push`/`pop` are the Forth-level effects they compile down to.

Push as commitment

Pushing x onto the data stack is a commitment. Before the push, many continuations of D remain possible; the push forecloses all but one:

$$\text{push}(x, D) \;:=\; \text{Pop}\big(D \cdot x\big).$$

Stack pop as refusal, not deletion

Because Spherepop's history is append-only, x cannot literally be erased when the Forth machine "pops" it. Consumption is a Refuse: x is marked inadmissible for further reference while the historical fact of its having been pushed remains in H. What the running machine sees is a Collapse of the push/consume log under the rule "cancel each push against its next Refuse and expose only what remains":

$$\text{pop}(D \cdot x) \;:=\; \text{Collapse}_{\text{cancel}}\big(\text{Refuse}(x)\big) = (D, x).$$

The visible data stack at any instant is therefore a quotient view onto an uncollapsed push/refuse log, the same relation the specification gives between an ordinary Merge and the Bind underneath it.

Duplication as coupling

dup introduces a second position whose value is dependent on the first rather than independently constructed:

$$\mathrm{dup}(D \cdot x) \;:=\; \mathrm{Pop}\Big(D \cdot x \cdot \mathrm{Bind}(x, x')\Big) = D \cdot x \cdot x'.$$

x' is bound to x, not identified with it, which is why an implementation with structural sharing can let the two cells share storage while the calculus still treats them as distinct positions.

Swap as collapse under a permutation rule

swap needs no primitive of its own. It is a Collapse of a Bound pair under the rule that exchanges their order:

$$\mathrm{swap}(D \cdot x \cdot y) \;:=\; \mathrm{Collapse}_{\pi}\big(\mathrm{Bind}(x, y)\big) = D \cdot y \cdot x,$$

with π the transposition rule. `over`, `rot`, and `nip`, taken up in the next section, are Collapse under other index permutations of the same Bound prefix.

Sequencing as monotone growth

Composition needs no primitive either; it is the append-only growth of the history itself:

$$(f; g)(\Sigma) = g(f(\Sigma)) \quad \Longleftrightarrow \quad H \mapsto H \cdot e_f \cdot e_g.$$

Where two independently constructed word bodies are stitched together rather than run in sequence on a single machine, that stitching is Meld, the monoidal composition of histories, rather than ordinary sequencing.

Call and return as Bind and Collapse

`call` couples the current instruction pointer to a return address without identifying them, then commits to entering the callee:

$$\mathrm{call}(a, \Sigma) = \mathrm{Pop}\Big(\mathrm{Bind}(ip, a)\Big) = \Sigma[R \mapsto R \cdot ip,\ ip \mapsto a].$$

`return` resolves that dependency by collapsing the Bound pair under the substitution rule that recovers the caller's instruction pointer:

$$\mathrm{return}\big(\Sigma[R = R' \cdot a]\big) = \mathrm{Collapse}_{c_{\mathrm{subst}}}\big(\mathrm{Bind}(ip, a)\big) = \Sigma[R \mapsto R',\ ip \mapsto a].$$

This is exactly the specification's canonical encoding of application, Application := $\text{Collapse}_{c_{\text{subst}}}(\text{Bind}(u, x)$ a Forth `call`/`return` pair is that same Bind/Collapse schema applied to control rather than data.

This establishes the essay's basic reduction: a Forth virtual machine is a Spherepop transition system built from exactly four primitives, whose stacks encode distinct forms of reachability, and whose familiar vocabulary (`push`, `pop`, `dup`, `swap`, `call`, `return`) names derived compositions rather than independent operations.

3 Reverse Polish Notation as Linear Composition

This section goes beyond a standard explanation of postfix notation. The interesting Spherepop point is that reverse Polish notation converts a tree-shaped dependency structure into a linearized sequence of admissible stack transitions.

For an operator f of arity two,

$$f : A \times B \to C,$$

its stack interpretation is

$$\widehat{f} : D \cdot a \cdot b \longrightarrow D \cdot f(a, b).$$

Thus the token sequence

$$a \ b \ f$$

is not merely an alternative syntax for $f(a, b)$. It is a serialized proof that the operands become available before the consuming operation occurs.

This lets us characterize stack underflow as a failed admissibility condition. For example,

$$+ : D \cdot n_1 \cdot n_2 \to D \cdot (n_1 + n_2)$$

is defined only when the data stack contains two compatible numerical values. A compiler can therefore associate each word with a symbolic stack signature:

`+`	(n_1 n_2 – n_3)
`dup`	(x – x x)
`swap`	(x y – y x)
`drop`	(x –)
`over`	(x y – x y x)

These signatures function as local proofs of stack-shape preservation.

4 Implementing the Data Stack

Several concrete stack representations can be compared.

4.1 Fixed Contiguous Stack

A fixed contiguous stack uses an array and stack pointer:

```
struct DataStack {
    cells: Vec<Cell>,
    sp: usize,
}
```

Conceptually,

$$D = (A, sp),$$

with push and pop defined by

$$A[sp] \leftarrow x, \qquad sp \leftarrow sp + 1,$$

and

$$sp \leftarrow sp - 1, \qquad x \leftarrow A[sp].$$

4.2 Linked Stack

A linked stack represents every pushed value as a node:

$$D = x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_n.$$

It makes persistence and structural sharing easier, but introduces allocation and pointer overhead.

4.3 Persistent Stack

A persistent stack is particularly relevant to Spherepop:

$$\text{push}(x, D) = \text{Node}(x, D).$$

Older states remain reachable, so a branch can preserve a prior stack without copying its entire contents. This makes speculative execution, rollback, debugging, and transactional words easier to describe. It is also the most literal implementation of the ontology's point about consumption from §2: a Forth `pop` was never a deletion, only a Refuse layered over an intact push, so a persistent representation simply stores that fact directly instead of hiding it behind reused memory.

4.4 Segmented Stack

A segmented stack combines contiguous speed with controlled growth. Each segment is a bounded region, and segment boundaries become explicit continuation or allocation points.

The essay should not merely compare performance. It should explain that each representation instantiates a different geometry of reachability.

5 The Return Stack and Control Stack

Forth traditionally separates the data stack from the return stack. Spherepop gives a principled explanation for the distinction.

The data stack contains values available to the current computation:

$$D = [d_0, \ldots, d_n].$$

The return stack contains suspended execution contexts:

$$R = [r_0, \ldots, r_m].$$

A call transfers the current instruction pointer into R, then enters the callee:

$$(ip, R) \mapsto (a, R \cdot ip).$$

A return reverses that transfer:

$$(ip, R \cdot r) \mapsto (r, R).$$

We may also introduce a third logical stack for structured loops and exception-like control:

$$C = [c_0, \ldots, c_k].$$

Although a practical implementation might store loop frames on the return stack, separating them analytically is useful. A call frame and a loop frame have different admissibility rules even when they share physical storage.

A call frame might be

$$\text{CallFrame}(ip),$$

whereas a counted-loop frame could be

$$\text{LoopFrame}(i, \ell, e),$$

with current index i, limit ℓ, and exit address e.

6 Dictionary Structure and Word Compilation

A Forth dictionary entry can be modeled as

$$W = (n, \tau, a, \kappa),$$

where n is the name, τ is the stack-effect signature, a is the code or threaded address, and κ identifies whether the word executes immediately or is compiled.

The dictionary becomes a mapping

$$\Gamma : \text{Name} \rightharpoonup W.$$

During interpretation, a token is processed by the rule

$$\mathrm{eval}(t, \Sigma) = \begin{cases} \mathrm{push}(\mathrm{number}(t), \Sigma), & t \text{ is a literal}, \\ \mathrm{execute}(\Gamma(t), \Sigma), & t \in \mathrm{dom}(\Gamma), \\ \mathrm{error}, & \text{otherwise}. \end{cases}$$

During compilation, ordinary words are appended to the current definition rather than immediately executed:

$$B \leftarrow B \cdot \mathrm{xt}(\Gamma(t)),$$

where B is the body under construction and xt is the execution token.

A colon definition such as

```
: square dup * ;
```

can be compiled into

```
CALL dup
CALL *
RETURN
```

or into a threaded representation:

$$[\mathrm{xt}(\texttt{dup}), \mathrm{xt}(\texttt{*}), \mathrm{xt}(\texttt{exit})].$$

The Spherepop interpretation is that a new word packages a sequence of primitive pops and pushes into a reusable higher-order transition.

7 Stack Words as Derived Spherepop Operations

This section derives the standard Forth stack vocabulary rather than simply listing it.

For example:

$$\mathrm{drop} = \mathrm{pop},$$
$$\mathrm{nip}(D \cdot x \cdot y) = D \cdot y,$$
$$\mathrm{over}(D \cdot x \cdot y) = D \cdot x \cdot y \cdot x,$$
$$\mathrm{rot}(D \cdot x \cdot y \cdot z) = D \cdot y \cdot z \cdot x.$$

These can be treated as permutations and contractions over a finite stack prefix. A general stack shuffle can be described by an index map

$$\pi : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\},$$

so that

$$[x_0, \ldots, x_{n-1}] \mapsto [x_{\pi(0)}, \ldots, x_{\pi(m-1)}].$$

Unlike a pure permutation, π may repeat indices, corresponding to duplication, or omit indices, corresponding to deletion. Standard stack words then become a small algebra of finite positional transformations.

8 Conditionals as Deferred Continuations

A conditional can be compiled as a branch over two possible continuations.

For

$$\texttt{flag IF true} - \texttt{part ELSE false} - \texttt{part THEN}$$

the runtime behavior is

$$\text{branch}(b, k_t, k_f) = \begin{cases} k_t, & b \neq 0, \\ k_f, & b = 0. \end{cases}$$

Compilation emits unresolved branch targets and repairs them once their destinations become known. This is an especially natural place to use Spherepop's repair language.

The branch itself is an instance of the specification's Choice encoding, Choice := Pop(A) ∥ Refuse(B). Taking the true branch commits, via Pop, to k_t, while k_f is Refused rather than erased, which is why the untaken branch's code and its unresolved-jump placeholder remain fully recoverable from the history even after compilation moves on.

When IF is encountered, the compiler emits a placeholder:

```
JUMP_IF_FALSE ?
```

and pushes its address onto the control-flow stack:

$$C \leftarrow C \cdot p_{\text{if}}.$$

At ELSE, it emits a second unresolved jump, repairs the first placeholder to point to the false branch, and replaces the control-stack entry:

$$C \cdot p_{\text{if}} \mapsto C \cdot p_{\text{else}}.$$

At THEN, the remaining placeholder is repaired to the current code address.

Thus control-flow compilation is not simply code emission. It is monotonic construction followed by local address repair.

9 Loops as Recursive Continuation Frames

The essay's most distinctive section treats loops as explicit recursive continuation structures.

A simple indefinite loop

```
BEGIN
    body
AGAIN
```

compiles to

$$L : \text{body}; \text{jump}(L).$$

Its denotation is the recursive equation

$$K = \text{body}; K.$$

A conditional loop

```
BEGIN
    body
UNTIL
```

has the denotation

$$K = \text{body}; \text{popflag}; \begin{cases} \text{return}, & \text{flag true}, \\ K, & \text{flag false}. \end{cases}$$

A counted loop such as

```
10 0 DO
    I .
LOOP
```

creates a loop frame

$$F = (i, \ell, k),$$

where i is the current index, ℓ the limit, and k the continuation at the loop body.

Each iteration performs

$$(i, \ell, k) \mapsto \begin{cases} (i+1, \ell, k), & i+1 < \ell, \\ \text{exit}(F), & i+1 \geq \ell. \end{cases}$$

The loop is recursive because its continuation refers back to the same body entry point. It is safe because the evolving index and limit are stored in an explicit frame rather than hidden in the host-language call stack.

In Spherepop terms the loop frame is a Bind of the body's continuation to itself: each iteration is not a fresh commitment but the same coupling observed again under an advancing index. This is the concrete case behind the specification's claim that ordinary concurrency and repetition are already expressible through Bind alone, with no separate looping primitive and no need for Meld.

10 Nested Recursive Loops

Nested loops can be represented as a stack of loop frames:

$$C = [\ldots, F_{\text{outer}}, F_{\text{inner}}].$$

The word I reads the index from the topmost frame:

$$\texttt{I}(C \cdot F_i) = i.$$

The word J reads the next enclosing frame:

$$\texttt{J}(C \cdot F_j \cdot F_i) = j.$$

A nested program such as

```
3 0 DO
    4 0 DO
        J I . .
    LOOP
LOOP
```

should be traced. The crucial point is that inner-loop termination pops only the inner frame:

$$C \cdot F_{\text{outer}} \cdot F_{\text{inner}} \longrightarrow C \cdot F_{\text{outer}}.$$

Execution then resumes at the continuation stored by the outer frame.

We can go further and define a recursive combinator:

$$\text{loop}(p, b, k) = \begin{cases} k, & \neg p(\Sigma), \\ b; \text{loop}(p, b, k), & p(\Sigma). \end{cases}$$

Nested loops are then nested fixed points:

$$\text{loop}(p_o, b_o; \text{loop}(p_i, b_i, k_i), k_o).$$

This connects practical Forth compilation with the more formal account of recursive reachability in Spherepop.

11 Direct, Indirect, and Subroutine Threading

The compiler section compares the principal Forth execution models.

With *indirect threading*, a word body contains addresses of execution tokens, and each execution token points to code:

$$ip \to xt \to code.$$

With *direct threading*, the word body points directly to code fields:

$$ip \to code.$$

With *subroutine threading*, the body is emitted as native calls:

```
call dup
call multiply
ret
```

Spherepop can describe each as a different number of reachability transitions between the current instruction and the executable transformation. Indirect threading adds one additional pop-like dereference. Direct threading contracts that layer. Subroutine threading delegates continuation management to the processor's call stack.

12 Static Stack-Effect Checking

The essay becomes much stronger if it includes a lightweight compile-time checker.

Suppose every word has a type

$$w : \alpha \cdot I \longrightarrow \alpha \cdot O,$$

where α is an unknown stack prefix, I is the required input suffix, and O is the produced output suffix.

For example,

$$\texttt{dup} : \alpha \cdot x \to \alpha \cdot x \cdot x,$$

$$\texttt{+} : \alpha \cdot n \cdot n \to \alpha \cdot n.$$

Composition is permitted when the output suffix of one word unifies with the input suffix of the next. Therefore,

```
dup *
```

has the derived effect

$$\alpha \cdot n \xrightarrow{\texttt{dup}} \alpha \cdot n \cdot n \xrightarrow{\texttt{*}} \alpha \cdot n.$$

Consequently,

```
: square dup * ;
```

receives the signature

$$\texttt{square} : (n \ -- \ n).$$

A malformed definition such as

```
: bad drop + ;
```

fails because after drop, the compiler cannot prove that two numerical operands remain available.

This makes the Spherepop compiler not merely an executor but an admissibility checker for stack trajectories.

13 A Small Compiler Architecture

The implementation can be divided into four transformations, and Spherepop gives them their natural description as stages of one growing history rather than as four unrelated compiler passes:

$$H_0 \xrightarrow{\text{lex}} H_1 \xrightarrow{\text{bind}} H_2 \xrightarrow{\text{collapse}} H_3.$$

Lexing performs $H_0 \to H_1$ by committing, via Pop, to a symbolic token for each lexeme in the source (the tokenizer itself need only distinguish names, numbers, and delimiters); nothing about a token's meaning is decided yet, only that this token exists at this position. Dictionary lookup performs $H_1 \to H_2$ by Binding each name token to the continuation

its dictionary entry denotes, coupling the two without yet resolving what executing that continuation will do. Code generation performs $H_2 \to H_3$ by Collapsing only as much of the Bound history as is needed to obtain concrete threaded code: an immediate word Collapses at once, while a forward reference such as an unresolved IF/ELSE placeholder stays Bound until the surrounding structure is complete, at which point its repair is itself a further, later Collapse under the address-resolution rule. Execution is not a fifth stage; it is the ordinary replay of H_3.

The interpreter/compiler mode

$$m \in \{\mathsf{Interpret}, \mathsf{Compile}\}$$

selects which Collapse rule is active at $H_2 \to H_3$: under Interpret, each Bound token is Collapsed immediately into an executed effect; under Compile, the Collapse is deferred and the token's execution token is instead appended to the body under construction, itself a further Bind rather than a Collapse.

A minimal machine state could be expressed as:

```
struct Machine {
    data: Vec<Cell>,
    returns: Vec<ReturnFrame>,
    control: Vec<ControlFrame>,
    dictionary: Dictionary,
    code: Vec<Instruction>,
    ip: usize,
    mode: Mode,
}
```

The instruction set could remain very small:

```
enum Instruction {
    Push(Cell),
    Primitive(Primitive),
    Call(usize),
    Jump(usize),
    JumpIfZero(usize),
    Return,
}
```

Nested recursion does not require a special recursive instruction. It follows from Call, Return, and the ability to place a word's own execution token in its body.

For example:

```
: countdown
    dup .
    1 -
    dup 0 >
    IF countdown
    ELSE drop
    THEN
;
```

is compiled as an ordinary self-reference. At runtime, each recursive call pushes a continuation onto the return stack. The recursion is therefore a dynamically growing path of suspended resumptions.

14 The Broader Spherepop Interpretation

The conclusion argues that Forth clarifies several general Spherepop ideas.

A value exists operationally when it occupies a reachable stack position. A word is an admissible transformation of those positions. A stack effect is a local contract over reachability. A call suspends one continuation and activates another. A return pops a preserved future. A loop is a continuation repaired so that it points backward. A nested loop is a stack of recursively maintained futures. Compilation is the progressive construction of a reachable code graph, including temporary holes whose targets are repaired once the surrounding structure becomes known.

The recurring picture beneath all of these translations is that nothing Forth calls removal is actually erasure. A dropped value, a taken branch, an exited loop: each survives in the history as a Refused option rather than a subtracted one, and the running machine's view of what remains is a Collapse of that history, not the history itself.

This gives us a clean final formulation:

Forth compilation is the construction of executable reachability.

The machine does not begin with expressions and reduce them into stack operations. It begins with stack transformations, and expressions emerge as stable compositions of those transformations. Spherepop supplies the primitive geometry; Forth supplies the executable language.