CHICAGO
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2015

goto; conference

Impossible Programs

Tom Stuart

Conference: May 11-12 / Workshops: 13-14

IMPOSSIBLE PROGRAMS

@tomstuart / GOTO Chicago / 2015-05-11

PROGRAMS

CAN'T DO

EVERYTHING

IMPOSSIBLE PROGRAMS

@tomstuart / GOTO Chicago / 2015-05-11

how can a

PROGRAM

be

IMPOSSIBLE?

WE DEMAND

UNIVERSAL SYSTEMS

Compare two programming languages, say Python and Ruby.

We can translate any Python program into Ruby.
We can translate any Ruby program into Python.

We can implement a Python **interpreter** in Ruby.
We can implement a Ruby **interpreter** in Python.

We can implement a Python interpreter in JavaScript.
We can implement a JavaScript interpreter in Python.

Tag systems

SKI calculus

Game of Life

Ruby

Lisp

Register machines

XSLT

JavaScript

Magic: The Gathering

Partial recursive functions

C

Python

Java

Lambda calculus

Turing machines

Rule 110

C++

Haskell

Universal systems can run **software**.

We don't just want machines, we want **general-purpose** machines.

PROGRAMS ARE DATA

```
>> puts 'hello world'
hello world
=> nil

>> program = "puts 'hello world'"
=> "puts 'hello world'"

>> bytes_in_binary = program.bytes.
                       map { |byte| byte.to_s(2).rjust(8, '0') }
=> ["01110000", "01110101", "01110100", "01110011", "00100000",
    "00100111", "01101000", "01100101", "01101100", "01101100",
    "01101111", "00100000", "01110111", "01101111", "01110010",
    "01101100", "01100100", "00100111"]

>> number = bytes_in_binary.join.to_i(2)
=> 979654384950070652110298049571774002183479 1
```

```
>> number = 97965438495007065211029804957174002183479 1
=> 97965438495007065211029804957174002183479 1

>> bytes_in_binary = number.to_s(2).scan(/.+?(?=.{8}*\z)/)
=> [ "1110000", "01110101", "01110100", "01110011", "00100000",
     "00100111", "01101000", "01100101", "01101100", "01101100",
     "01101111", "00100000", "01110111", "01101111", "01110010",
     "01101100", "01100100", "00100111"]

>> program = bytes_in_binary.map { |string| string.to_i(2).chr }.join
=> "puts 'hello world'"

>> eval program
hello world
=> nil
```

UNIVERSAL SYSTEMS

+

PROGRAMS ARE DATA

=

INFINITE LOOPS

Every universal system can simulate every other universal system, including itself.

More specifically: every universal programming language can implement its own interpreter.

```ruby
def evaluate(program, input)
  # parse program
  # evaluate program on input while capturing output
  # return output
end
```

```
>> evaluate('print $stdin.read.reverse', 'hello world')
=> "dlrow olleh"
```

```ruby
def evaluate(program, input)
  # parse program
  # evaluate program on input while capturing output
  # return output
end

def evaluate_on_itself(program)
  evaluate(program, program)
end
```

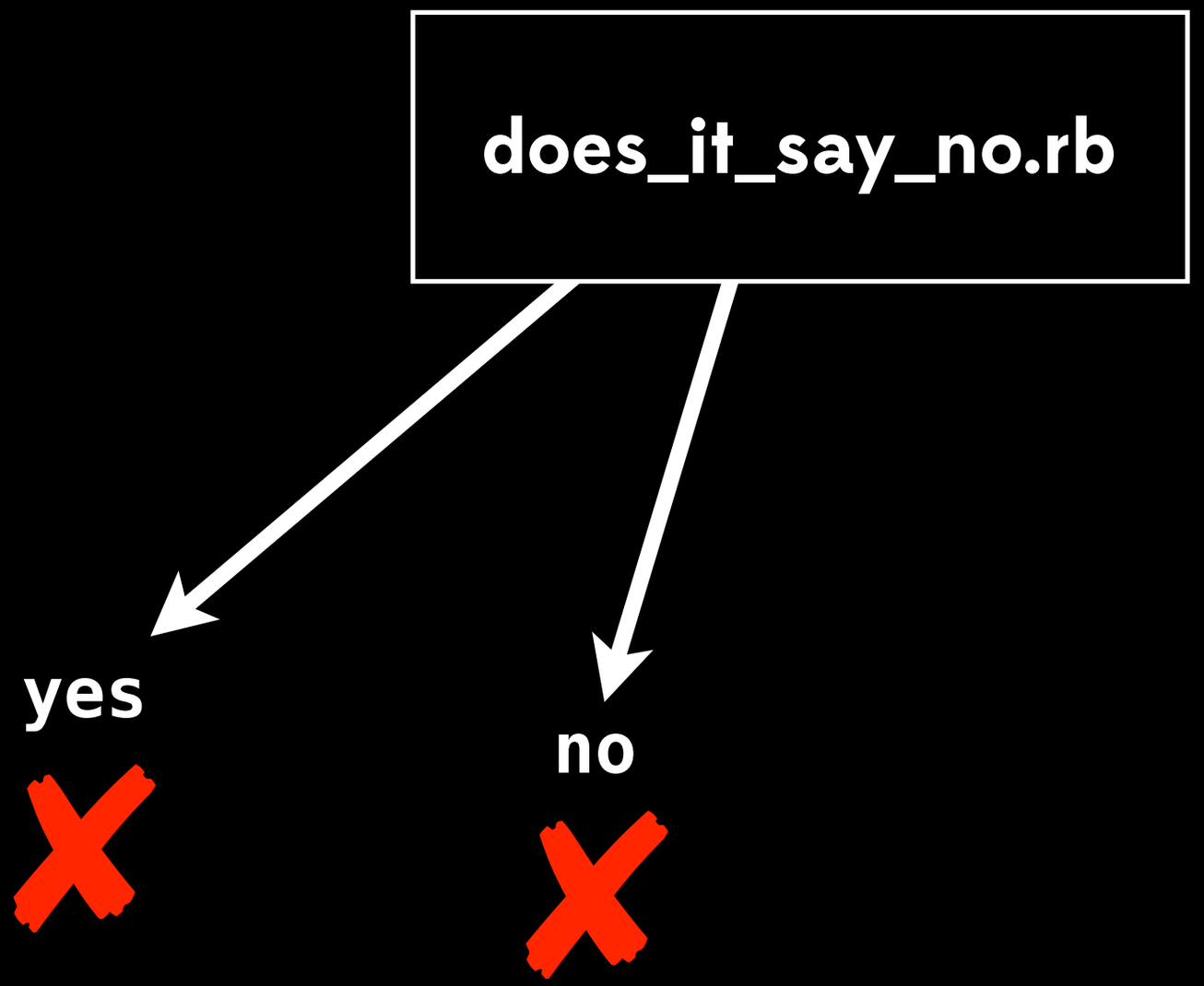
```
>> evaluate_on_itself('print $stdin.read.reverse')
=> "esrever.daer.nidts$ tnirp"
```

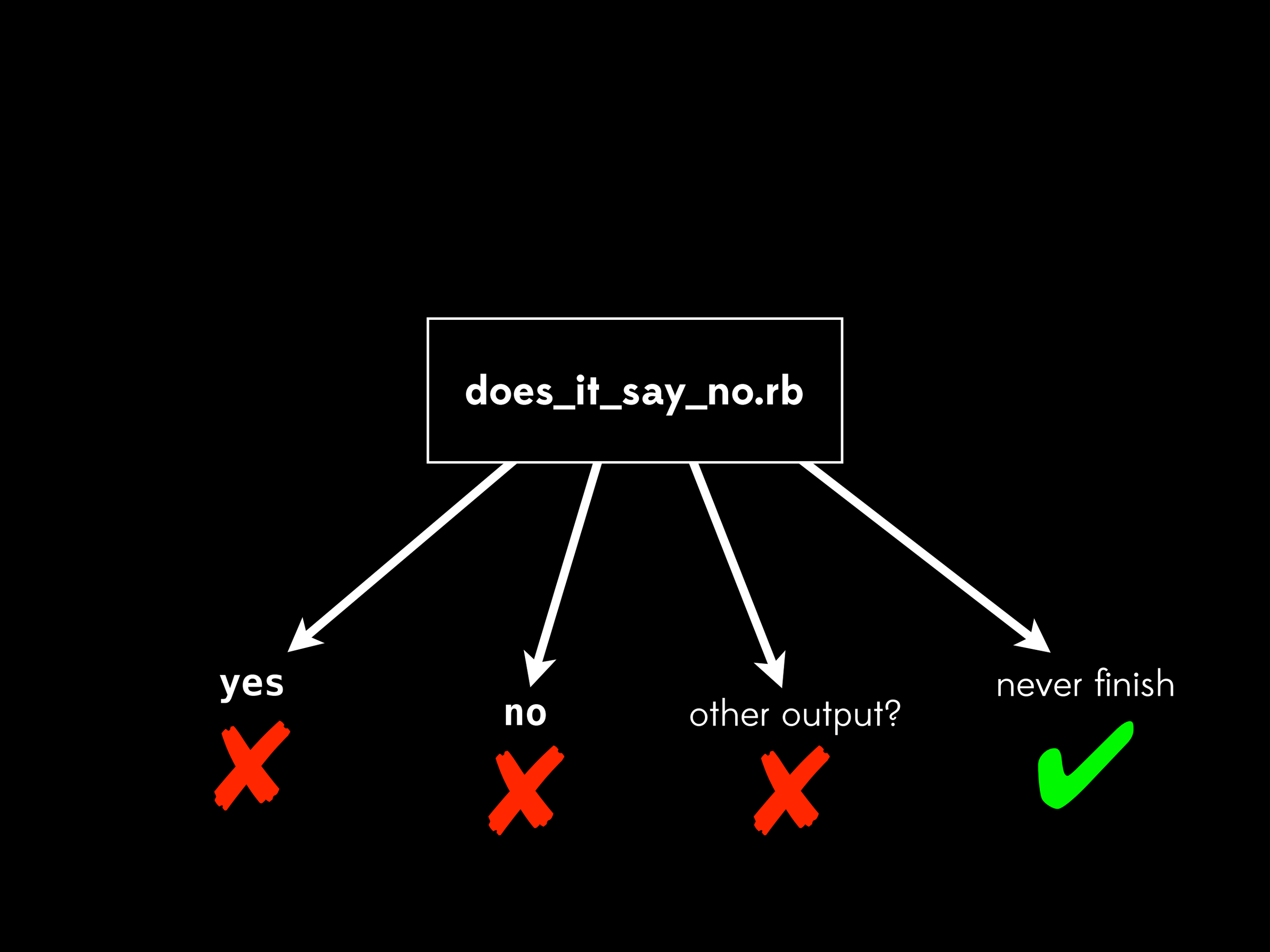
```ruby
def evaluate(program, input)
  # parse program
  # evaluate program on input while capturing output
  # return output
end

def evaluate_on_itself(program)
  evaluate(program, program)
end
```

```ruby
program = $stdin.read

if evaluate_on_itself(program) == 'no'
  print 'yes'
else
  print 'no'
end
```

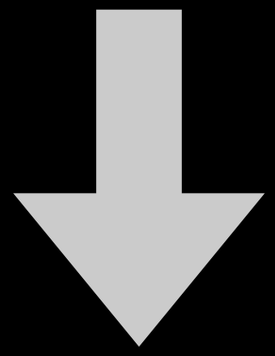
does_it_say_no.rb

```
$ echo 'print $stdin.read.reverse' | ruby does_it_say_no.rb
no

$ echo 'print "no" if $stdin.read.include?("no")' | ruby does_it_say_no.rb
yes

$ ruby does_it_say_no.rb < does_it_say_no.rb
???
```

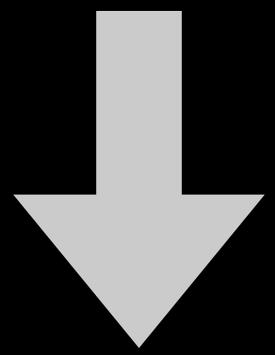

does_it_say_no.rb
yes
no


does_it_say_no.rb
yes
no
other output?
never finish

Ruby is universal



so we can write **`#evaluate`** in it



so we can construct a special program that loops forever

so here's one

IMPOSSIBLE PROGRAM

Sometimes infinite loops are bad.

We could remove features from a language until there's no way to cause an infinite loop.

- **No unlimited iteration**

  remove **`while`** loops etc, only allow iteration over finite data structures

- **No lambdas**

  to prevent **`(λx.x x)(λx.x x)`**

- **No recursive function calls**

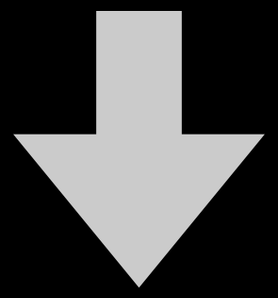
  e.g. only allow a function to call other functions whose names come later in the alphabet

- **No blocking I/O**
- **…**

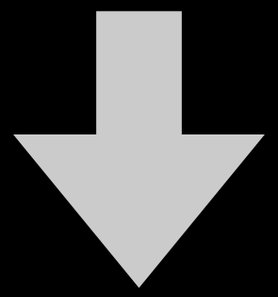
The result is called a **total** programming language.

It must be impossible to write an interpreter for a total language in itself.

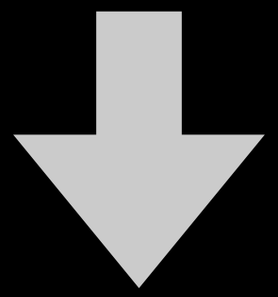
if we could write **`#evaluate`** in a total language



then we could use it to construct a special program
that loops forever



but a total language doesn't let you write programs
that loop forever



so it must be impossible to write **`#evaluate`** in one

(That's weird, because a total language's interpreter always finishes eventually, so it feels like the kind of program we should be able to write.)

We **could** write an interpreter for a total language in a universal language, or in **some other** more powerful total language.

okay but

WHAT ABOUT REALITY?

`#evaluate` is an impossible program for any total language, which means that total languages can't be universal.

Universal systems have impossible programs too.

```
input = $stdin.read
puts input.upcase
```

This program always finishes.*

* assuming **`STDIN`** is finite & nonblocking

```
input = $stdin.read

while true
  # do nothing
end

puts input.upcase
```

This program always loops forever.

Can we write a program that can decide this in general?

(This question is called the **halting problem**.)

```
input = $stdin.read
output = ''

n = input.length

until n.zero?
  output = output + '*'
  n = n - 1
end

puts output
```

```ruby
require 'prime'

def primes_less_than(n)
  Prime.each(n - 1).entries
end

def sum_of_two_primes?(n)
  primes = primes_less_than(n)
  primes.any? { |a| primes.any? { |b| a + b == n } }
end

n = 4

while sum_of_two_primes?(n)
  n = n + 2
end

print n
```

```ruby
def halts?(program, input)
  # parse program
  # analyze program
  # return true if program halts on input, false if not
end
```

```
>> halts?('print $stdin.read', 'hello world')
=> true
```

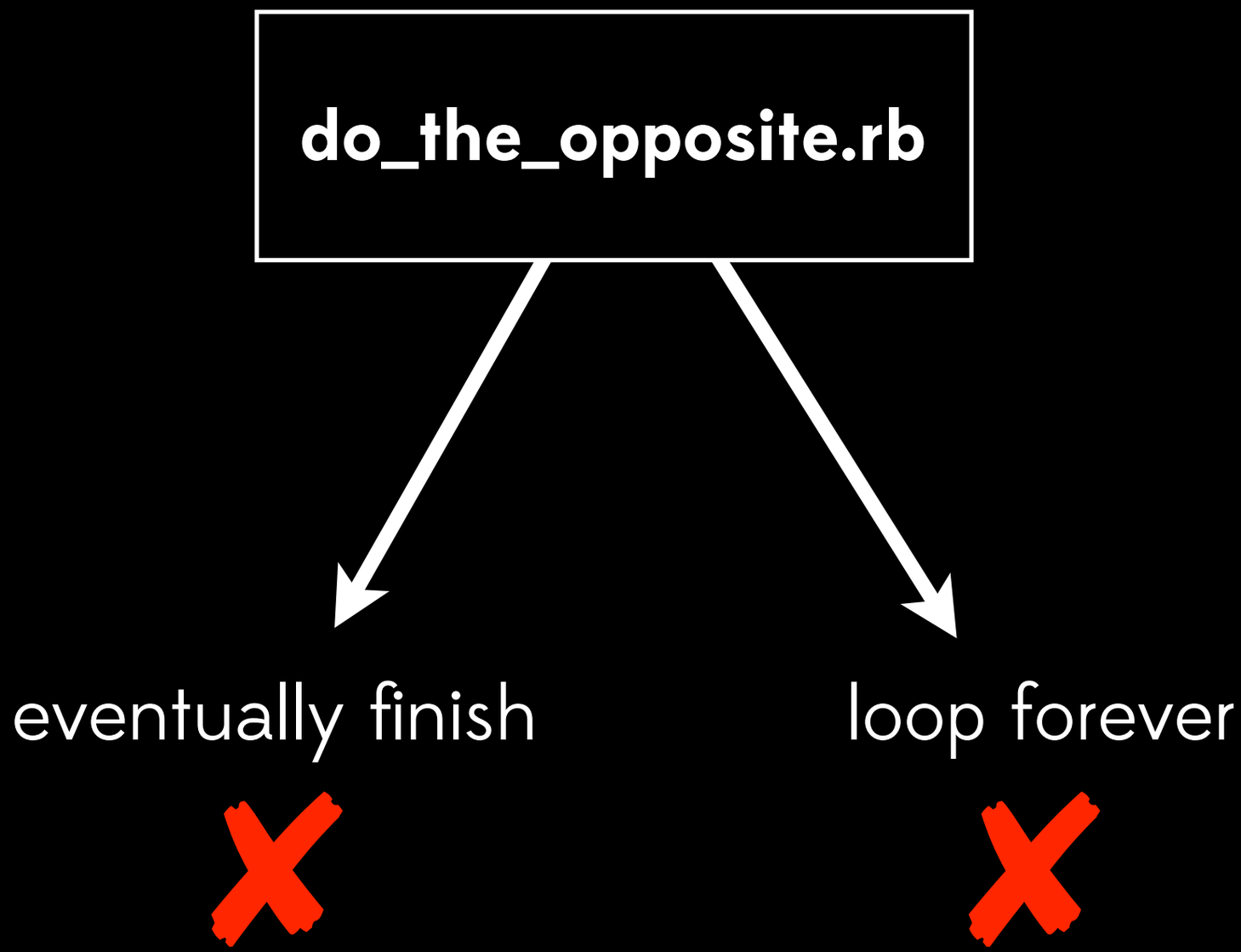
```
>> halts?('while true do end', 'hello world')
=> false
```

```ruby
def halts?(program, input)
  # parse program
  # analyze program
  # return true if program halts on input, false if not
end

def halts_on_itself?(program)
  halts?(program, program)
end
```

```ruby
program = $stdin.read

if halts_on_itself?(program)
  while true
    # do nothing
  end
end
```

do_the_opposite.rb

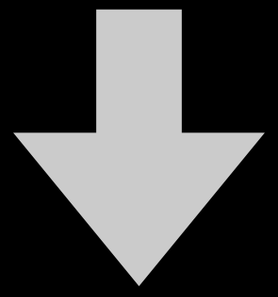
```
$ ruby do_the_opposite.rb < do_the_opposite.rb
```


do_the_opposite.rb
eventually finish
loop forever

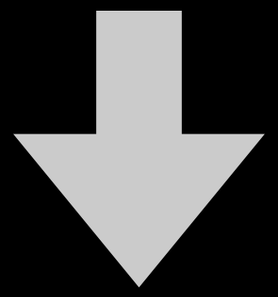
Every real program must either loop forever or not, but whichever happens, **`#halts?`** will be wrong about it.

do_the_opposite.rb forces **`#halts?`** to give the wrong answer.

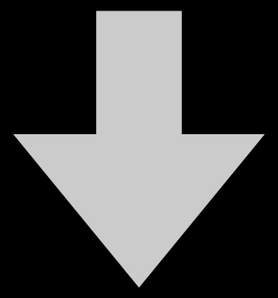
if we could write **`#halts?`**



then we could use it to construct a special program
that forces **`#halts?`** to give the wrong answer



but a correct implementation of **`#halts?`**
would always give the right answer



so it must be impossible to write **`#halts?`**

okay but

WHO CARES?

We never actually want to ask a computer whether a program will loop forever.

But we often want to ask computers other questions about programs.

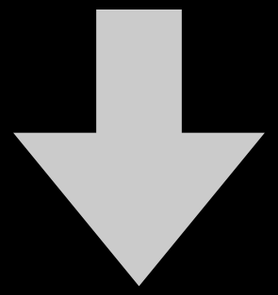
```ruby
def prints_hello_world?(program, input)
  # parse program
  # analyze program
  # return true if program prints "hello world", false if not
end
```

```
>> prints_hello_world?('print $stdin.read.reverse', 'dlrow olleh')
=> true
```

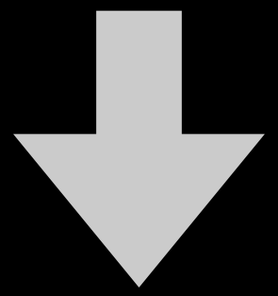
```
>> prints_hello_world?('print $stdin.read.upcase', 'dlrow olleh')
=> false
```

```ruby
def prints_hello_world?(program, input)
  # parse program
  # analyze program
  # return true if program prints "hello world", false if not
end

def halts?(program, input)
  hello_world_program = %Q{
    program = #{program.inspect}
    input = $stdin.read
    evaluate(program, input)
    print 'hello world'
  }

  prints_hello_world?(hello_world_program, input)
end
```

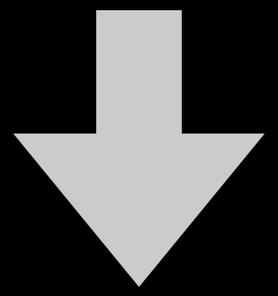
if we could write **`#prints_hello_world?`**



then we could use it to construct a correct implementation of **`#halts?`**



but it's impossible to correctly implement **`#halts?`**



so it must be impossible to write
`#prints_hello_world?`

Not only can we not ask
“does this program halt?”,
we also can’t ask
“does this program do
what I want it to do?”.

This is Rice's theorem:

Any interesting property of program behavior is undecidable.

WHY DOES THIS HAPPEN?

We can't look into the future and predict
what a program will do.

The only way to find out for sure is to run it.

But when we run a program, we don't know
how long we have to wait for it to finish.
(Some programs never will.)

Any system with enough power to be self-referential can't correctly answer every question about itself.

We need to step outside the self-referential system and use a different, more powerful system to answer questions about it.

But there **is** no more powerful system to upgrade to.

HOW CAN WE COPE?

- Ask undecidable questions, but give up if an answer can't be found in a reasonable time.
- Ask several small questions whose answers provide evidence for the answer to a larger question.
- Ask decidable questions by being conservative.
- Approximate a program by converting it into something simpler, then ask questions about the approximation.

From Simple Machines to Impossible Programs

Understanding Computation

O'REILLY®

Tom Stuart

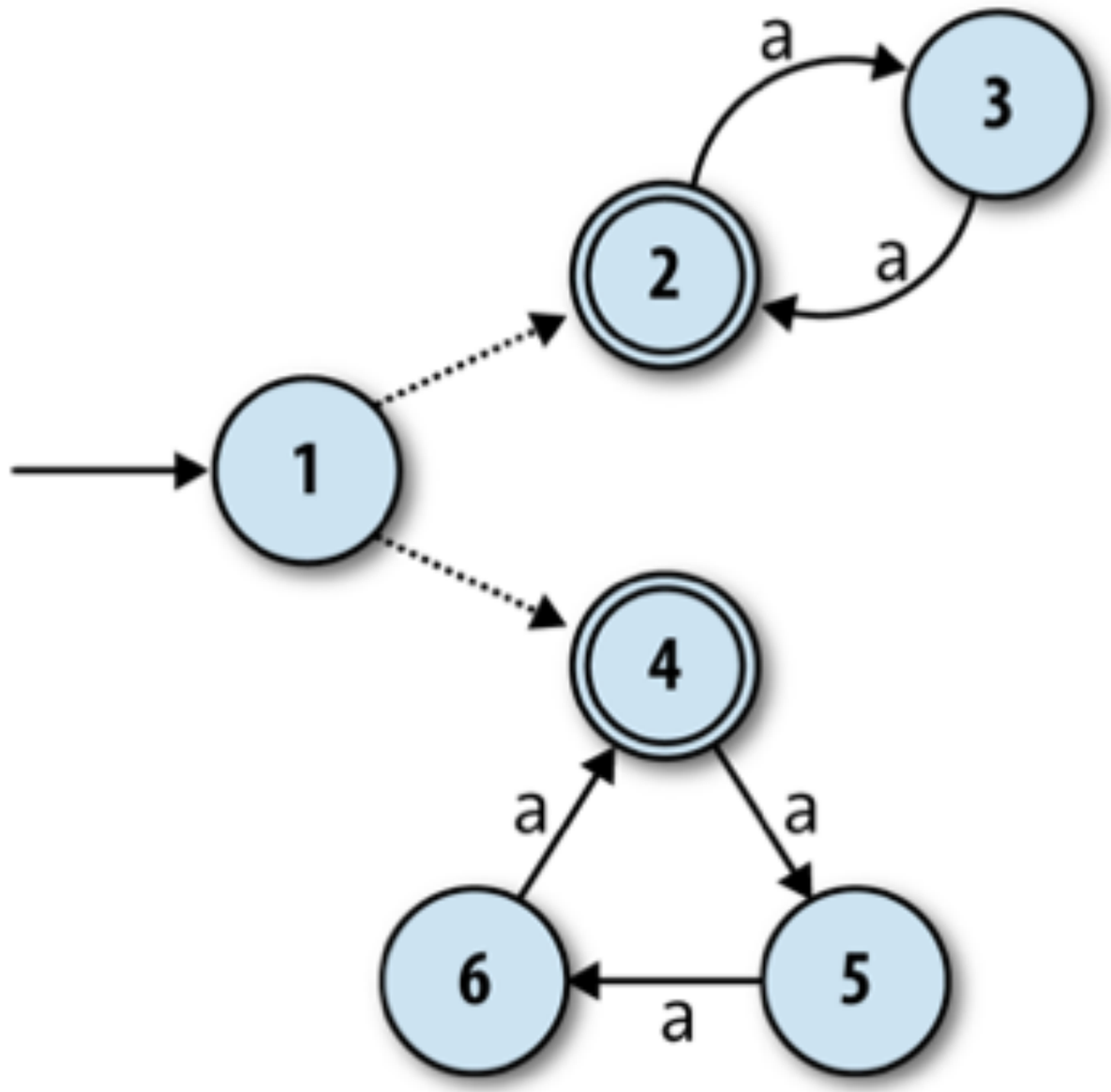
computationbook.com

```
    end
  end
```

This gives the virtual machine the opportunity to evaluate the condition and body as many times as necessary:

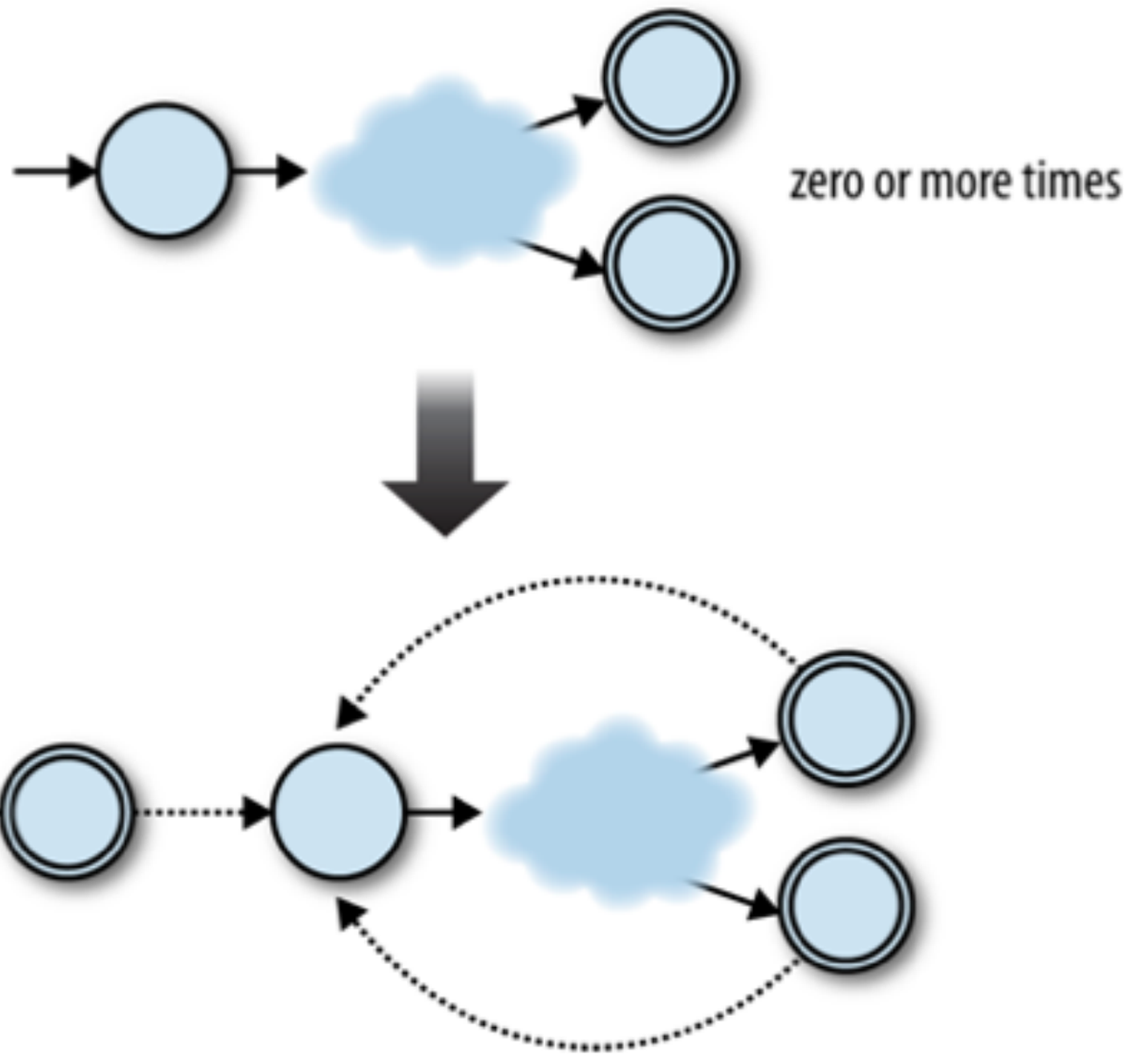
```
>> Machine.new(
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
     ),
     { x: Number.new(1) }
   ).run
while (x < 5) { x = x * 3 }, {:x=>«1»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
if (1 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 1 * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«3»}
while (x < 5) { x = x * 3 }, {:x=>«3»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
if (3 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 3 * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 9; while (x < 5) { x = x * 3 }, {:x=>«3»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«9»}
while (x < 5) { x = x * 3 }, {:x=>«9»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«9»}
```

can address the problem by introducing another machine feature called *free ves*. These are rules that the machine may spontaneously follow without reading input, and they help here because they give the NFA an initial choice between two arate groups of states:



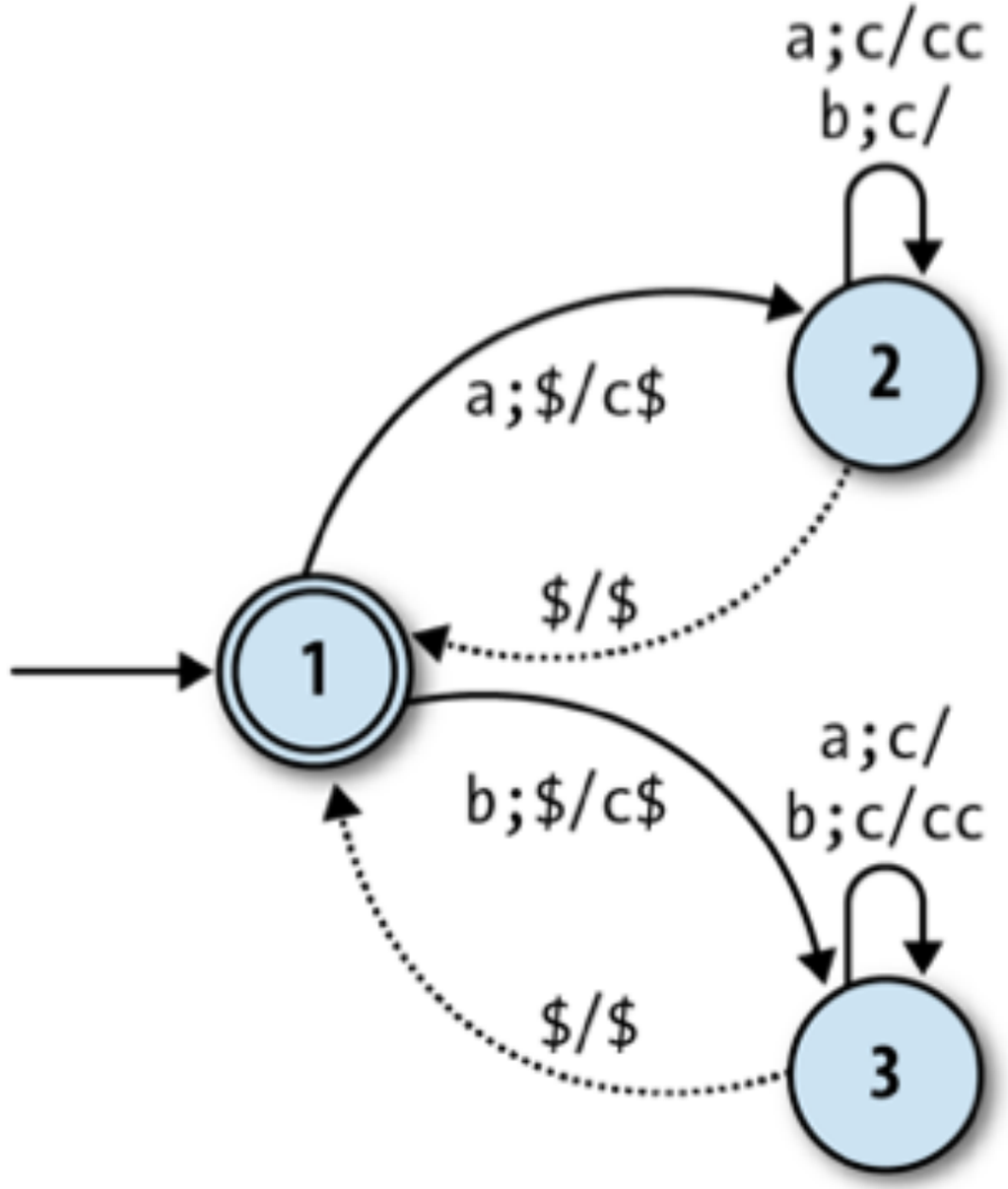
e free moves are shown by the dotted unlabeled arrows from state 1 to states 2 and This machine can still accept the string 'aaaa' by spontaneously moving into state and then moving between states 2 and 3 as it reads the input, and likewise for

match several times instead of just once (`'aa'`, `'aaa'`, etc.), and the new start state allows it to match the empty string without affecting what other strings it can accept.[5] We can do the same for any NFA as long as we connect each old accept state to the old start state with a free move:



This time we need:

- A new start state, which is also an accept state
- All the accept states from the old NFA



To really exploit the potential of the stack, we need a tougher problem that'll force us to store structured information. The classic example is recognizing palindromes: as we read the input string, character by character, we have to remember what we see; once we pass the halfway point, we check our memory to decide whether the characters we saw earlier are now appearing in reverse order. Here's a DPDA that can recognize pal-

2. Use the PDA's stack to store characters that represent grammar symbols (`S`, `W`, `A`, `E`, ...) and tokens (`w`, `v`, `=`, `*`, ...). When the PDA starts, have it immediately push a symbol onto the stack to represent the structure it's trying to recognize. We want to recognize SIMPLE statements, so our PDA will begin by pushing `S` onto the stack:

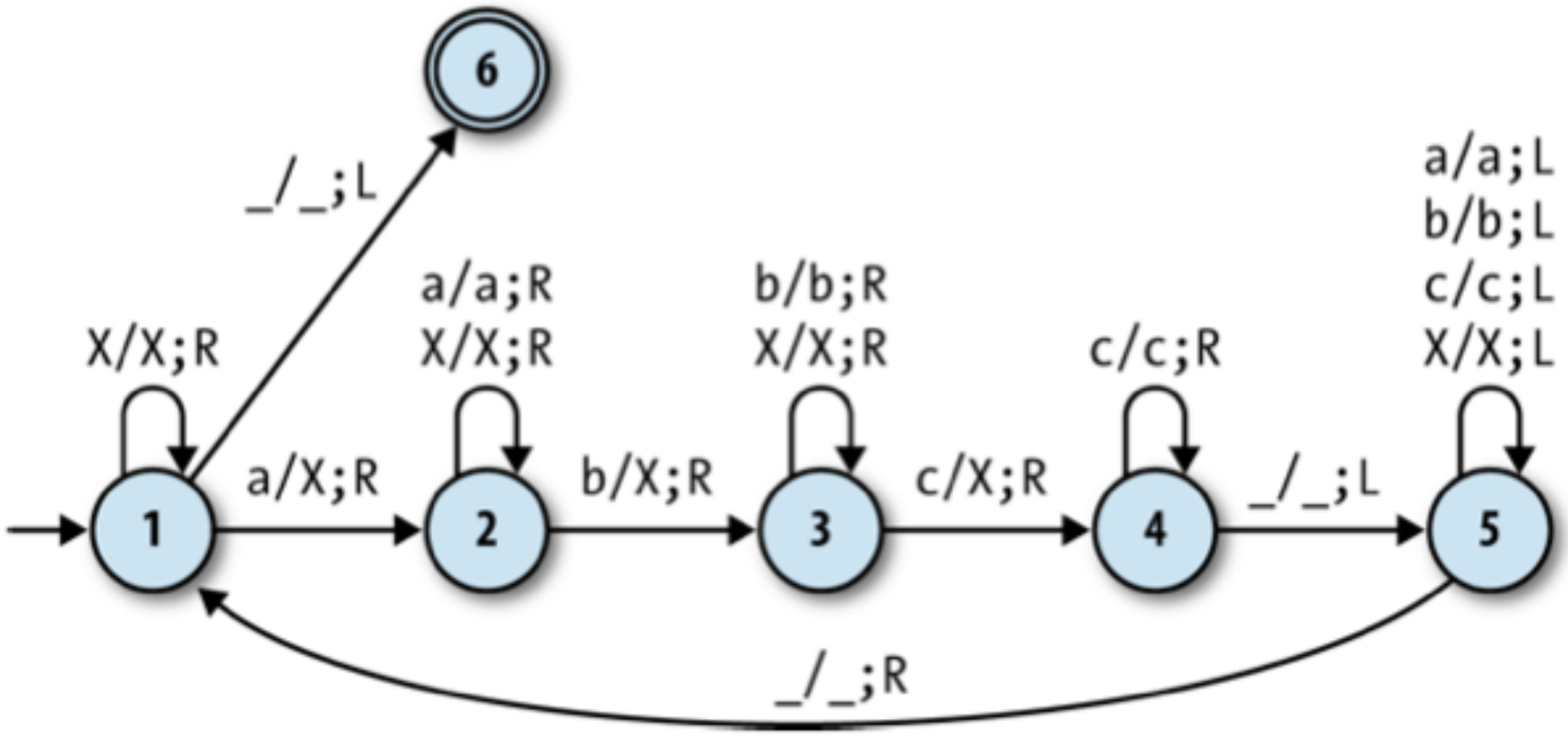
```
>> start_rule = PDARule.new(1, nil, 2, '$', ['S', '$'])
=> #<struct PDARule …>
```

3. Translate the grammar rules into PDA rules that expand symbols on the top of the stack without reading any input. Each grammar rule describes how to expand a single symbol into a sequence of other symbols and tokens, and we can turn that description into a PDA rule that pops a particular symbol's character off the stack and pushes other characters on:

```
>> symbol_rules = [
     # <statement> ::= <while> | <assign>
     PDARule.new(2, nil, 2, 'S', ['W']),
     PDARule.new(2, nil, 2, 'S', ['A']),

     # <while> ::= 'w' '(' <expression> ')' '{' <statement> '}'
     PDARule.new(2, nil, 2, 'W', ['w', '(', 'E', ')', '{', 'S', '}']),

     # <assign> ::= 'v' '=' <expression>
     PDARule.new(2, nil, 2, 'A', ['v', '=', 'E']),

     # <expression> ::= <less-than>
     PDARule.new(2, nil, 2, 'E', ['L']),
```

b, and then moves the tape head one square to the left; a rule labelled a/b;R does almost the same, but moves the head to the right instead of the left.

Let's see how to use a Turing machine to solve a string-recognition problem that push-down automata can't handle: identifying inputs that consist of one or more a characters followed by the same number of bs and cs (e.g., 'aaabbbccc'). The Turing machine that solves this problem has 6 states and 16 rules:



It works roughly like this:

1. Scan across the input string by repeatedly moving the tape head to the right until an a is found, then cross it out by replacing it with an X (state 1).
2. Scan right looking for a b, then cross it out (state 2).

subexpressions is reducible, we reduce that; if not, we actually perform the call by calling the left subexpression (which should be a `LCFunction`) with the right one as its argument. This strategy is known as *call-by-value* evaluation—first we reduce the argument to an irreducible value, then we perform the call.

Let's test our implementation by using the lambda calculus to calculate one plus one:

```
>> expression = LCCall.new(LCCall.new(add, one), one)
=> -> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[-> p { -> x { p[x] } ↵
}][-> p { -> x { p[x] } }]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[-> p { -> x { p[x] } }]↵
[-> p { -> x { p[x] } }]
-> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][-> p { -> x { p[x] } }] }[-> p { -> x ↵
{ p[x] } }]
-> p { -> x { p[x] } }[-> n { -> p { -> x { p[n[p][x]] } } }][-> p { -> x { p[x] } }]
-> x { -> n { -> p { -> x { p[n[p][x]] } } }[x] }[-> p { -> x { p[x] } }]
-> n { -> p { -> x { p[n[p][x]] } } }[-> p { -> x { p[x] } }]
-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }
=> nil
```

Well, something definitely happened, but we didn't get quite the result we wanted: the final expression is `-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }`, but the lambda calculus representation of the number two is supposed to be `-> p { -> x { p[p[x]] } })`. What went wrong?

The mismatch is caused by the evaluation strategy we're using. There are still reducible

```
> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p
{ p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-
> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x
{ x } }][-> p { -> x { x } }]]] }][m] } }[m][n]][n][x] }][m] } } }][n][-> p { -
> x { p[p[p[x]]] } }]]][-> 1 { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x
{ -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][1]] } }[-> 1 { -> x { ->
x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x]
[y] } } }[x][1]] } }[-> 1 { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -
> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][1]] } }[-> 1 { -> x { -> x
{ -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x]
[y] } } }[x][1]] } }[-> x { -> y { -> f { f[x][y] } } }[-> x { -> y { x } }][-
> x { -> y { x } }]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m
{ -> n { n[-> n { -> p { -> x { p[n[p]
[x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p
{ -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n
{ -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -
> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { ->
x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m
{ -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]
```

symbols `S`, `K`, and `I` (called *combinators*), each of which has its own reduction rule:

- Reduce `S[`*a*`][`*b*`][`*c*`]` to *a*`[`*c*`][`*b*`[`*c*`]]`, where *a*, *b*, and *c* can be any SKI calculus expressions.
- Reduce `K[`*a*`][`*b*`]` to *a*.
- Reduce `I[`*a*`]` to *a*.

For example, here's one way of reducing the expression `I[S][K][S][I[K]]`:

```
I[S][K][S][I[K]] → S[K][S][I[K]]  (reduce I[S] to S)
                 → S[K][S][K]     (reduce I[K] to K)
                 → K[K][S[K]]     (reduce S[K][S][K] to K[K][S[K]])
                 → K              (reduce K[K][S[K]] to K)
```

Notice that there's no lambda-calculus-style variable replacement going on here, just symbols being reordered, duplicated, and discarded according to the reduction rules.

It's easy to implement the abstract syntax of SKI expressions:

```ruby
class SKISymbol < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    to_s
  end
end

class SKICall < Struct.new(:left, :right)
```

Does the SKI calculus expression `S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]]` do the same thing as the lambda calculus expression `-> p { -> x { p[p[x]] } }`? Well, it's supposed to call its first argument twice on its second argument, so we can try giving it some arguments to see whether it actually does that, just like we did in "Semantics" on page 199:

```
>> inc, zero = SKISymbol.new(:inc), SKISymbol.new(:zero)
=> [inc, zero]
>> expression = SKICall.new(SKICall.new(two.to_ski, inc), zero)
=> S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]][inc][zero]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]][inc][zero]
S[K[S]][S[K[K]][I]][inc][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
K[S][inc][S[K[K]][I][inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[S[K[K]][I][inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[K][inc][I[inc]]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[I[inc]]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[inc]][S[K[S]][S[K[K]][I]][inc][K[I][inc]]][zero]
S[K[inc]][K[S][inc][S[K[K]][I][inc]][K[I][inc]]][zero]
S[K[inc]][S[S[K[K]][I][inc]][K[I][inc]]][zero]
S[K[inc]][S[K[K][inc][I[inc]]][K[I][inc]]][zero]
S[K[inc]][S[K[I[inc]]][K[I][inc]]][zero]
S[K[inc]][S[K[inc]][K[I][inc]]][zero]
S[K[inc]][S[K[inc]][I]][zero]
K[inc][zero][S[K[inc]][I][zero]]
```

Watch how this tag system behaves when started with the string `aabbbb`, representing the number 2:

```
aabbbb → bbbbaa
       → bbaabbbb
       → aabbbbbbbb  (representing the number 4)
       → bbbbbbbbaa
       → bbbbbbaabbbb
       → bbbbaabbbbbbbb
       → bbaabbbbbbbbbbb
       → aabbbbbbbbbbbbbbbb  (the number 8)
       → bbbbbbbbbbbbbbbbaa
       → bbbbbbbbbbbbbbaabbbb
       ⋮
```

The doubling is clearly happening, but this tag system runs forever—doubling the number represented by its current string, then doubling it again, then again—which isn't really what we had in mind. To design a system that doubles a number just once and then halts, we need to use different characters to encode the result so that it doesn't trigger another round of doubling. We can do this by relaxing our encoding scheme to allow `c` and `d` characters in place of `a` and `b`, and then modifying the rules to append `cc` and `dddd` instead of `aa` and `bbbb` when creating the representation of the doubled number.

With those changes, the computation looks like this:

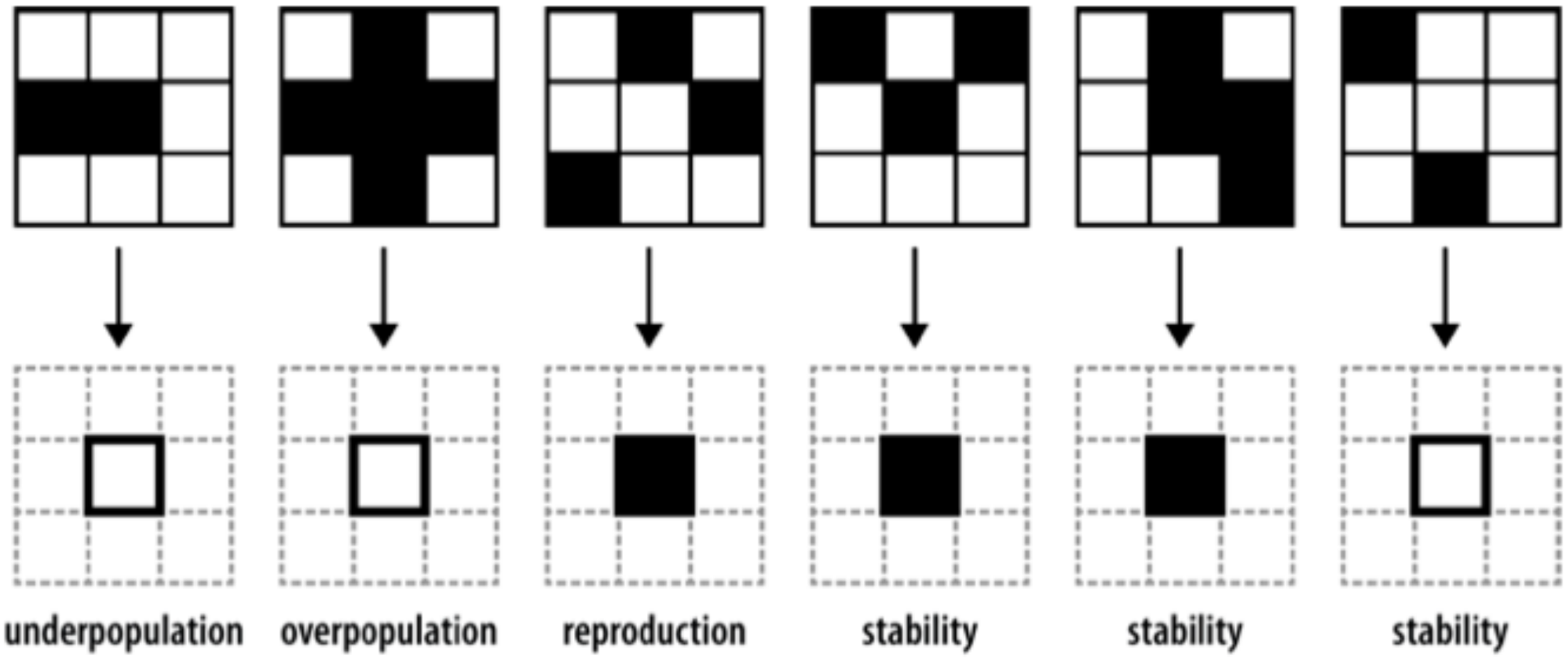
```
aabbbb → bbbbcc
       → bbccdddd
       → ccddddddddd  (the number 4, encoded with c and d instead of a and b)
```

11	append the character 1	yes
11	append the characters 0010	yes
10010	append the characters 10	yes
00101**0**	append the character 1	no
01010	append the characters 0010	no
1010	append the characters 10	yes
0101**0**	append the character 1	no
1010	append the characters 0010	yes
0100**010**	append the characters 10	no
100010	append the character 1	yes
00010**1**	append the characters 0010	no
00101	append the characters 10	no
0101	append the character 1	no
101	append the characters 0010	yes
01**0010**	append the characters 10	no
10010	append the character 1	yes
00101	append the characters 0010	no
⋮	⋮	⋮

Despite the extreme simplicity of this system, we can see a hint of complex behavior: it's not obvious what will happen next. With a bit of thought we can convince ourselves

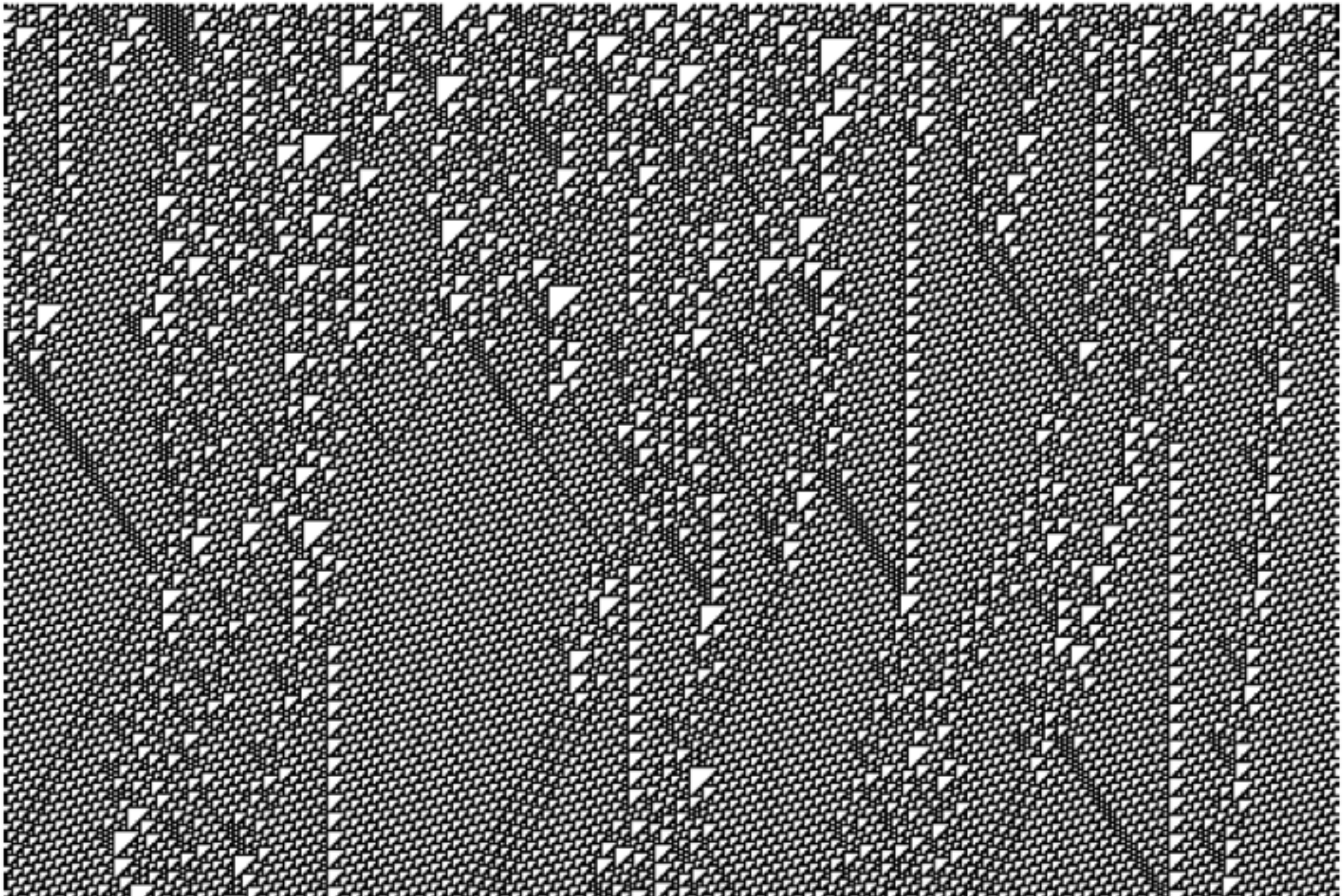
that are triggered by the current state of the cell itself and the states of its neighbors. The rules are simple: a living cell dies if it has fewer than two living neighbors (underpopulation) or more than three (overpopulation), and a dead cell comes to life if it has exactly three living neighbors (reproduction).

Here are six examples[11] of how the Game of Life rules affect a cell's state over the course of a single step, with living cells shown in black and dead ones in white:



A system like this, consisting of an array of cells and a set of rules for updating a cell's state at each step, is called a *cellular automaton*.

Alternatively, running rule 110 from an initial state consisting of a random pattern of living and dead cells reveals all kinds of shapes moving around and interacting with each other:



```ruby
def halts?(program, input)
  if program.include?('while true')
    if program.include?('input.include?(\'goodbye\')')
      if input.include?('goodbye')
        false
      else
        true
      end
    else
      false
    end
  else
    true
  end
end
```

Now we have a checker that gives the correct answers for all three programs and any possible input strings:

```ruby
>> halts?(always, 'hello world')
=> true
>> halts?(never, 'hello world')
=> false
>> sometimes = "input = $stdin.read\nif input.include?('goodbye')\nwhile true\n↵
# do nothing\nend\nelse\nputs input.upcase\nend"
=> "input = $stdin.read\nif input.include?('goodbye')\nwhile true\n# do nothing\n↵
end\nelse\nputs input.upcase\nend"
>> halts?(sometimes, 'hello world')
=> true
>> halts?(sometimes, 'goodbye world')
=> false
```

```
    def +(other_sign)
      if self == other_sign || other_sign == ZERO
        self
      elsif self == ZERO
        other_sign
      else
        UNKNOWN
      end
    end
  end
```

This gives us the behavior we want:

```
>> Sign::POSITIVE + Sign::POSITIVE
=> #<Sign positive>
>> Sign::NEGATIVE + Sign::ZERO
=> #<Sign negative>
>> Sign::NEGATIVE + Sign::POSITIVE
=> #<Sign unknown>
```

In fact, this implementation happens to do the right thing when the sign of one of the *inputs* is unknown:

```
>> Sign::POSITIVE + Sign::UNKNOWN
=> #<Sign unknown>
>> Sign::UNKNOWN + Sign::ZERO
=> #<Sign unknown>
>> Sign::POSITIVE + Sign::NEGATIVE + Sign::NEGATIVE
=> #<Sign unknown>
```

We do need to go back and fix up our implementation of `Sign#*`, though, so that it

```
    class LessThan
      def type(context)
        if left.type(context) == Type::NUMBER && right.type(context) == Type::NUMBER
          Type::BOOLEAN
        end
      end
    end
```

This lets us ask for the type of expressions that involve variables, as long as we provide a context that gives them the right types:

```
>> expression = Add.new(Variable.new(:x), Variable.new(:y))
=> «x + y»
>> expression.type({})
=> nil
>> expression.type({ x: Type::NUMBER, y: Type::NUMBER })
=> #<Type number>
>> expression.type({ x: Type::NUMBER, y: Type::BOOLEAN })
=> nil
```

That gives us implementations of `#type` for all forms of expression syntax, so what about statements? Evaluating a SIMPLE statement returns an environment, not a value, so how do we express that in the static semantics?

The easiest way to handle statements is to treat them as a kind of inert expression: assume that they don't return a value (which is true) and ignore the effect they have on the environment. We can come up with a new type that means "doesn't return a value" and associate that type with any statement as long as all its subparts have the right types. Let's give this new type the name `Type::VOID`:

```
class Type
  VOID = new(:void)
```

From Simple Machines to Impossible Programs

Understanding Computation

O'REILLY®

Tom Stuart

computationbook.com

thanks!

@tomstuart / tom@codon.com / computationbook.com

CHICAGO
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2015

goto; conference

Questions?

Please remember to evaluate via the GOTO Guide App

follow us @gotochgo

Conference: May 11-12 / Workshops: 13-14