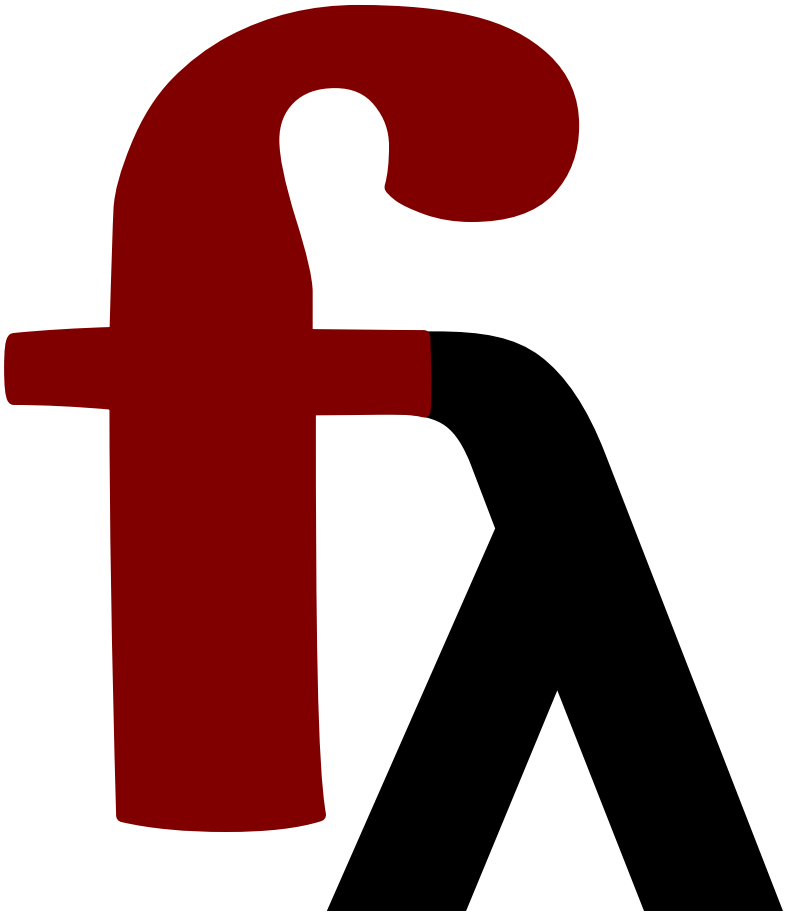
Kalafiorem w JVM!

(czyli o eksperymentach w języku Frege)

#30 Toruń JUG, 22.02.2017

Skonieczka Piotr



/ˈfreɪgə/

Gottlob Frege

From Wikipedia, the free encyclopedia

Friedrich Ludwig Gottlob Frege (/ˈfreɪgə/;[9] German: [ˈgɔtlo:p ˈfre:gə]; 8 November 1848 – 26 July 1925) was a German philosopher, logician, and mathematician.

Considered a major figure in mathematics, he is responsible for the development of modern logic and making contributions to the foundations of mathematics. He is also understood by many to be the father of analytic philosophy, where he concentrated on the philosophy of language and mathematics. Though largely ignored during his lifetime, Giuseppe Peano (1858–1932) and Bertrand Russell (1872–1970) introduced his work to later generations of logicians and philosophers.

Contents [hide]

Gottlob Frege

Frege in c. 1879

Born	8 November 1848 Wismar, Mecklenburg-Schwerin, Germany
Died	26 July 1925 (aged 76) Bad Kleinen, Mecklenburg-Schwerin, Germany
Alma mater	University of Jena University of Göttingen (PhD)
Notable work	*Begriffsschrift* (1879)

Search GitHub Pull requests Issues Gist

Frege

(BSD LICENSE)

Repositories People 5

Find a member…


Dierk König
Dierk
Follow
Switzerland


James Earl Douglas
earldouglas
Follow
USA


Ingo Wechsung
Ingo60
DESIGNER
Follow
Germany


Marimuthu Madasamy
mmhelloworld
Follow
India

Yorick Laupa
YoEight
Follow
France

+46 MORE

What is Frege?

IS LIKE A

„Frege ~~is a~~ Haskell for the JVM.

Like any Haskell, it is **purely functional**, enjoys a **strong static type** system with global type inference and non-strict - also known as **lazy** - **evaluation**.

Frege compiles to Java, **runs on the JVM**, and uses any Java library you want. It can be used inside any Java project.”

CAUTION!

This presentation contains mathematical formulas!

http://www.wallpapers4u.org/wp-content/uploads/light_glare_triangle_stripes_colorful_shade_15678_1920x1080.jpg

Syntax?

Frege is based on **Haskell 2010**

https://wiki.haskell.org/Language_and_library_specification

Standard Libraries?

- Haskell (204 in 10 major categories)

  https://hackage.haskell.org/package/base-4.9.1.0#modules

- Frege (88 in 18 major categories)

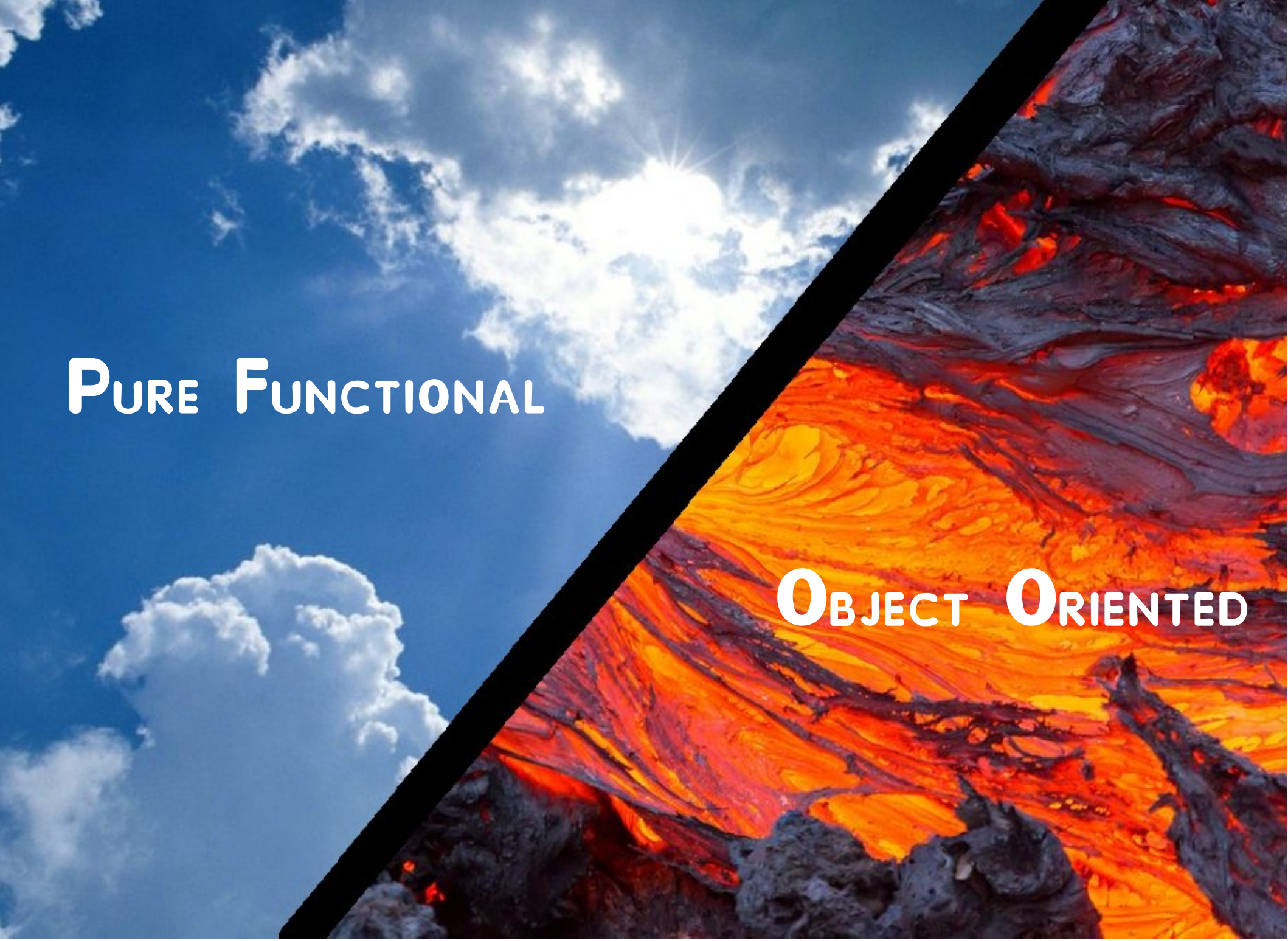
  http://www.frege-lang.org/doc/fregedoc.html

  https://github.com/Frege/frege/wiki/Libraries

- Where it’s possible – Frege uses native Java types!

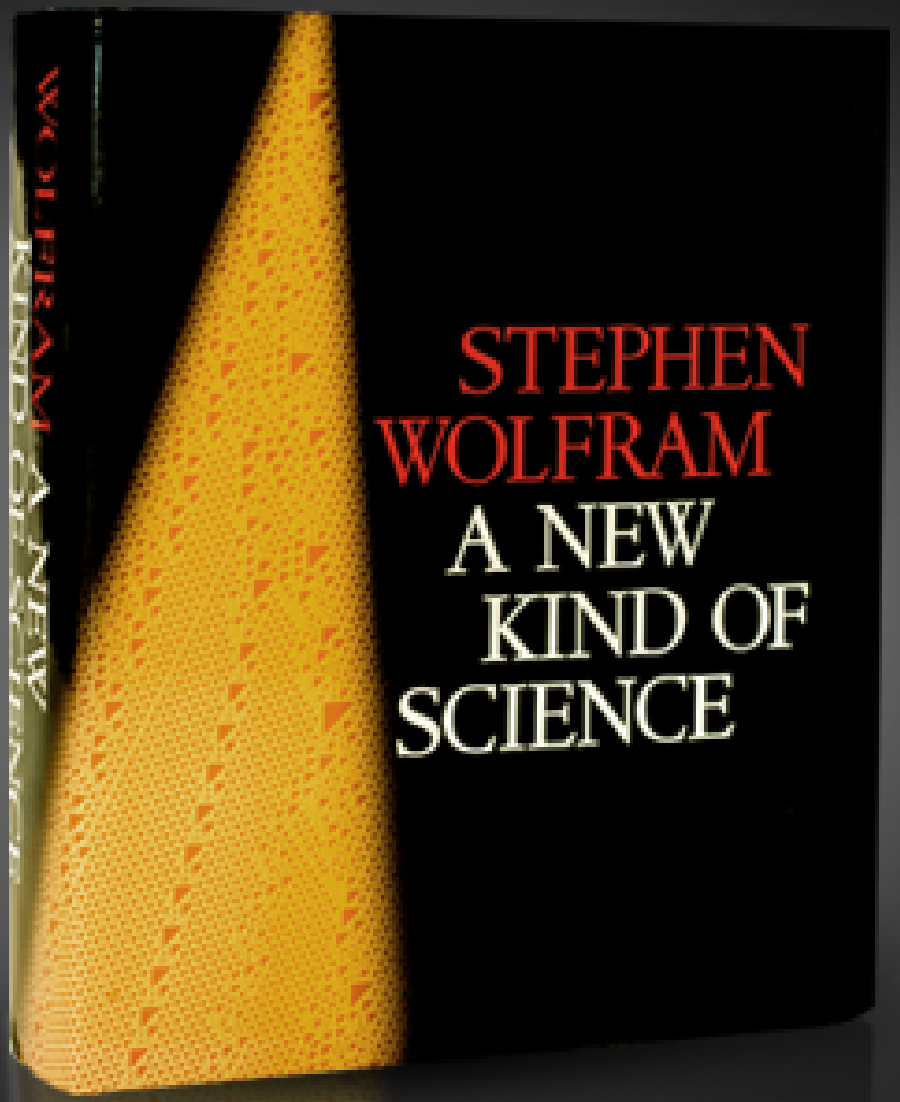
Technicalities?

Frege works with both **Java 7** & **Java 8** JRE/JDK



http://smallify.it/wp-content/uploads/2013/06/Salesforce.jpg https://media.mnn.com/assets/images/2017/01/hot-flowing-lava.jpg.1000x0_q80_crop-smart.jpg

How to check this out?


STEPHEN
WOLFRAM
A NEW
KIND OF
SCIENCE

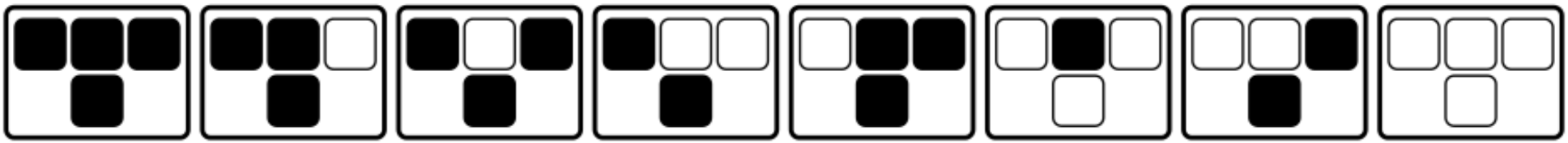
WolframAlpha computational knowledge engine

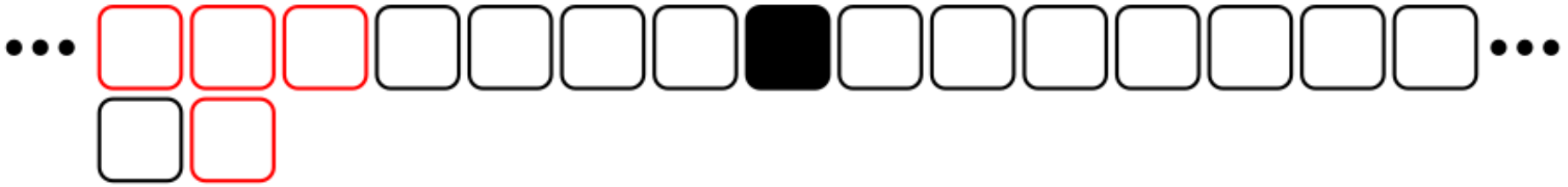
Enter what you want to calculate or know about:

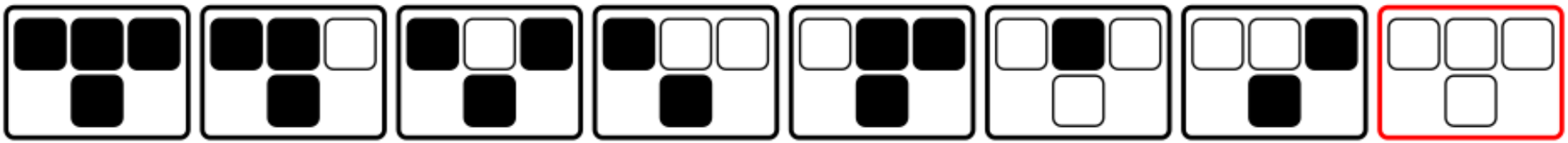
Web Apps Examples Random

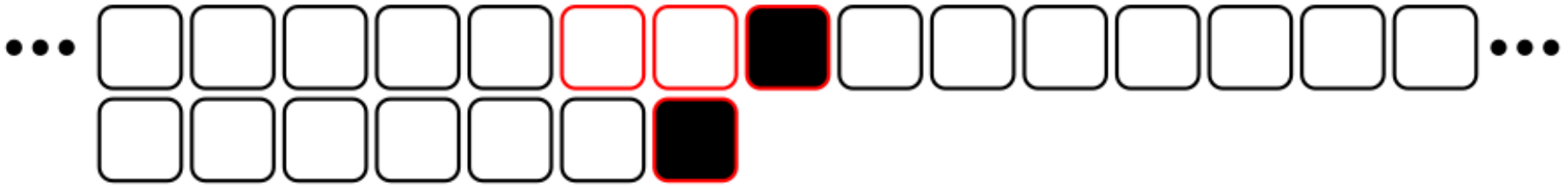
Explore some of the things Wolfram|Alpha can do:

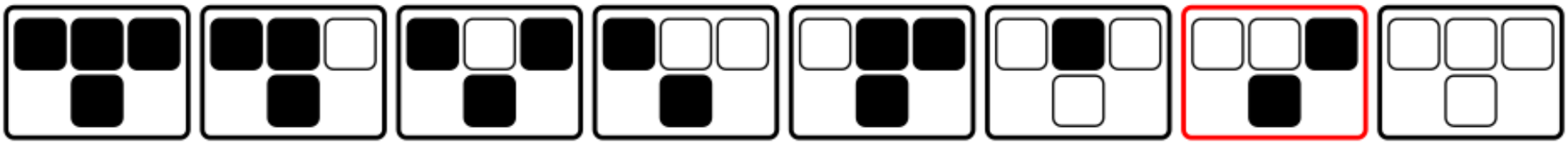
- sin(x) Mathematics
- Step-by-step Solutions
- Words & Linguistics
- Units & Measures
- Statistical & Data Analysis
- People & History
- Dates & Times
- Chemistry
- Culture & Media
- Money & Finance
- Physics
- Art & Design

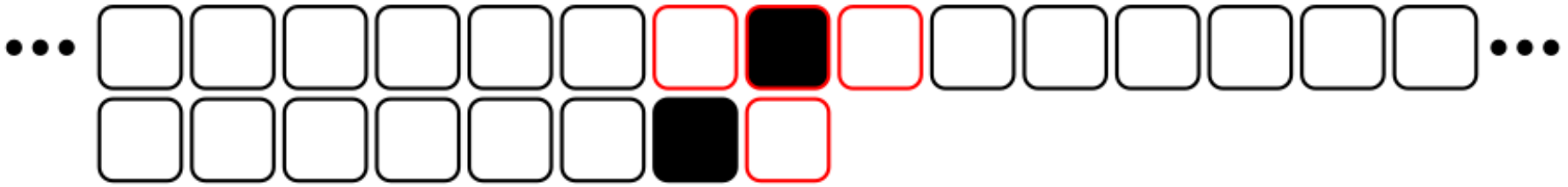


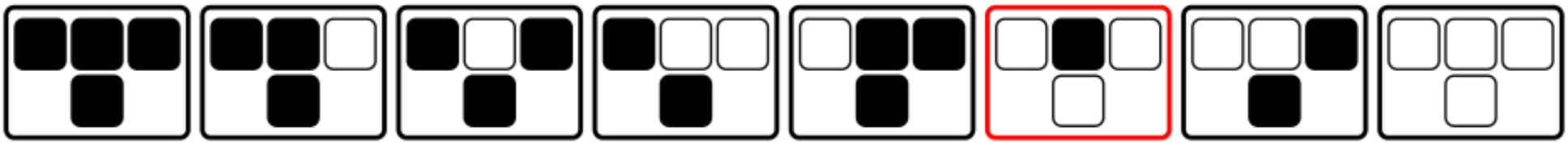


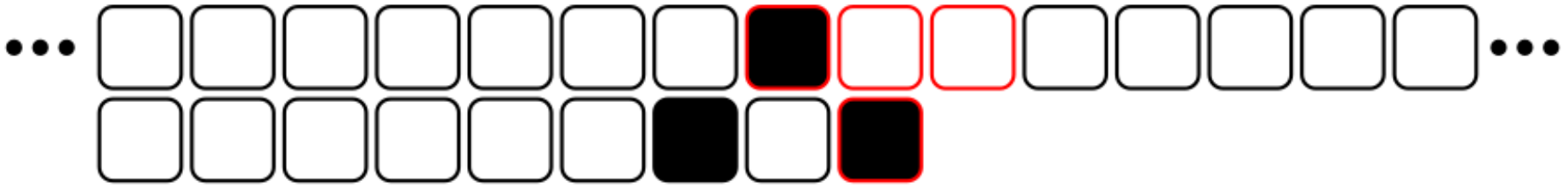


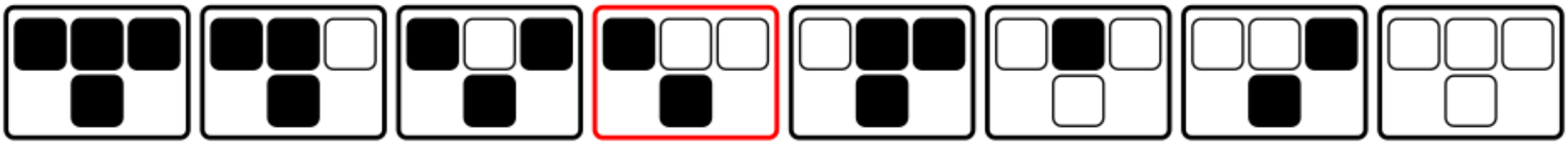


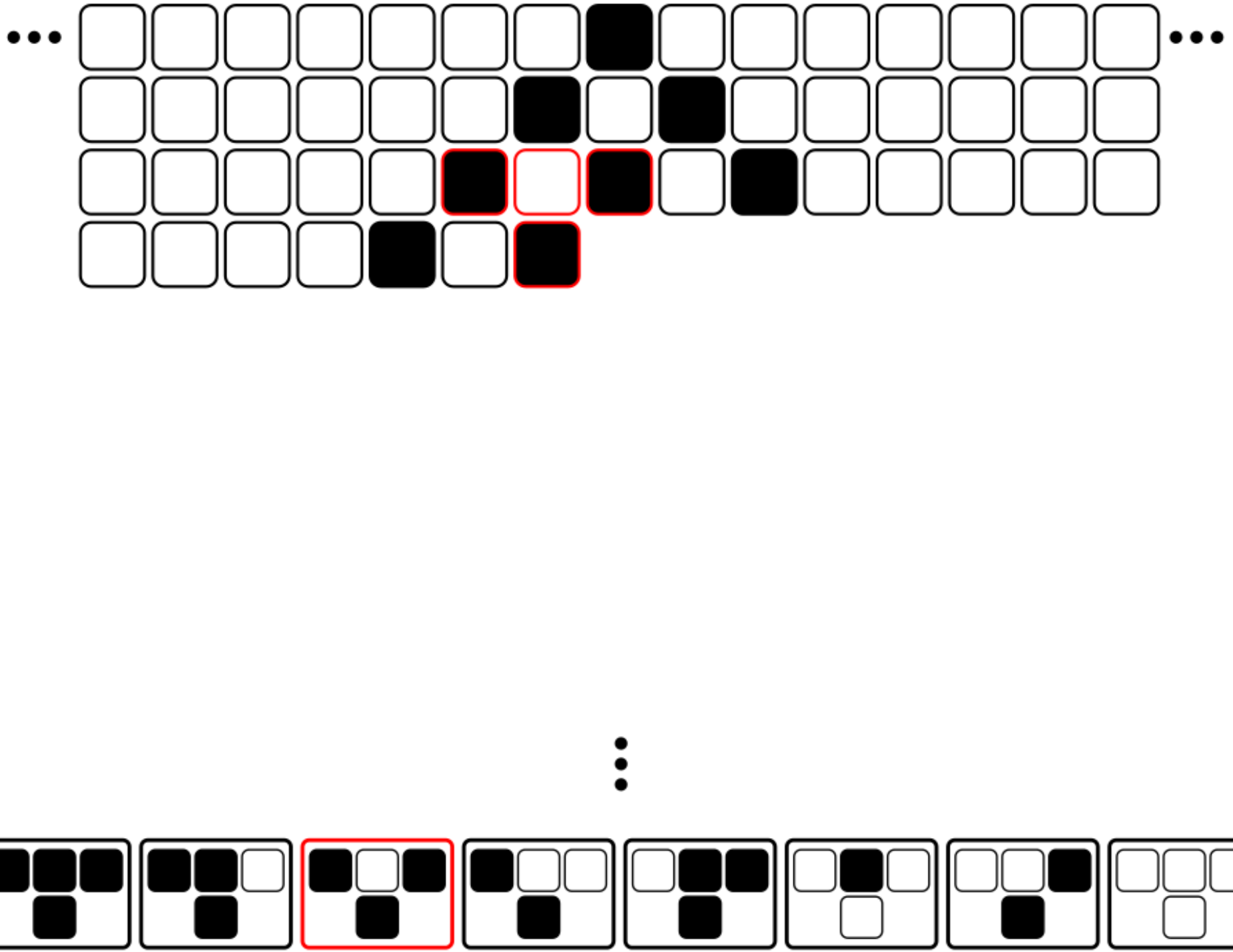


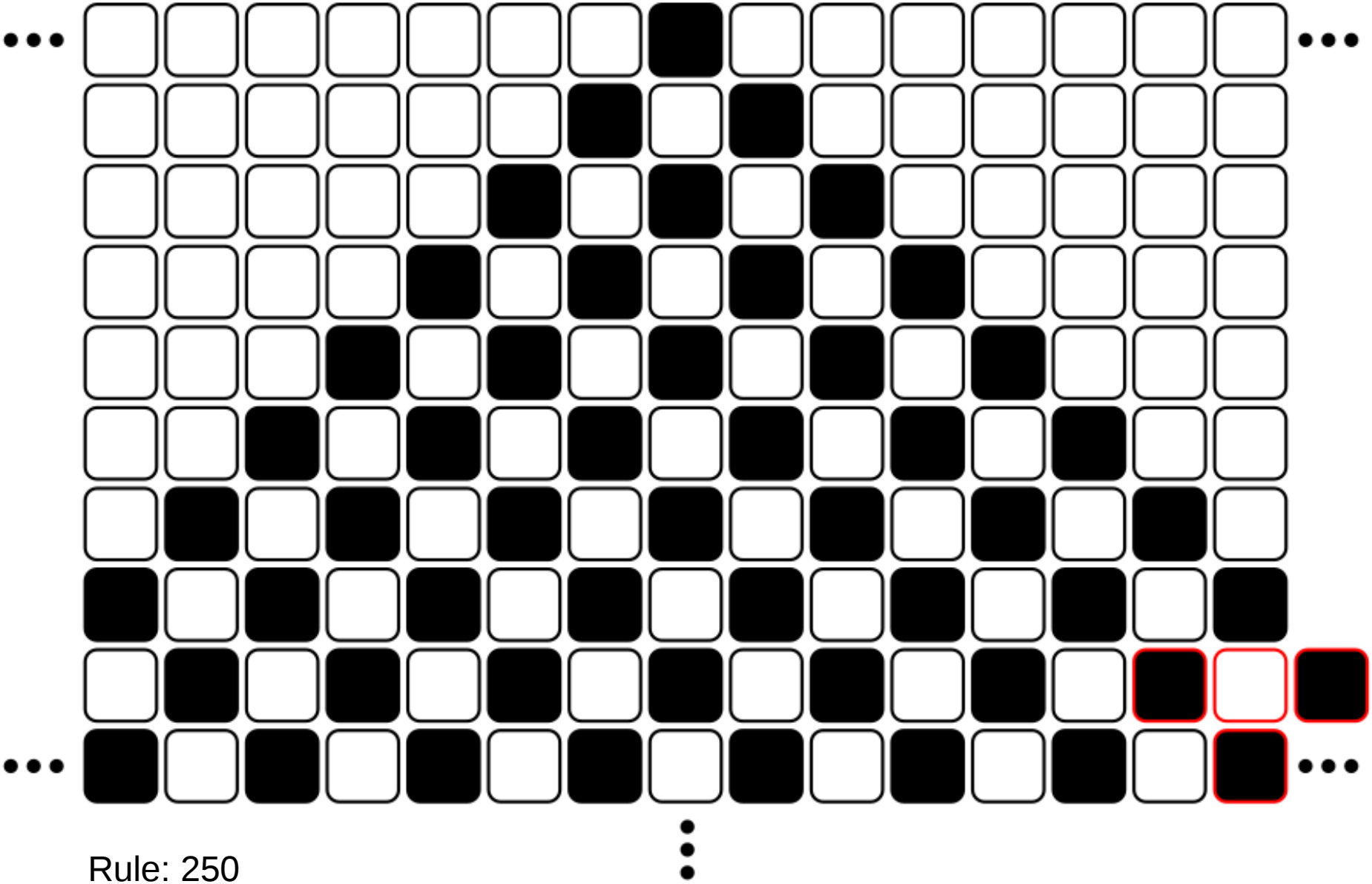




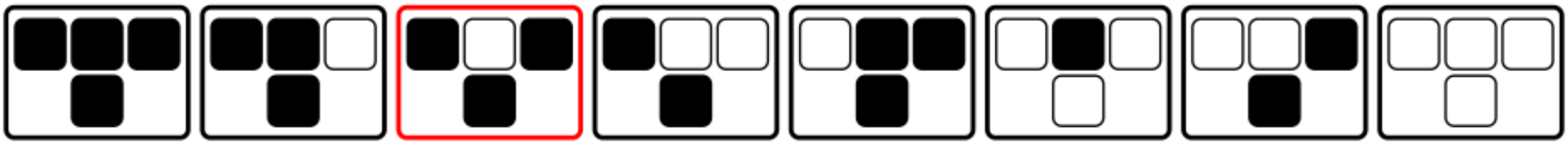


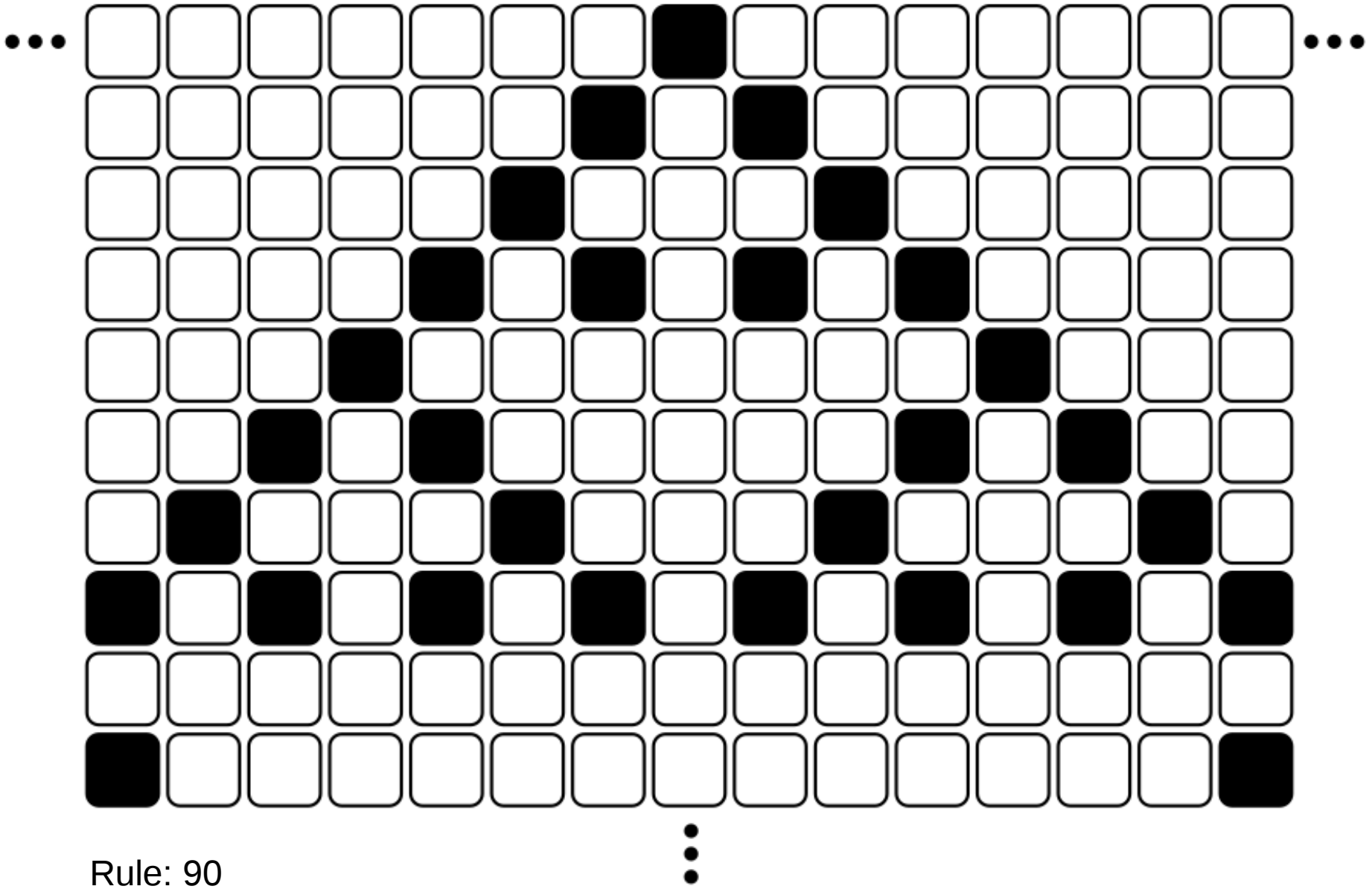




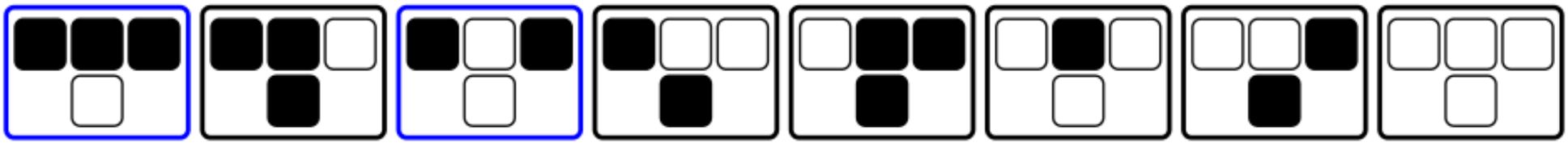


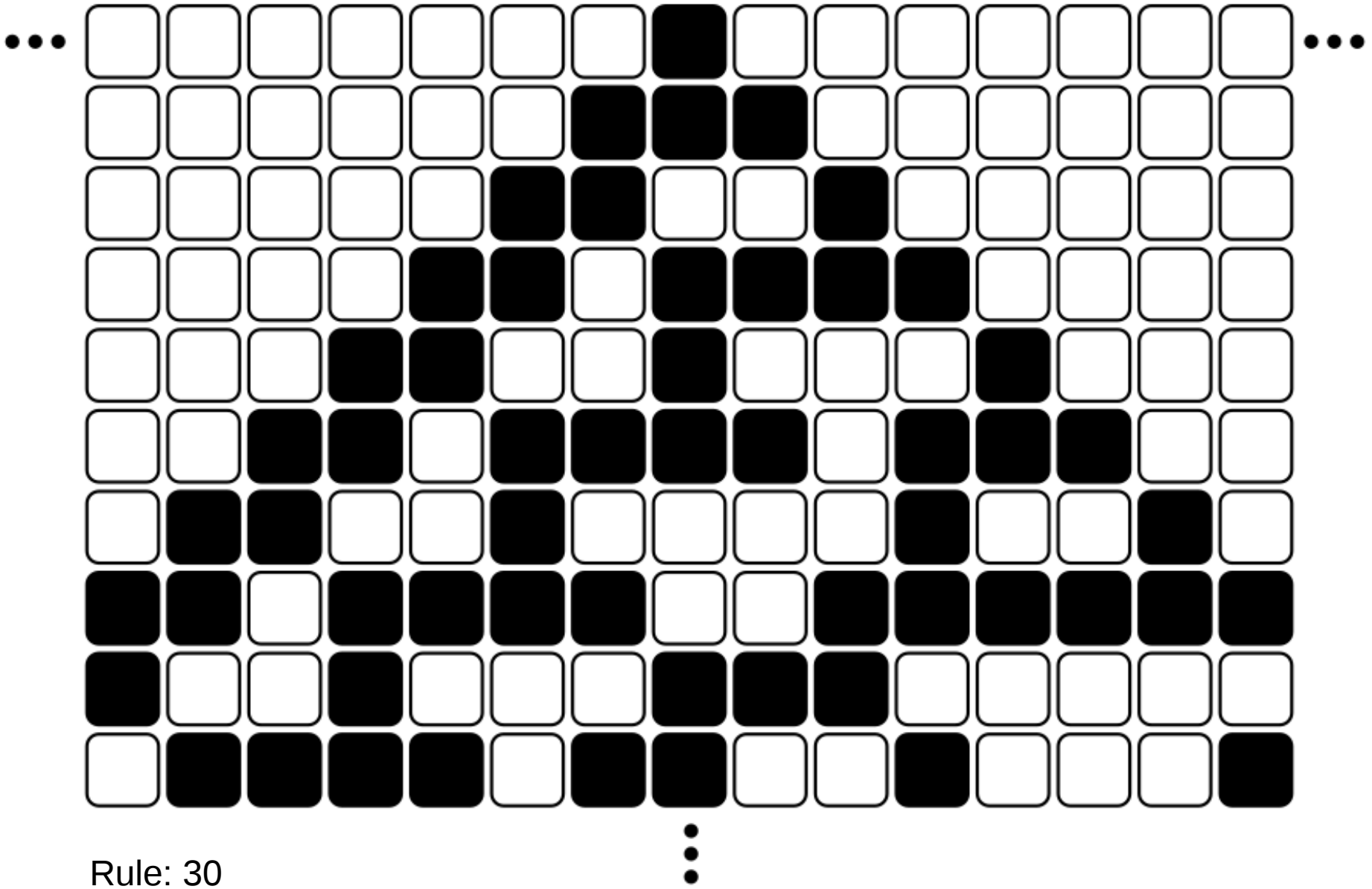
Rule: 250



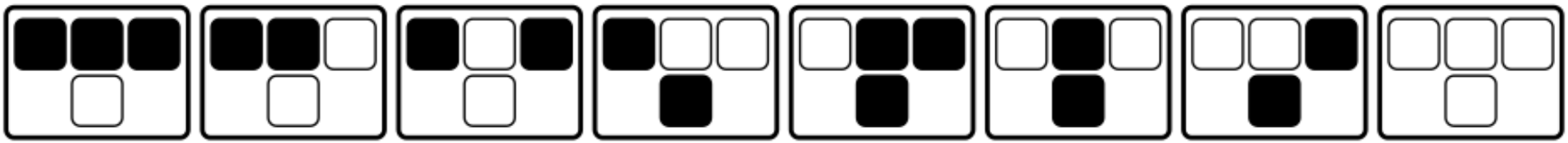


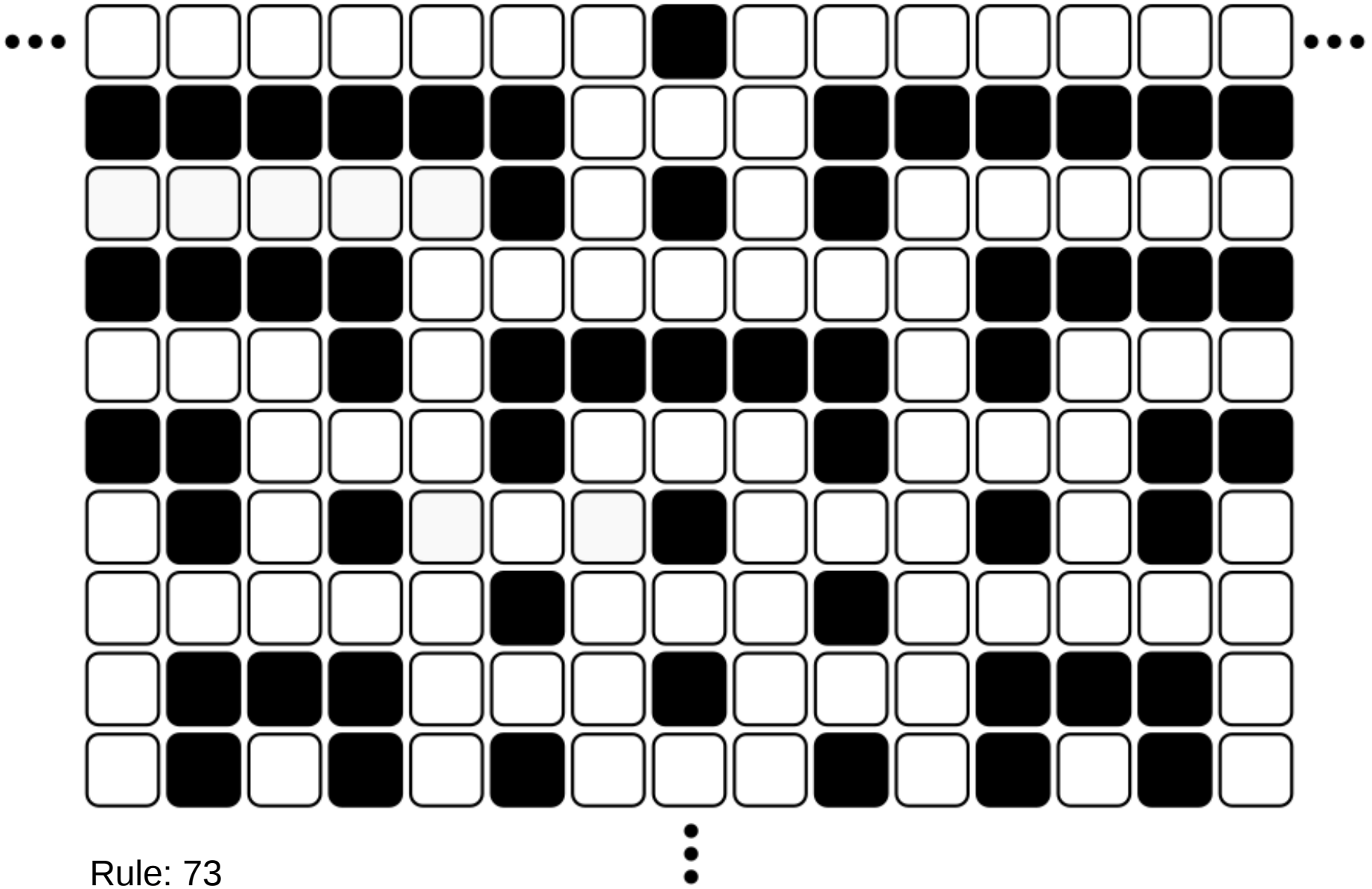
Rule: 90



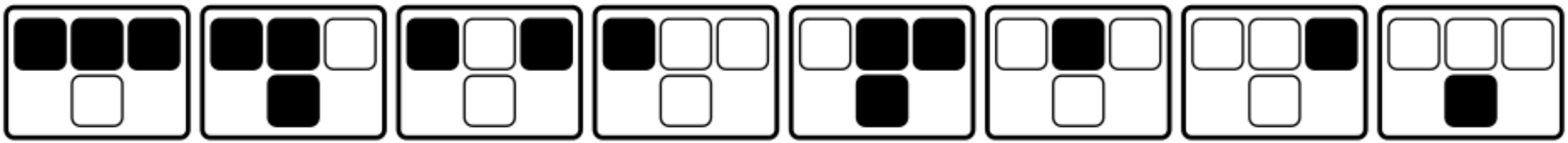


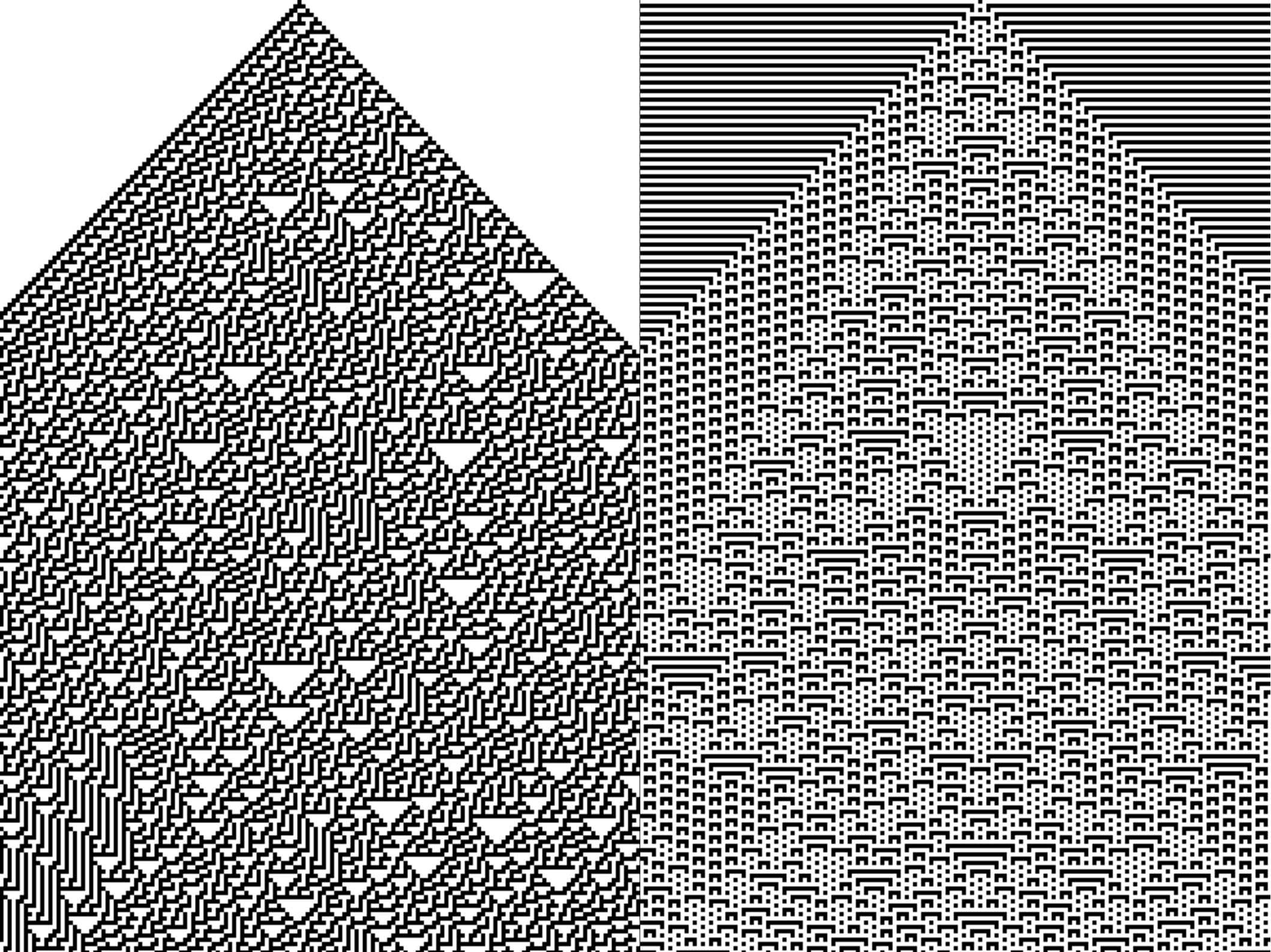
Rule: 30

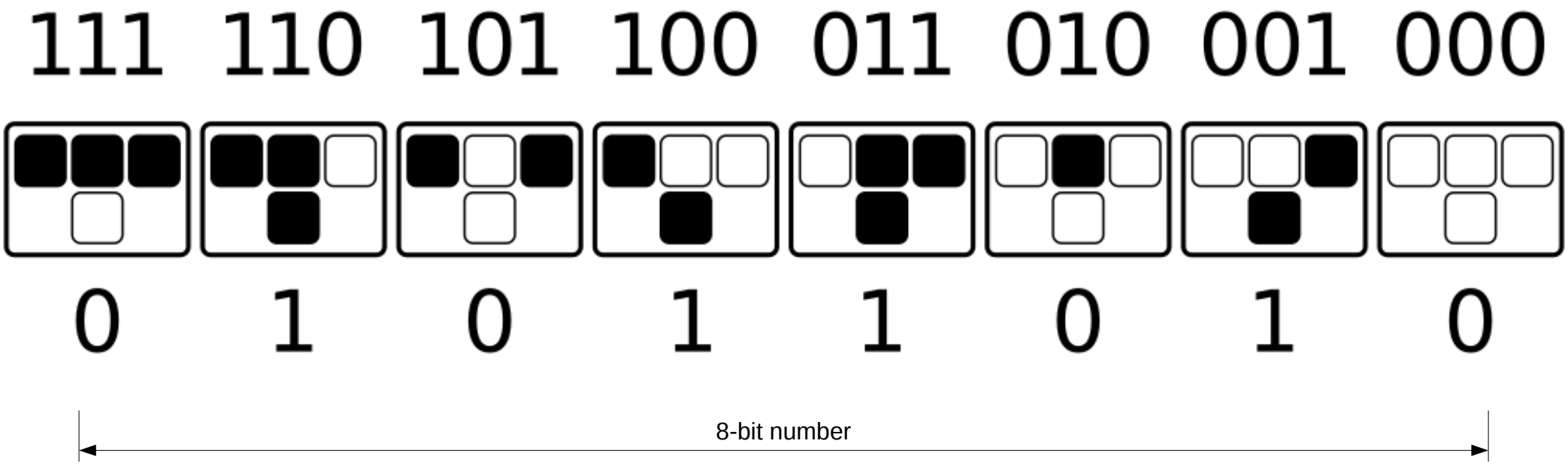




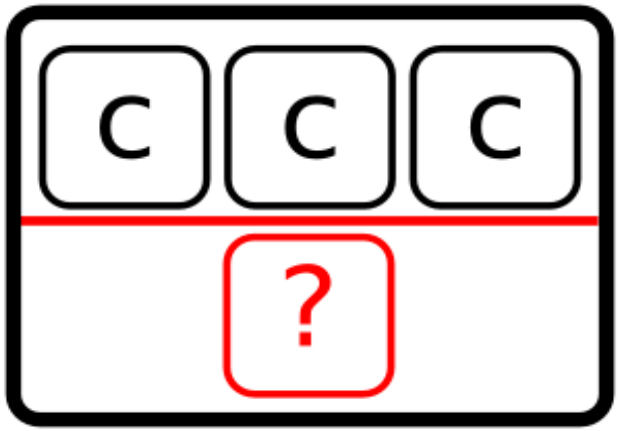
Rule: 73






111 110 101 100 011 010 001 000
0 1 0 1 1 0 1 0
8-bit number

General case:



C-colors:

$$cell = (rule \ / \ C^{v}) \ mod \ C$$

2-colors:

$$cell = (rule \gg v) \ mod \ 2$$

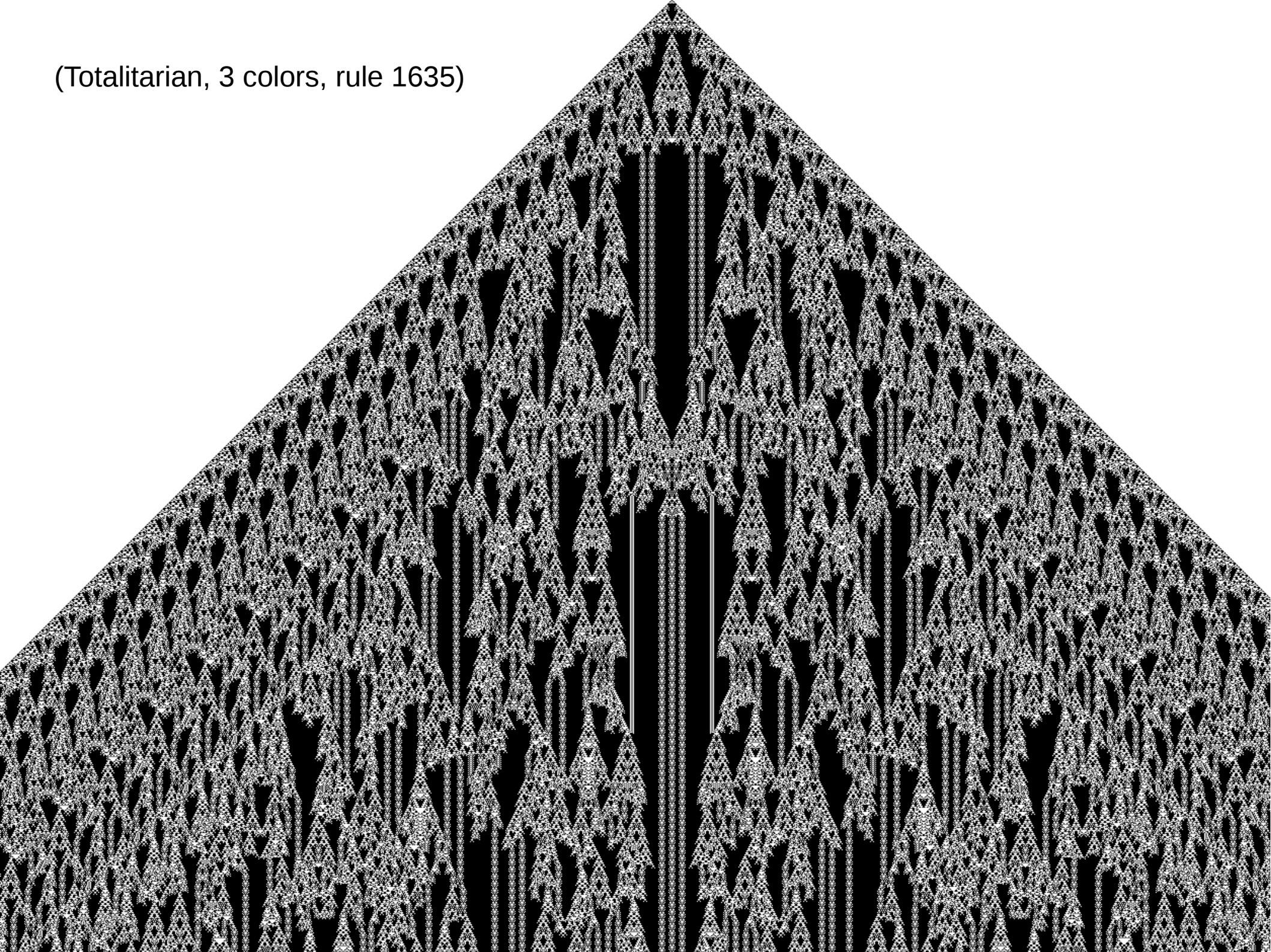
(Where v is an decimal value* of 3 cells above)

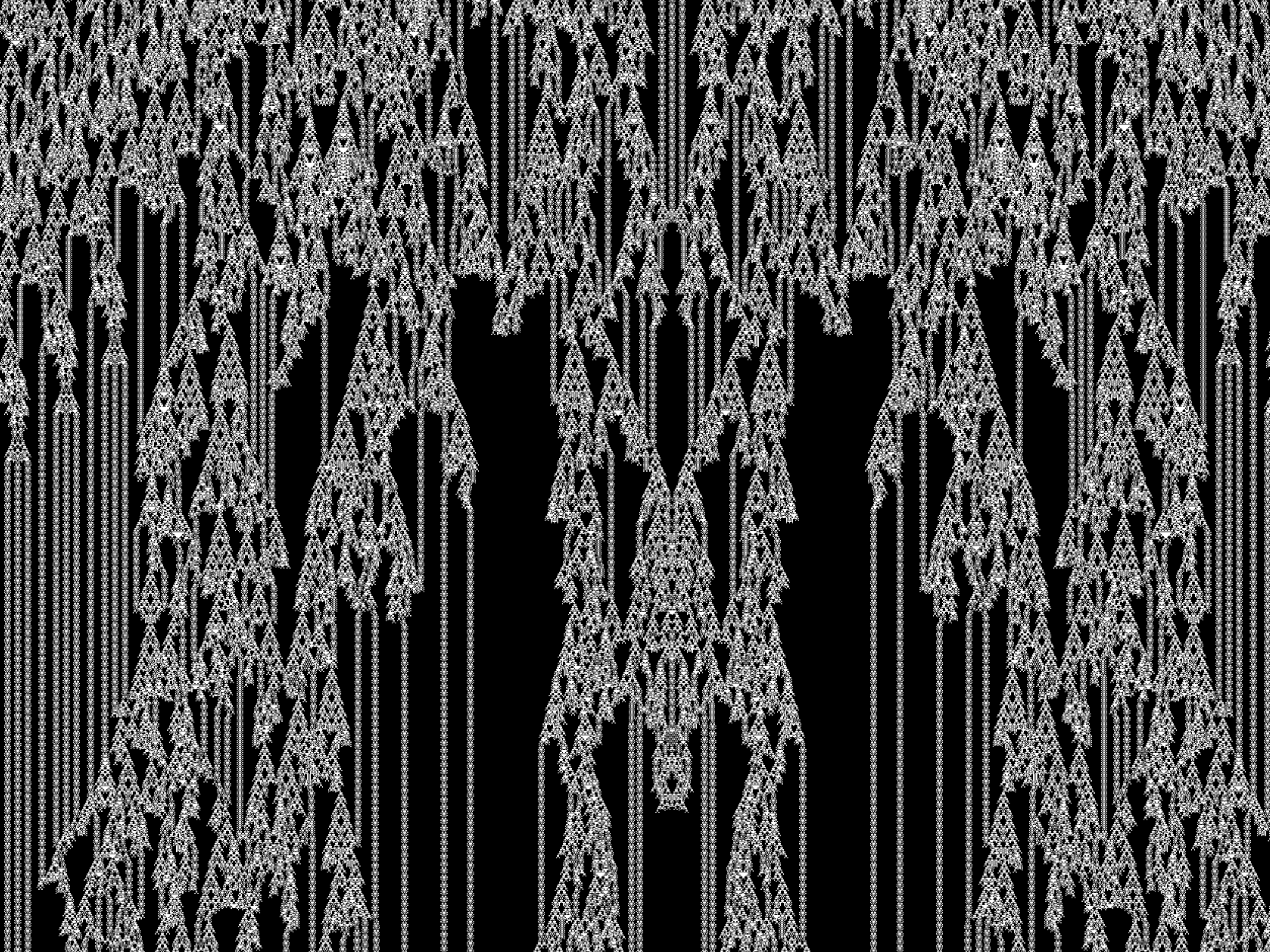
Number of colors	Number of rules	
	Simple CA	Totalitarian CA
2	256	---
3	7.6×10^{12}	2 187
4	3.4×10^{38}	1 048 576
5	2.3×10^{87}	1 220 703 125
6	1.2×10^{168}	2 821 109 907 456
7	7.3×10^{289}	11 398 895 185 373 100
8	2.4×10^{462}	73 786 976 294 838 200 000

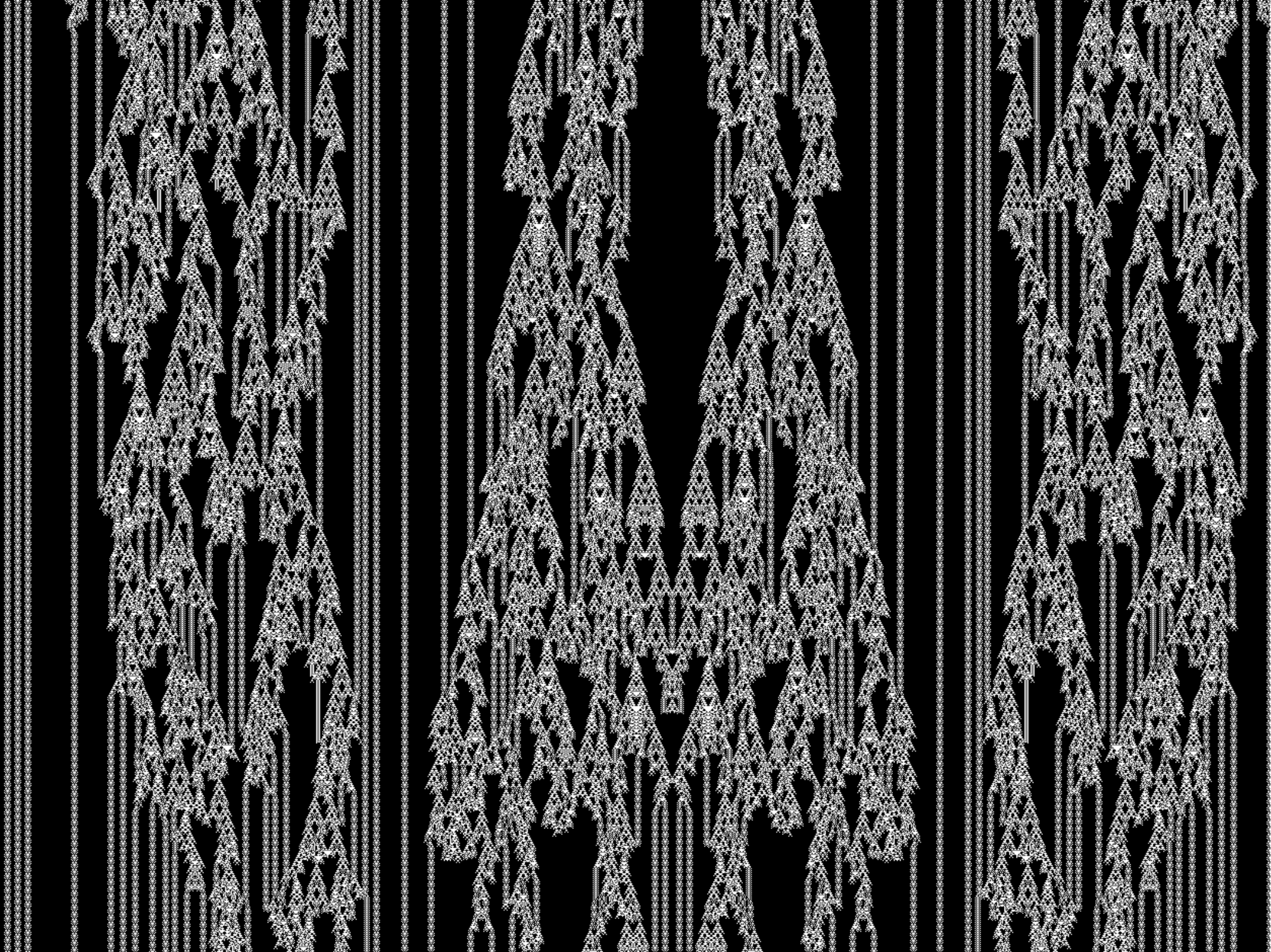
~7.3×10^{19}

$$v_s = \left(C^2 \times c_1 + C^1 \times c_2 + C^0 \times c_3\right) \qquad v_t = \left(c_1 + c_2 + c_3\right)$$

(Totalitarian, 3 colors, rule 1635)








code 451


code 452


code 453

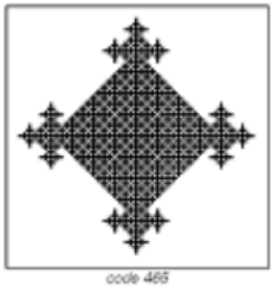

code 454


code 457

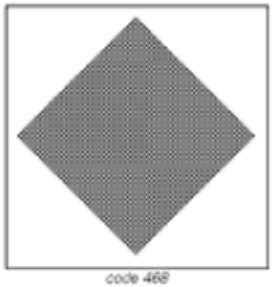

code 459


code 461


code 462


code 465


code 467


code 468


code 470


code 473


code 475


code 478

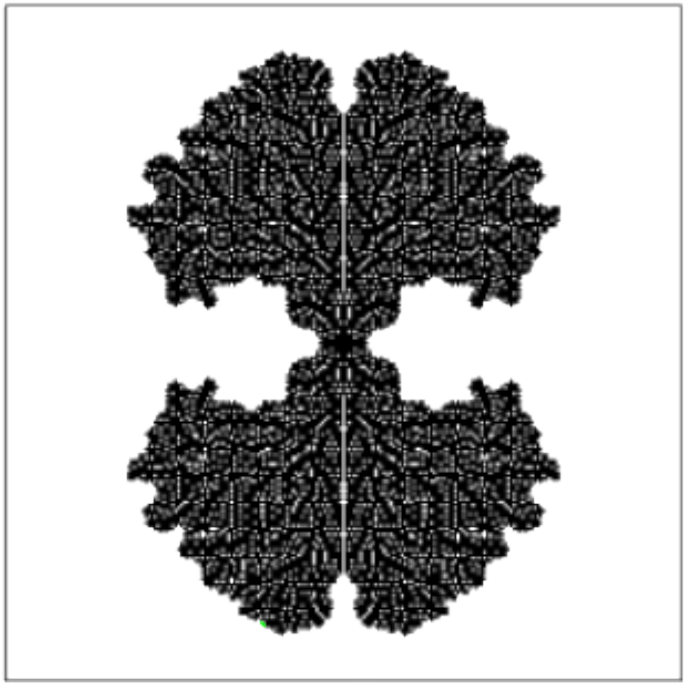

code 481

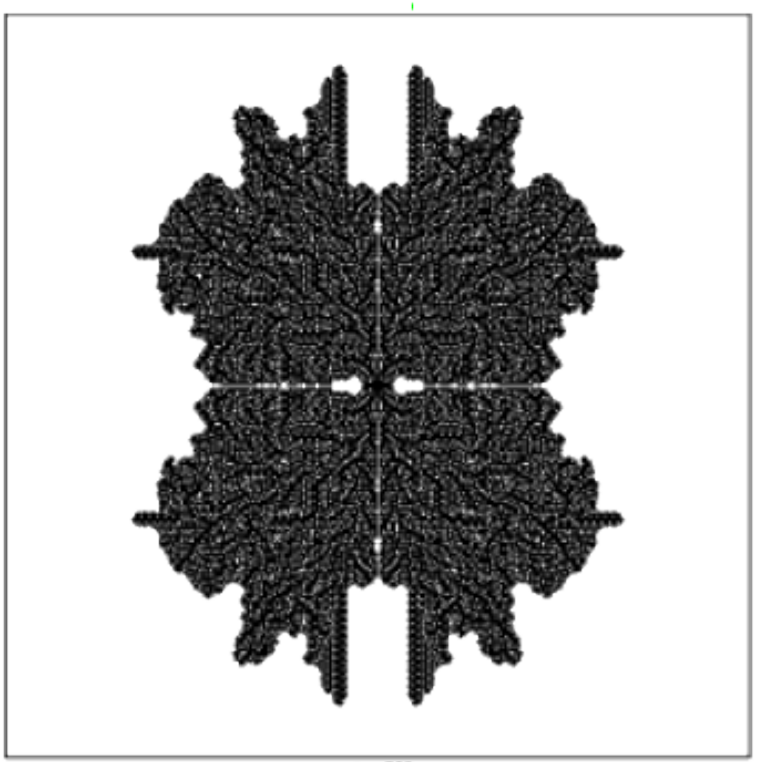

code 483

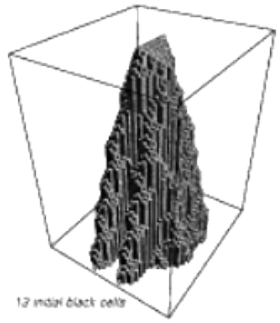

code 489


code 491

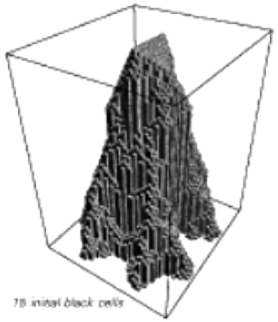

code 493


step 300

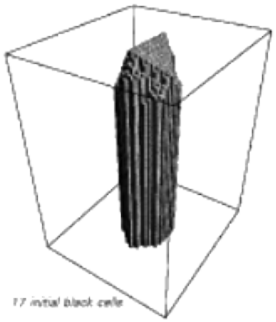

step 500



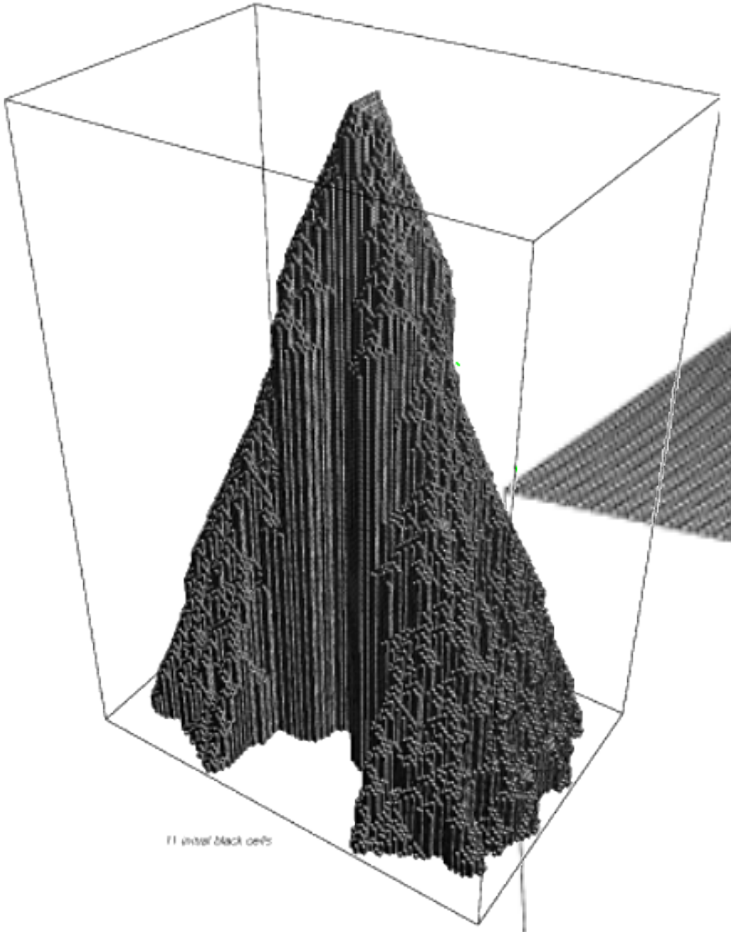
12 initial black cells



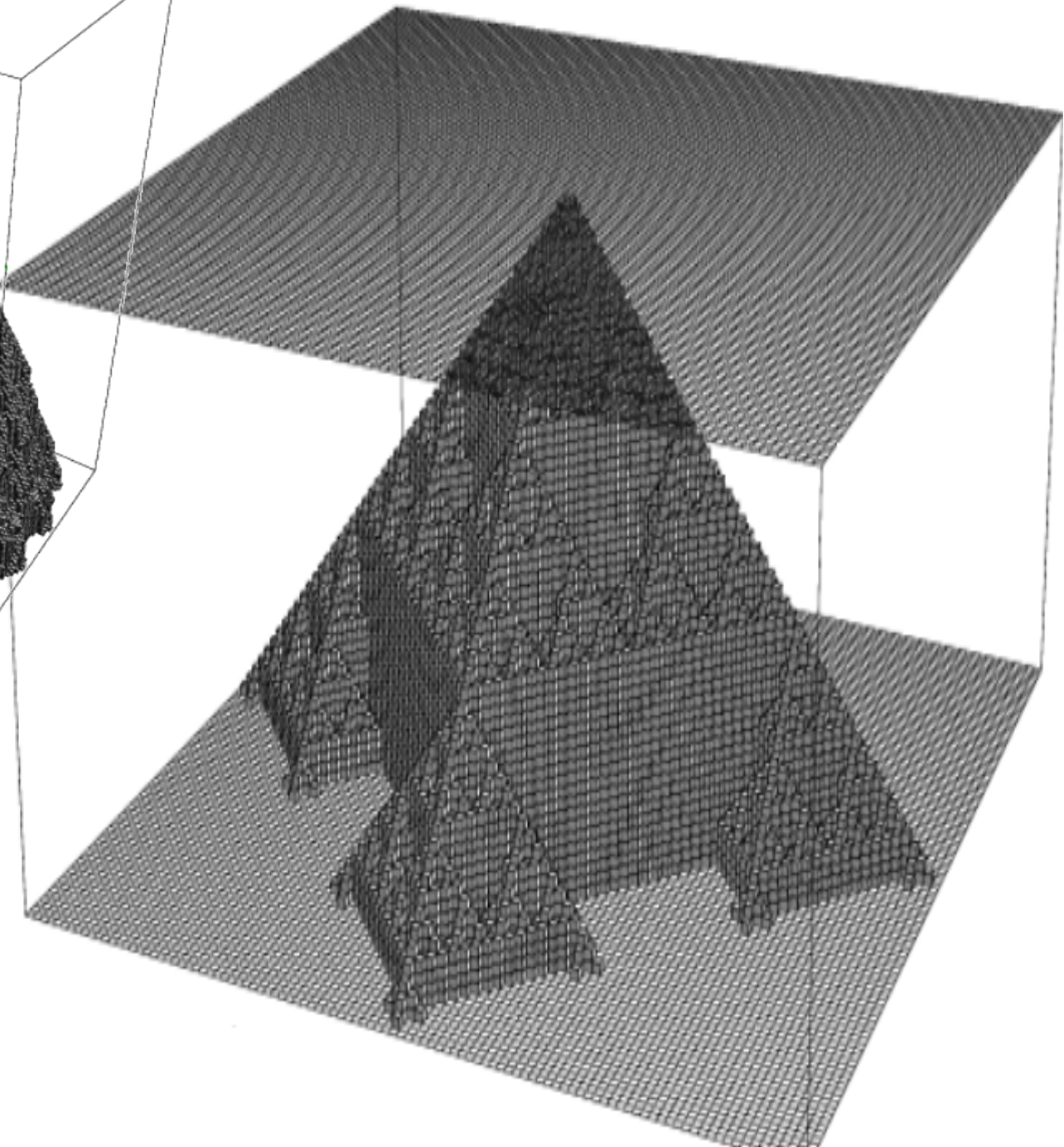
19 initial black cells

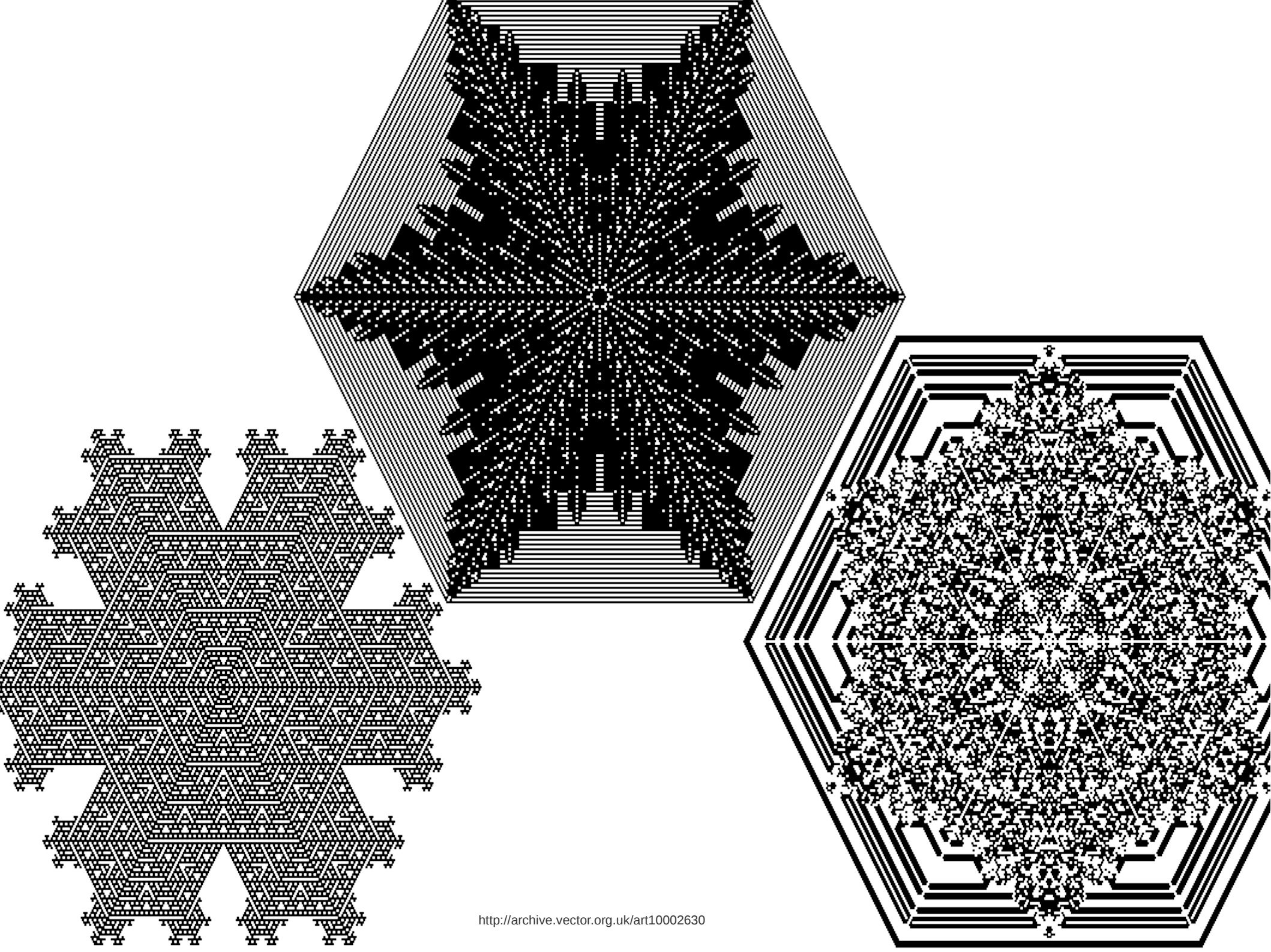


17 initial black cells

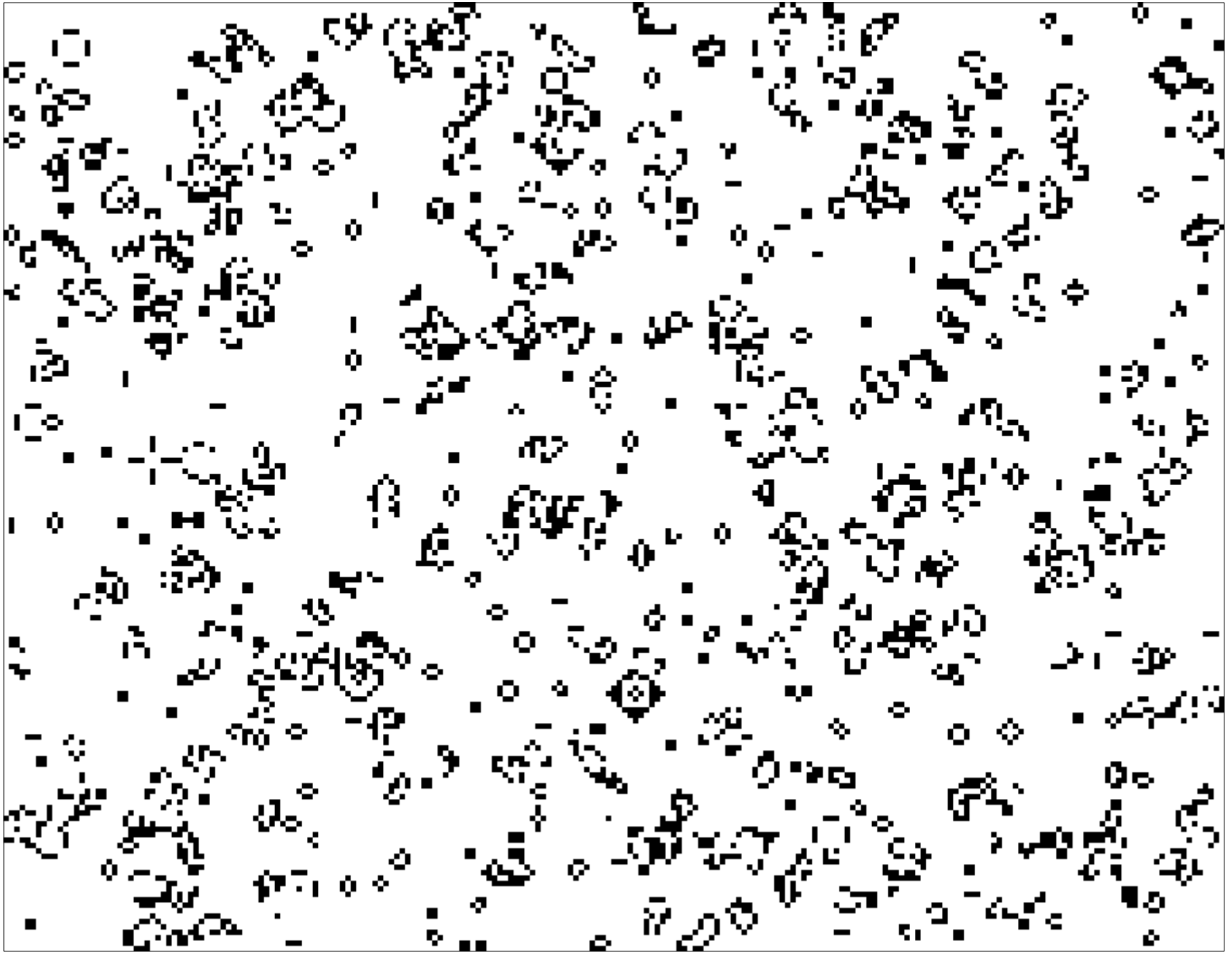


11 initial black cells





Game of life:



Let’s code!

(Haskell → Frege → Java)

Assumptions and goals:

- **Width**, **height**, **rule** and **zoom** parameters will be given as a command line arguments.

- As a result we would like to see **a picture**!

How to start?

- **Download fregec.jar from** (v. 3.24.100):

  https://github.com/Frege/frege/releases

- **Try it in REPL** (read-eval-print-loop):

  http://try.frege-lang.org/

  https://github.com/Frege/frege-repl/releases

Compiling Frege sources:

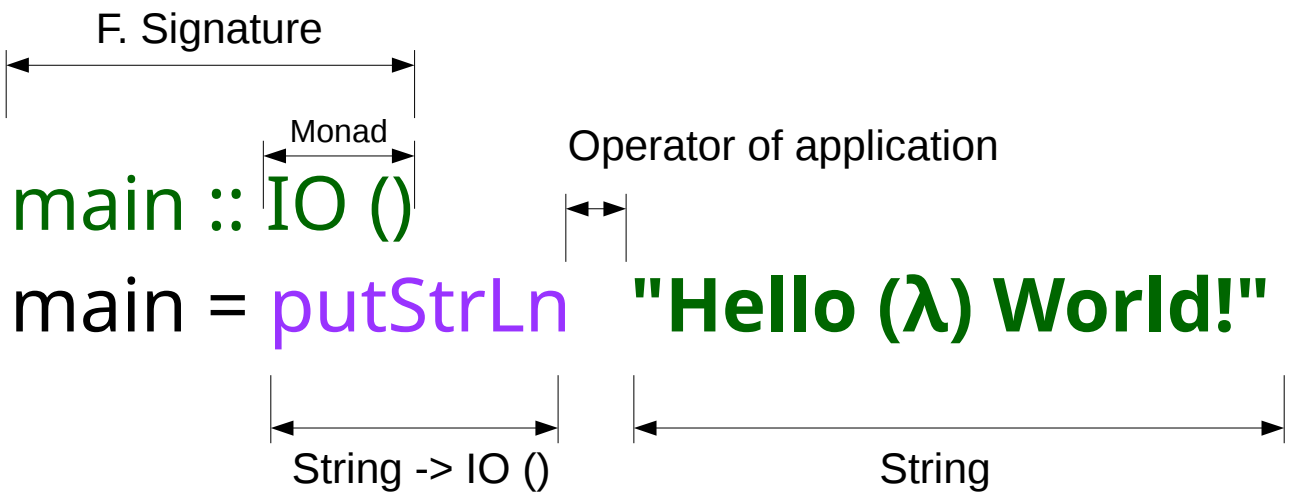
java [-Xss1m] **-jar lib/fregec.jar** [-d classes/] **File.fr**

File.class & **File.java**

Step 0.

(„Hello World”)

Haskell/Frege



Laws...

- Functors

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

- Applicative Functors

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

- Monads

```
return x >>= f = f x
m >>= return = m
```

Java?

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello (λ) World!");
    }
}
```

Step 1.

(Command line arguments)

„if” statement

if <condition> **then** <true-expression> **else** <false-expression>

- It’s acts like expression
- Cannot live without „**else**”

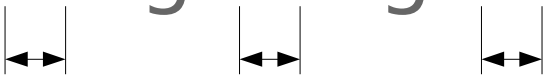
Function application!

- In Java:

```
foo(arg1, arg2, arg2);
```

- In Haskell & Frege:

```
foo  arg1  arg2  arg3
```



- left-associative
- most silent character
- the highest precedence of all operators!

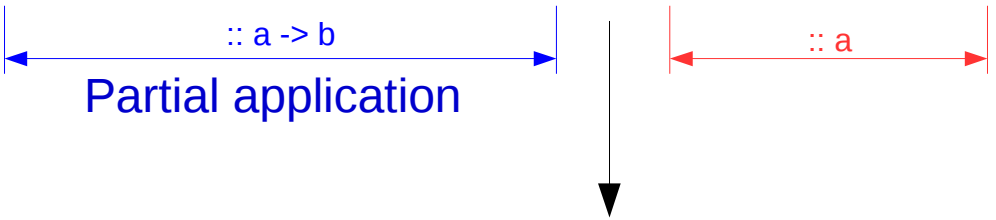
(Weak) function application with $

```
foo :: a -> a -> a -> b

bar :: a -> a

foo  arg1  arg2  (bar arg3)

foo  arg1  arg2  $  bar arg3
```




```
($) :: (a -> b) -> a -> b
```

„Let” and „Where”

let <bindings> **in** <expression>

- it is an expression
- all bindings are visible after „in”
- they are very „local”

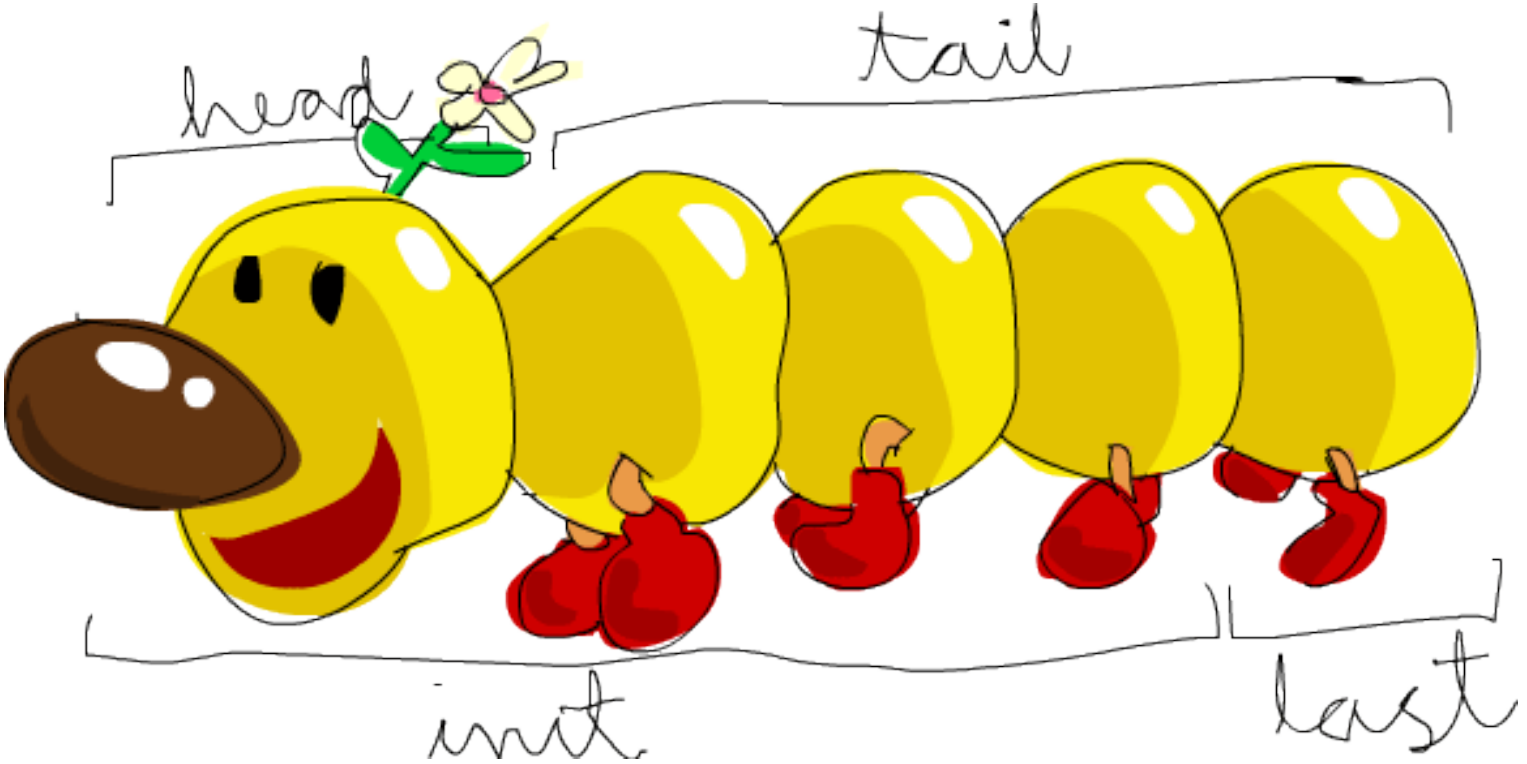
<expresion> **where** <bindings>

- syntactic sugar only
- whole function can see them, including all the „guards”

Step 2.

(First row of cells and first function composition!)

Lists



Infinite:

```
[1..]
```

Finite:

```
[1, 2, 3, 4, 5]
```

```
1:2:3:4:5:[]
```

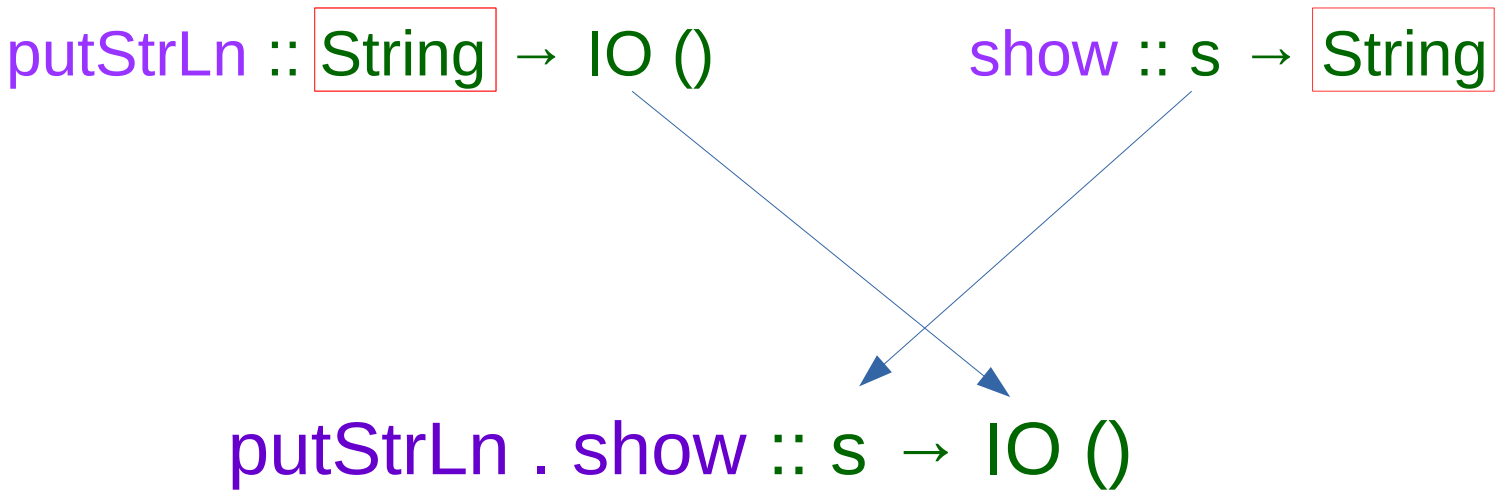
Operators:

```
(++) :: [a] -> [a] -> [a]
```

```
(!!) :: [a] -> Int -> a
```

Function composition!

(f ∘ g) (x) = f(g(x))



Step 3.

(Recursive calculations, mapping and pattern matching)

Type aliases?

Haskell/Frege

```
type SomeAlias = SomeType
```

Java

```
public class SomeType {
    ...
}

public class SomeAlias extends SomeType {
}
```

Type aliases?

Alias	Actual type
`type Rule = Int`	`Int`
`type Cell = Int`	`Int`
`type CellularTrio = (Cell, Cell, Cell)`	`(Int, Int, Int)`
`type CellularAutomataRow = [Cell]`	`[Int]`
`type CellularAutomata = [CellularAutomataRow]`	`[[Int]]`

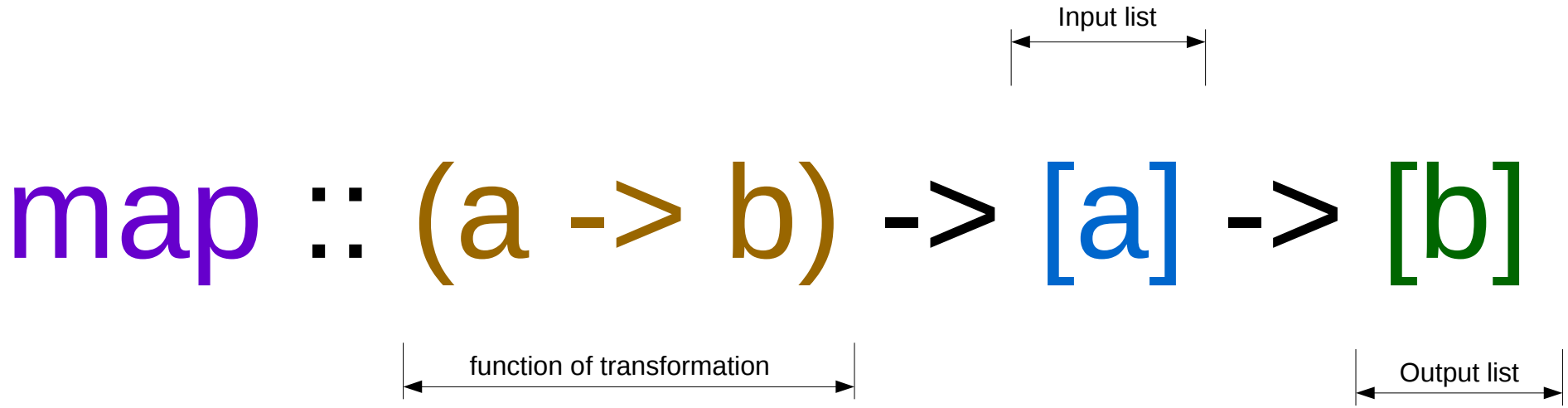
More readable:

```
initialRow :: Int -> Cell -> CellularAutomataRow
nextRow :: Rule -> CellularAutomataRow -> CellularAutomataRow
getTriplesFromRow :: CellularAutomataRow -> [CellularTrio]
generateCellularAutomata :: Int -> Int -> Rule -> CellularAutomata
```

Less readable:

```
initialRow :: Int -> Int -> [Int]
nextRow :: Int -> [Int] -> [Int]
getTriplesFromRow :: [Int] -> [(Int, Int, Int)]
generateCellularAutomata :: Int -> Int -> Int -> [[Int]]
```

Like a stream!



Step 4.

(Modules)

One module

One class

Step 5.

(Binary data & Images)

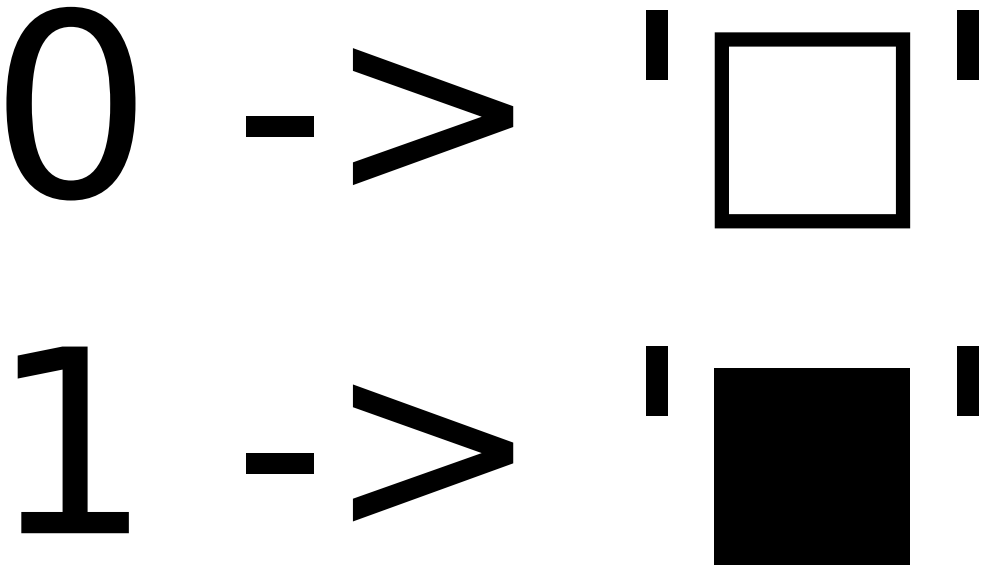
3rd party PNG library

https://wiki.haskell.org/Library/PNG

Step 6.

(So maybe ASCII Art...?)

Characters are encoded in **UTF-8**!

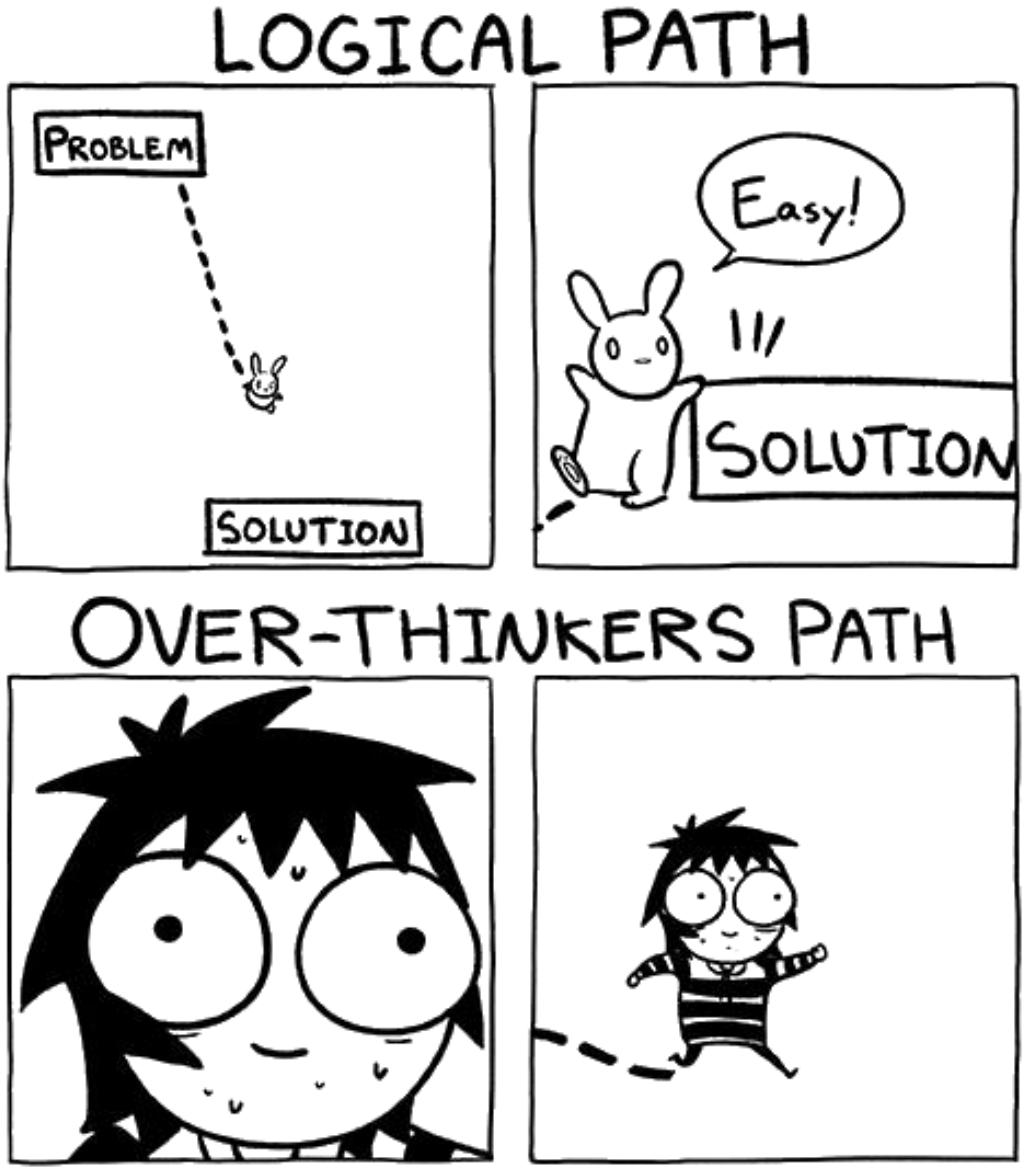


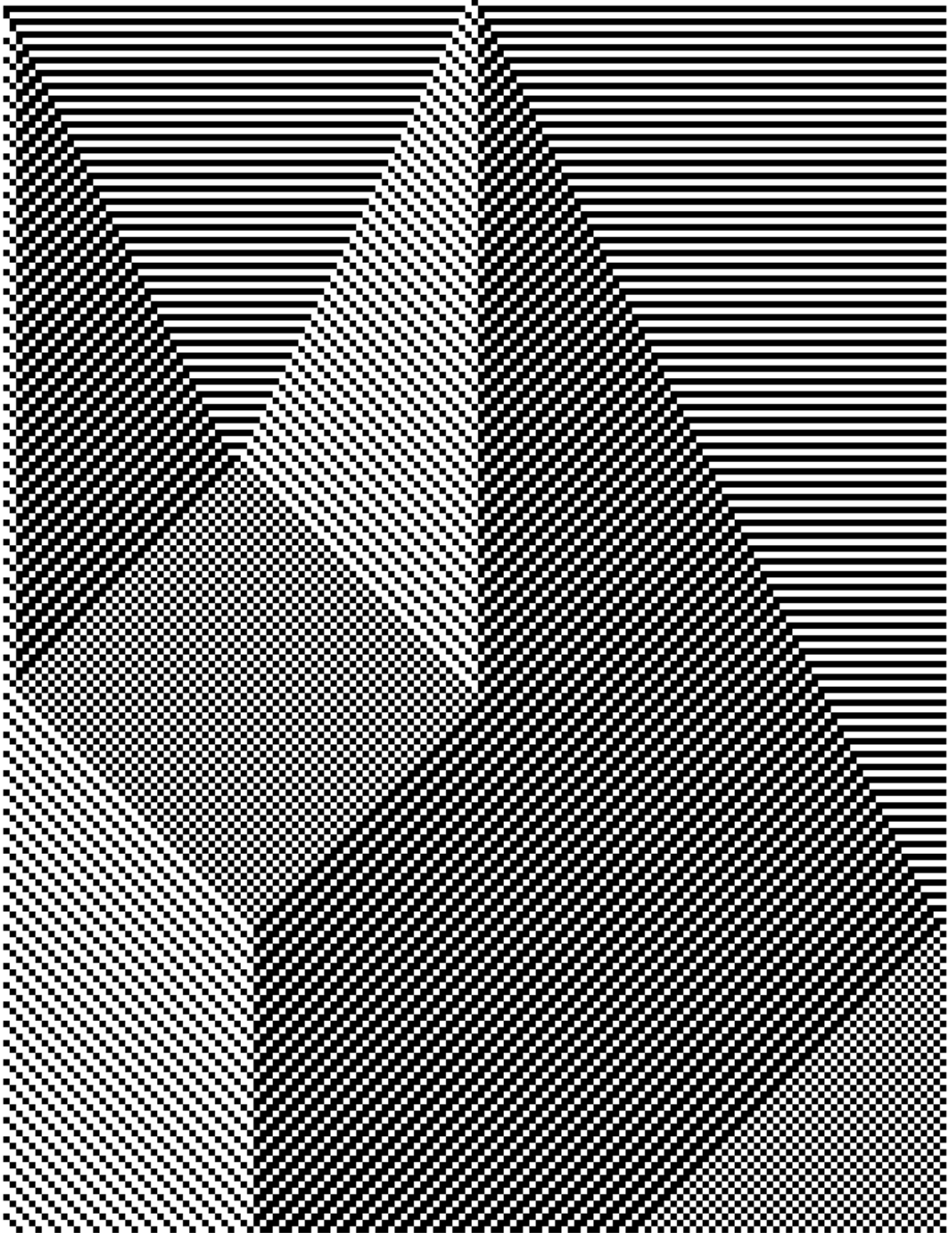
Compatybility difference!

- In Haskell, **String** type is an alias for **[Char]**
- In Frege it’a **java.lang.String**!

Step 7.

(Dealing with anomalies!)

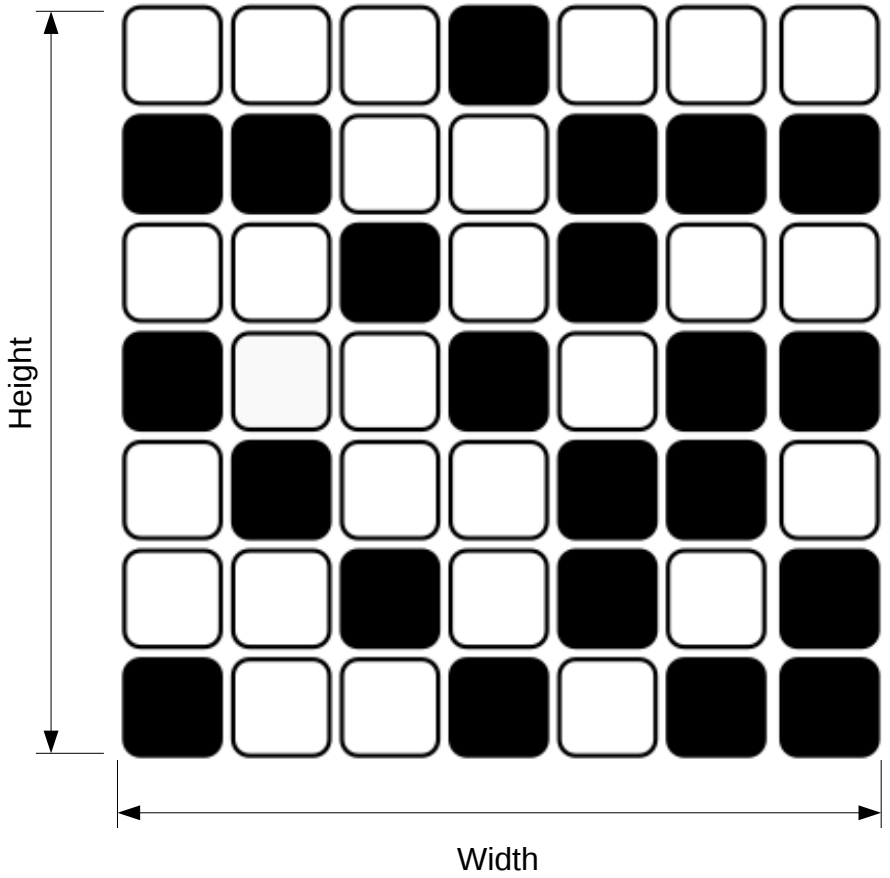




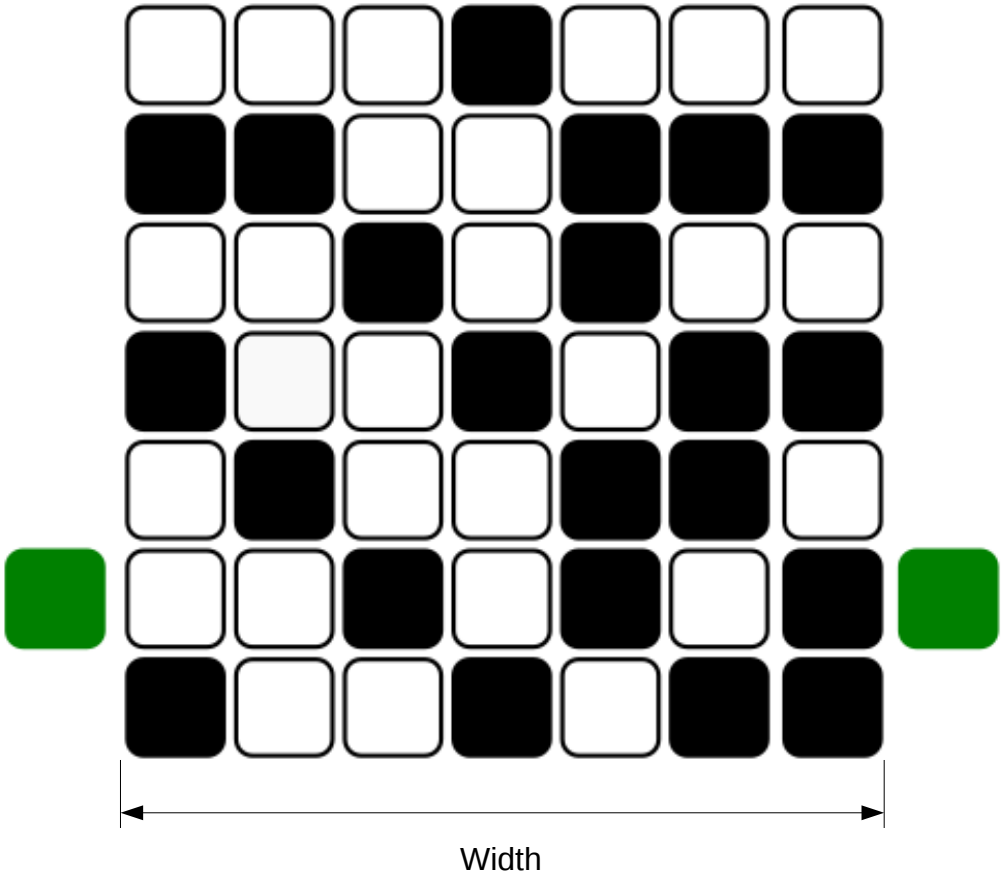
Rule: 57

Something goes wrong...

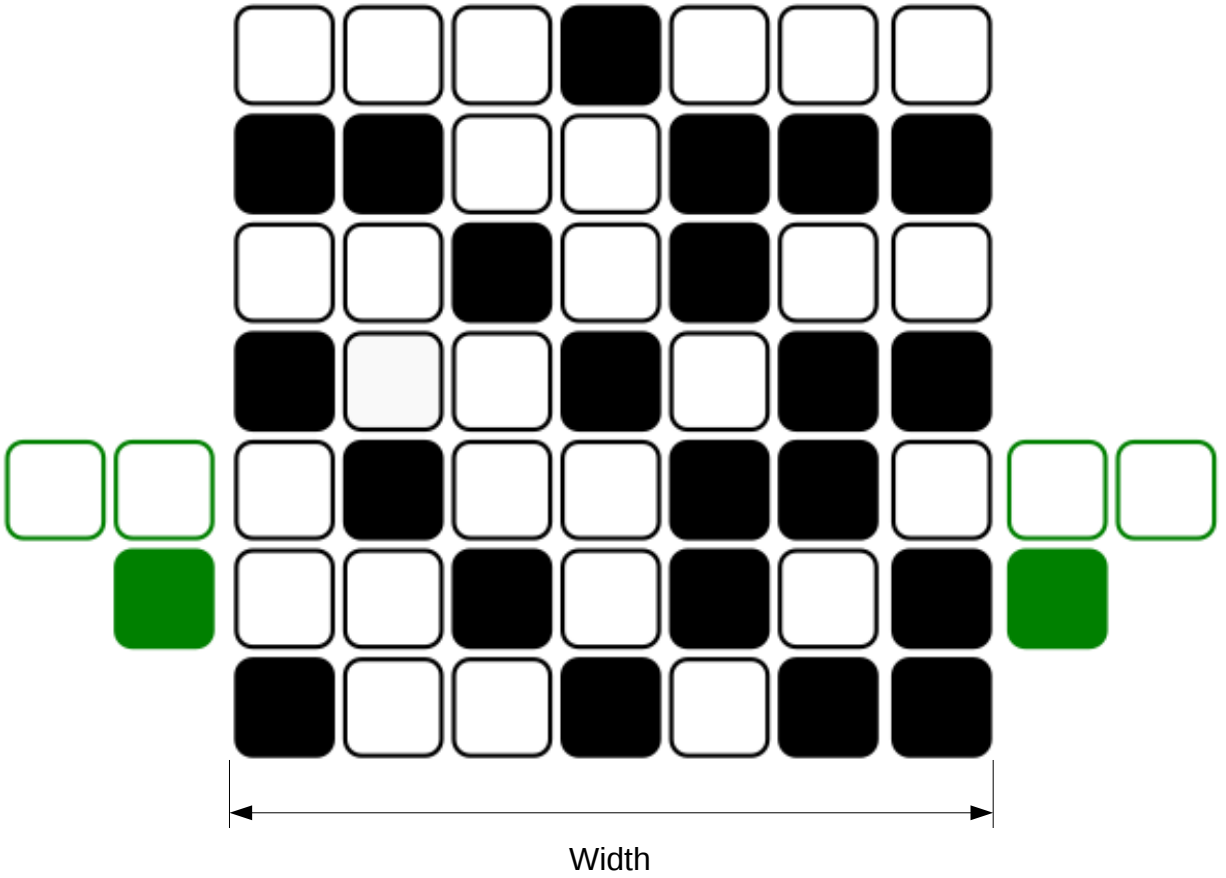
Avoiding of anomalies



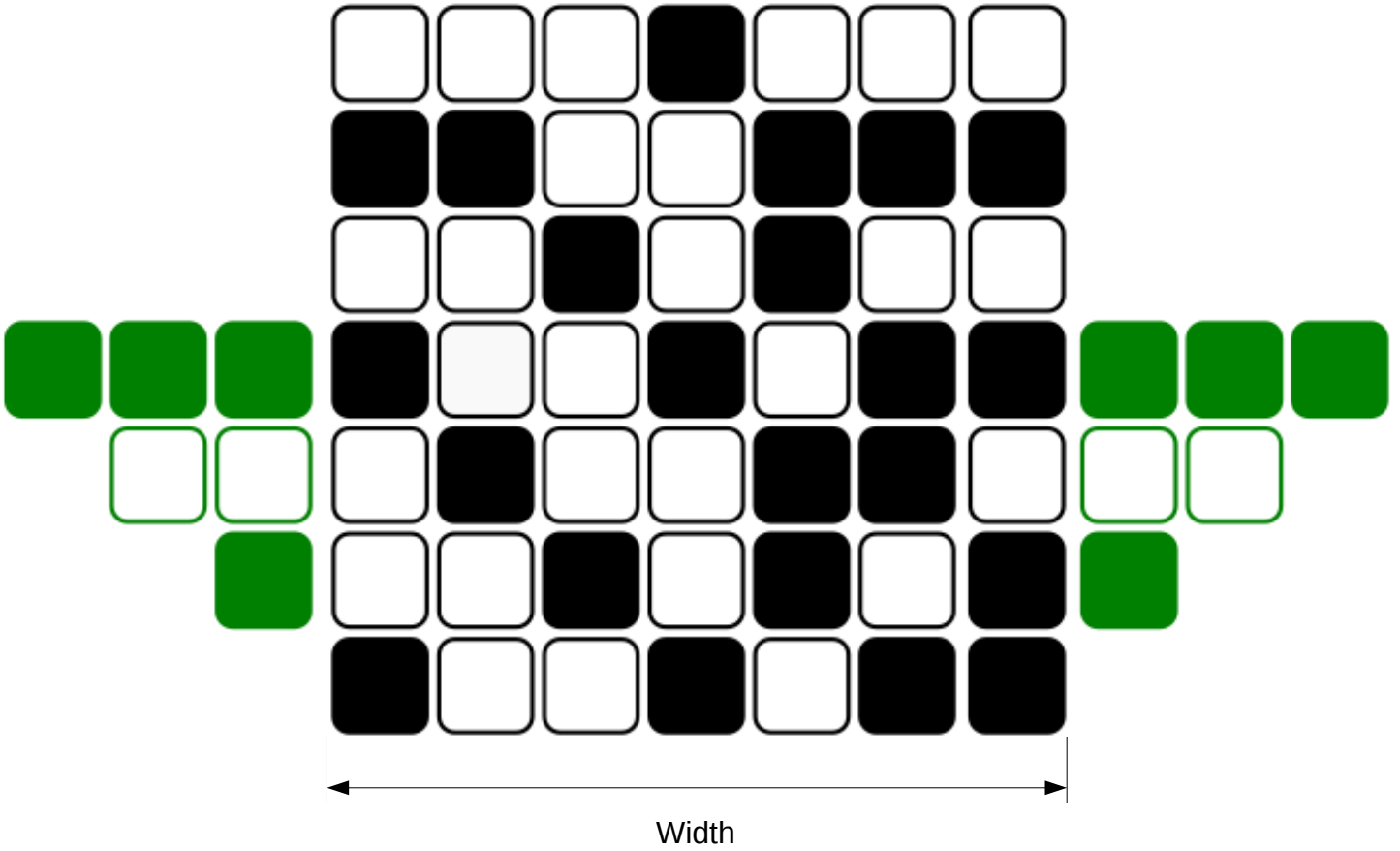
Avoiding of anomalies



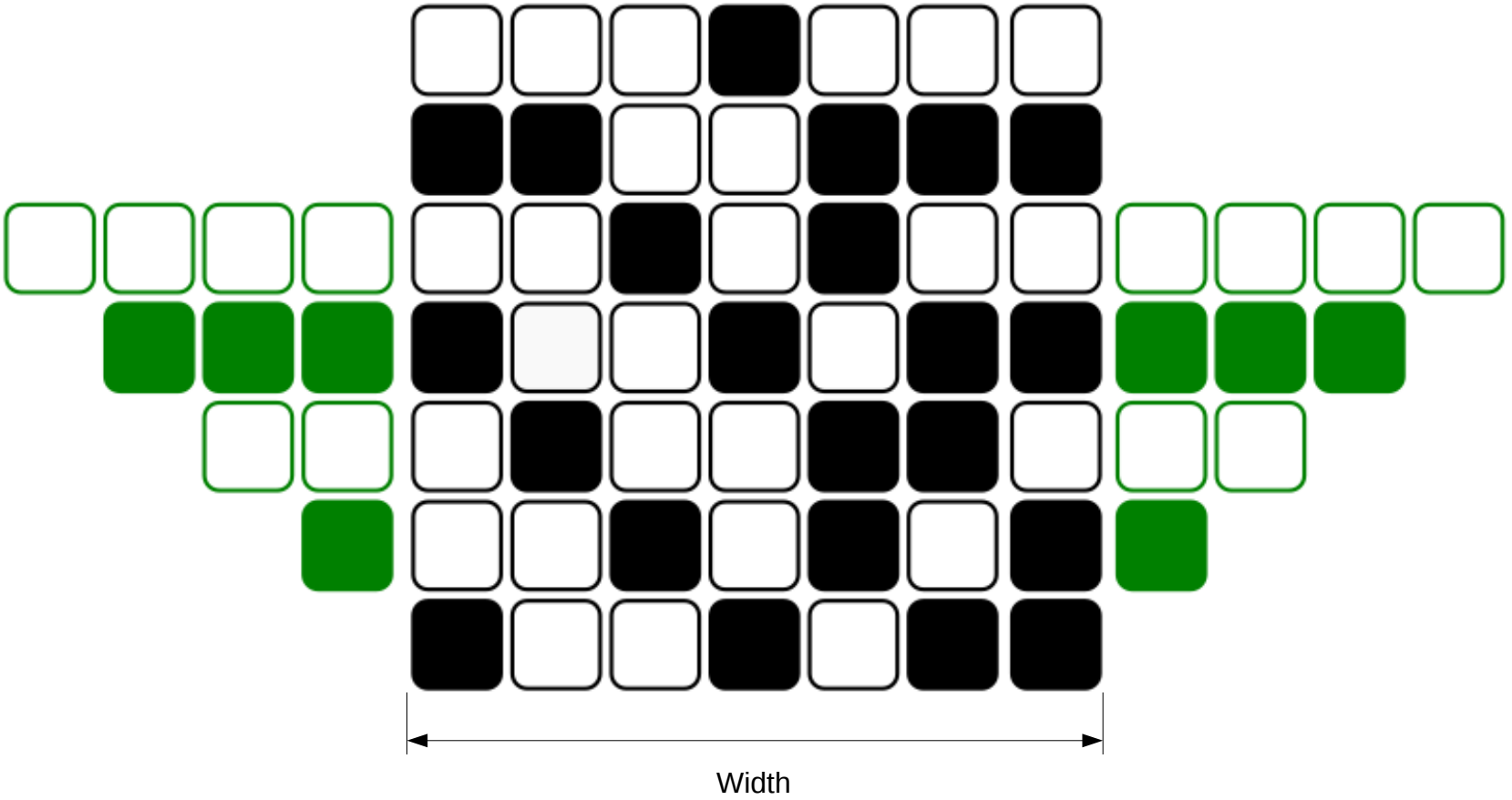
Avoiding of anomalies



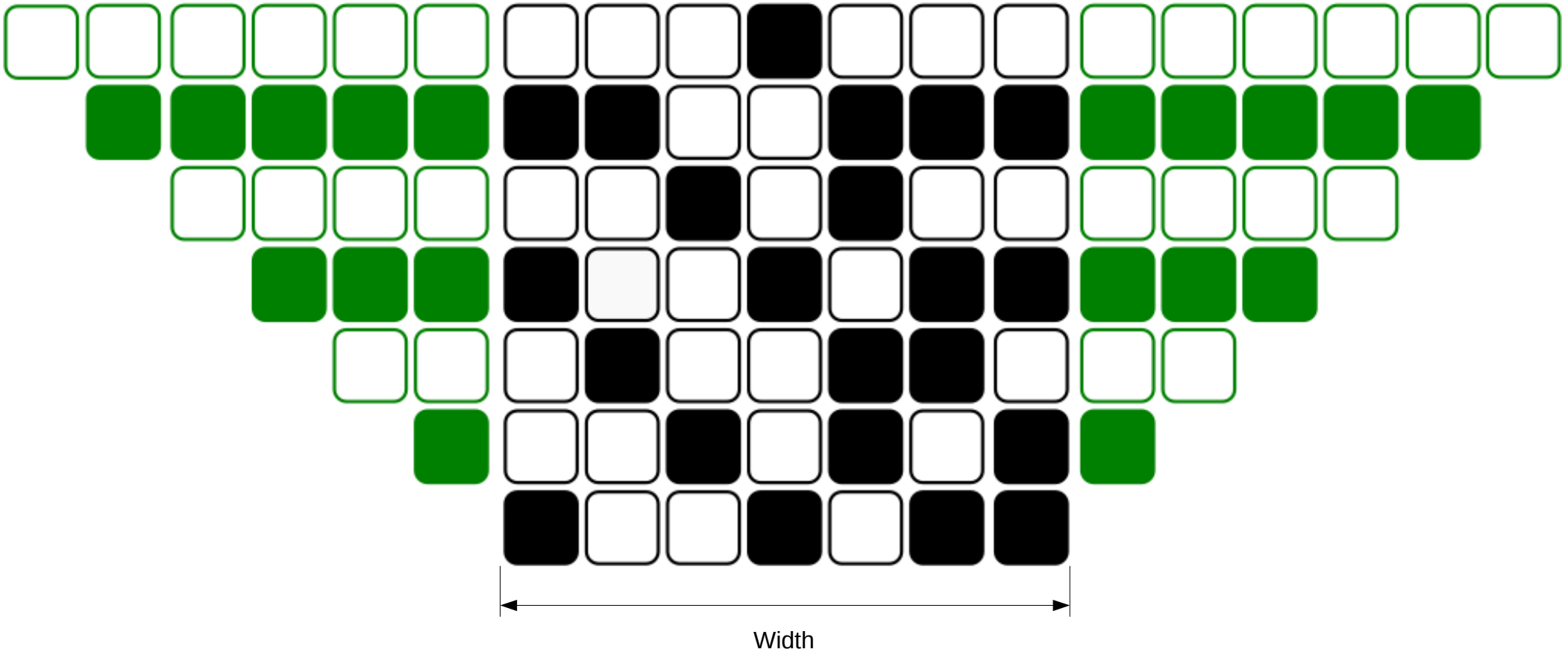
Avoiding of anomalies

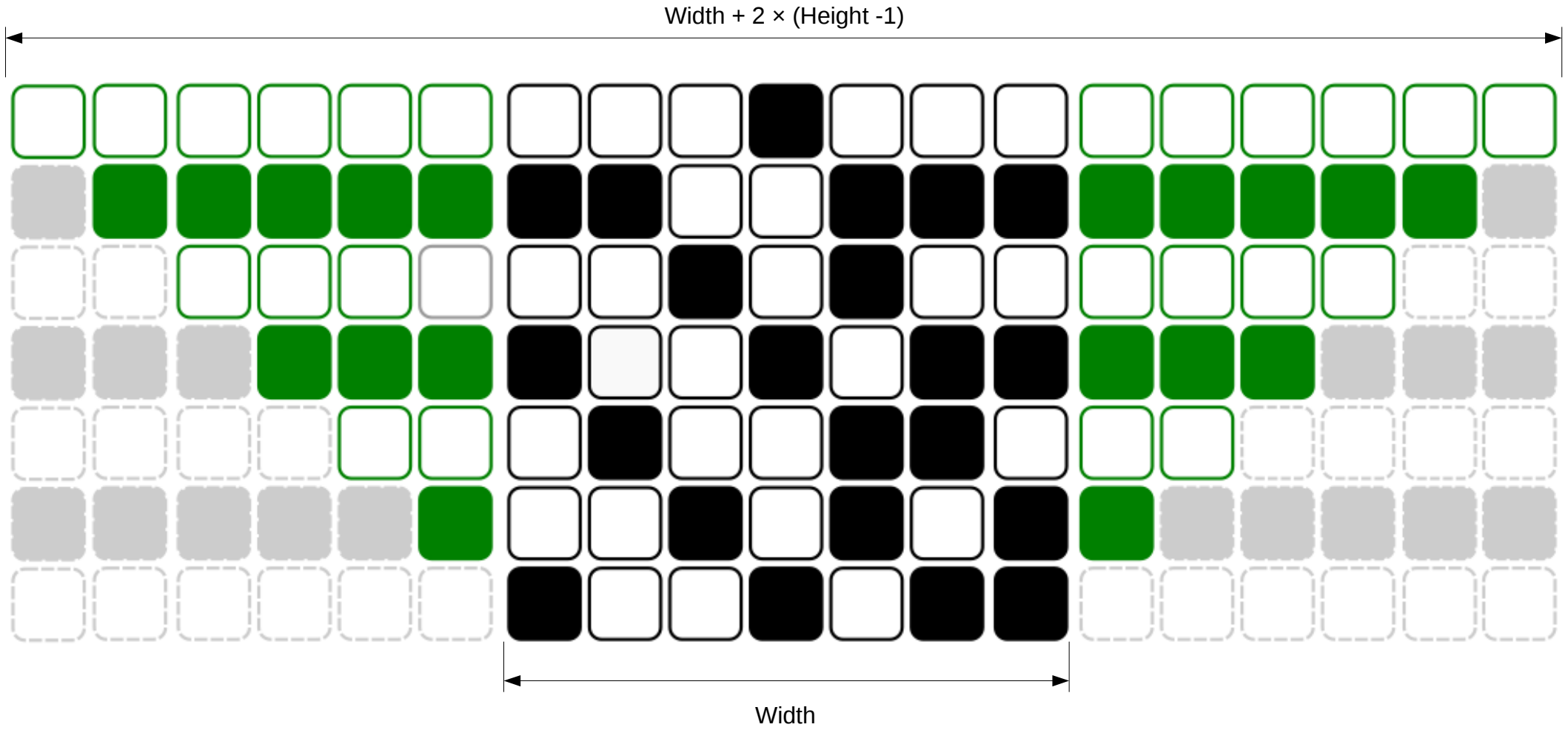


Avoiding of anomalies

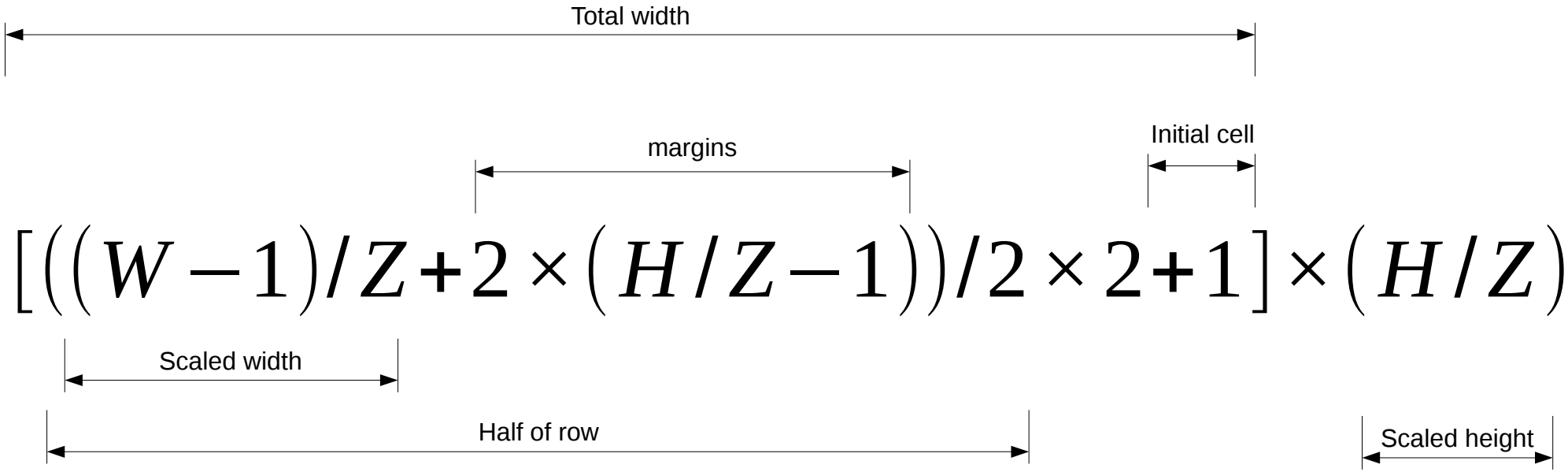


Avoiding of anomalies




Width + 2 × (Height -1)
Width

Total number of automata cells



Step 8: Benchmark!

Testing environment

MSI PE60

Intel Core i7 CPU

16GB RAM (DDR4)

35GB SWAP (SSD)

Linux Mint 18.1

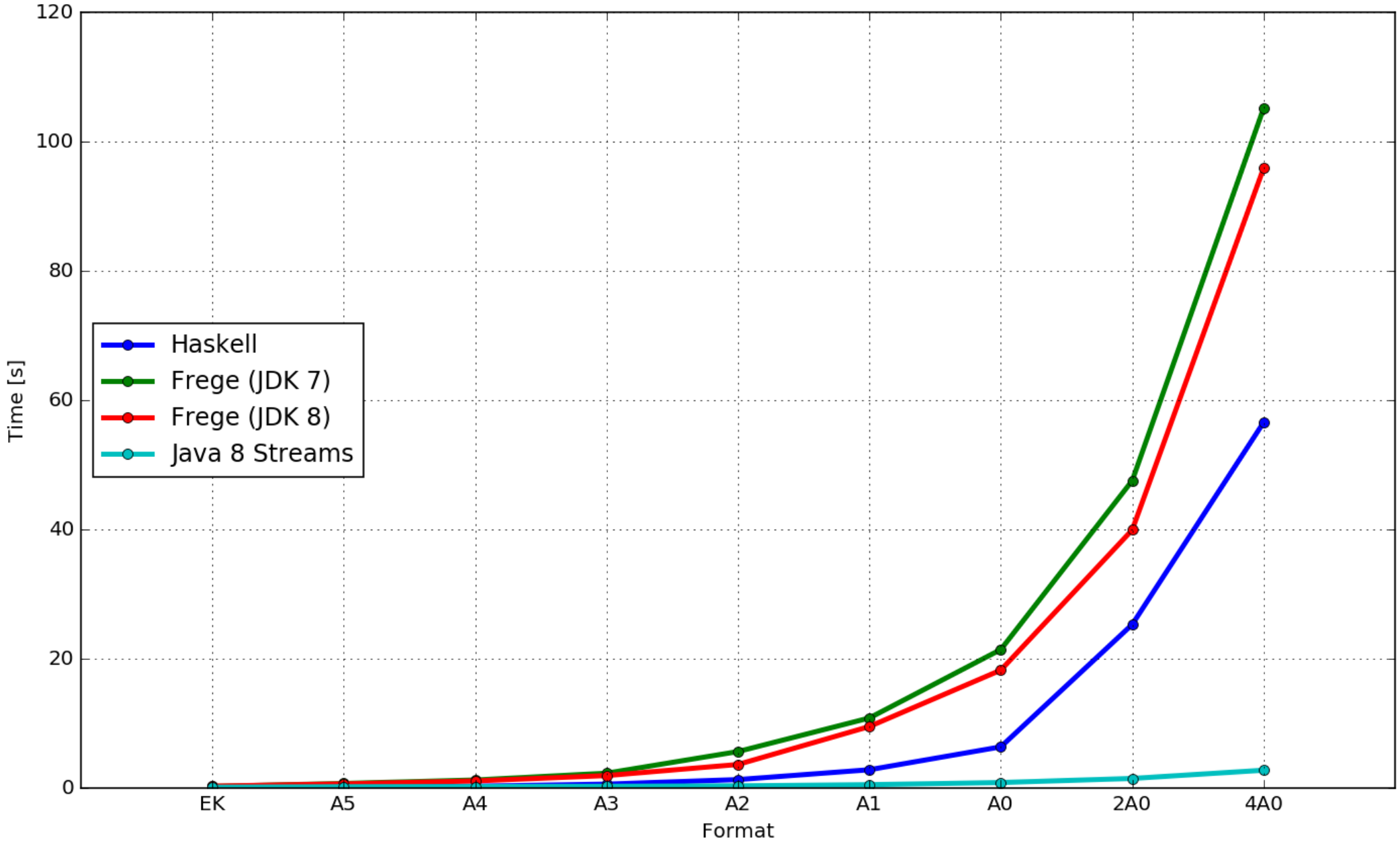
Tests:

x10

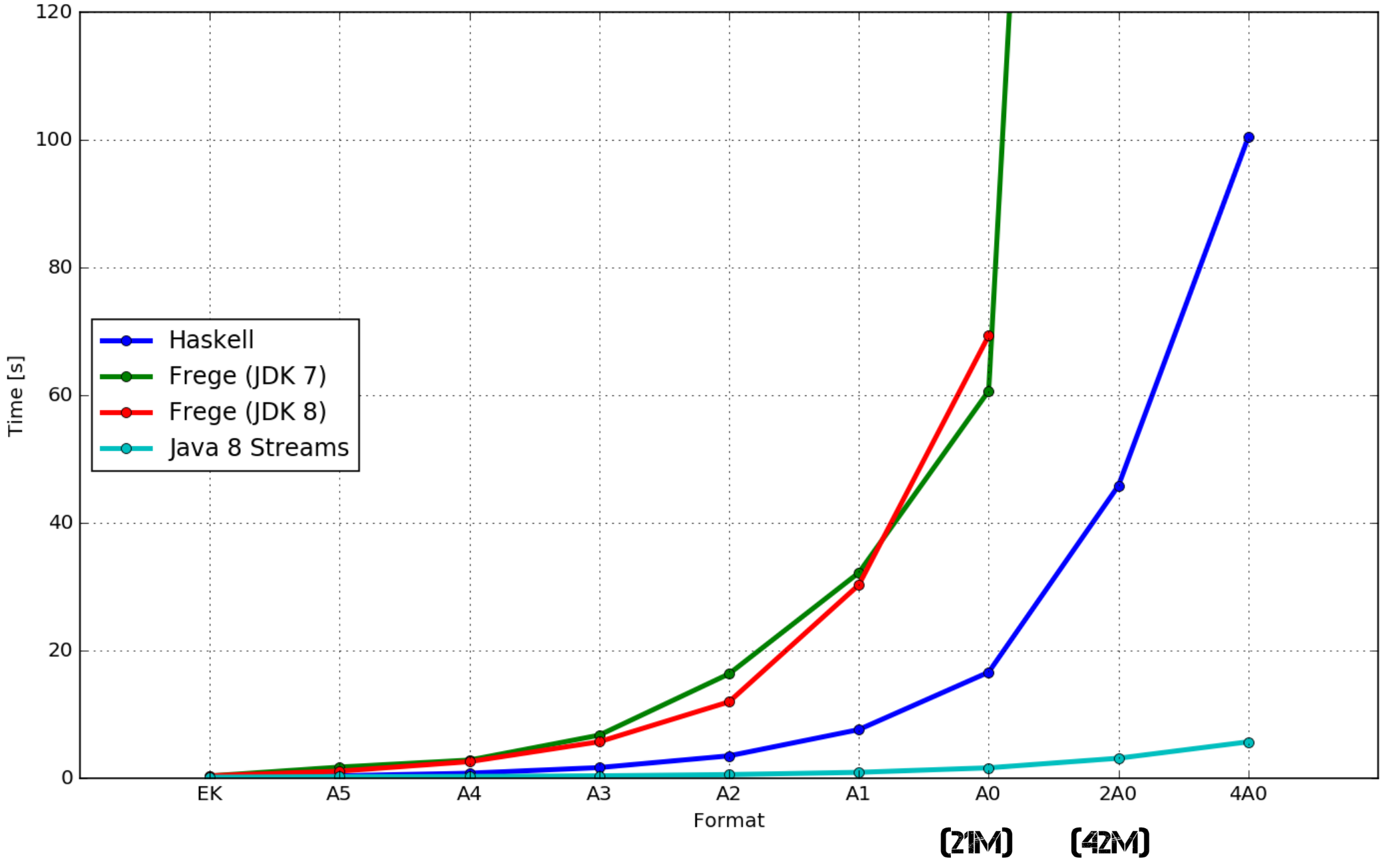
Format	Page [300 dpi]		Zoom			
	Width	Height	12	4	2	1
EK	1 024	768	13 504	122 304	490 368	1 963 776
A5	2 480	1 748	71 775	651 567	2 608 890	10 440 804
A4	3 508	2 480	144 818	1 311 300	5 246 440	20 993 200
A3	4 960	3 508	290 540	2 623 107	10 497 690	42 001 284
A2	7 016	4 960	581 917	5 246 440	20 993 200	83 987 680
A1	9 924	7 016	1 163 912	10 501 198	42 008 300	168 054 248
A0	14 038	9 924	2 332 967	21 011 589	84 061 242	336 254 892
2A0	19 856	14 038	4 665 479	42 034 311	168 196 297	672 827 302
4A0	28 068	19 856	9 336 830	84 105 052	336 430 136	1 345 780 112

TIME LIMIT: 15MIN

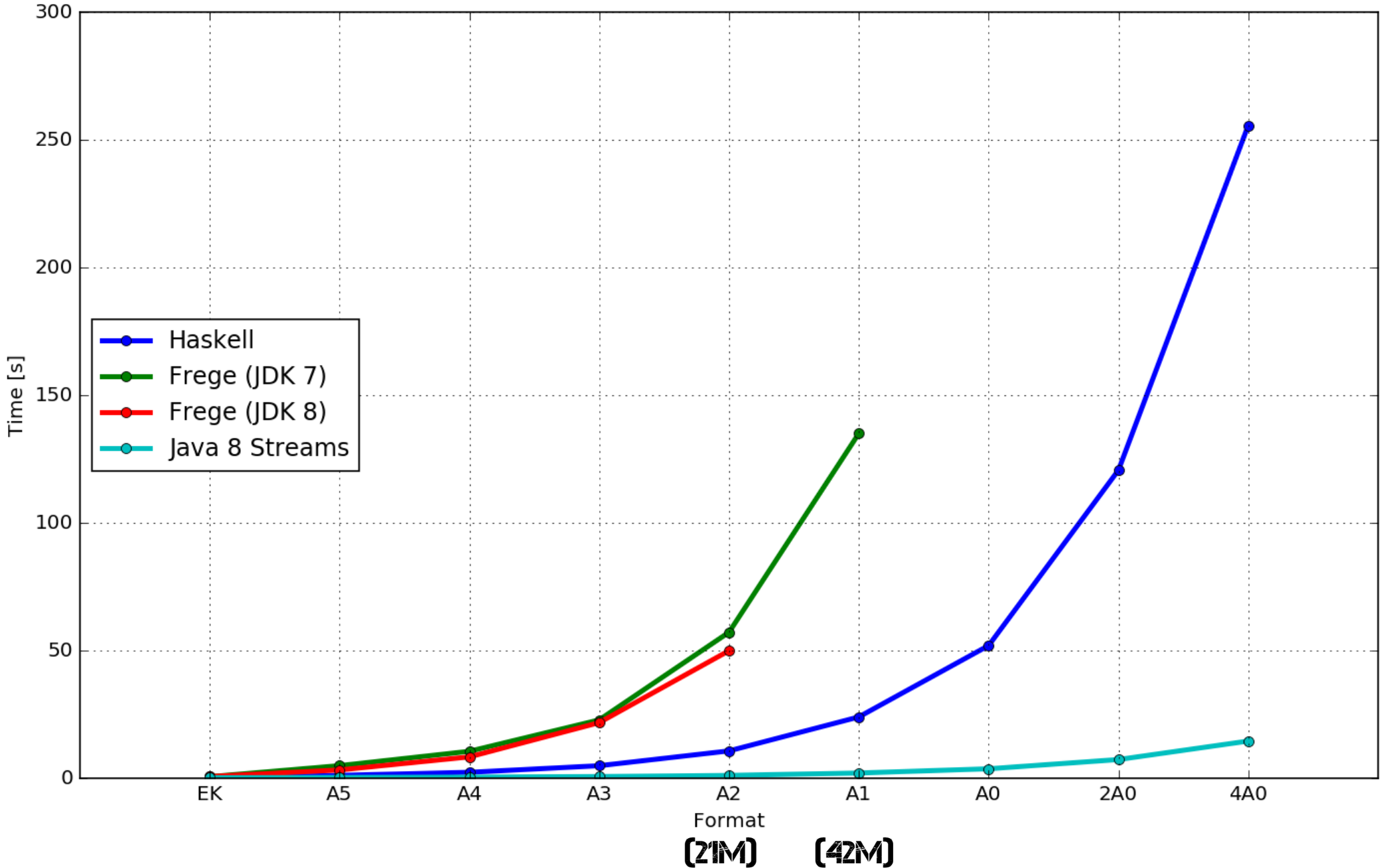
Zoom x12 (13K – 9M cells)



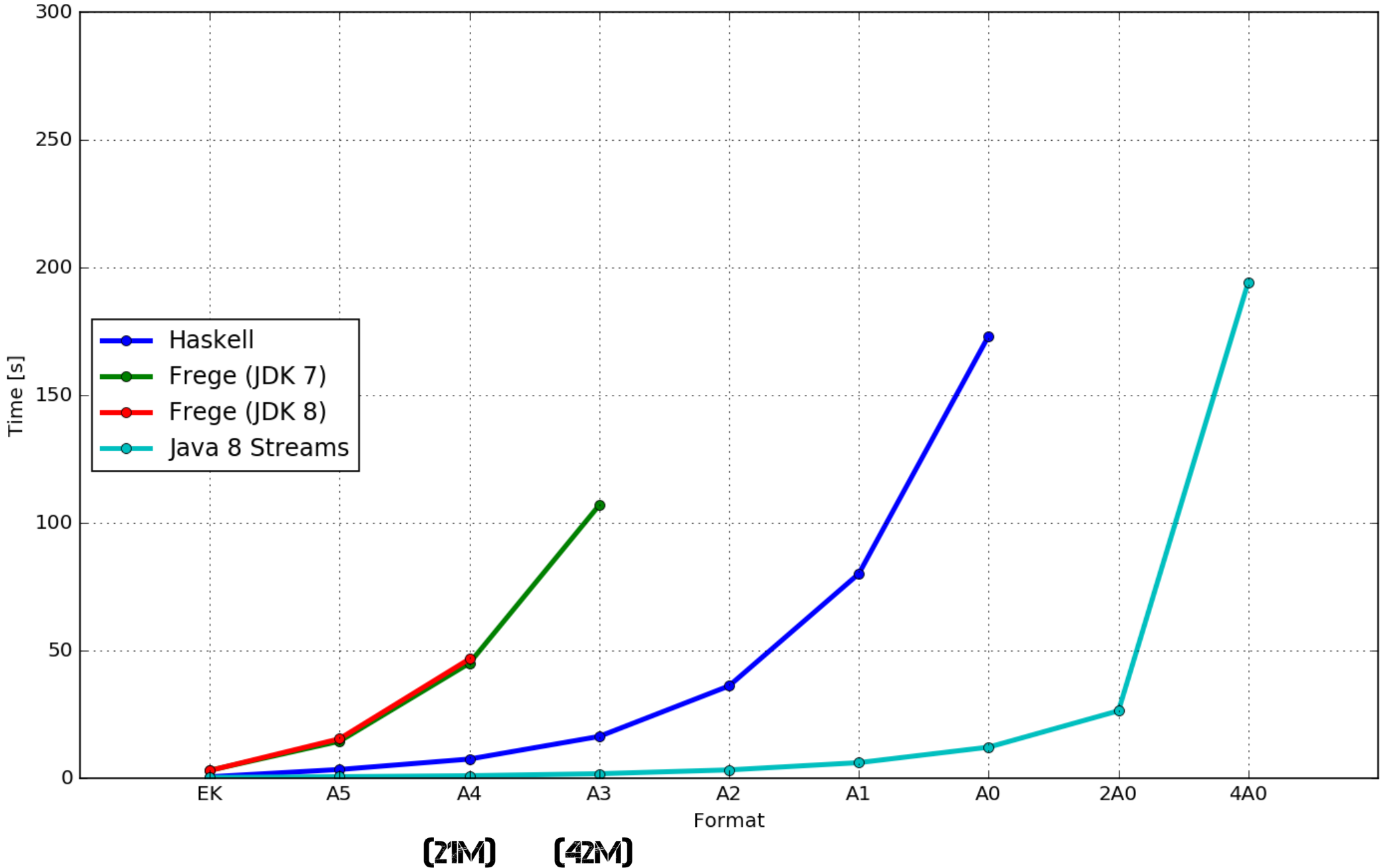
Zoom x4 (122K – 84M cells)



Zoom x2 (0.5M – 336M cells)



Zoom x1 (2M – 1.3B cells)



```
$ JAVA_HOME=/opt/jdk8 PATH=${JAVA_HOME}/bin java -jar ./bin/app8.jar 3508 2480 22 1

Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at frege.run8.Thunk.shared(Thunk.java:297)
    at ModuleAutomata.lambda$null$17(ModuleAutomata.java:265)
    at ModuleAutomata$$Lambda$73/445884362.apply(Unknown Source)
    at frege.prelude.PreludeList.lambda$map$13(PreludeList.java:1912)
    at frege.prelude.PreludeList$$Lambda$17/83954662.call(Unknown Source)
    at frege.run8.Thunk.call(Thunk.java:231)
    at frege.prelude.PreludeBase$INum_Int.f$plus(PreludeBase.java:8341)
    at frege.prelude.PreludeBase$INum_Int.f$plus(PreludeBase.java:8304)
    at frege.prelude.PreludeList.lambda$null$24(PreludeList.java:2105)
    at frege.prelude.PreludeList$$Lambda$57/355629945.call(Unknown Source)
    at frege.run8.Thunk.call(Thunk.java:231)
    at frege.prelude.PreludeList.fold(PreludeList.java:2092)
    at frege.prelude.PreludeList.sum(PreludeList.java:2102)
    at ModuleAutomata.lambda$null$21(ModuleAutomata.java:291)
    at ModuleAutomata$$Lambda$58/1327763628.call(Unknown Source)
    at frege.run8.Thunk.call(Thunk.java:231)
    at frege.prelude.PreludeBase$INum_Int.f$plus(PreludeBase.java:8341)
    at frege.prelude.PreludeBase$INum_Int.f$plus(PreludeBase.java:8304)
    at frege.prelude.PreludeList.lambda$null$24(PreludeList.java:2105)
    at frege.prelude.PreludeList$$Lambda$57/355629945.call(Unknown Source)
    at frege.run8.Thunk.call(Thunk.java:231)
    at frege.prelude.PreludeList.fold(PreludeList.java:2092)
    at frege.prelude.PreludeList.sum(PreludeList.java:2102)
    at ModuleAutomata.lambda$null$19(ModuleAutomata.java:283)
    at ModuleAutomata$$Lambda$40/200006406.call(Unknown Source)
    at frege.run8.Thunk.call(Thunk.java:231)
    at ModuleAutomata.getHashCode(ModuleAutomata.java:298)
    at Main.lambda$null$9(Main.java:176)
    at Main$$Lambda$37/1512981843.call(Unknown Source)
    at frege.run8.Thunk.call(Thunk.java:231)
    at Main.lambda$null$7(Main.java:149)
    at Main$$Lambda$36/1768305536.call(Unknown Source)
```

```
$ JAVA_HOME=/opt/jdk8 PATH=${JAVA_HOME}/bin java -Xmx50G -jar ./bin/app8.jar 19856 14038 22 4

Exception in thread "main" java.lang.StackOverflowError
	at frege.run8.Thunk.call(Thunk.java:210)
	at ModuleAutomata.getTriplesFromRow(ModuleAutomata.java:217)
	at ModuleAutomata.lambda$getTriplesFromRow$15(ModuleAutomata.java:241)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList.lambda$map$14(PreludeList.java:1914)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList$IListMonoid_$lbrack$rbrack.lambda$$plus$plus$3(PreludeList.j
	at frege.run8.Thunk.call(Thunk.java:231)
	at ModuleAutomata.getTriplesFromRow(ModuleAutomata.java:217)
	at ModuleAutomata.lambda$getTriplesFromRow$15(ModuleAutomata.java:241)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList.lambda$map$14(PreludeList.java:1914)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList$IListMonoid_$lbrack$rbrack.lambda$$plus$plus$3(PreludeList.j
	at frege.run8.Thunk.call(Thunk.java:231)
	at ModuleAutomata.getTriplesFromRow(ModuleAutomata.java:217)
	at ModuleAutomata.lambda$getTriplesFromRow$15(ModuleAutomata.java:241)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList.lambda$map$14(PreludeList.java:1914)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList$IListMonoid_$lbrack$rbrack.lambda$$plus$plus$3(PreludeList.j
	at frege.run8.Thunk.call(Thunk.java:231)
	at ModuleAutomata.getTriplesFromRow(ModuleAutomata.java:217)
	at ModuleAutomata.lambda$getTriplesFromRow$15(ModuleAutomata.java:241)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList.lambda$map$14(PreludeList.java:1914)
	at frege.run8.Thunk.call(Thunk.java:231)
	at frege.prelude.PreludeList$IListMonoid_$lbrack$rbrack.lambda$$plus$plus$3(PreludeList.j
	at frege.run8.Thunk.call(Thunk.java:231)
	at ModuleAutomata.getTriplesFromRow(ModuleAutomata.java:217)
	at ModuleAutomata.lambda$getTriplesFromRow$15(ModuleAutomata.java:241)
	...
```

Execution time [s]

„-Xmx50G"

Haskell	Zoom			
	12	4	2	1
EK	0,023	0,056	0,175	0,558
A5	0,131	0,380	1,117	3,412
A4	0,279	0,780	2,310	7,494
A3	0,602	1,665	4,892	16,358
A2	1,305	3,503	10,696	36,134
A1	2,828	7,615	23,996	80,093
A0	6,370	16,606	51,956	173,236
2A0	25,299	45,826	120,645	TL
4A0	56,564	100,396	255,658	TL

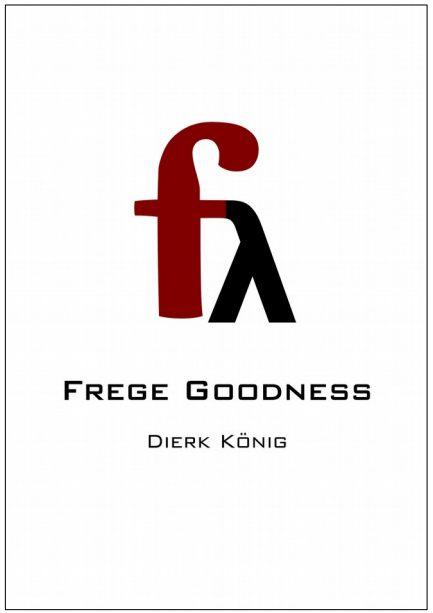
Frege (JDK 8)	Zoom			
	12	4	2	1
EK	0,300	0,380	0,660	3,007
A5	0,638	1,110	3,238	15,440
A4	1,096	2,595	8,357	46,696
A3	1,922	5,712	21,772	x
A2	3,642	11,962	49,935	x
A1	9,515	30,285	x	x
A0	18,237	69,318	x	x
2A0	39,925	x	x	x
4A0	95,896	x	x	x

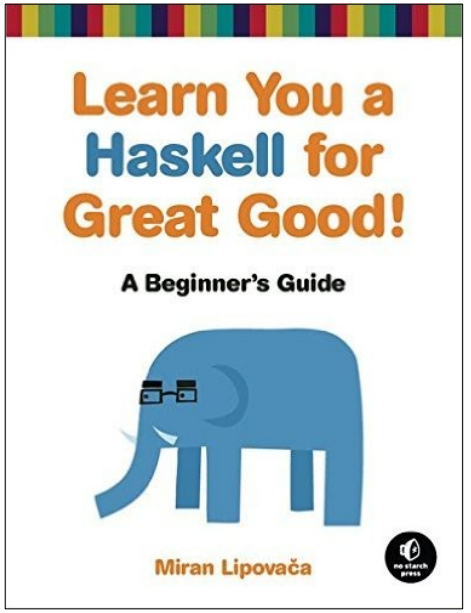
Frege (JDK 7)	Zoom			
	12	4	2	1
EK	0,248	0,350	0,601	3,088
A5	0,704	1,740	4,919	14,472
A4	1,239	2,838	10,542	45,045
A3	2,292	6,741	22,833	106,921
A2	5,640	16,347	57,165	x
A1	10,836	32,190	135,026	x
A0	21,425	60,658	x	x
2A0	47,543	432,005	x	x
4A0	105,167	x	x	x

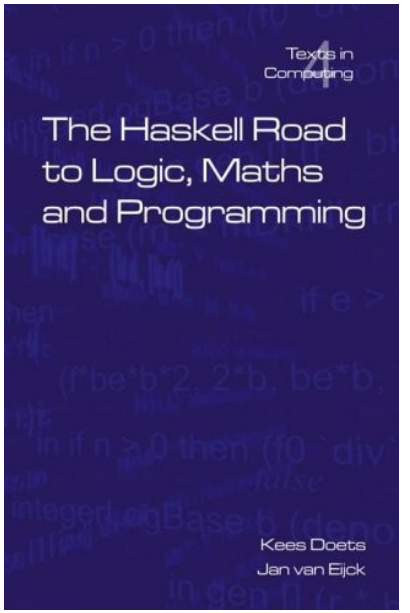
Java	Zoom			
	12	4	2	1
EK	0,131	0,156	0,203	0,291
A5	0,196	0,236	0,325	0,594
A4	0,223	0,279	0,423	0,973
A3	0,265	0,380	0,646	1,757
A2	0,342	0,565	1,097	3,253
A1	0,516	0,919	2,031	6,091
A0	0,847	1,628	3,694	12,159
2A0	1,479	3,130	7,313	26,392
4A0	2,752	5,698	14,551	194,250

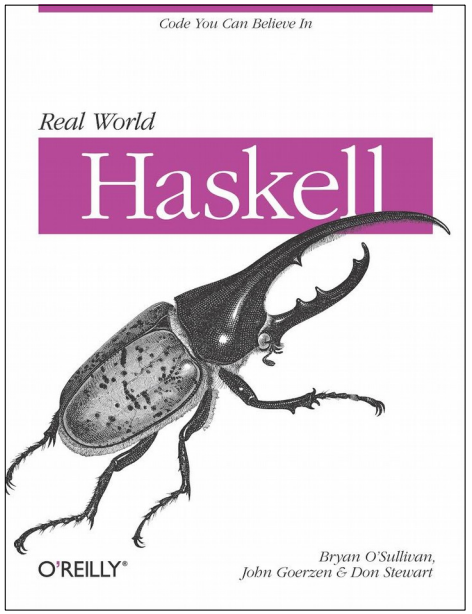
Conclusion...

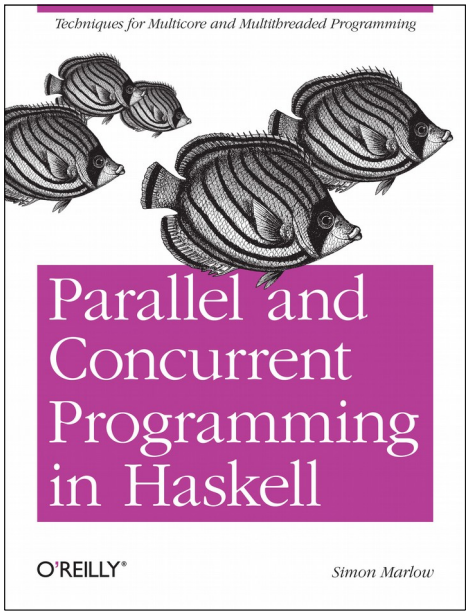
-Books -


FREGE GOODNESS
DIERK KÖNIG


Learn You a Haskell for Great Good!
A Beginner's Guide
Miran Lipovača
no starch press

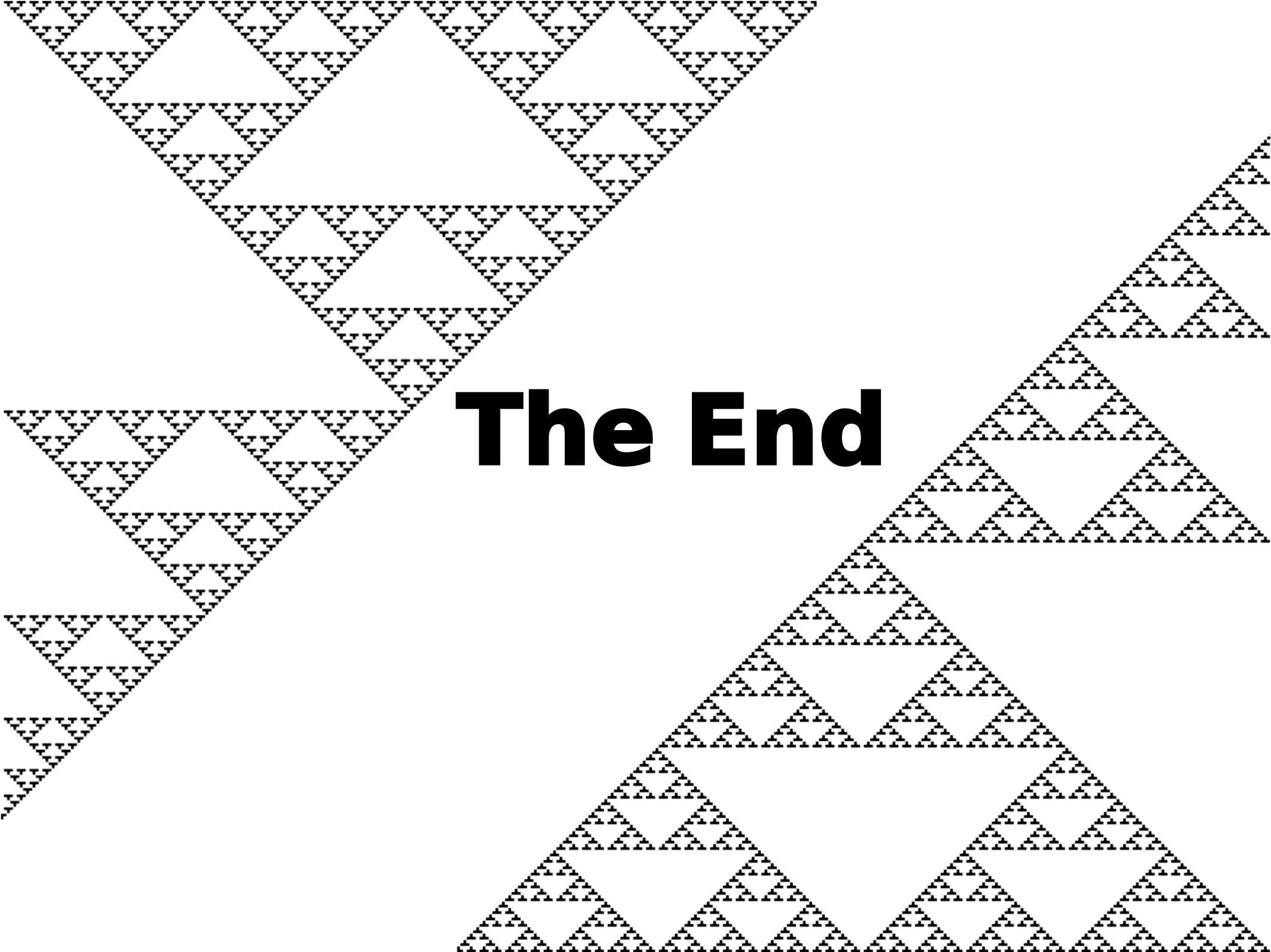

Texts in Computing
The Haskell Road to Logic, Maths and Programming
Kees Doets
Jan van Eijck


Code You Can Believe In
Real World
Haskell
O'REILLY
Bryan O'Sullivan,
John Goerzen & Don Stewart


Techniques for Multicore and Multithreaded Programming
Parallel and Concurrent Programming in Haskell
O'REILLY
Simon Marlow

All examples are placed on GitHub:

https://github.com/phiotr/JUG30--Examples


The End