# Probabilistic Programming

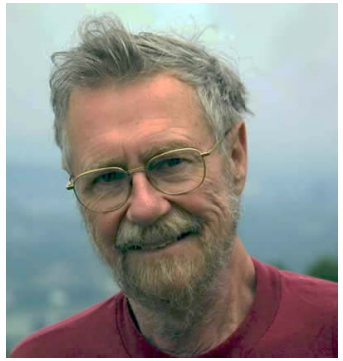
## Lecture #4: Probabilistic GCL

Joost-Pieter Katoen


RWTH Lecture Series on Probabilistic Programming 2018

# Overview

# Dijkstra's guarded command language: Syntax



- `skip` — empty statement
- `diverge` — divergence
- `x := E` — assignment
- `prog1 ; prog2` — sequential composition
- `if (G) prog1 else prog2` — choice
- `prog1 [] prog2` — non-deterministic choice
- `while (G) prog` — iteration

# Elementary pGCL ingredients

- Program variables $x \in \mathit{Vars}$ whose values are fractional numbers
- Arithmetic expressions $E$ over the program variables $2x+y$
- Boolean expressions $G$ (guarding a choice or loop) over the program variables $x < y$
- A distribution expression $\mu : \Sigma \to \mathit{Dist}(\mathbb{Q})$ $\mu(\sigma) = \mathit{Dist}(\mathbb{Q})$
- A probability expression $p : \Sigma \to [0,1] \cap \mathbb{Q}$ $\frac{1}{x}$

# Probabilistic GCL: Syntax

Kozen

McIver

Morgan

- `skip` — empty statement
- `diverge` — divergence
- `x := E` — assignment
- `x :r= mu` — **random assignment** $(x :\approx \mu)$
- `prog1 ; prog2` — sequential composition
- `if (G) prog1 else prog2` — choice
- `prog1 [p] prog2` — **probabilistic choice**
- `while (G) prog` — iteration

$p = \frac{1}{2}$ $p = \frac{x}{1+x}$ $x = 2$ $x = 4$ $p = \frac{2}{3}$ $p = \frac{4}{5}$

Conditioning in the form of observe-statements omitted for now.

# Let's start simple

```
x := 0 [0.5] x := 1;   // flip a fair coin
y := -1 [0.5] y := 0   // flip a fair coin
```

This program admits four runs and yields the outcome:

$$Pr[x=0, y=0] = Pr[x=0, y=-1] = Pr[x=1, y=0] = Pr[x=1, y=-1] = 1/4$$

# A loopy program

For $0 < p < 1$ an arbitrary probability:

```
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

// flip c biased coin

The loopy program models a geometric distribution with parameter $p$.

$$Pr[i = N] = (1-p)^{N-1} \cdot p \quad \text{for } N > 0$$

# On termination

```
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

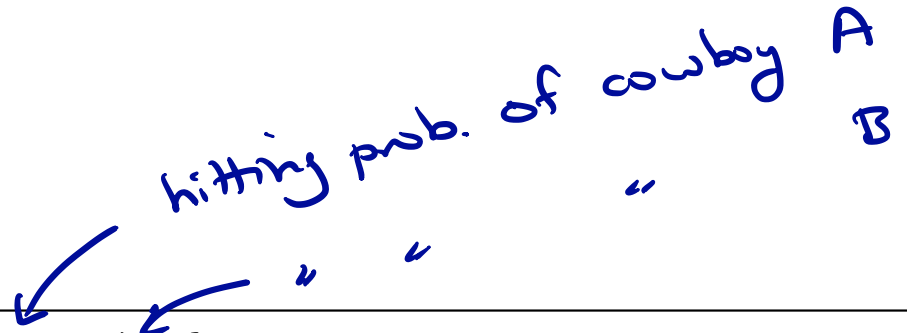
This program does not always terminate. It almost surely terminates.

$Pr\{\text{one non-terminating run}\} = 0$

# The good, the bad, and the ugly

# The good, the bad, and the ugly


# Duelling cowboys




```
int cowboyDuel(float a, b) {
  int t := A [0.5] t := B;
  bool c := true;
  while (c) {
    if (t = A) {
      (c := false [a] t := B);
    } else {
      (c := false [b] t := A);
    }
  }
  return t;
}
```

turn

// survivor

# Random assignments

The random assignment $x :\approx \mu$ works as follows:

1. evaluate distribution expression $\mu$ in the current program state $s$
2. sample from the resulting probability distribution $\mu(s)$ yielding value $v$ with probability $\mu(s)(v)$
3. assign the value $v$ to the variable $x$.

For denoting distribution expressions, we use the bra–ket notation.

$$\frac{1}{2} \cdot [a\rangle + \frac{1}{3} \cdot [b\rangle + \frac{1}{6} \cdot [c\rangle$$

denotes the distribution $\mu$ with $\mu(a) = 1/2$, $\mu(b) = 1/3$, and $\mu(c) = 1/6$. The support set of $\mu$ equals $\{ a, b, c \}$

Examples on the black board.

$$x :\approx \frac{1}{2}[x+1\rangle + \frac{1}{2}[x-1\rangle$$

equivalent

$$x := x+1 \quad [\tfrac{1}{2}] \quad x := x-1$$

---

$$x :\approx \underbrace{\frac{1}{|x|+1}[x+1\rangle}_{} + \frac{|x|}{|x|+1}\cdot[x-1\rangle$$

$x = 2$

$S(x) = 2$

$\frac{1}{3} \rightarrow x := x+1$

$\frac{2}{3} \rightarrow x := x-1$

# Overview

# Why formal semantics matters

- Unambiguous meaning to all programs
- Basis for proving correctness
  - of programs
  - of program transformations
  - of program equivalence
  - of static analysis
  - of compilers
  - .....

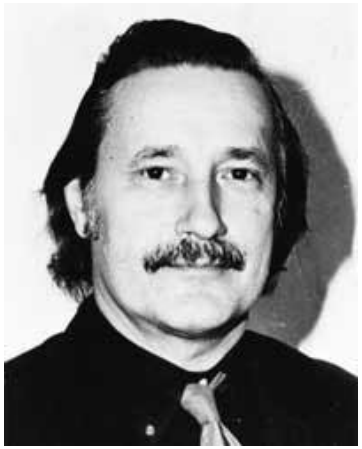
# The inventors of semantics

Tony Hoare 196?

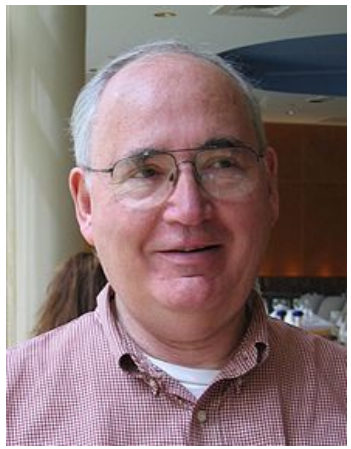
Robert W. Floyd

1981

Gordon Plotkin



Christopher Strachey 1969



Dana Scott

# Approaches to semantics

- Operational semantics: (developed by Plotkin)
  - The meaning of a program in terms of how it executes on an abstract machine.
  - Useful for modelling the execution behaviour of a program.

# Approaches to semantics

- Operational semantics: (developed by Plotkin)
  - The meaning of a program in terms of how it executes on an abstract machine.
  - Useful for modelling the execution behaviour of a program.

- Axiomatic semantics: (developed by Floyd and Hoare)
  - Provides correctness assertions for each program construct.
  - Useful for verifying that a program's computed results are correct with respect to the specification.

$\{pre\}\ P\ \{post\}$

# Approaches to semantics

- Operational semantics: (developed by Plotkin)
  - The meaning of a program in terms of how it executes on an abstract machine.
  - Useful for modelling the execution behaviour of a program.
- Axiomatic semantics: (developed by Floyd and Hoare)
  - Provides correctness assertions for each program construct.
  - Useful for verifying that a program's computed results are correct with respect to the specification.
- Denotational semantics: (developed by Strachey and Scott)
  - Provides a mapping of language constructs onto mathematical objects.
  - Useful for obtaining an abstract insight into the working of a program.

# Approaches to semantics

- Operational semantics: (developed by Plotkin)
  - The meaning of a program in terms of how it executes on an abstract machine.
  - Useful for modelling the execution behaviour of a program.
- Axiomatic semantics: (developed by Floyd and Hoare)
  - Provides correctness assertions for each program construct.
  - Useful for verifying that a program's computed results are correct with respect to the specification.
- Denotational semantics: (developed by Strachey and Scott)
  - Provides a mapping of language constructs onto mathematical objects.
  - Useful for obtaining an abstract insight into the working of a program.

Today: operational semantics of `pGCL` in terms of Markov chains.

# Approaches to semantics

- Operational semantics: (developed by Plotkin)
  - The meaning of a program in terms of how it executes on an abstract machine.
  - Useful for modelling the execution behaviour of a program.
- Axiomatic semantics: (developed by Floyd and Hoare)
  - Provides correctness assertions for each program construct.
  - Useful for verifying that a program's computed results are correct with respect to the specification.
- Denotational semantics: (developed by Strachey and Scott)
  - Provides a mapping of language constructs onto mathematical objects.
  - Useful for obtaining an abstract insight into the working of a program.

Today: operational semantics of `pGCL` in terms of Markov chains.

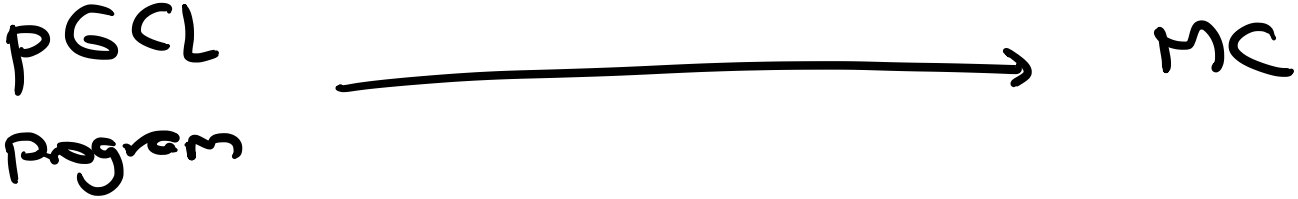
Later: denotational semantics of `pGCL` in terms of weakest preconditions.

# Structural operational semantics: ingredients

- Variable valuation $s : \mathit{Vars} \to \mathbb{Q}$ maps each program variable onto a value (here: rational numbers)

$s(x) = \frac{2}{3}$

$s(y) = 1$




[1] Here, we will not go into the details of this (simple) part.

# Structural operational semantics: ingredients

- Variable valuation $s : \mathit{Vars} \to \mathbb{Q}$ maps each program variable onto a value (here: rational numbers)

- Expression valuation[1], let $[\![ E ]\!]$ denote the valuation of expression $E$

$s(x) = \frac{2}{3}$

$s(y) = 1$

$E = 2x^2 + y$

$[\![ E ]\!](s) = 2 \cdot \left(\frac{2}{3}\right)^2 + 1$

[1] Here, we will not go into the details of this (simple) part.

# Structural operational semantics: ingredients

- Variable valuation $s : \mathit{Vars} \to \mathbb{Q}$ maps each program variable onto a value (here: rational numbers)
- Expression valuation[1], let $[\![ E ]\!]$ denote the valuation of expression $E$
- Configuration (aka: MC state) $\langle P, s \rangle$ denotes that
  - program $P$ is about to be executed (aka: program counter)
  - and the current variable valuation equals $s$.

[1] Here, we will not go into the details of this (simple) part.

# Structural operational semantics: ingredients

- Variable valuation $s : \mathit{Vars} \to \mathbb{Q}$ maps each program variable onto a value (here: rational numbers)
- Expression valuation[1], let $[\![ E ]\!]$ denote the valuation of expression $E$
- Configuration (aka: state) $\langle P, s \rangle$ denotes that
  - program $P$ is about to be executed (aka: program counter)
  - and the current variable valuation equals $s$.
- Transition rules for the execution of commands: $\langle P, s \rangle \longrightarrow \langle P', s' \rangle$

  transition rules are written as $\dfrac{\text{premise}}{\text{conclusion}}$

  where the premise is omitted if it is vacuously true.

[1] Here, we will not go into the details of this (simple) part.

# Recall: Markov chains

A Markov chain (MC) is a triple $(\Sigma, \sigma_I, \mathbf{P})$ with:

- $\Sigma$ being a countable set of states
- $\sigma_I \in \Sigma$ the initial state, and
- $\mathbf{P} : \Sigma \to \mathit{Dist}(\Sigma)$ the transition probability function

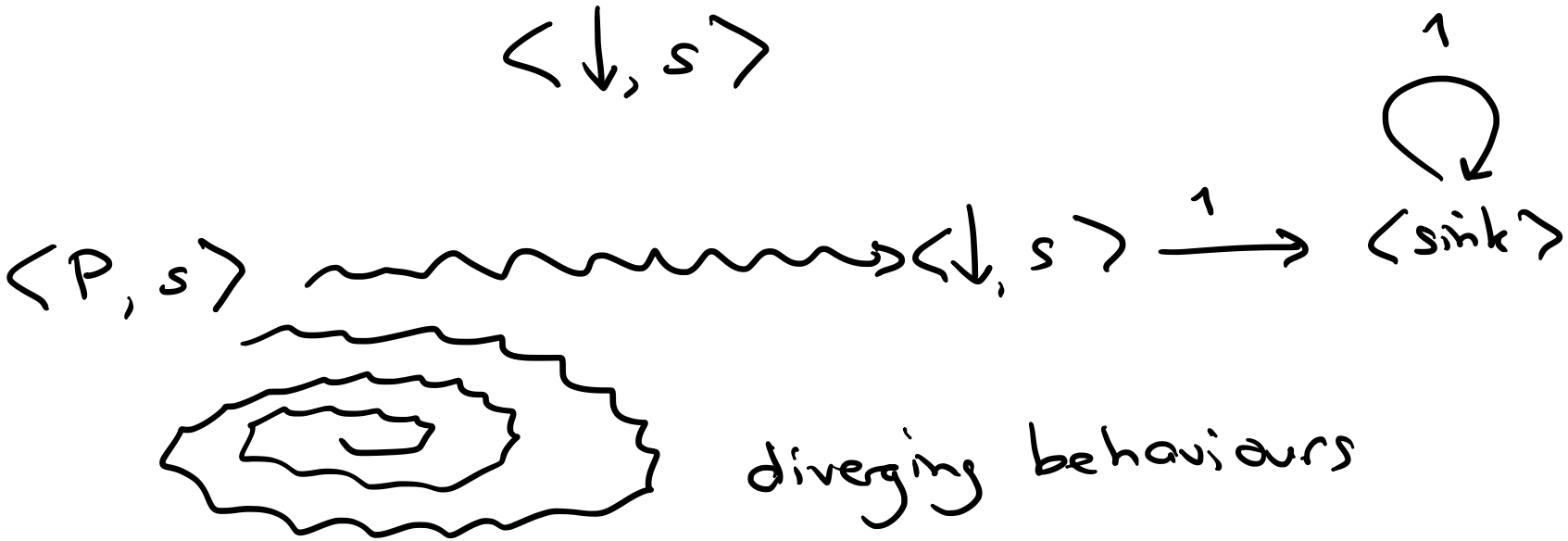
where $\mathit{Dist}(\Sigma)$ is a discrete probability measure on $\Sigma$.

# Operational semantics

Aim: Model the behaviour of a pGCL program $P$ by the MC $[\![ P ]\!]$.

Approach:

- Take states of the form
  - $\langle Q, s \rangle$ with program $Q$ or $\downarrow$, and variable valuation $s : \mathit{Vars} \to \mathbb{Q}$
  - $\langle \mathit{sink} \rangle$ models program termination (successful or violated observation)



# Operational semantics

Aim: Model the behaviour of a pGCL program $P$ by the MC $[\![ P ]\!]$.

Approach:

- Take states of the form
  - $\langle Q, s \rangle$ with program $Q$ or $\downarrow$, and variable valuation $s : \mathit{Vars} \to \mathbb{Q}$
  - $\langle \mathit{sink} \rangle$ models program termination (successful or violated observation)
- Take initial state $\sigma_I = \langle P, s \rangle$ where $s$ fulfils the initial conditions

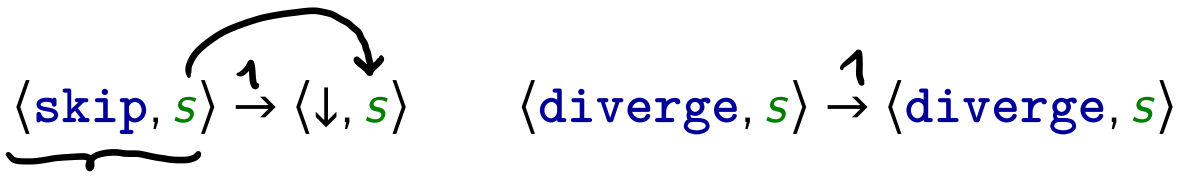
# Operational semantics

Aim: Model the behaviour of a pGCL program $P$ by the MC $[\![ P ]\!]$.

Approach:

- Take states of the form
  - $\langle Q, s \rangle$ with program $Q$ or $\downarrow$, and variable valuation $s : \mathit{Vars} \to \mathbb{Q}$
  - $\langle \mathit{sink} \rangle$ models program termination (successful or violated observation)
- Take initial state $\sigma_I = \langle P, s \rangle$ where $s$ fulfils the initial conditions
- Transition relation $\overset{1}{\to}$ is the smallest relation satisfying the SOS rules on the next slides
- Where transition probabilities equal to one are omitted

# Transition rules (1)



# Transition rules (1)

$$\langle \texttt{skip}, s \rangle \to \langle \downarrow, s \rangle \qquad \langle \texttt{diverge}, s \rangle \to \langle \texttt{diverge}, s \rangle$$

$$\langle \downarrow, s \rangle \xrightarrow{1} \langle \mathit{sink} \rangle \qquad \langle \mathit{sink} \rangle \xrightarrow{1} \langle \mathit{sink} \rangle$$

# Transition rules (1)

$$\langle \texttt{skip}, s \rangle \to \langle \downarrow, s \rangle \qquad \langle \texttt{diverge}, s \rangle \to \langle \texttt{diverge}, s \rangle$$

$$\langle \downarrow, s \rangle \to \langle \mathit{sink} \rangle \qquad \langle \mathit{sink} \rangle \to \langle \mathit{sink} \rangle$$

$$\langle x := E, s \rangle \to \langle \downarrow, s[x := s([\![ E ]\!])] \rangle$$

evaluate E in s

$2x^2+y$

assign that value to x

$$s\,[x := v]\,(z) = \begin{cases} v & \text{if } x = z \\ s(z) & \text{if } x \neq z \end{cases}$$

# Transition rules (1)

$$\langle \texttt{skip}, s\rangle \to \langle \downarrow, s\rangle \qquad \langle \texttt{diverge}, s\rangle \to \langle \texttt{diverge}, s\rangle$$

$$\langle \downarrow, s\rangle \to \langle \mathit{sink}\rangle \qquad \langle \mathit{sink}\rangle \to \langle \mathit{sink}\rangle$$

$$\langle x := E, s\rangle \to \langle \downarrow, s[x := s([\![ E ]\!])]\rangle$$

$$\frac{\mu(s)(v) = a > 0}{\langle x :\approx \mu, s\rangle \xrightarrow{a} \langle \downarrow, s[x := v]\rangle}$$

$x :\approx \mu$

$s(y) = 2$

$x := \text{unif}[1..2y]$

$v = 3$ $\qquad a = \frac{1}{4}$

# Transition rules (1)

$$\langle \texttt{skip}, s\rangle \to \langle \downarrow, s\rangle \qquad \langle \texttt{diverge}, s\rangle \to \langle \texttt{diverge}, s\rangle$$

$$\langle \downarrow, s\rangle \to \langle \mathit{sink}\rangle \qquad \langle \mathit{sink}\rangle \to \langle \mathit{sink}\rangle$$

$$\langle x := E, s\rangle \to \langle \downarrow, s[x := s([\![ E ]\!])]\rangle$$

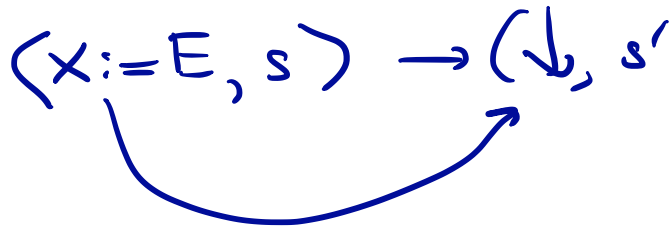
$$\frac{\mu(s)(v) = a > 0}{\langle x :\approx \mu, s\rangle \xrightarrow{a} \langle \downarrow, s[x := v]\rangle}$$

$$\langle P[\, p\,] Q, s\rangle \to \mu \text{ with } \mu(\langle P, s\rangle) = p \text{ and } \mu(\langle Q, s\rangle) = 1-p$$

$\downarrow$

$\in \mathrm{Dist}(\Sigma)$

# Transition rules (2)




$$\frac{\langle P, s\rangle \to \mu}{\langle P; Q, s\rangle \to \nu} \text{ with } \nu(\langle P'; Q, s'\rangle) = \mu(\langle P', s'\rangle) \text{ where } \downarrow; Q = Q$$

$\mu \in Disk(\Sigma)$

$$\frac{(P, s) \xrightarrow{\frac{1}{2}} (P', s')}{(P; Q, s) \xrightarrow{\frac{1}{2}} (P'; Q, s')}$$

$$\frac{(P, s) \to (\downarrow, s')}{(P; Q, s) \to (\downarrow; Q, s')} \qquad \downarrow; Q = Q$$

# Transition rules (2)

$$\frac{\langle P, s\rangle \to \mu}{\langle P; Q, s\rangle \to \nu} \text{ with } \nu(\langle P'; Q', s'\rangle) = \mu(\langle P', s'\rangle) \text{ where } \downarrow; Q = Q$$

$$\frac{s \vDash G}{\langle \text{if } (G)\{P\} \text{ else } \{Q\}, s\rangle \to \langle P, s\rangle} \qquad \frac{s \nvDash G}{\langle \text{if } (G)\{P\} \text{ else } \{Q\}, s\rangle \to \langle Q, s\rangle}$$

$s \vDash G$ $\quad s(x) = \frac{2}{3}$ $\quad s(y) = 1$

$G = x > 2y$

$s \overset{?}{\vDash} G \quad s(x) > 2\,s(y)$

$\Leftrightarrow \frac{2}{3} > 2 \cdot 1$

$s \nvDash G$

# Transition rules (2)

$$\frac{\langle P, s\rangle \to \mu}{\langle P; Q, s\rangle \to \nu} \text{ with } \nu(\langle P'; Q', s'\rangle) = \mu(\langle P', s'\rangle) \text{ where } \downarrow; Q = Q$$

$$\frac{s \models G}{\langle \text{if } (G)\{P\} \text{ else } \{Q\}, s\rangle \to \langle P, s\rangle} \qquad \frac{s \not\models G}{\langle \text{if } (G)\{P\} \text{ else } \{Q\}, s\rangle \to \langle Q, s\rangle}$$

$$\frac{s \models G}{\langle \text{while}(G)\{P\}, s\rangle \to \langle P; \text{while } (G)\{P\}, s\rangle} \qquad \frac{s \not\models G}{\langle \text{while}(G)\{P\}, s\rangle \to \langle \downarrow, s\rangle}$$

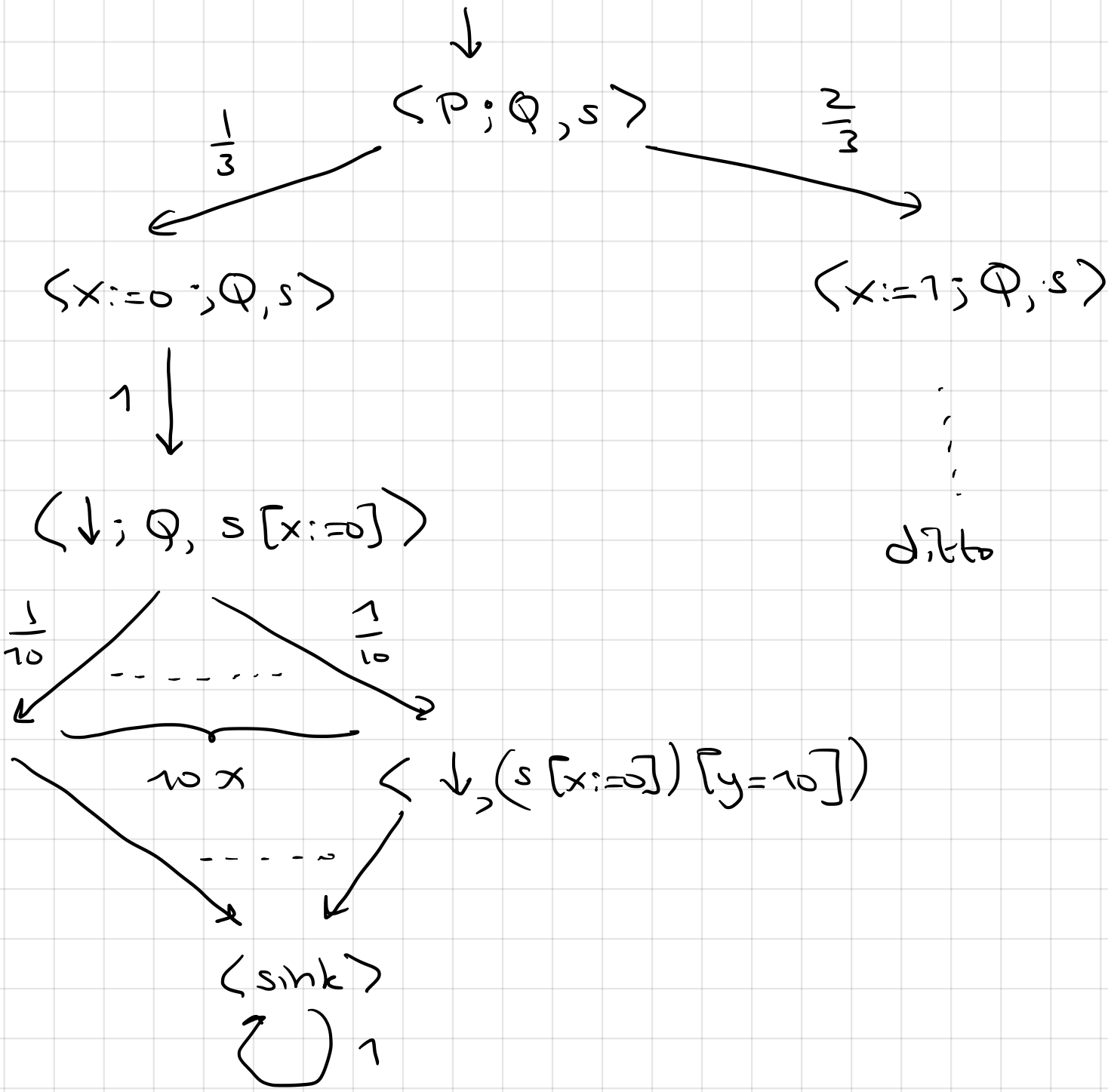
# Example

$$\underbrace{x := 0 \;[\tfrac{1}{3}]\; x := 1}_{P} \;;\; \underbrace{y :\approx \mathrm{unif}[1..10]}_{Q}$$

$$\frac{\langle P, s\rangle \xrightarrow{\frac{1}{3}} \langle x := 0, s\rangle}{\langle P; Q, s\rangle \xrightarrow{\frac{1}{3}} \langle x := 0; Q, s\rangle}$$

$$\frac{\langle x := 0, s\rangle \xrightarrow{1} \langle \downarrow, \overbrace{s[x := 0]}^{= s'}\rangle}{\langle x := 0; Q\rangle \xrightarrow{1} \langle \underbrace{\downarrow; Q}_{= Q}, s[x := 0]\rangle}$$

$$\frac{\mathrm{unif}[1..10](s')(6) = \frac{1}{10} > 0}{\langle y :\approx \mathrm{unif}[1..10], s'\rangle \xrightarrow{\frac{1}{10}} \langle \downarrow, \underbrace{s'[y := 6]}_{\begin{array}{l} s'(x) = 0 \\ s'(y) = 6 \end{array}}\rangle}$$




What changes if

$$x := 0 \ [\tfrac{1}{3}] \ x := 1 \ ; \ y :\approx \mathrm{unif}[1..x]$$

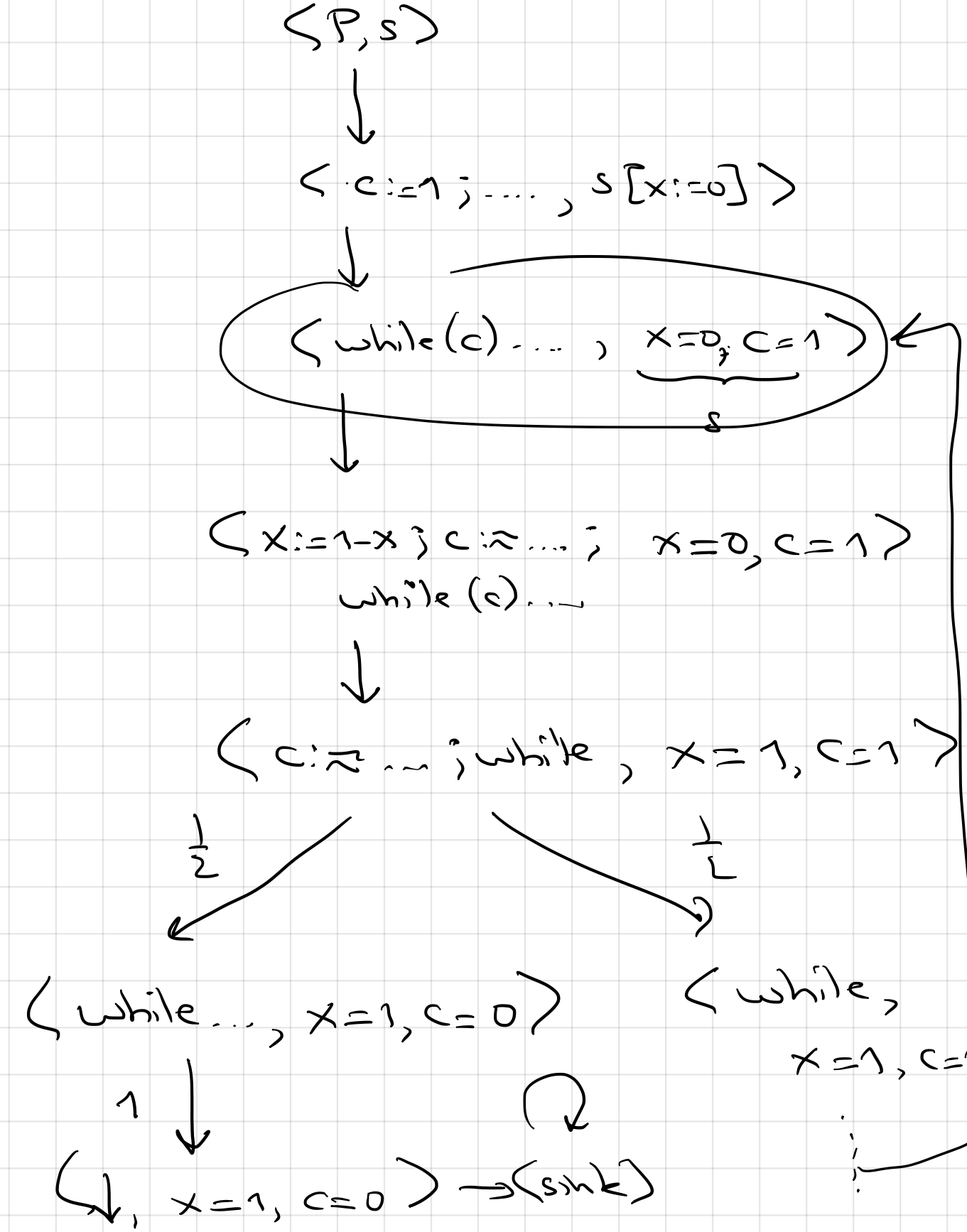
$P ::=$

$x := 0 ;\ c := 1 ;\ \text{while}\ (c)\ \{\ x := 1 - x\ ;$

$c :\approx [\tfrac{1}{2}) \cdot 0 +$

$[\tfrac{1}{2}) \cdot 1\ \}$

$X := X + 1$

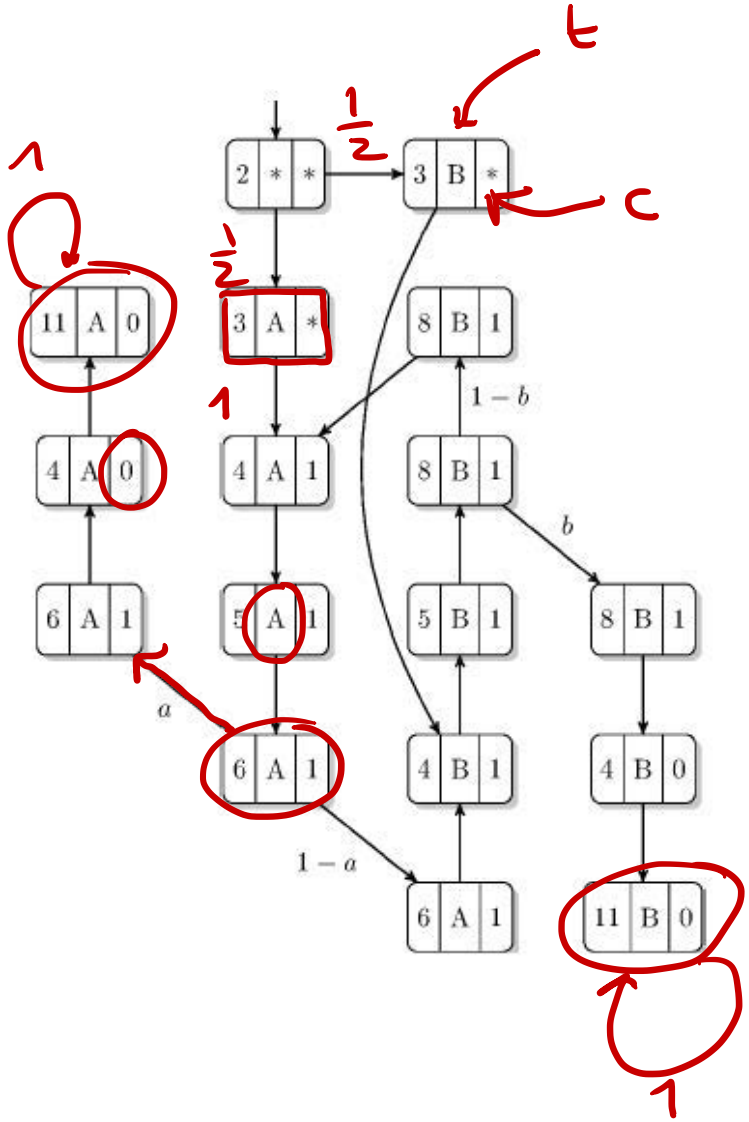


# Duelling cowboys

```
int cowboyDuel(float a, b) {
  int t := A [0.5] t := B;
  bool c := true;
  while (c) {
    if (t = A) {
      (c := false [a] t := B);
    } else {
      (c := false [b] t := A);
    }
  }
  return t;
}
```

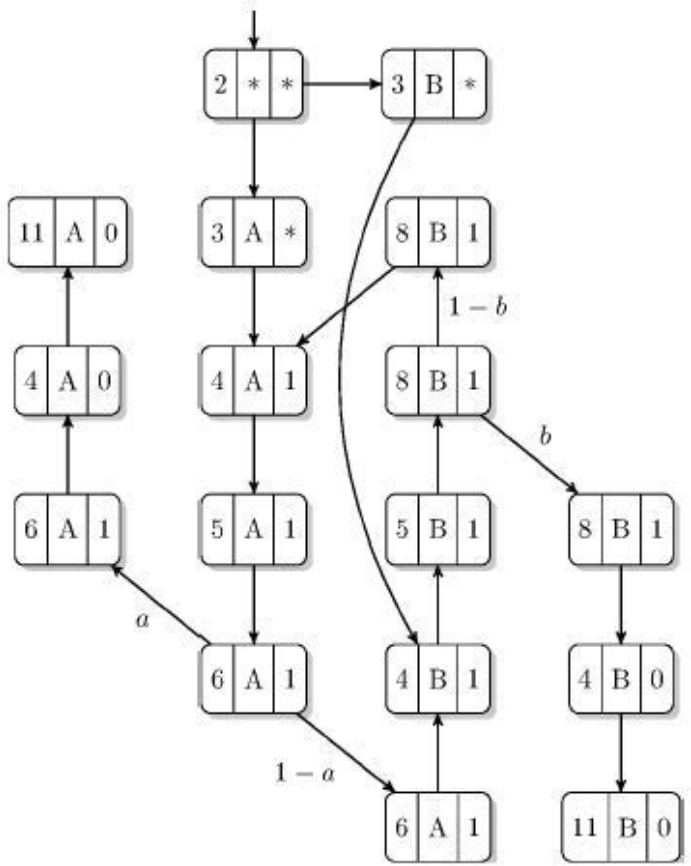
# Duelling cowboys

```
int cowboyDuel(float a, b) {
  int t := A [0.5] t := B;
  bool c := true;
  while (c) {
    if (t = A) {
      (c := false [a] t := B);
    } else {
      (c := false [b] t := A);
    }
  }
  return t;
}
```



# Duelling cowboys

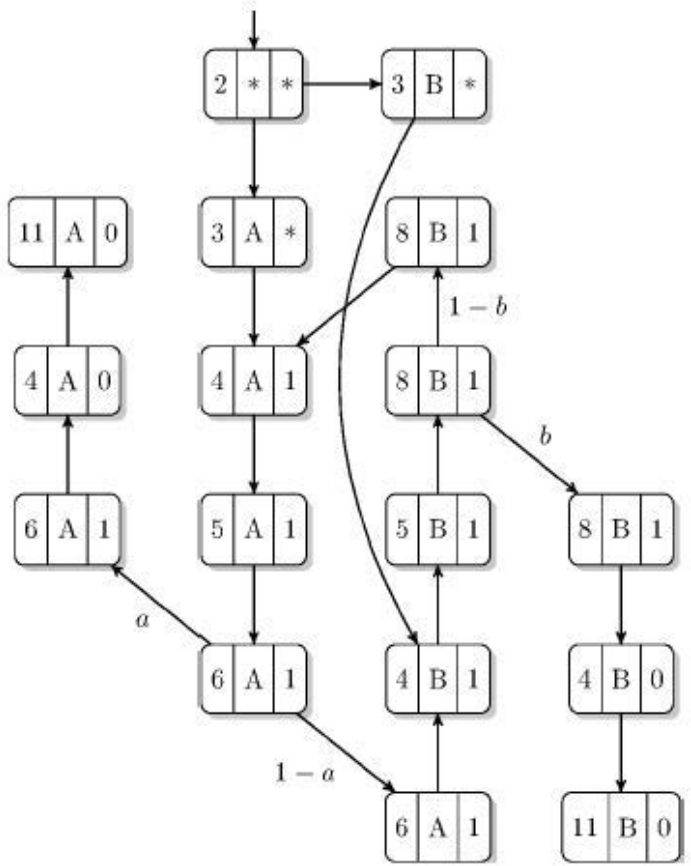
```
int cowboyDuel(float a, b) {
  int t := A [0.5] t := B;
  bool c := true;
  while (c) {
    if (t = A) {
      (c := false [a] t := B);
    } else {
      (c := false [b] t := A);
    }
  }
  return t;
}
```




This (parametric) MC is finite.

# Duelling cowboys

```
int cowboyDuel(float a, b) {
  int t := A [0.5] t := B;
  bool c := true;
  while (c) {
    if (t = A) {
      (c := false [a] t := B);
    } else {
      (c := false [b] t := A);
    }
  }
  return t;
}
```




This (parametric) MC is finite. Once we count the number of shots before one of the cowboys dies, the MC becomes countably infinite.

# Playing with geometric distributions

# Playing with geometric distributions

- $X$ is a random variable, geometrically distributed with parameter $p$

# Playing with geometric distributions

- $X$ is a random variable, geometrically distributed with parameter $p$
- $Y$ is a random variable, geometrically distributed with parameter $q$

# Playing with geometric distributions

- $X$ is a random variable, geometrically distributed with parameter $p$
- $Y$ is a random variable, geometrically distributed with parameter $q$

Q: generate a sample $x$, say, according to the random variable $X - Y$

# Playing with geometric distributions

- $X$ is a random variable, geometrically distributed with parameter $p$
- $Y$ is a random variable, geometrically distributed with parameter $q$

Q: generate a sample $x$, say, according to the random variable $X - Y$

```
int XminY1(float p, q){ // 0 <= p, q <= 1
  int x := 0;
  bool flip := false;
  while (not flip) { // take a sample of X to increase x
    (x +:= 1 [p] flip := true);
  }
  flip := false;
  while (not flip) { // take a sample of Y to decrease x
    (x -:= 1 [q] flip := true);
  }
  return x; // a sample of X-Y
}
```

# An alternative program

```
int XminY2(float p, q){
  int x := 0;
  bool flip := false;
  (flip := false [0.5] flip := true); // flip a fair coin
  if (not flip) {
    while (not flip) { // sample X to increase x
      (x +:= 1 [p] flip := true);
    }
  } else {
    flip := false; // reset flip
    while (not flip) { // sample Y to decrease x
      x -:= 1;
      (skip [q] flip := true);
    }
  }
return x; // a sample of X-Y
}
```

# Program equivalence: $X - Y$

# Program equivalence: $X - Y$

```
int XminY1(float p, q){
  int  x, c := 0, 1;
  while (c) {
    (x +:= 1 [p] c := 0);
  }
  c := 1;
  while (c) {
    (x -:= 1 [q] c := 0);
  }
  return x;
}
```

# Program equivalence: $X - Y$

```
int XminY1(float p, q){
  int x, c := 0, 1;
  while (c) {
    (x +:= 1 [p] c := 0);
  }
  c := 1;
  while (c) {
    (x -:= 1 [q] c := 0);
  }
  return x;
}
```

```
int XminY2(float p, q){
  int x := 0;
  (c := 0 [0.5] c := 1);
  if (c) {
    while (c) {
      (x +:= 1 [p] c := 0);
    }
  } else {
    c := 1;
    while (c) {
      x -:= 1;
      (skip [q] c := 0);
    }
  }
return x;
}
```

# Program equivalence: $X - Y$

```
int XminY1(float p, q){
  int x, c := 0, 1;
  while (c) {
    (x +:= 1 [p] c := 0);
  }
  c := 1;
  while (c) {
    (x -:= 1 [q] c := 0);
  }
  return x;
}
```

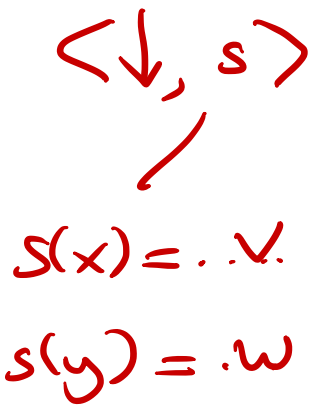
```
int XminY2(float p, q){
  int x := 0;
  (c := 0 [0.5] c := 1);
  if (c) {
    while (c) {
      (x +:= 1 [p] c := 0);
    }
  } else {
    c := 1;
    while (c) {
      x -:= 1;
      (skip [q] c := 0);
    }
  }
return x;
}
```

The probability that $x = k$ for some $k \in \mathbb{Z}$ coincides for both programs if and only if $q = \frac{1}{2-p}$.

# The outcome of a pGCL program

# The outcome of a pGCL program

Unlike a deterministic program, a pGCL program $P$ has not an output for a given input. Instead, it yields a unique probability distribution over its final states.



# The outcome of a pGCL program

$\langle \downarrow, s \rangle$

Unlike a deterministic program, a pGCL program $P$ has not an output for a given input. Instead, it yields a unique probability distribution over its final states.

In fact, this is a sub-distribution (probability mass at most one), as with a certain probability $P$ may diverge.

$$\mu : \Sigma \to [0,1]$$

$$\sum_{\sigma \in \Sigma} \mu(\sigma) = 1$$

$$\sum_{\sigma \in \Sigma} \mu(\sigma) \leq 1$$

# The outcome of a pGCL program

Unlike a deterministic program, a pGCL program $P$ has not an output for a given input. Instead, it yields a unique probability distribution over its final states.

In fact, this is a sub-distribution (probability mass at most one), as with a certain probability $P$ may diverge.

Let $P$ be a pGCL program and $s$ an input state. Then the distribution over final states obtained by running $P$ starting in $s$ is given by $Pr(s \vDash \Diamond \langle \downarrow, \cdot \rangle)$.

# Reachability probabilities

If the MC $[\![ P ]\!]$ of pGCL program $P$ has finitely many states, reachability probabilities can be obtained in an automated manner. This applies to the cowboy example for given probabilities $a$ and $b$.

$\frac{1}{2}$ $\frac{1}{3}$

The same holds for expected rewards, e.g., the expected number of steps until termination of a finite-state program $P$.

# Overview

# Probabilistic GCL with recursion: Syntax

| | |
|---|---|
| `skip` | empty statement |
| `diverge` | divergence |
| `x := E` | assignment |
| `x :r= mu` | random assignment ($x :\approx \mu$) |
| `prog1 ; prog2` | sequential composition |
| `if (G) prog1 else prog2` | choice |
| `prog1 [p] prog2` | probabilistic choice |
| `while (G) prog` | iteration |
| `P = prog` | **process definition** |
| `call P` | **process invocation** |

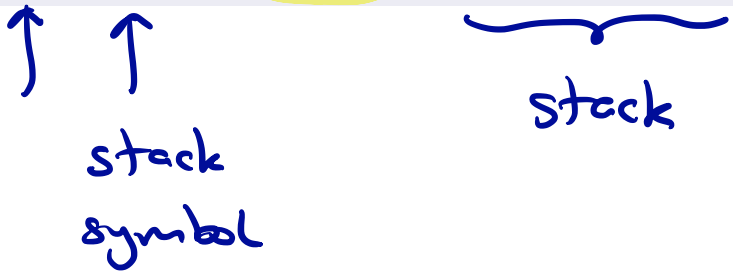
Recursion does not increase the expressive power, but is often convenient.

# Pushdown Markov chains

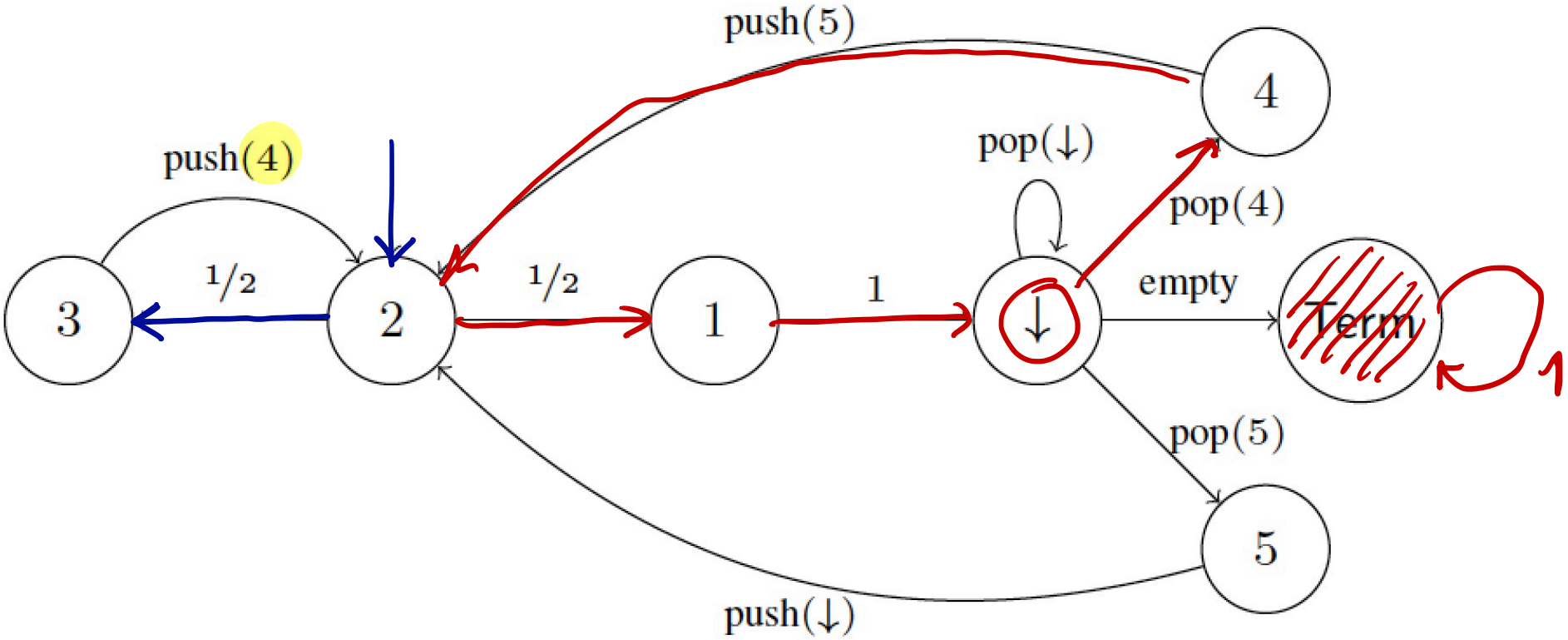
## Pushdown Markov chain

A pushdown Markov chain $D$ is a tuple $(\Sigma, \sigma_I, \Gamma, \gamma_0, \Delta)$ where:

- $\Sigma$ is a countable set of (control) states
- $\sigma_I \in \Sigma$ is the initial (control) state
- $\Gamma$ is a finite stack alphabet
- $\gamma_0 \in \Gamma$ is the bottom-of-the-stack symbol
- $\Delta : \Sigma \times \Gamma \to \mathit{Dist}(\Sigma) \times (\Gamma \setminus \{\gamma_0\})^*$ is the probability transition relation



# Recursion: pushdown Markov chains




$P ::= \{\text{skip}^1\}\ [1/2]^2\ \{\text{call } P^3; \text{call } P^4; \text{call } P^5\}$

prog = call P