Homespring Updated Language Standard

Cal Henderson, Joe Neeman & Jeff Binder

April 18, 2026

1 Introduction

1.1 Slogan

"Because programming isn't like a river, but it damn well ought to be."

1.2 History

The original version of this document was included with Jeff Binder's Homespring interpreter and contained many omissions and ambiguities. A revised version was created by Joe Neeman in conjunction with his improved interpreter. This version was created by Cal Henderson to correct a handful of mistakes and ambiguities. The language documented here complies with the behaviour documented in the official language standard. It also correctly runs all examples provided with the official interpreter. However, it does *not* proscribe the same behaviour as the official interpreter in every case. That is, the documented behaviour of this standard does not comply with the undocumented behaviour of the official interpreter except to the extent that it was necessary in order for the official examples to work.

A Homespring interpreter that matches this documentation is included with this document.

1.3 Motivation

One of the problems with current programming languages is that they're too abstract. Although they frequently use metaphors to explain their concepts to users, these metaphors do not hold up very well in the long run. Enter Homespring, or Hatchery Oblivion through Marshy Energy from Snowmelt Powers Rapids Insulated but Not Great. It is also sometimes referred to as HOtMEfSPRIbNG.

1.3.1 Revolution Information

So what we have here is a new programming paradigm: Metaphor Oriented Programming, or MOP. MOP languages are built around a unified metaphor, and

stick rigorously to its real-world properties and limits. This allows languages to be created that are both high-level and simple, offering exciting new abstractions and ideas that are familiar as they are powerful. As such, Homespring disposes of outmoded concepts such as classes, sequential execution, evaluation, assignment, binding, variables, numbers, and calculations.

1.3.2 Consequences of Failure to Learn

The Homespring language is the archetype of MOP, and it shows off all aspects of this revolutionary new concept. Learn it now or be left behind! Your current favourite language stands no chance! Now it is time to learn HOtMEfSPRIbNG, your next favourite language!

2 Lexical Structure

Before we get into the wonderful new concepts that you are impatiently awaiting, we must discuss Homespring's soon-to-be highly influential lexical structure.

2.1 Tokens

Homespring has exactly one (1) types of tokens: tokens. This simplicity will be greatly appreciated, once you try it. Tokens consist of zero (0) or a number greater than zero (> 0) of non-whitespace characters, separated by one (1) character of whitespace.

Many inferior languages include highly complex escape sequences and quoting rules. For example, how is one expected to remember that the 'n' in
n stands for 'newline', when the inept designers who thought of this could just as well have chosen 'e', 'w', or any of the other character in that word? Homespring's system is far superior, as well as being intellectually stimulating.

2.1.1 Escaping

Homespring offers one (1) meta-character, namely, the period ('.'). To include a newline[1] or a space in a token, precede it with a period. To include a period in a token, precede it with a space. It is not possible to start a token with a period. If a period precedes any character other than a space or a newline, it terminates the previous token (if any) and produces a blank token. The use of tabs is discouraged, as it is not possible in HOtMEfSPRIbNG.

2.1.2 Paradox

The sequences ' . ' and '. .' are required to cause a causality paradox in all conforming implementations. As such, there are no conforming implementations.

[1] A newline character causes a token to end even if it is escaped. Therefore it is impossible to include a newline except at the end of a token.

2.1.3 Example

Although Homespring's lexical rules are so simple that you don't need an example, one is provided anyway as a service to our customers. The following sequence:

Hello,. World ..

is interpreted as:

```
(Hello, )()(World.
)
```

Note the conveniently easy to add blank token.

3 Syntax

Homespring disposes of the outdated notion of syntax, taking the burden of program design off the shoulders of the programmer and putting it nowhere in particular.

4 Program structure

4.1 Inferiority of Other Approaches

In a bold and dynamic move, Homespring has only a single structure which is used by all programs. In the traditional languages which you are now free from and will never have to use again, you would waste most of your valuable time creating a structure for your program which does what you want it to do. HOtMEfSPRIbNG liberates you from this, allowing you to spend your time in a few mega-productive fits of work, and the get back to slacking off. You see, with Homespring you simply use the language's built-in structure, and come up with a way to force it to do what you want. The superiority of the approach is so obvious that it need not be mentioned.

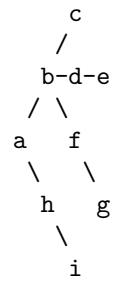
4.2 The Ideal Approach

The tokens of a Homespring program are automatically formed into a tree structure, with the first token as the root, and the rest added one branch at a time. Blank tokens are used to jump up in the tree. A blank token that would traverse tree-building beyond the root node adds a blank token as a child of the root token instead.

By these simple rules, the program

```
a b c  d e   f g    h i
```

is parsed into this tree:



Remember that the outmoded concept of indentation is not present in Homespring, since two spaces does not have the same meaning as one space. This allows you to avoid worrying about program style and focus on what programming is really about, the reproductive behaviour of salmon.

A program with no tokens obviously can't be treated normally. Such a program will, as expected, print the message:

```
In Homespring, the null program is not a quine.
```

and exit.

5 The River Paradigm

Homespring uses the paradigm of a river to create its astoundingly user-friendly semantics. Each program is a river system which flows into the watershed (the terminal output). Information is carried by salmon (which represent string values), which swim upstream trying to find their home river. Terminal input causes a new salmon to be spawned at the river mouth; when a salmon leaves the river system for the ocean, its value is output to the terminal. In this way, terminal I/O is neatly and elegantly represented within the system metaphor.

The river is represented in an n-ary tree structure[2] Each node in the tree is associated with

1. a name
2. a list of salmon present at that node
3. a power state
4. a water state
5. a snow state
6. a destroyed state

[2]In the HOtMEfSPRIbNG reference implementation, the parser allows for n-ary trees but the interpreter only runs the first two children of each node. This implementation runs over the full n-ary tree.

where each of the "states" in the above list is a boolean value. The name of the node determines its behaviour; a list of reserved names is in section 7.2. Every node whose name is not a reserved name is a spring: it creates water.

Each salmon in the river system is associated with

1. a name
2. an age (young or mature)
3. a direction (upstream or downstream)

6 Program flow

The execution of a HOtMEfSPRIbNG program consists of several distinct stages, called "ticks". The order of ticks is as follows:

1. snow tick
2. water tick
3. power tick
4. fish tick
5. miscellaneous tick
6. input tick

6.1 Snow tick

In the snow tick, the snow state of each node is updated. A node becomes snowy if it is not currently blocking snowmelts and if one of its children is snowy. The snow tick is propagated in a pre-order fashion.

Certain nodes will be destroyed when snowmelt reaches them. A node that is destroyed loses its abilities, but keeps its name.

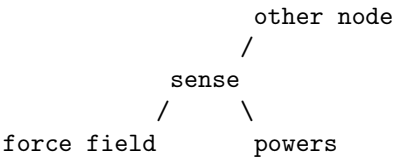
6.2 Water tick

In the water tick, the water state of each node is updated. A node becomes watered if it is not currently blocking water and if one of its children is watered. The water tick is propagated in a pre-order fashion.

6.3 Power tick

Unlike the snow and water ticks, the power tick does not calculate the power state of each node; it merely calculates, for each node, whether that node should *generate* power. The power state of each node is calculated on demand by checking for powered children.

The significance of this difference lies in the following example:



Suppose there is a young, upstream salmon at `force field` and a mature, downstream salmon at `other node`. During the down fish tick (see next section), the mature fish will move down to `sense`. This will immediately block electricity to `force field`, allowing the young fish to move up to `sense` in the fish tick up. If the power state of each node was calculated only during the power state, the behaviour of this example would be different (the young salmon would be stuck until the next turn).

6.4 Fish tick

The fish tick is divided into 3 stages:

6.4.1 Fish tick down

This part of the fish tick only affects downstream salmon. Each downstream salmon is moved to the parent of its current node if it is not blocked from doing so. If its current node is the mouth of the river, the salmon is removed from the river system and its name is printed to the terminal.

This tick propagates in a pre-order fashion.

6.4.2 Fish tick up

This part of the fish tick only affects upstream salmon. For each upstream salmon, an in-order search of the river system is conducted in order[3] to find a river node with the same name as the salmon. If the current node matches the salmon name, it spawns at the current node. If there is such a node and the salmon is not prevented from moving towards it, the salmon moves towards that node. If there is no such node or if the salmon is prevented from moving towards that node, the salmon will attempt to move (in order) to each child of the current node. If the salmon cannot move to any child of the current node or if there are no children of the current node, the salmon will spawn at the current node.

When a salmon spawns, it becomes mature and its direction becomes downstream. Spawning happens only when the upstream salmon is ready to leave the node; on the second tick after entering a shallows or rapids node. A new salmon is created at the current node. The new salmon is young, downstream and its name is the name of the current node.

This tick propagates in a post-order fashion.

[3]heh heh. Sorry.

6.4.3 Fish tick hatch

This tick activates hatcheries. It propagates in a pre-order fashion.

6.4.4 Miscellaneous tick

All other nodes that need to perform some action perform it in this tick, which propagates in a pre-order fashion.

6.4.5 Input tick

If any input is available on the terminal, an upstream, mature fish is created at the mouth of the river with the input text as its name.

6.4.6 Additional notes

When a fish is added to a node, it is added at the head of that node's salmon list. Therefore, if a list of salmon move in unison, its order is reversed at every step.

7 Reference

7.1 Terminology

We say a node "blocks water" if water is prevented from entering it.

We say a node "blocks snow" if snow is prevented from entering it.

We say a node "blocks salmon" if salmon are prevented from *leaving* it. We say a node "very blocks" salmon if salmon are prevented from entering it.

We say a node "blocks power" if it is considered to be unpowered even if it has a child which is powered.

7.2 Node Reference

This is a list of all the node types that can be located on rivers.

7.2.1 hatchery

When powered, creates a young, upstream salmon named "homeless". Operates during the fish tick hatch step. Can be destroyed by snowmelt.

7.2.2 hydro power

Creates electricity when watered. Can be destroyed by snowmelt.

7.2.3 snowmelt

Creates a snowmelt at the end of each snow tick.

7.2.4 shallows

Mature salmon take two turns to pass through.

7.2.5 rapids

Young salmon take two turns to pass through.

7.2.6 append down

For each downstream salmon that did not arrive from the first child, destroy that salmon and append its name to each downstream salmon that did arrive from the first child.

7.2.7 bear

Eats mature salmon.

7.2.8 force field

Blocks water, snowmelt and salmon when powered.

7.2.9 sense

Blocks electricity when mature salmon are present.

7.2.10 clone

For each salmon, create a young, downstream salmon with the same name.

7.2.11 young bear

Eats every other mature salmon (the first mature salmon doesn't get eaten, the second one does, etc.). Young salmon are moved to the beginning of the list because they don't have to take the time to evade the bear.

7.2.12 bird

Eats young salmon.

7.2.13 upstream killing device

When powered and if it contains more than one child, kills all the salmon in the last child.

7.2.14 waterfall

Blocks upstream salmon.

7.2.15 universe

If destroyed by a snowmelt, the program terminates. The program is terminated in the miscellaneous tick following the snow tick in which the Universe is destroyed.

7.2.16 powers

Generates power.

7.2.17 marshy

Snowmelts take two turns to pass through.

7.2.18 insulated

Blocks power.

7.2.19 upstream sense

Blocks the flow of electricity when upstream, mature salmon are present.

7.2.20 downstream sense

Blocks the flow of electricity when downstream, mature salmon are present.

7.2.21 evaporates

Blocks water and snowmelt when powered.

7.2.22 youth fountain

Makes all salmon young.

7.2.23 oblivion

When powered, changes the name of each salmon to “”. Can be destroyed by snowmelt.

7.2.24 pump

Very blocks salmon unless powered.

7.2.25 range sense

Blocks electricity when mature salmon are here or upstream.

7.2.26 fear

Very blocks salmon when powered.

7.2.27 reverse up

For each downstream salmon that arrived from the second child, move it to the first child unless it is prevented from moving there (and change its direction to upstream).

7.2.28 reverse down

For each downstream salmon that arrived from the first child, move it to the second child unless it is prevented from moving there (and change its direction to upstream).

7.2.29 time

Makes all salmon mature.

7.2.30 lock

Very blocks downstream salmon and blocks snowmelt when powered.

7.2.31 inverse lock

Very blocks downstream salmon and blocks snowmelt when not powered.

7.2.32 young sense

Blocks electricity when young salmon are present.

7.2.33 switch

Blocks electricity unless mature salmon are present.

7.2.34 young switch

Blocks electricity unless young salmon are present.

7.2.35 narrows

Very blocks salmon if another salmon is present.

7.2.36 append up

For each downstream salmon that did not arrive from the first child, destroy that salmon and append its name to each upstream salmon.

7.2.37 young range sense

Blocks electricity when young salmon are here or upstream.

7.2.38 net

Very blocks mature salmon.

7.2.39 force down

For each downstream salmon that arrived from the first child, move it to the second child unless it is prevented from moving there.

Also blocks upstream salmon from moving to the last child.

7.2.40 force up

For each downstream salmon that arrived from the second child, move it to the first child unless it is prevented from moving there.

Also blocks upstream salmon from moving to the first child.

7.2.41 spawn

When powered, makes all salmon upstream spawn.

7.2.42 power invert

This node is powered if and only if none of its children are powered. Can be destroyed by snowmelt.

7.2.43 current

Very blocks young salmon.

7.2.44 bridge

If destroyed by snowmelt, blocks snowmelt and water and very blocks salmon.

7.2.45 split

Splits each salmon into a new salmon for each letter in the original salmon's name. The original salmon are destroyed. The new salmon are added to the bottom of the list.

7.2.46 range switch

Blocks electricity unless mature salmon are here or upstream.

7.2.47 young range switch

Blocks electricity unless young salmon are here or upstream.

8 Examples

The first example program is the simplest useful Homespring program:

This program is similar to the cat utility, but it doesn't print the newlines like cat irrationally does. Here is a version of the inferior old cat utility:

.

Here are several possible implementations of the important and useful UNIX utility 'hello'. This is the simplest possible one:

```
Universe␣bear␣hatchery␣Hello.␣World!.
␣Powers␣␣␣marshy␣marshy␣snowmelt
```

This is the same program written in professional style, with a more cohesive sentence structure:

```
Universe␣of␣bear␣hatchery␣says␣Hello.␣World!.
␣It␣␣␣powers␣␣␣␣␣the␣marshy␣things;
the␣power␣of␣the␣snowmelt␣overrides.
```

Here's the alternative, more complicated and less efficient preferred method:

```
Universe␣of␣marshy␣force.␣Field␣sense
shallows␣the␣hatchery␣saying␣Hello,.␣World!.
␣Hydro.␣Power␣spring␣␣sometimes;␣snowmelt
␣␣␣␣␣␣powers␣␣␣snowmelt␣always.
```

This is the somewhat less common but still often useful, "Hi. What's your name? Hi, xxx!" program.

```
Universe␣marshy␣now.␣The␣marshy␣stuff␣evaporates␣downstream.␣Sense␣rapids
upstream.␣Killing.␣Device␣downstream.␣Sense␣shallows␣and␣say␣Hi,.
␣␣␣That␣powers␣the␣␣␣␣␣force.␣Field␣sense␣shallows␣hatchery␣power.
Hi␣..␣What's.␣your.␣name?.
␣␣Hydro.␣Power␣spring␣␣when␣snowmelt␣then␣␣␣␣␣␣␣powers
␣␣␣␣insulated␣bear␣hatchery␣!.
␣Powers␣felt;␣␣␣␣␣␣␣powers␣feel␣␣␣␣␣snowmelt␣themselves.
```

This program tests whether the user knows what six times four is, and get this: the *program* knows what six times four is!

Universe␣alive␣with␣youth.␣Fountain␣bear␣Marshy
evaporates␣downstream.␣Sense␣rapids
upstream.␣Killing.␣Device␣downstream.␣Sense␣shallows␣you.␣lie!.
␣Powers␣␣␣force.␣Field␣sense␣shallows␣the␣hatchery␣but
what's.␣six.␣times.␣four?.
␣␣Hydro.␣Power␣spring␣␣with␣snowmelt␣which␣has
␣␣␣␣␣␣␣powers␣enough.
␣␣␣␣␣␣␣␣It␣powers␣␣␣␣snowmelt␣at␣least.
␣␣␣␣␣␣␣Marshy␣lock␣upstream.␣Sense␣bear␣now.
24␣␣powers␣drive␣␣␣snowmelt␣away.
␣␣␣Insulated␣bear␣hatchery␣time,␣rightyo!.
␣HYDRO.␣Power␣spring␣␣with␣snowmelt␣first.

This extremely powerful program can actually add two arbitrary digits together, in only twenty seconds or so on a fast machine!:

```
Universe␣is␣marshy␣but␣evaporates␣downstream.␣Sense␣the␣rapids␣reverse.␣Down
bridge␣is␣now␣marsh:
Marshy␣marshy␣marshy␣marshy␣marshy␣marshy␣marshy␣marshy␣marshy␣marshy␣now.
All␣evaporates␣downstream.␣Sense
the␣rapids␣now:
Rapids␣rapids␣rapids␣rapids␣rapids␣rapids␣rapids␣rapids␣sensed.
Ugh␣+.
␣Take␣powers␣from␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣snowmelt␣␣therefore;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣the␣␣␣current␣time␣is␣of␣youth.␣Fountain␣is␣young.␣Bear␣cannot
reverse.␣Down␣inverse.␣Lock␣young.␣Switch␣young.␣Range.␣Switch␣clone␣to␣the
switch␣itself.␣Now␣inverse.␣Lock␣narrows␣down:
␣␣␣␣Powers
␣␣␣␣␣␣␣␣to␣␣␣append.␣Up␣go␣all␣young.␣Bear␣time␣evaporates
then.␣Therefore:
Spawn␣power.␣Invert␣evaporates␣it.␣Down␣force.␣Down␣reverse.␣Down␣net.␣The
␣net␣reverses␣force.
Now␣try:
Add␣add␣add␣add␣add␣add␣add␣now.
It␣is␣not␣possible;␣now␣count:
0.
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18+.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣You␣␣␣can␣␣␣now␣␣␣pump
in␣reverse.␣Down␣lock␣goes;␣narrows␣lock␣down:
Inverse.␣Lock␣young.␣Range.␣Sense␣0n␣1n␣2n␣3n␣4n␣5n␣6n␣7n␣8n␣9n
␣␣␣␣␣␣␣␣␣␣Powers␣␣␣␣␣␣␣␣␣␣lock␣time␣now.
Inverse.␣Lock␣young.␣Range.␣Sense␣0n␣1n␣2n␣3n␣4n␣5n␣6n␣7n␣8n␣9n
```

␣␣␣␣␣␣␣␣␣␣Powers␣␣␣␣␣␣␣␣␣␣␣␣␣snowmelt␣␣␣now.
␣␣␣␣Powers
␣␣␣␣␣␣all:
Bear␣hatchery␣n
␣powers
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣insulated␣bear␣hatchery␣?.
␣Hydro.␣Power␣spring␣as
␣snowmelt␣␣␣␣␣␣␣␣␣powers␣␣␣snowmelt␣␣then,␣and␣disengage.
HYDRO!!

This program is the language's name. It prints a bunch of various stuff:

```
Hatchery
Oblivion␣through
Marshy
Energy␣from
Snowmelt
Powers
Rapids
Insulated␣but
Not
Great
```

You can see that Homespring programs have a very poetic and expressive quality. Although it is said that artists must suffer for their work, this does not apply to HOtMEfSPRIbNG as suffering is not included among its features. Writing programs in any one of the flawed 'other' languages is a painful and disturbing ordeal that is best avoided at all costs.