What is static analysis?

Matt Might

matt.might.net

University of Utah

@mattmight

interpreter

abstract interpreter

static analyzer

What

Why

Too slow.

Too buggy.

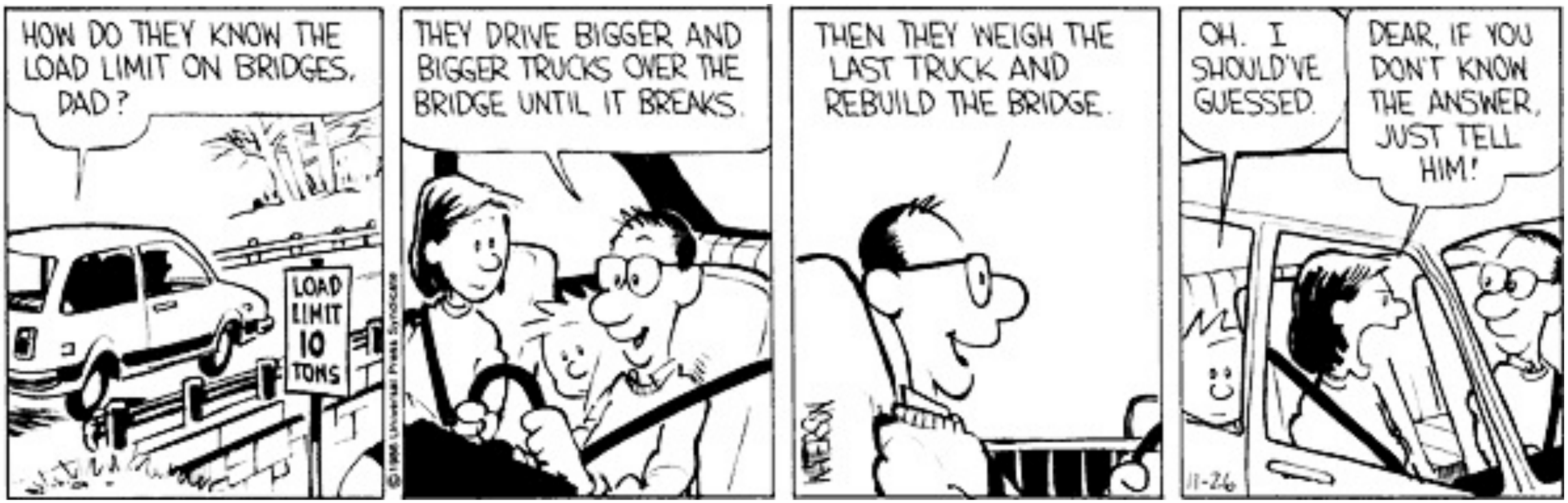
Too insecure.

Why?

We can’t engineer.

Software engineering?

Software engineering?



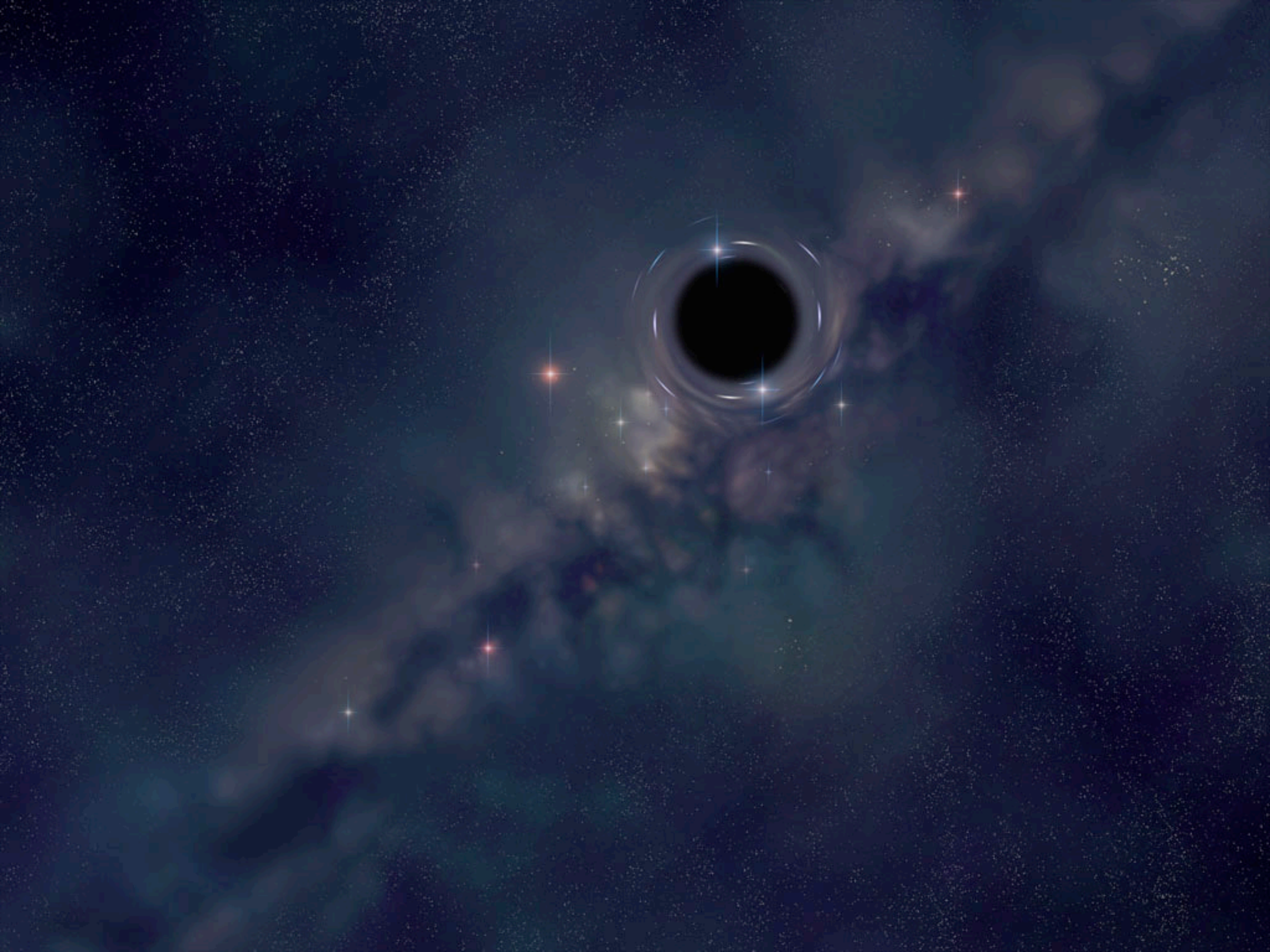
We can’t engineer.

We can’t predict.

Why?

Because Alan Turing said so!

“Thou shalt not write an algorithm which determines whether a program halts.”



“the loop hole”

while P(x)

Interesting question?

Interesting question?

Undecidable.

Why we need software *engineering*

```
class MyActivity {

 public MyActivity() {
  activateMic();
 }

}
```

```
class MyActivity {

 public MyActivity() {
  activateMic();
 }

}
```

There’s a loop hole...

There's a loop hole...

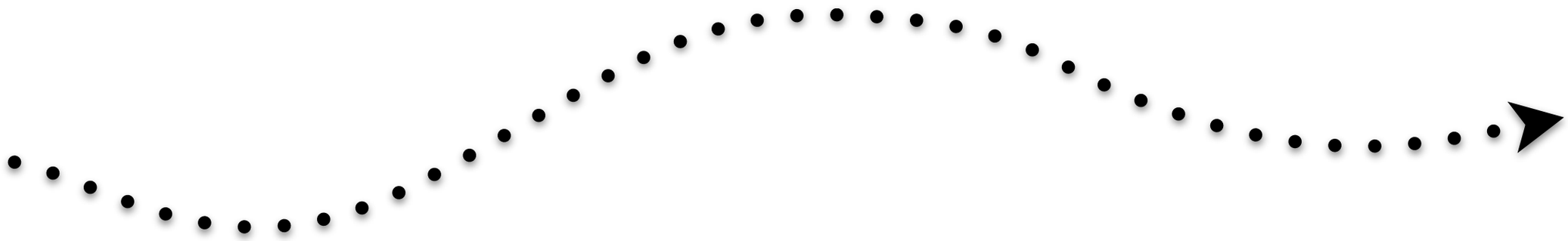
...in the loop hole.

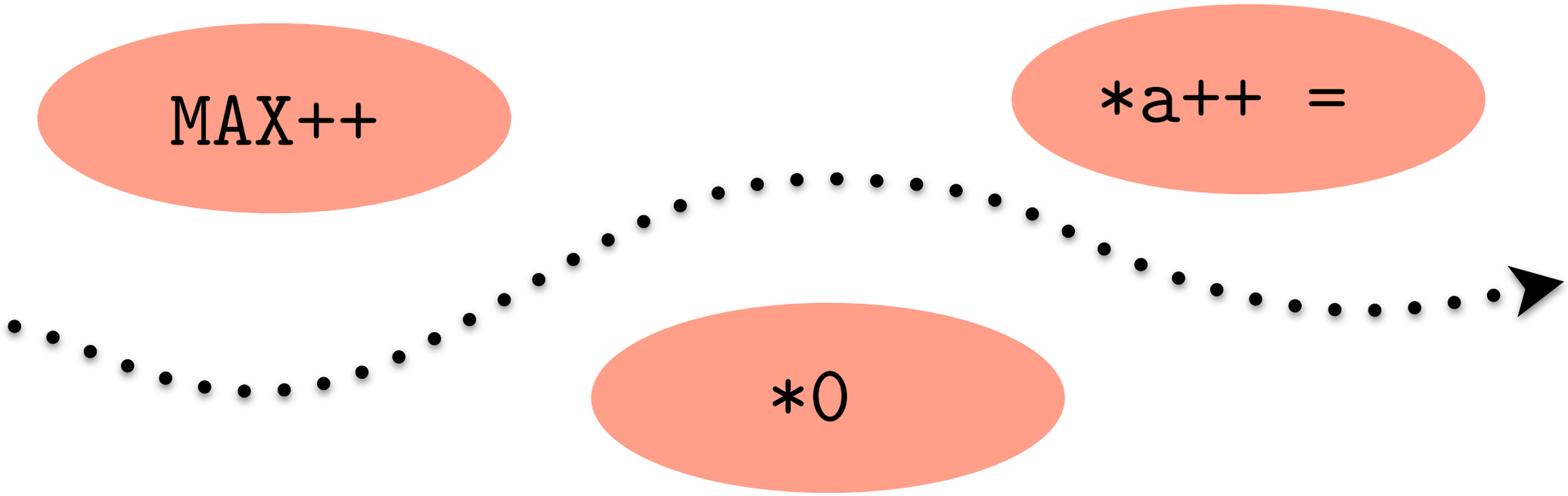

Yes
No

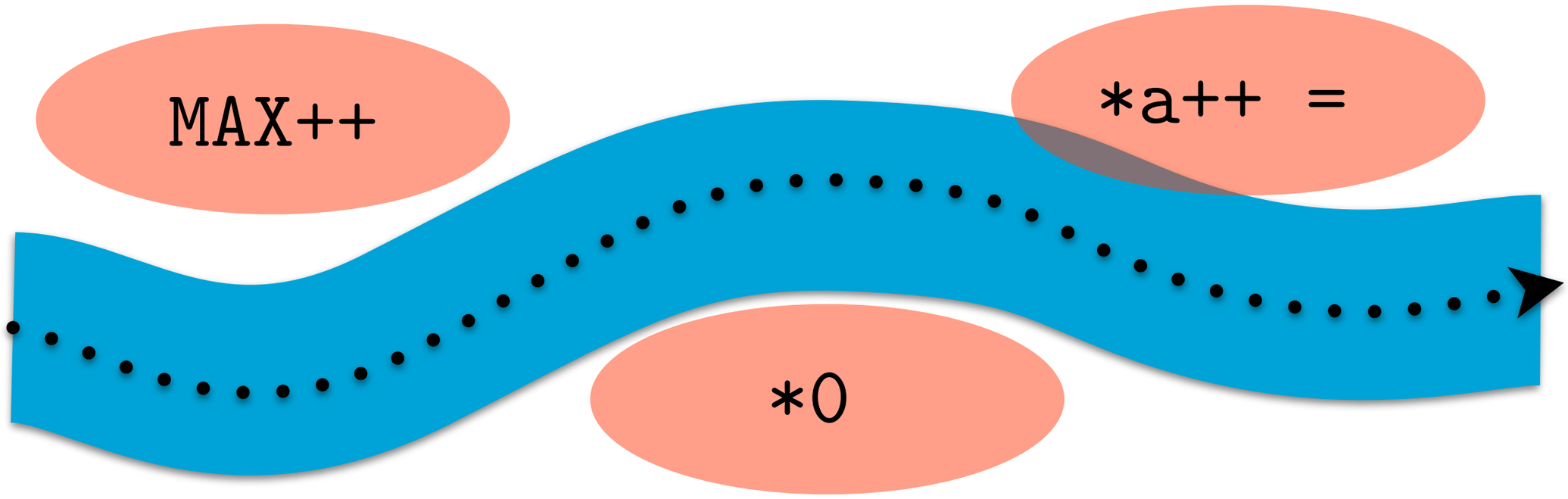
Yes

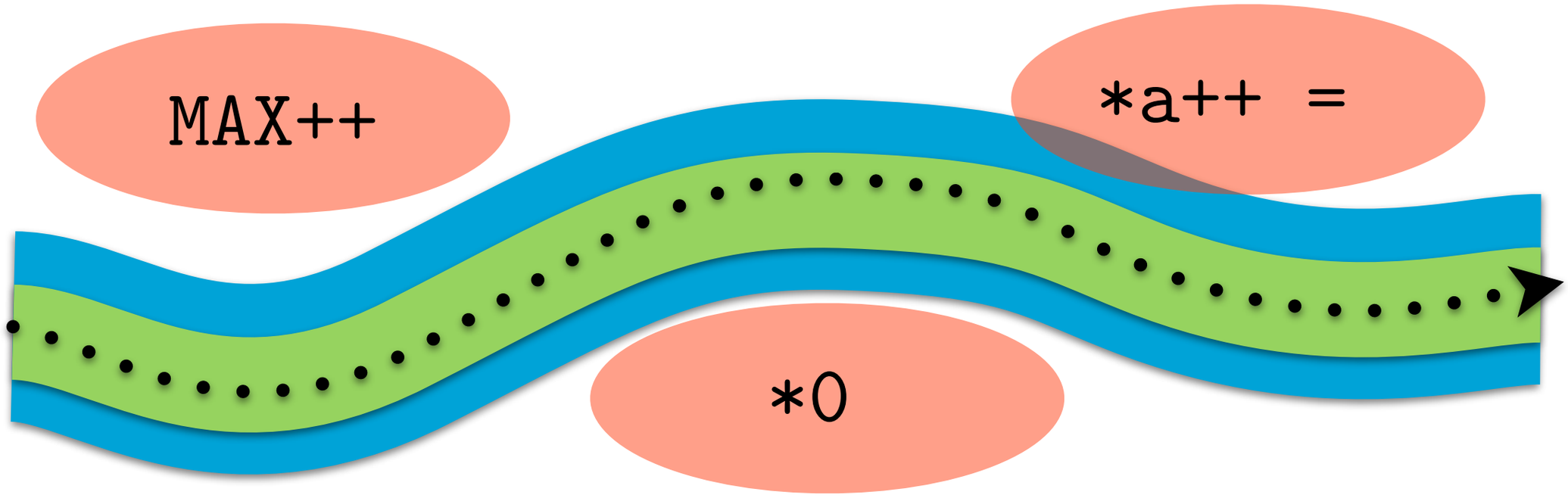
? No

static analysis = reasoning




MAX++
*a++ =
*0


MAX++
*a++ =
*0

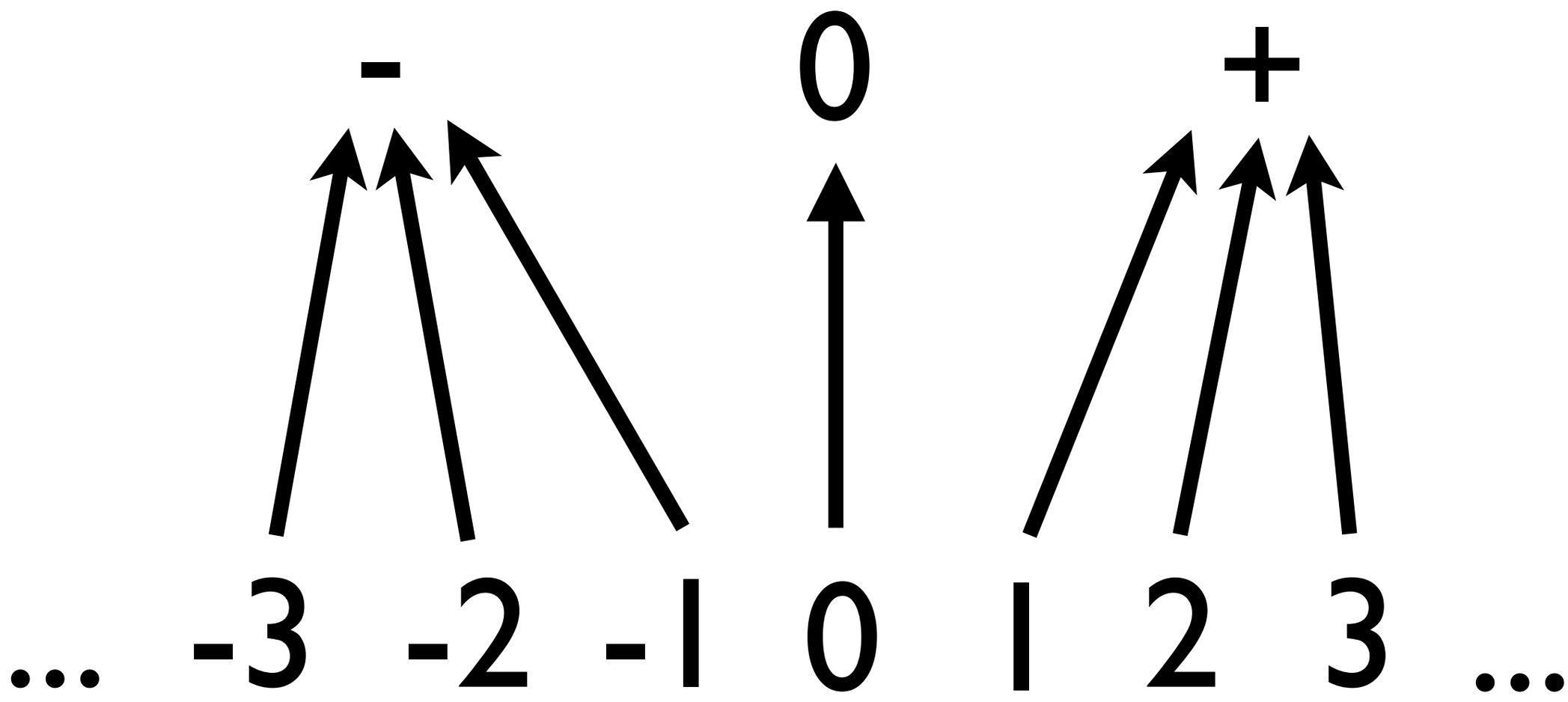

MAX++
*a++ =
*0

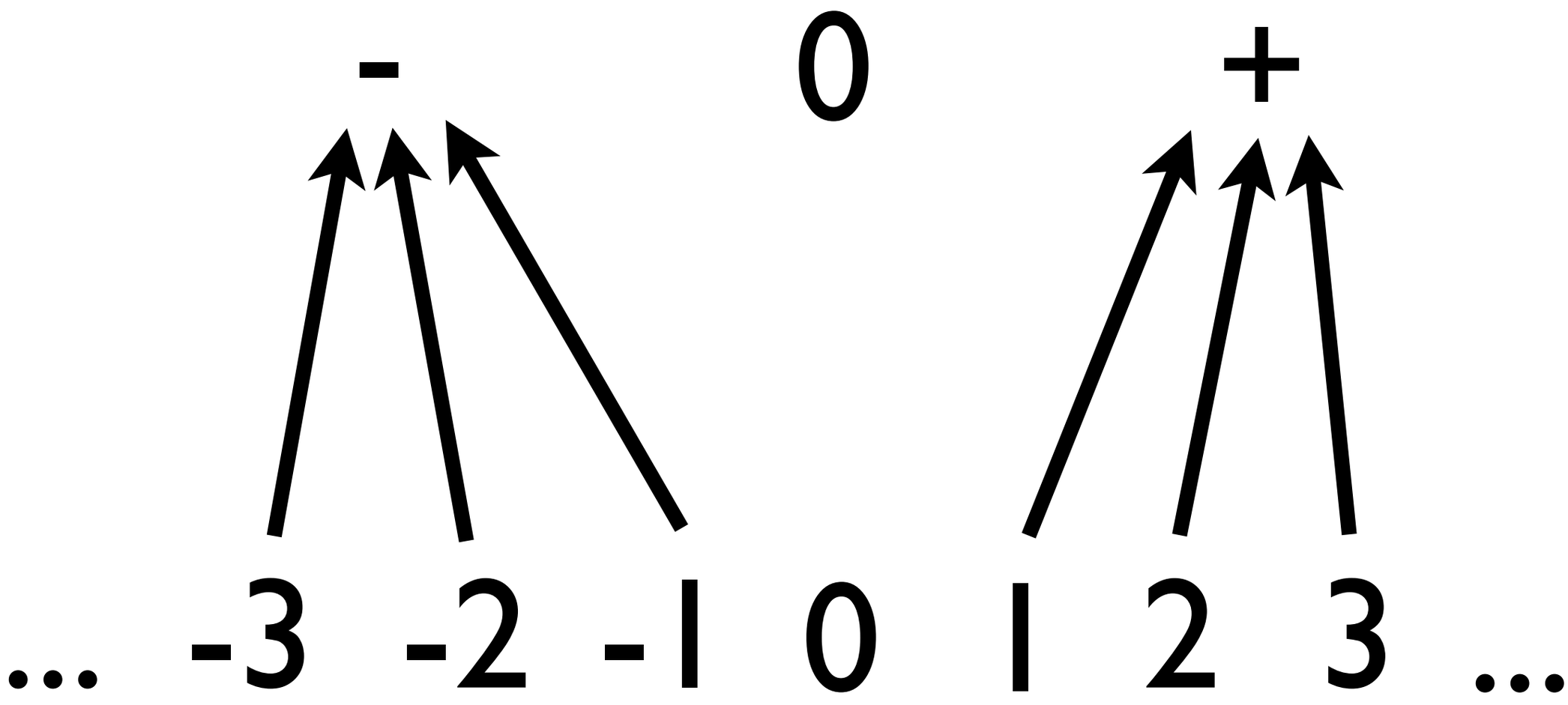
Example: Sign analysis

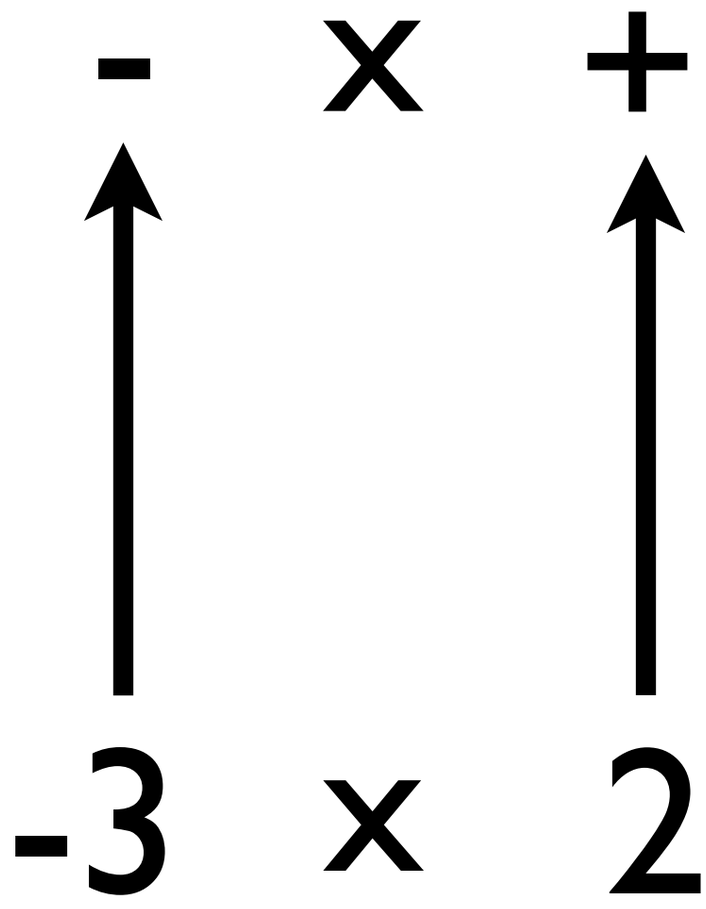
What is the sign of -3 x 2?

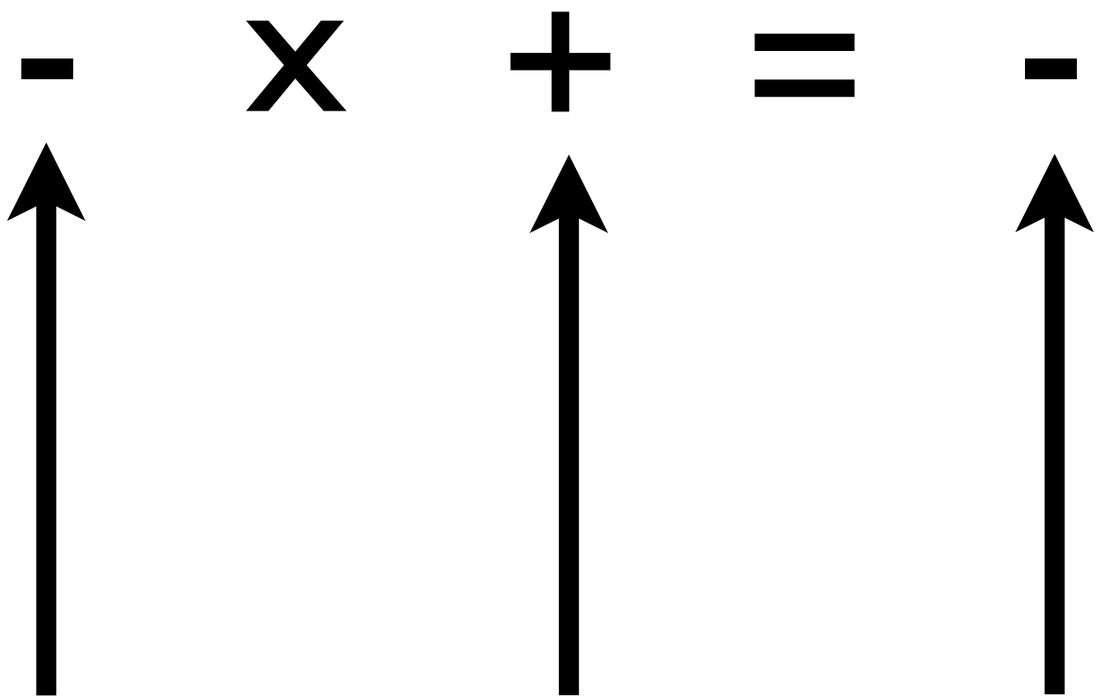
… -3 -2 -1 0 1 2 3 …

… -3 -2 -1 0 1 2 3 …


-
0
+
... -3 -2 -1 0 1 2 3 ...


-
0
+
... -3 -2 -1 0 1 2 3 ...


-
x
+
-3 x 2



-3 x 2 = -6

Let’s build it!

```
<exp> ::= <int>
        |   <exp> * <exp>
        |   <exp> = <exp>
        |   <exp> + <exp>
```

```
simple-eval : exp -> integer
```

```
simple-eval^ : exp -> abstract-integer
```

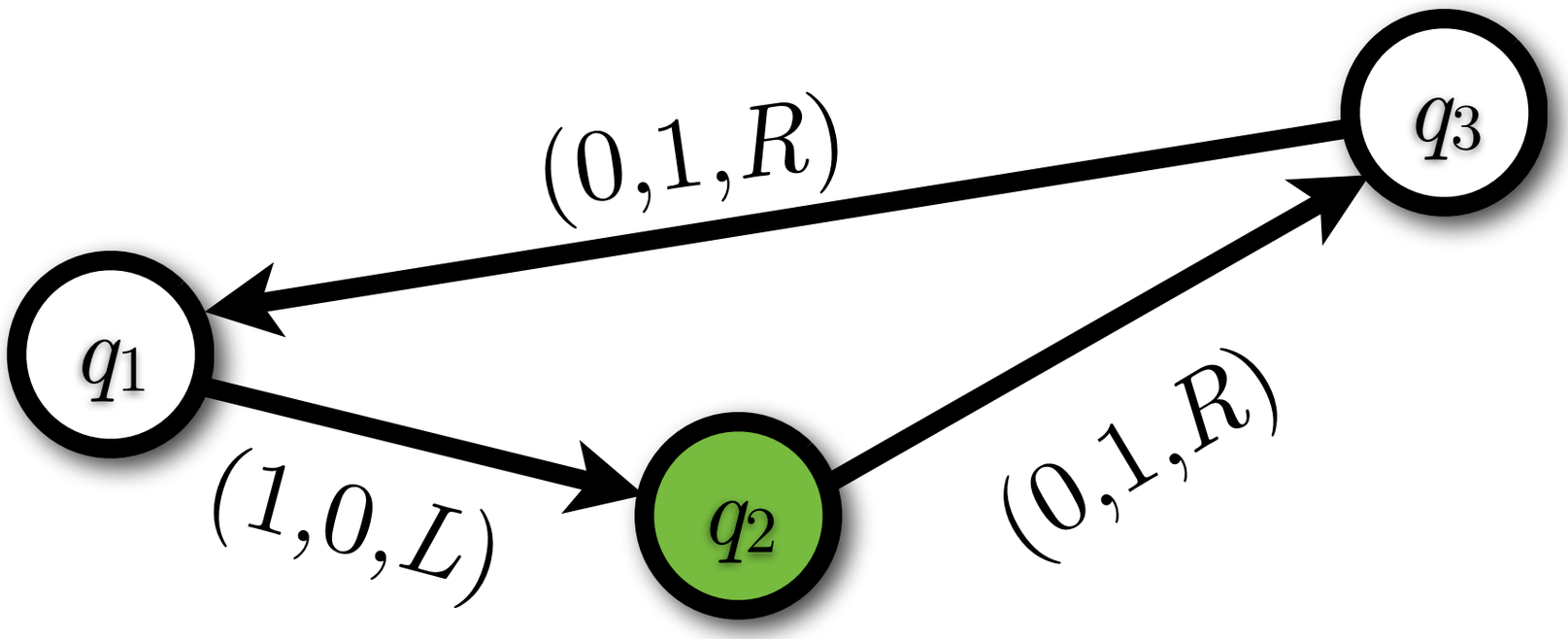
```
simple-eval : exp -> integer
```

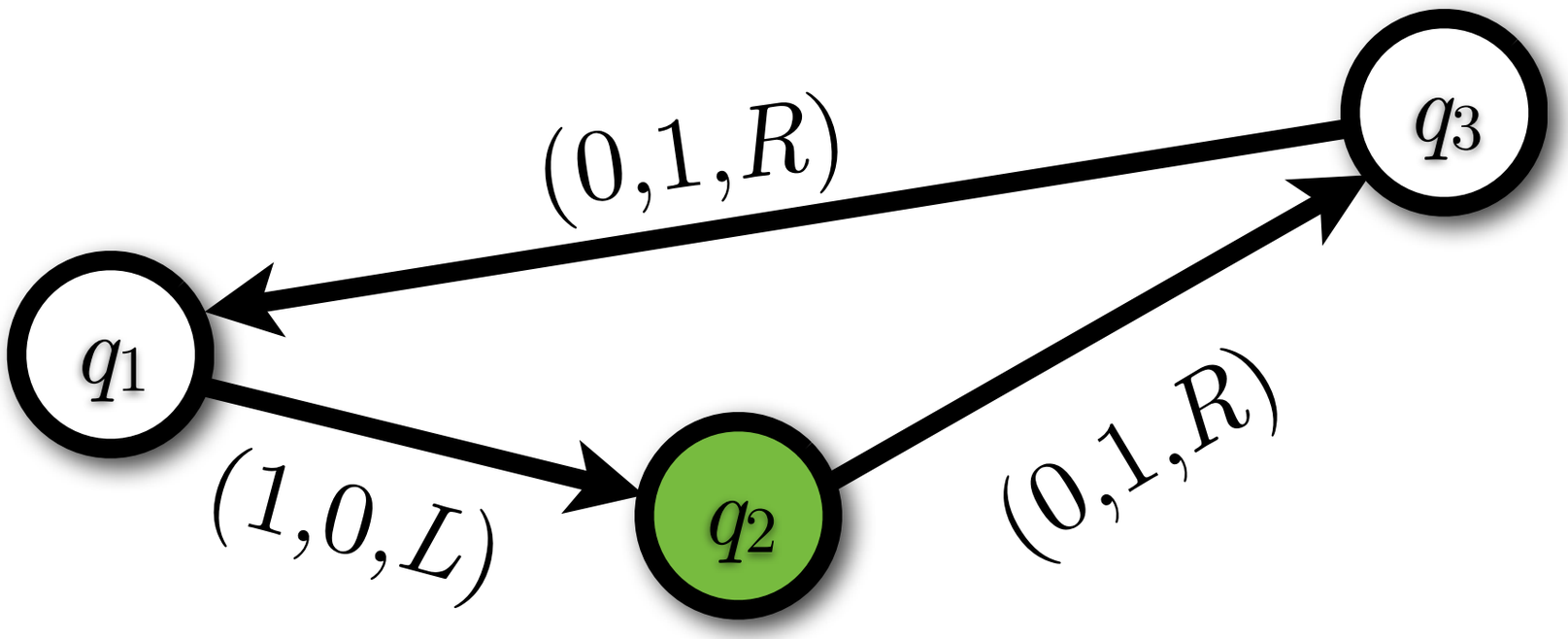
```
α : integer -> abstract-integer
```

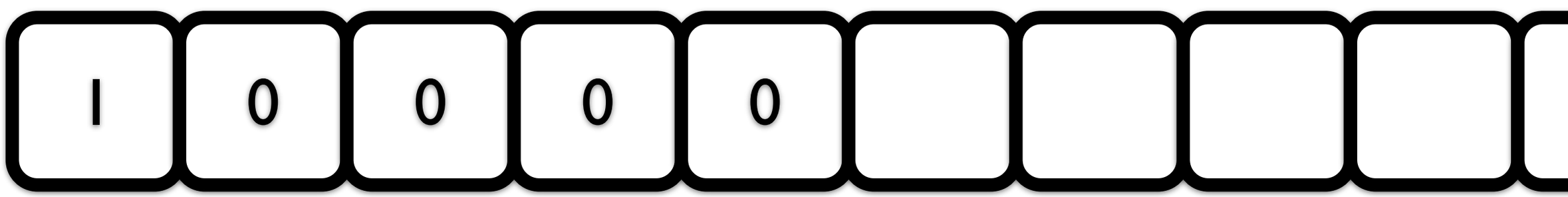
```
simple-eval^ : exp -> abstract-integer
```

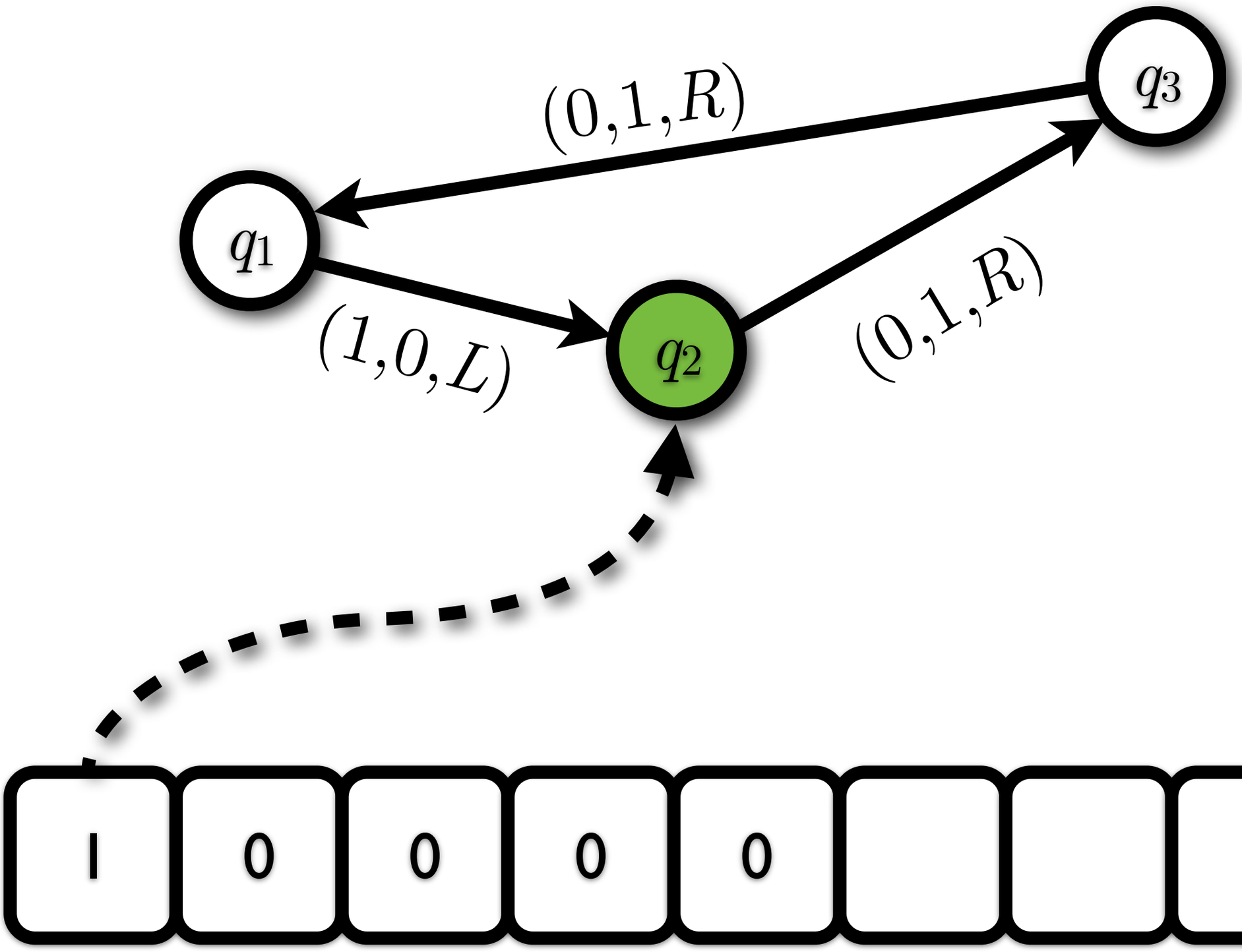
Example: Turing machines


q_1
q_2
q_3


q_1
q_2
q_3
$(0,1,R)$
$(1,0,L)$
$(0,1,R)$

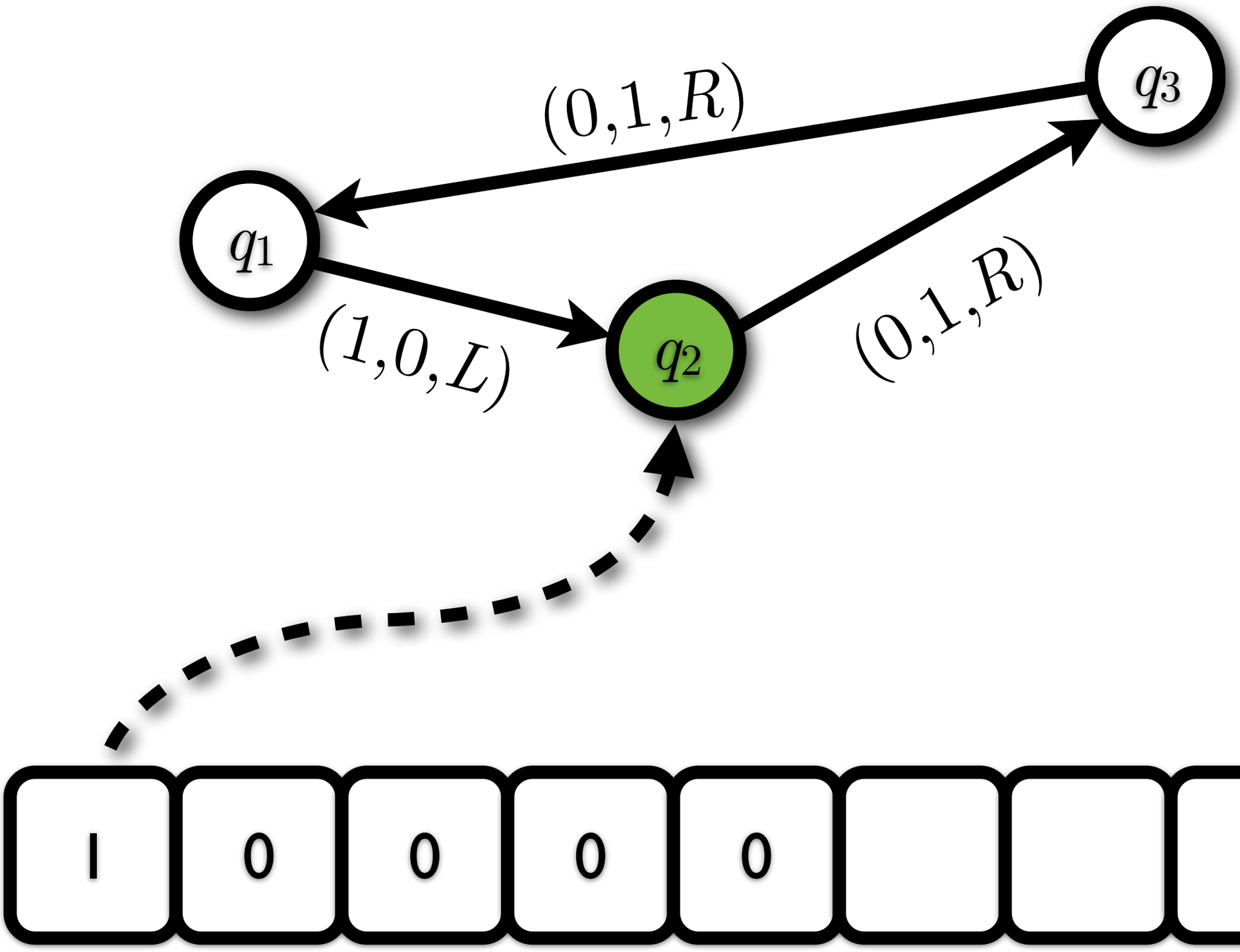

q_1
q_2
q_3
$(0,1,R)$
$(1,0,L)$
$(0,1,R)$

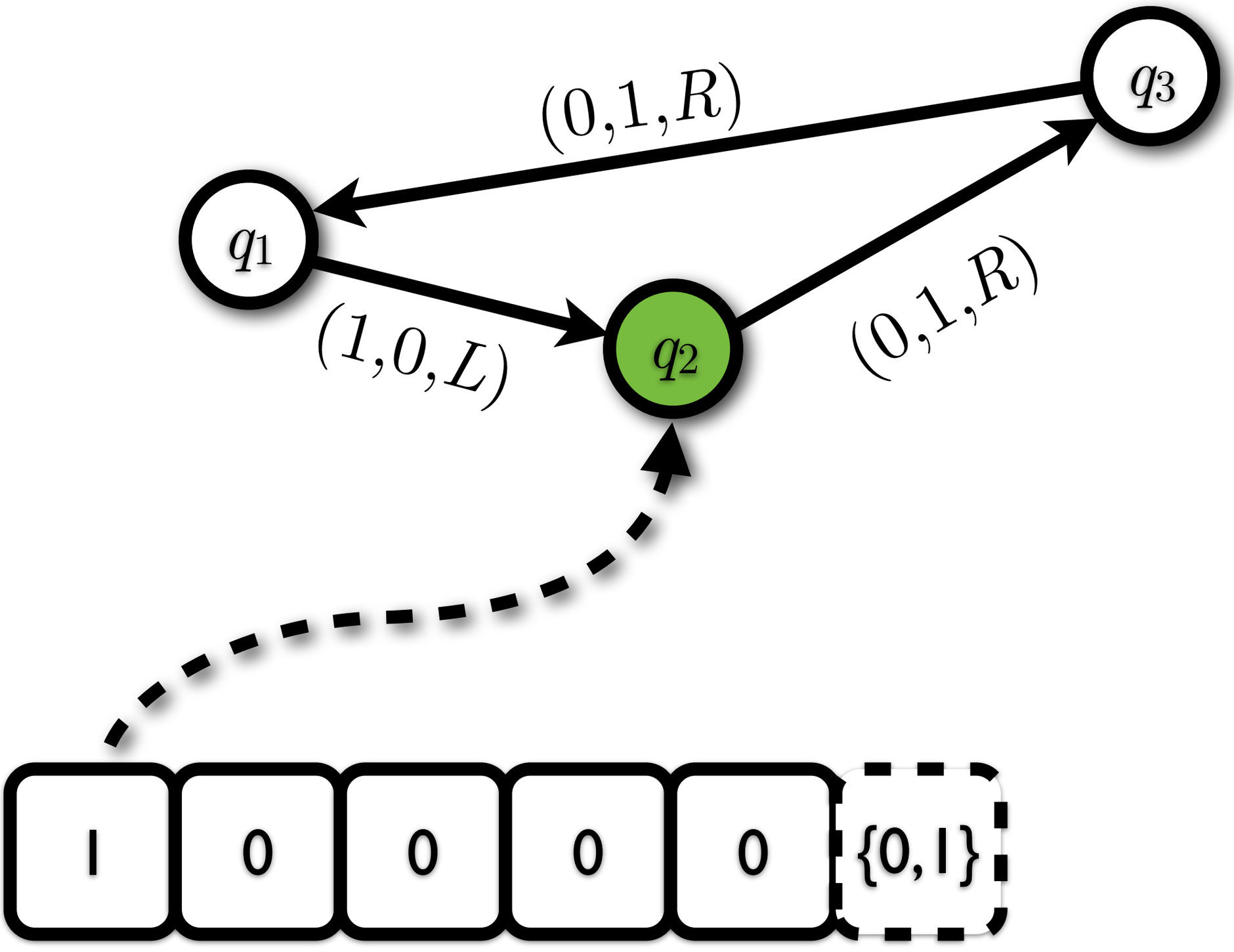

1
0
0
0
0

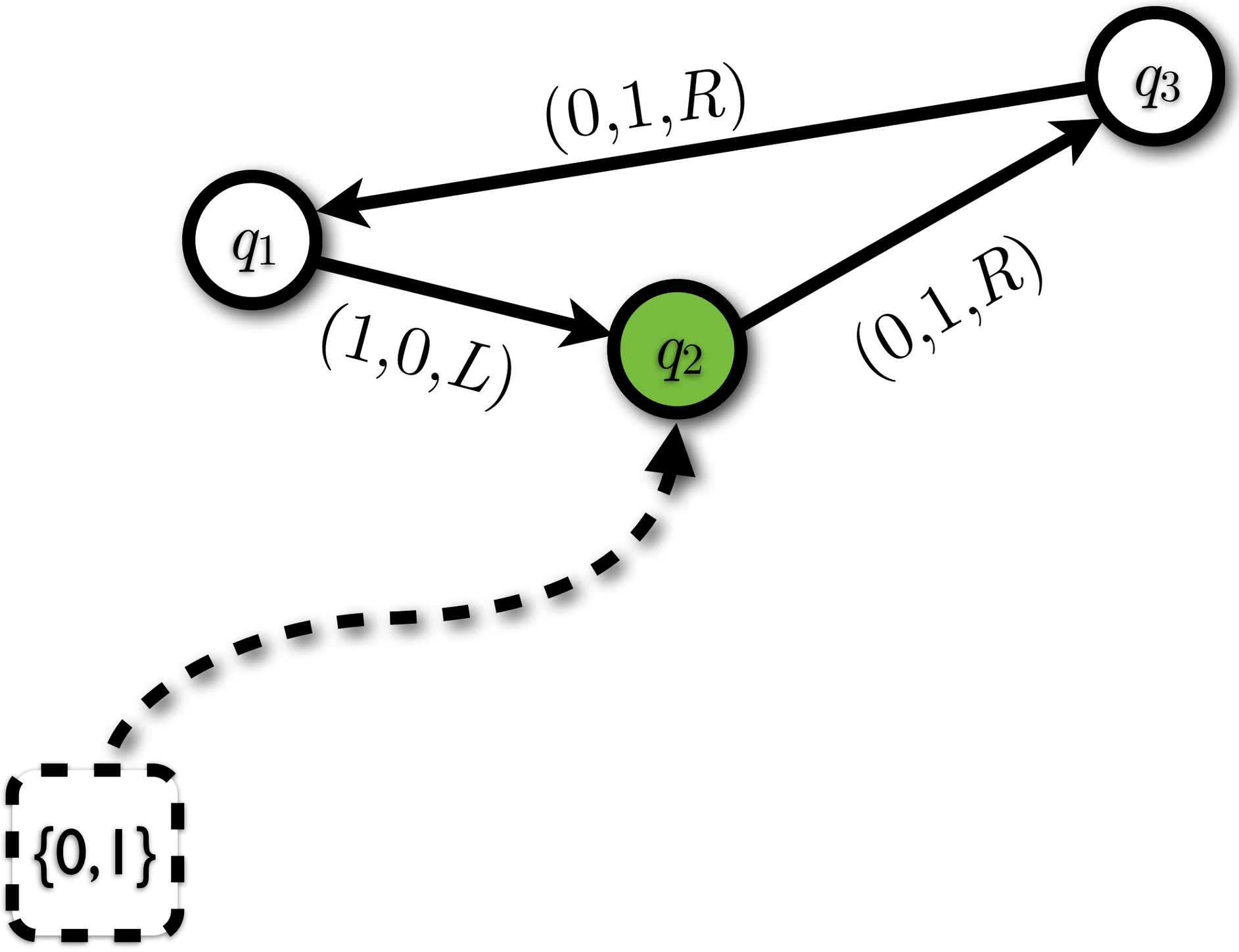

q_1
q_2
q_3
$(0,1,R)$
$(1,0,L)$
$(0,1,R)$
1
0
0
0
0

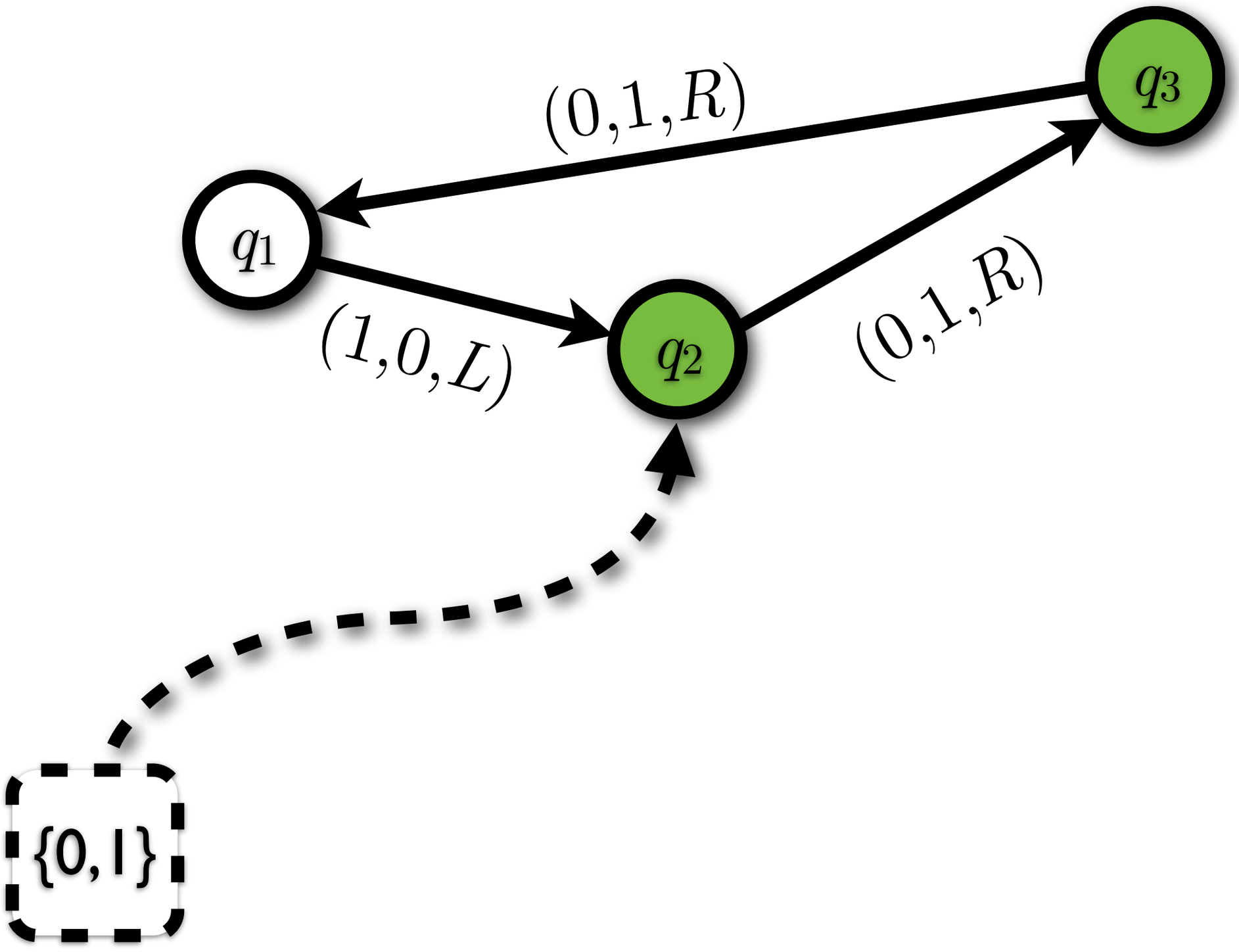
How to approximate?

Make it finite!


q_1
q_2
q_3
$(0,1,R)$
$(1,0,L)$
$(0,1,R)$
1
0
0
0
0


q_1
q_2
q_3
$(0,1,R)$
$(1,0,L)$
$(0,1,R)$
1
0
0
0
0
{0,1}


q_1
q_2
q_3
$(0,1,R)$
$(1,0,L)$
$(0,1,R)$
$\{0,1\}$


q_1
q_2
q_3
$(0,1,R)$
$(1,0,L)$
$(0,1,R)$
{0,1}

Let’s do it for RTL.

What is static program analysis?

The halting problem asks whether the execution of a specific program for a given input will terminate.

The halting problem is famous for being undecidable.

That is, no algorithm can solve it for all programs and all inputs.

This complicates any attempt to predict program behavior: we can make predicting almost any program behavior equivalent to predicting the termination of a nearly identical program.

Static analyses are algorithms that do their best to defy the undecidability of the halting problem: they attempt to predict program behavior.

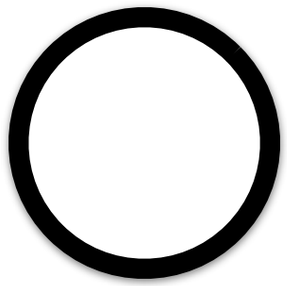
Predicting program behavior enables program optimization, security audits, automatic parallelization and, if accurate enough, correctness verification.

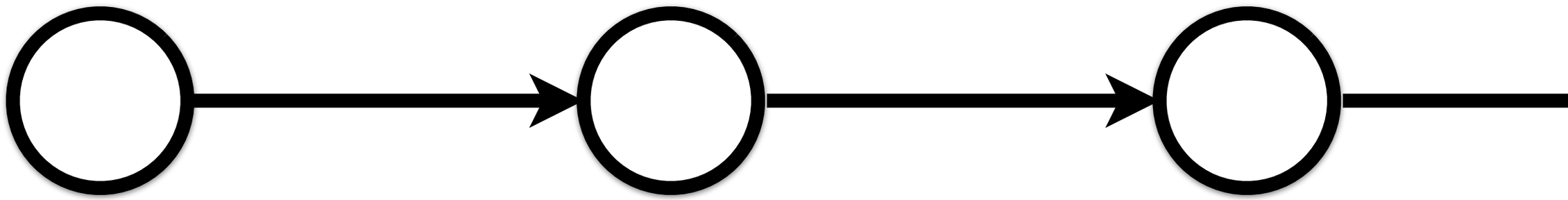
```
<prog> ::= <stmt> ...

<stmt> ::=  <label> :
         |  goto <label> ;
         |  <var> := <exp> ;
         |  if <exp> goto <label> ;
```

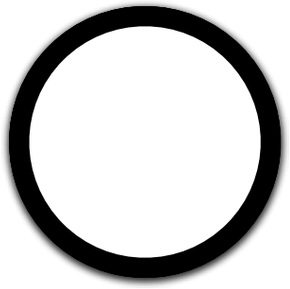
```
<exp> ::= <exp> + <exp>
        |  <exp> * <exp>
        |  <exp> = <exp>
        |  <int>
        |  <var>
```

```
<prog> ::= <stmt> ...

<stmt> ::= (label <label>)
         |  (goto <label>)
         |  (:= <var> <exp>)
         |  (if <exp> goto <label>)
```

```
<exp> ::= (+ <exp> <exp>)
        |  (* <exp> <exp>)
        |  (= <exp> <exp>)
        |  <int>
        |  <var>
```







```
(struct state {stmts env})
```

```
stmts = stmt*
```

```
env = var #=> integer
```

```
inject : prog -> state
```

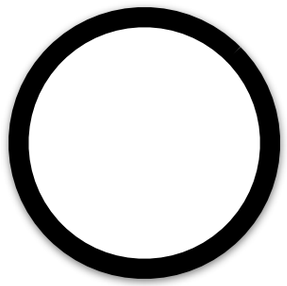
```
step : state -> state
```

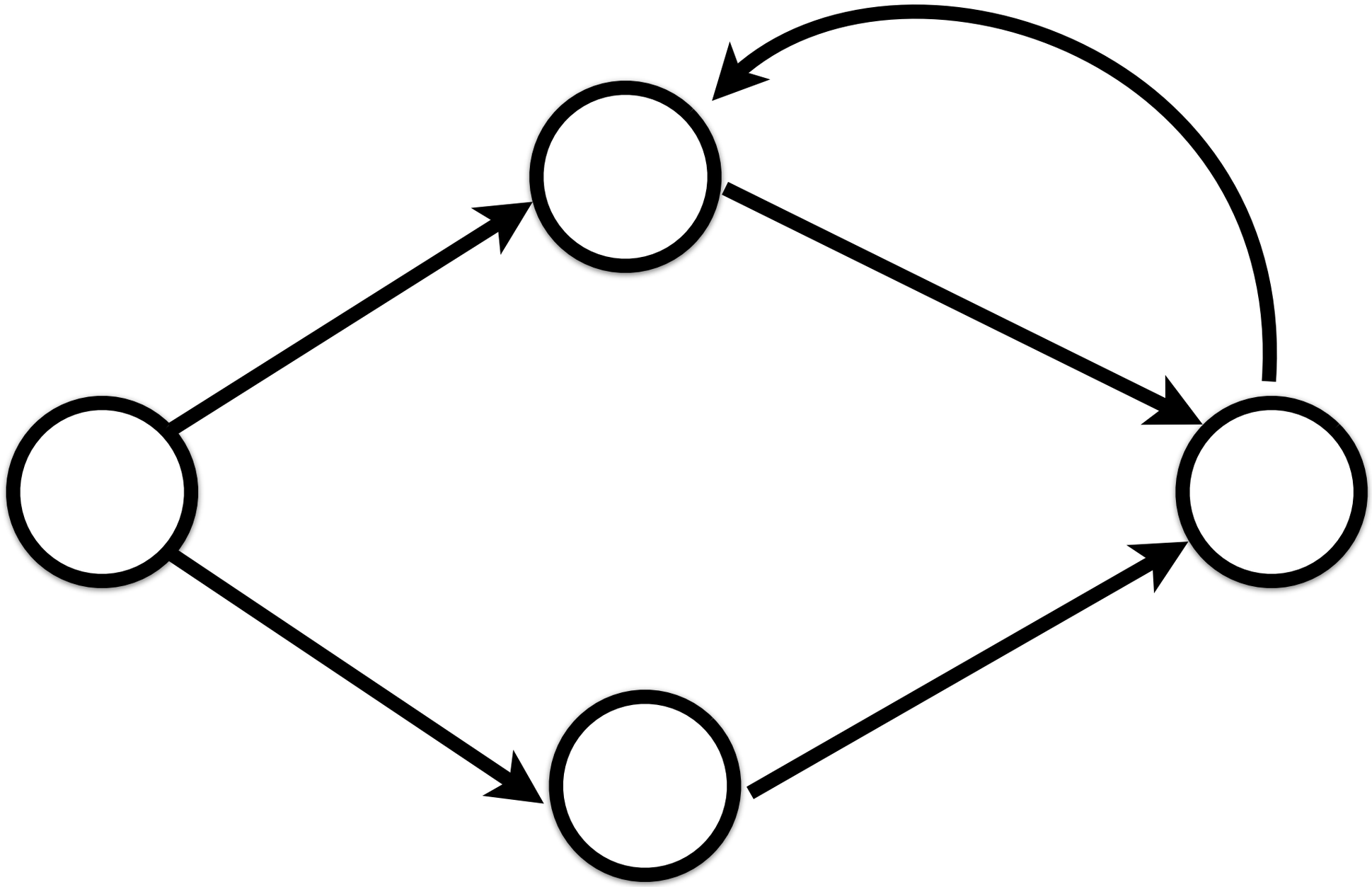
```
; stmt-map : label => stmt*
(define stmt-map (make-hasheq))
```

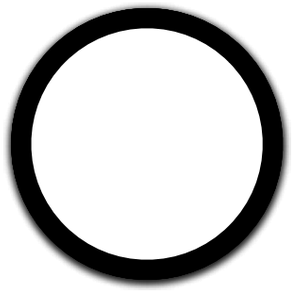
```
preprocess : prog -> void
```

```
(define (preprocess stmts)
  (match stmts
    [(cons `(label ,label) rest)
     (hash-set! stmt-map label stmts)
     (preprocess rest)]

    [(cons _ rest)
     (preprocess rest)]

    ['()
     (void)]))
```

Make it finite!







```
(struct state^ { stmts^ env^})
```

```
stmts^ = stmt*
```

```
env^ = var #=> abstract-integer
```

```
inject^ : prog -> state^
```

```
step^ : state^ -> state^
```

Questions?

matt.might.net

@mattmight