A BRIEF INTRODUCTION TO GENETIC PROGRAMMING

Luca Manzoni

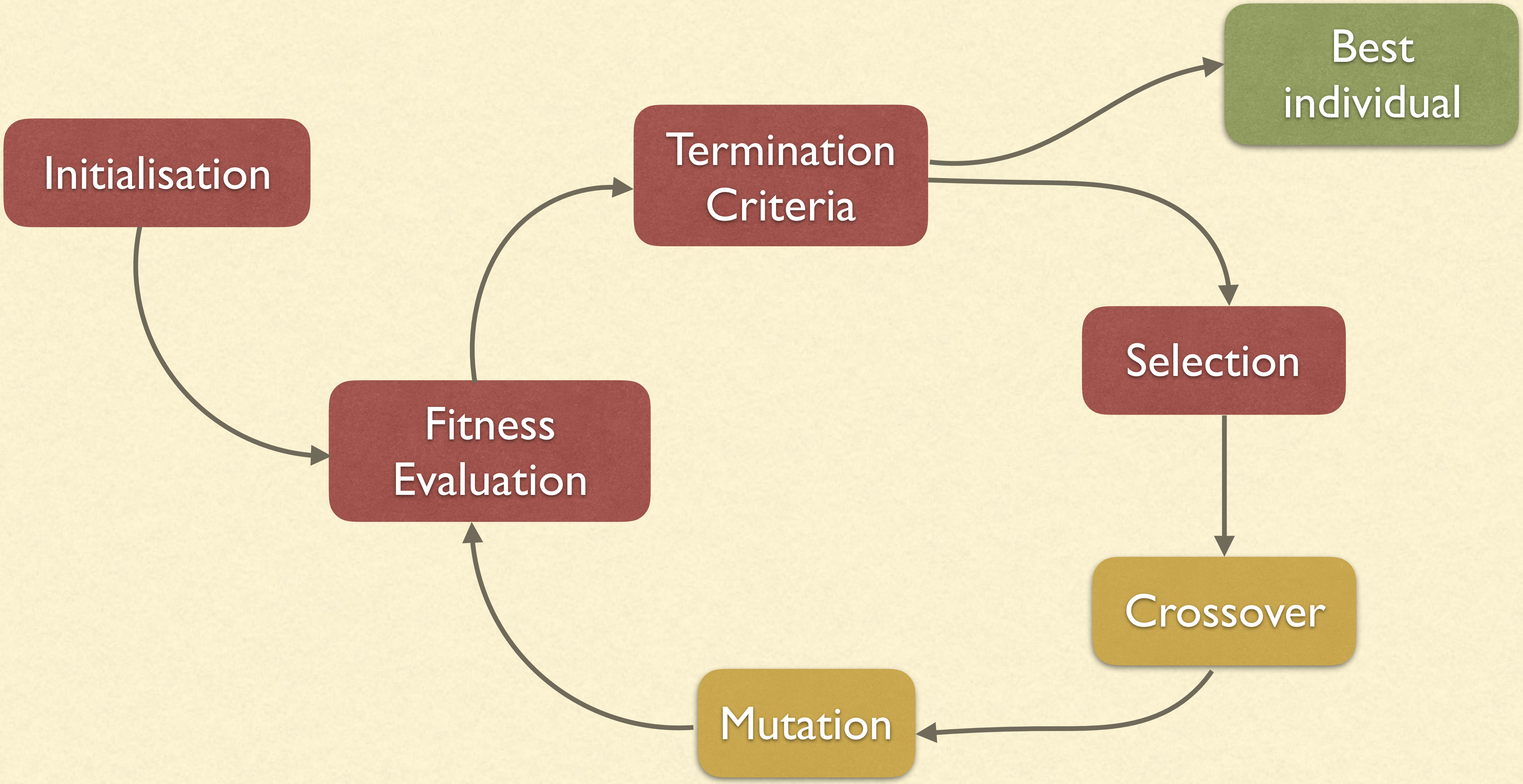
OUTLINE

- How programs are represented: tree-based GP
- Initialisation, Crossover, and Mutation
- Overfitting and how to combat bloat
- Linear GP
- Cartesian GP

WHAT'S GP?

- GP is a technique to *stochastically* evolve a *population* (multiset) of individuals encoding *computer programs*
- John Koza was one of the firsts to talk about evolving computer programs in the late 80s
- While the main "evolutionary cycle" is like the one for GA…
- …GP has many peculiar properties, starting from the representation used.

EVOLUTION CYCLE



PROGRAM REPRESENTATION

How can we represent programs that evolve?

Parent 1

```
if (a > 0){
  do_foo();
}
```

Crossover point

Parent 2

```
while (x != 4){
  do_baz();
}
```

Crossover point

Offspring 1

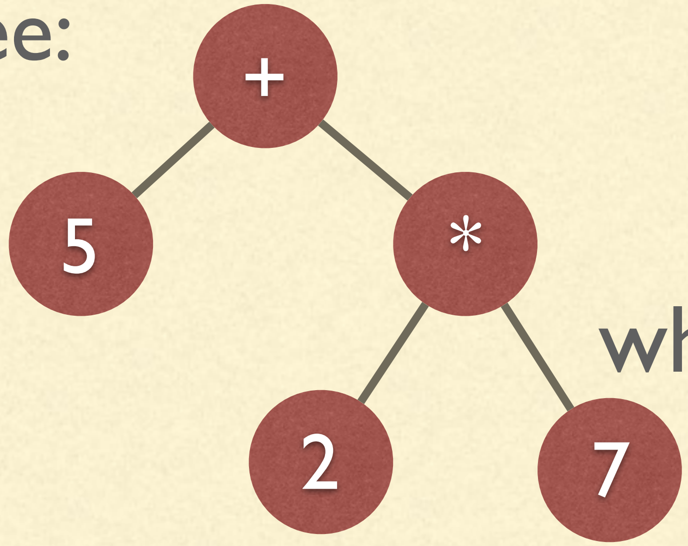
```
if (a > 0){
  do_foo(baz();
}
```

Offspring 2

```
while (x != 4){
  do_);
}
```

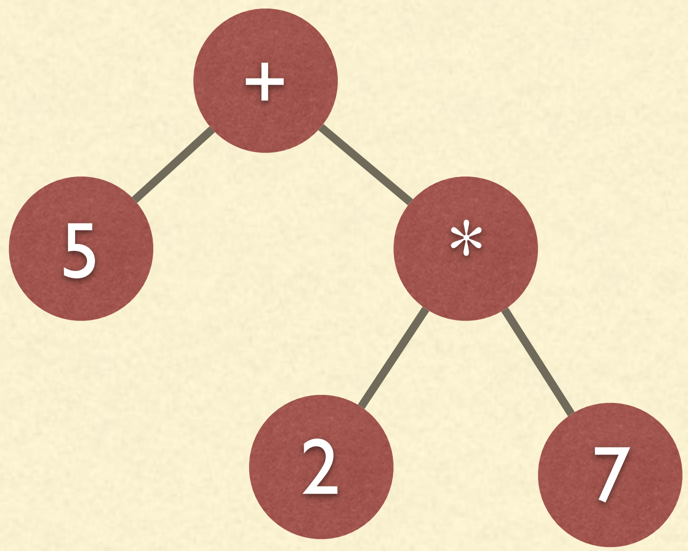
PROGRAM REPRESENTATION

- We must select a representation where crossover and mutation can easily work…
- …which means that “a string” is usually not the best choice
- If we start with an expression like “5 + (7 * 2)” we can represent it as its parsing tree:



which seems a more suitable representation

PROGRAM REPRESENTATION



Crossover and mutation can be written as operations on subtrees

Evaluation of a tree is quite simple

What to we need to encode a program as a tree?

What are the possible internal nodes and leaves?

TERMINAL SET

The terminal set contains all the possible leaves, for example:

Constants	Input variables
$\{0,1,2,3,4,\ldots\}$	$\{x_0, x_1, \ldots\}$
$\{\text{true, false}\}$	
$\{e, \pi, -1, \ldots\}$	

FUNCTION SET

The terminal set contains the possible inner nodes, for example:

Arithmetical operations	$\{ + , - , \times , \div \}$
Trigonometric functions	$\{\sin, \cos, \tan\}$
Boolean operators	$\{ \wedge , \vee , \neg \}$
Choice/conditional	{if … then … else …}

CLOSURE

- The primitives sets (functionals and terminals) must respect the property of closure:
- Intuitively, we must be able to "mix" the primitives without problems.
- *Type consistency*: the terminals and the output of any functional must be valid inputs to all functionals
- *Evaluation safety*: primitives that can fail at runtime (e.g., division) should be "protected" to avoid runtime failures.

SUFFICIENCY

- To find a solution we must be able to represent it, which means that the primitives must be sufficient to write a solution
- For example: with real constants, variables, and {+, -, *,} we can represent any polynomial…
- …which is not very useful if the function that we must fit is an exponential
- Usually we cannot assure sufficiency, but we might still obtain solutions that are good approximations

INITIALISATION METHODS

INITIALISATION

- For binary strings (traditional GA), a random generation is easy: select each bit independently with a uniform probability
- Finite trees are infinite in number…
- …and to generate them we must recall that:
- “*Random numbers should not be generated with a method chosen at random.*” — Donald Knuth
- …also holds for randomly generating trees.

GROW

Up to a maximum depth, select randomly across all primitives
Once the maximum depth is reached select a terminal

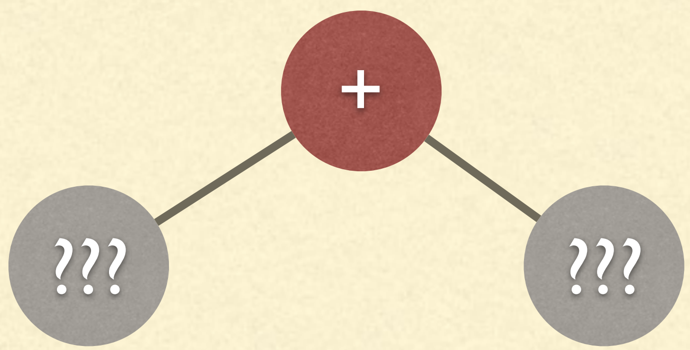
$$\mathcal{F} = \{ + , * , - , / \}$$
$$\mathcal{T} = \{x, y, 1\}$$

max_depth $= 2$

GROW

Up to a maximum depth, select randomly across all primitives
Once the maximum depth is reached select a terminal

$$\mathcal{F} = \{ + , * , - , / \}$$
$$\mathcal{T} = \{x, y, 1\}$$

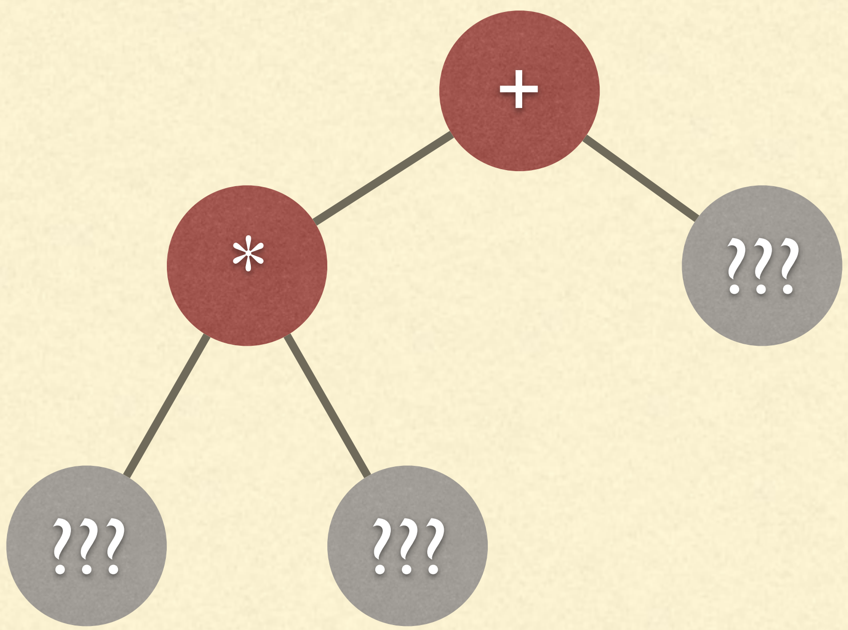


max_depth $= 2$

GROW

Up to a maximum depth, select randomly across all primitives
Once the maximum depth is reached select a terminal

$$\mathcal{F} = \{ + , * , - , / \}$$
$$\mathcal{T} = \{x, y, 1\}$$



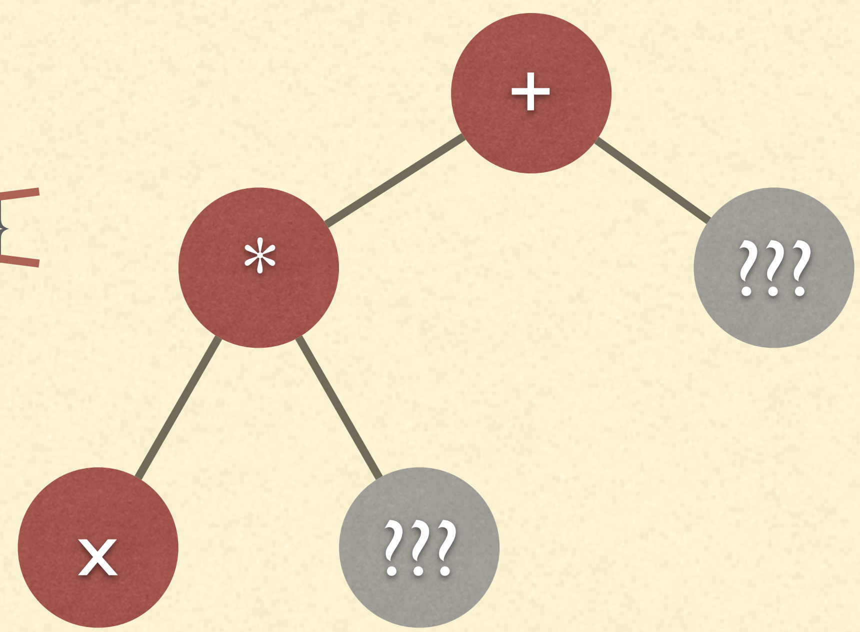
max_depth $= 2$

GROW

Up to a maximum depth, select randomly across all primitives
Once the maximum depth is reached select a terminal

$\cancel{\mathcal{F} = \{+, *, -, /\}}$

$\mathcal{T} = \{x, y, 1\}$



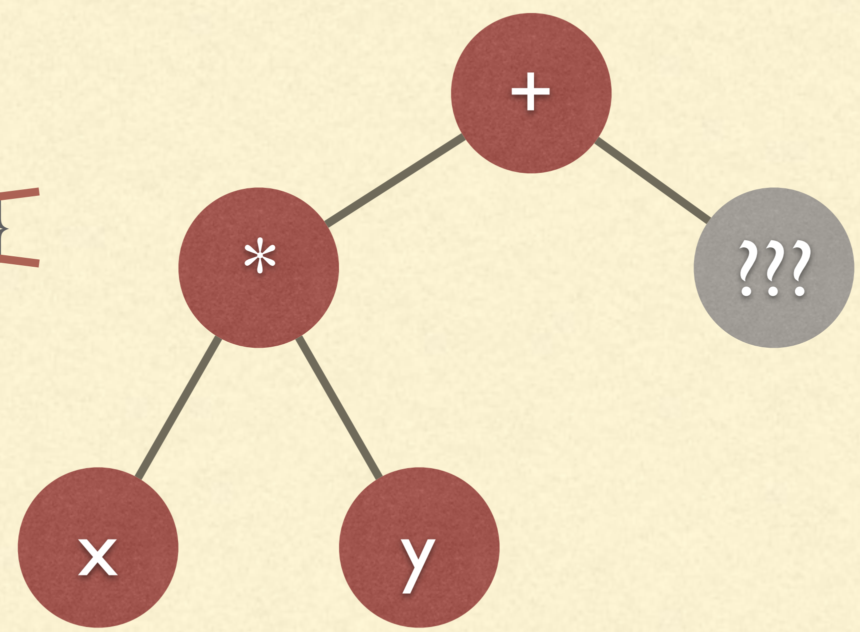
max_depth $= 2$

GROW

Up to a maximum depth, select randomly across all primitives
Once the maximum depth is reached select a terminal

$$\cancel{\mathcal{F} = \{+, *, -, /\}}$$

$$\mathcal{T} = \{x, y, 1\}$$



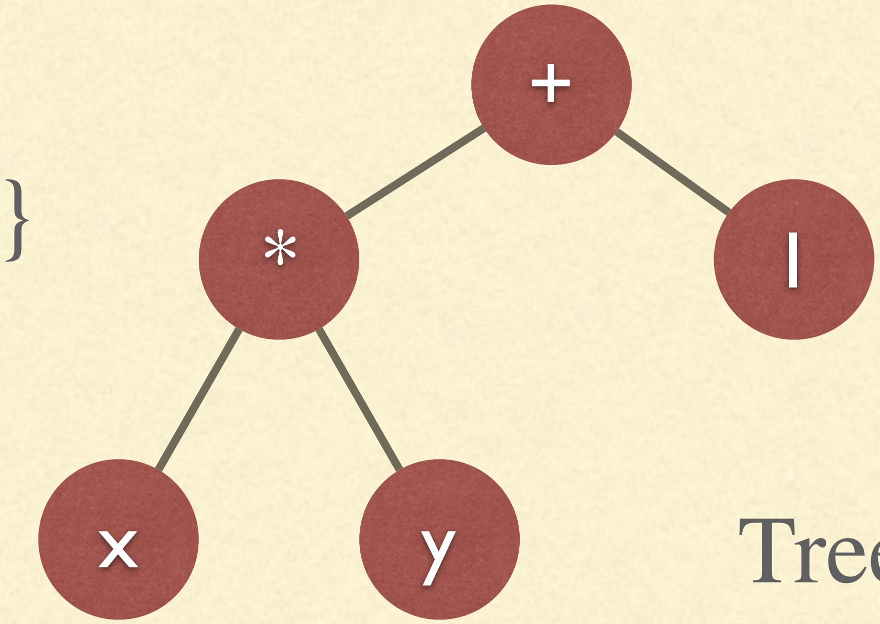
max_depth $= 2$

GROW

Up to a maximum depth, select randomly across all primitives
Once the maximum depth is reached select a terminal

$\mathcal{F} = \{+, *, -, /\}$

$\mathcal{T} = \{x, y, 1\}$



Tree $= xy + 1$

max_depth $= 2$

FULL

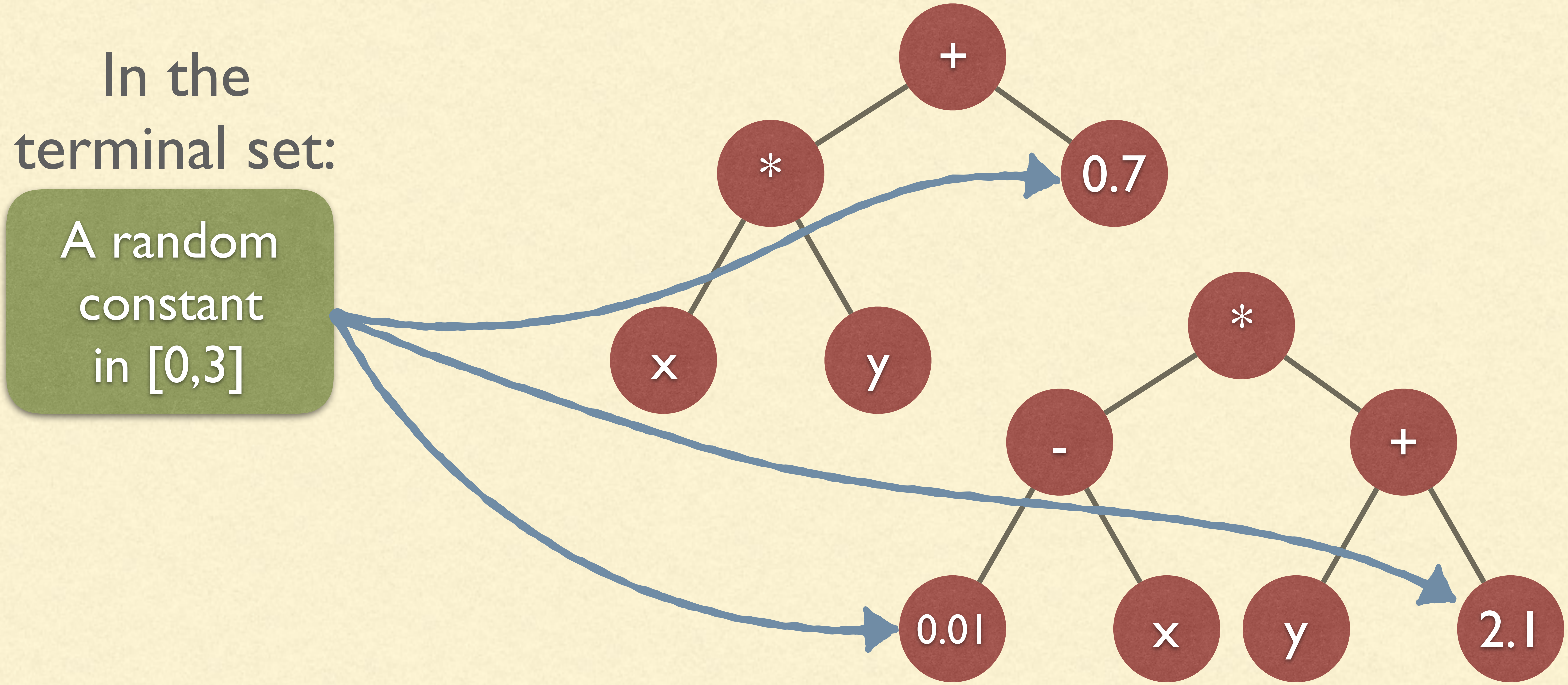
- Like the “grow” method, but only functional symbols are selected before reaching the maximum depth
- This means that terminals appears only in the last level of the tree
- Different distribution of trees w.r.t. the “grow” method (obviously)
- Generally bigger trees

RAMPED HALF AND HALF

- Randomly select between "grow" and "full"…
- …with a random maximum depth between a minimum and a maximum
- Generally trees with a better "variability" among them (non all representing similar functions)

EPHEMERAL CONSTANTS

The terminals might include constant. But how to chose them?
An alternative to the choice is the use of *ephemeral constants*



CROSSOVER

HOMOLOGOUS CROSSOVER

Lets move back to GA

$x =$ | 0 | 1 | 0 | 1 | 1 | 0 |

$x =$ | 0 | 1 | 0 | 1 | 1 | 0 |

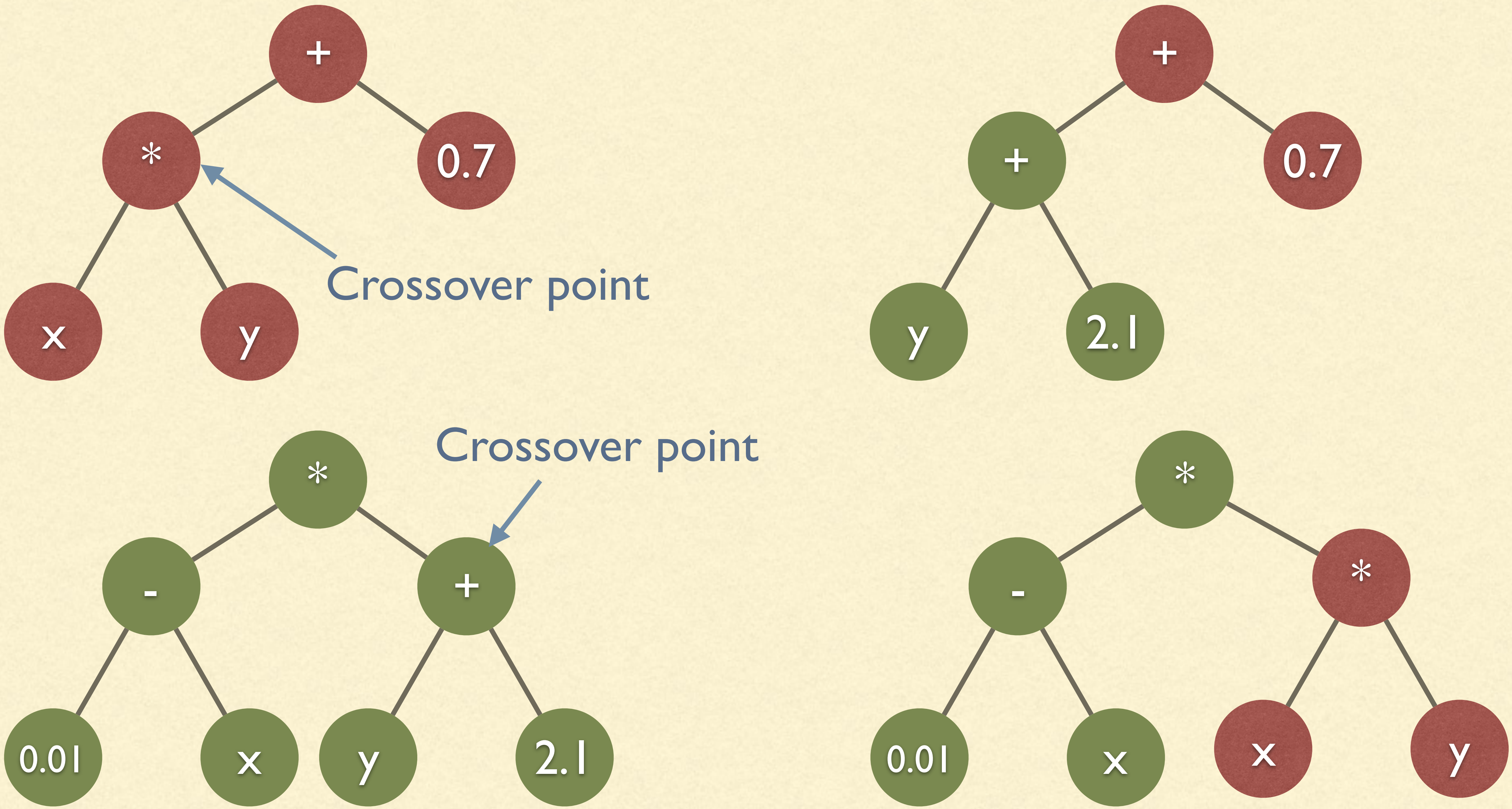
Can we obtain a different individual by crossing x with itself?

NO

The classical GA crossovers are homologous

This is not usually the case for GP

SUBTREE CROSSOVER



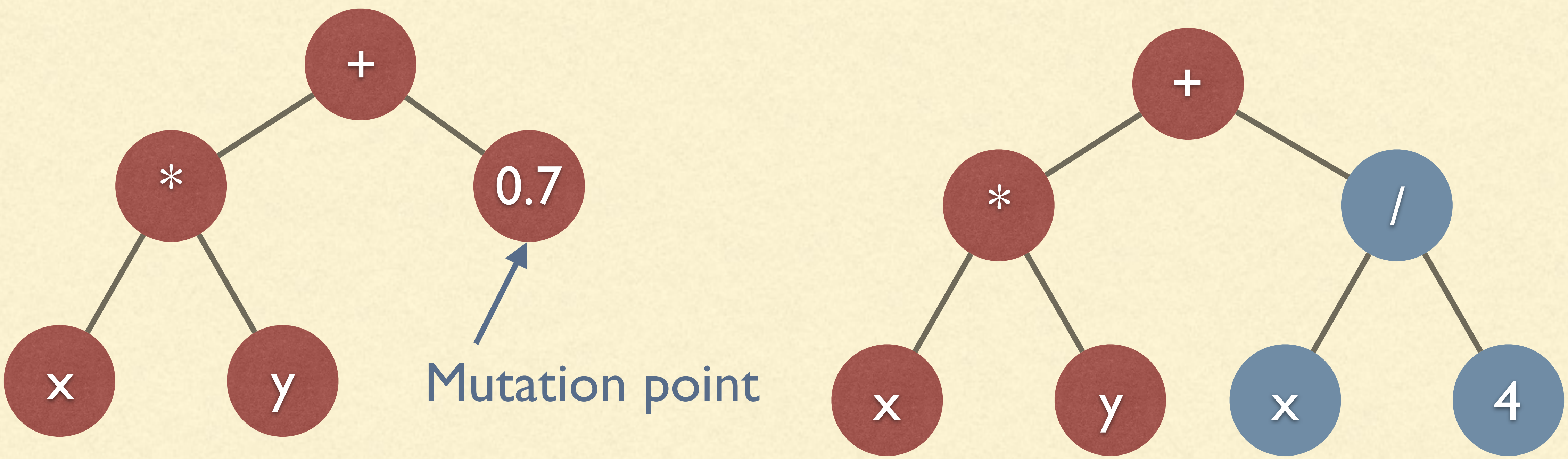
SUBTREE CROSSOVER

- In many cases trees have a maximum depth during evolution
- Which means that subtree crossover must not exceed it
- Subtree crossover is non-homologous…
- …but there exist crossovers for GP that *are* homologous

MUTATION(S)

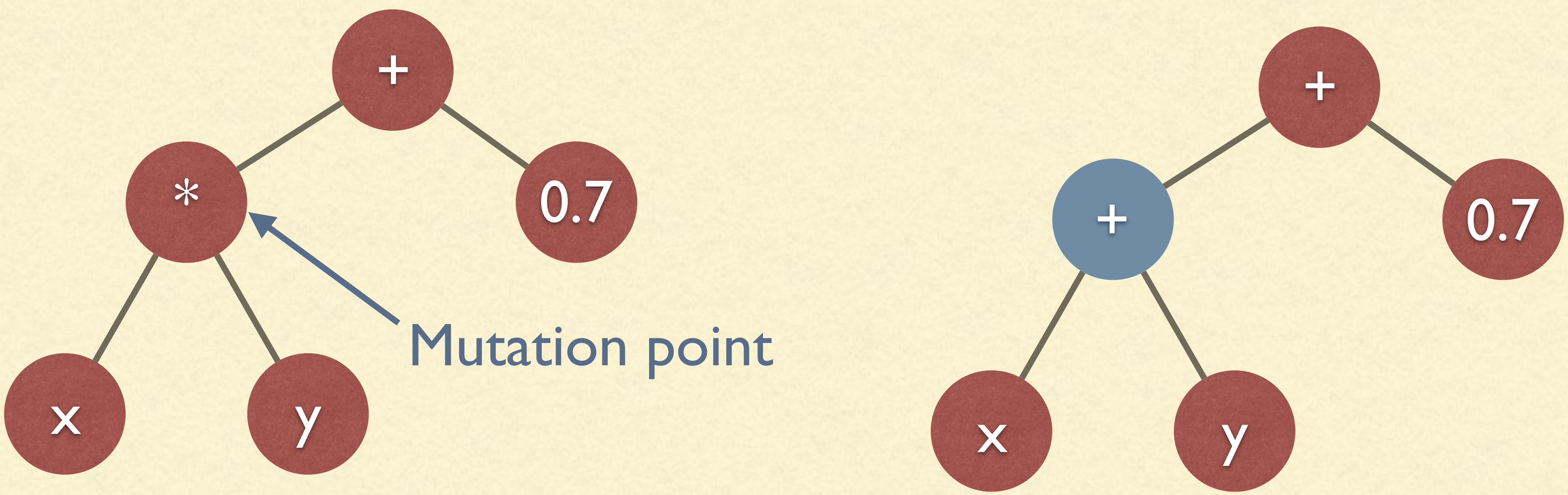
SUBTREE MUTATION

Replacement of a randomly selected subtree
with a new random subtree



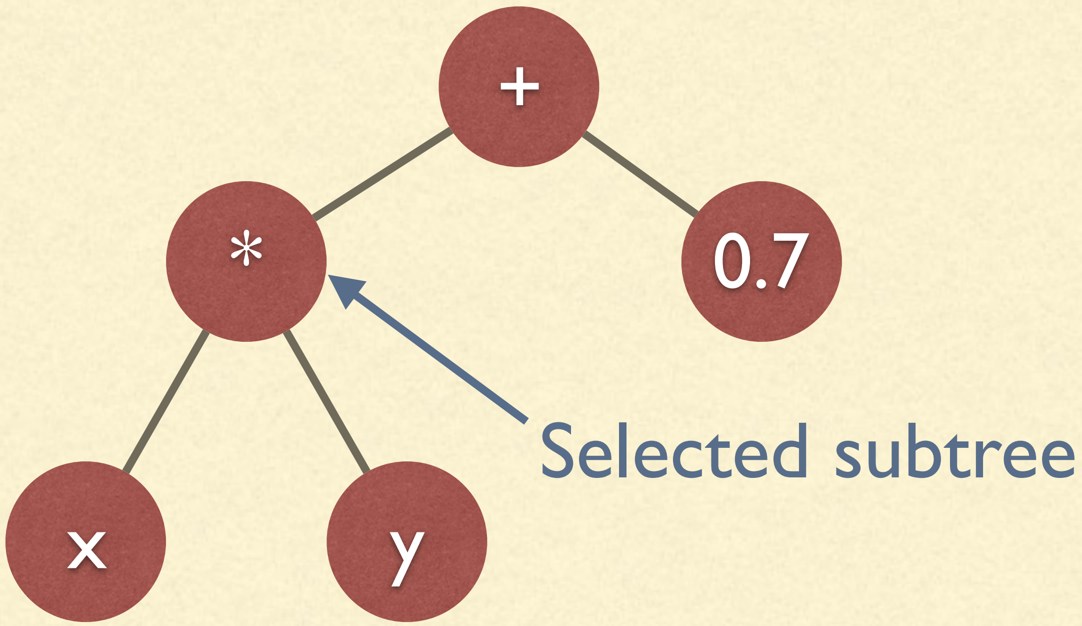
POINT MUTATION

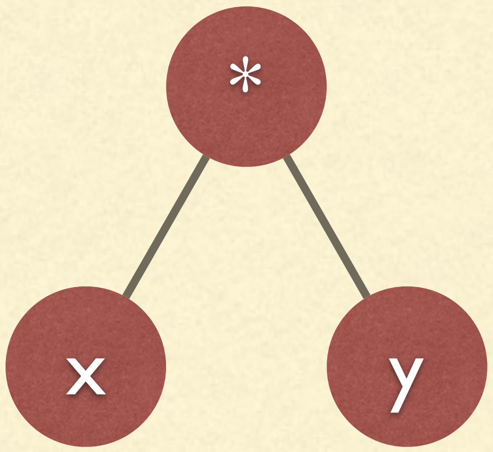
Replacement of a randomly selected node with a compatible randomly selected node



HOIST MUTATION

Replacement of the entire tree
with one of its subtree

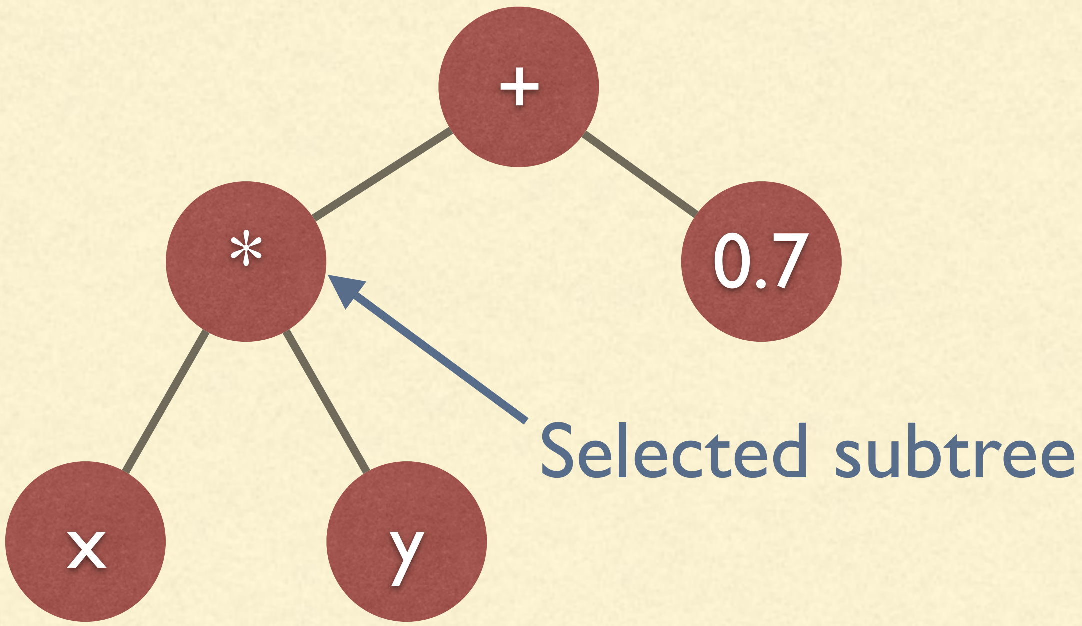


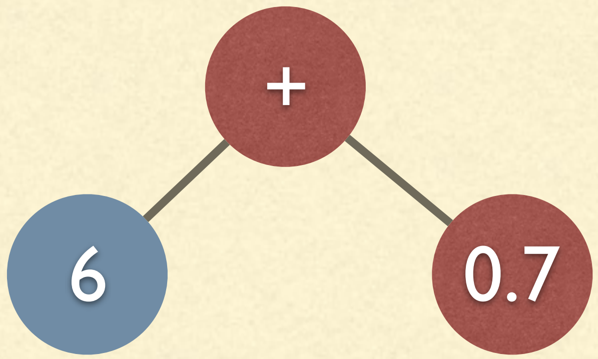


Used to reduce program size

SHRINK MUTATION

Replacement a randomly selected subtree
with a randomly selected terminal



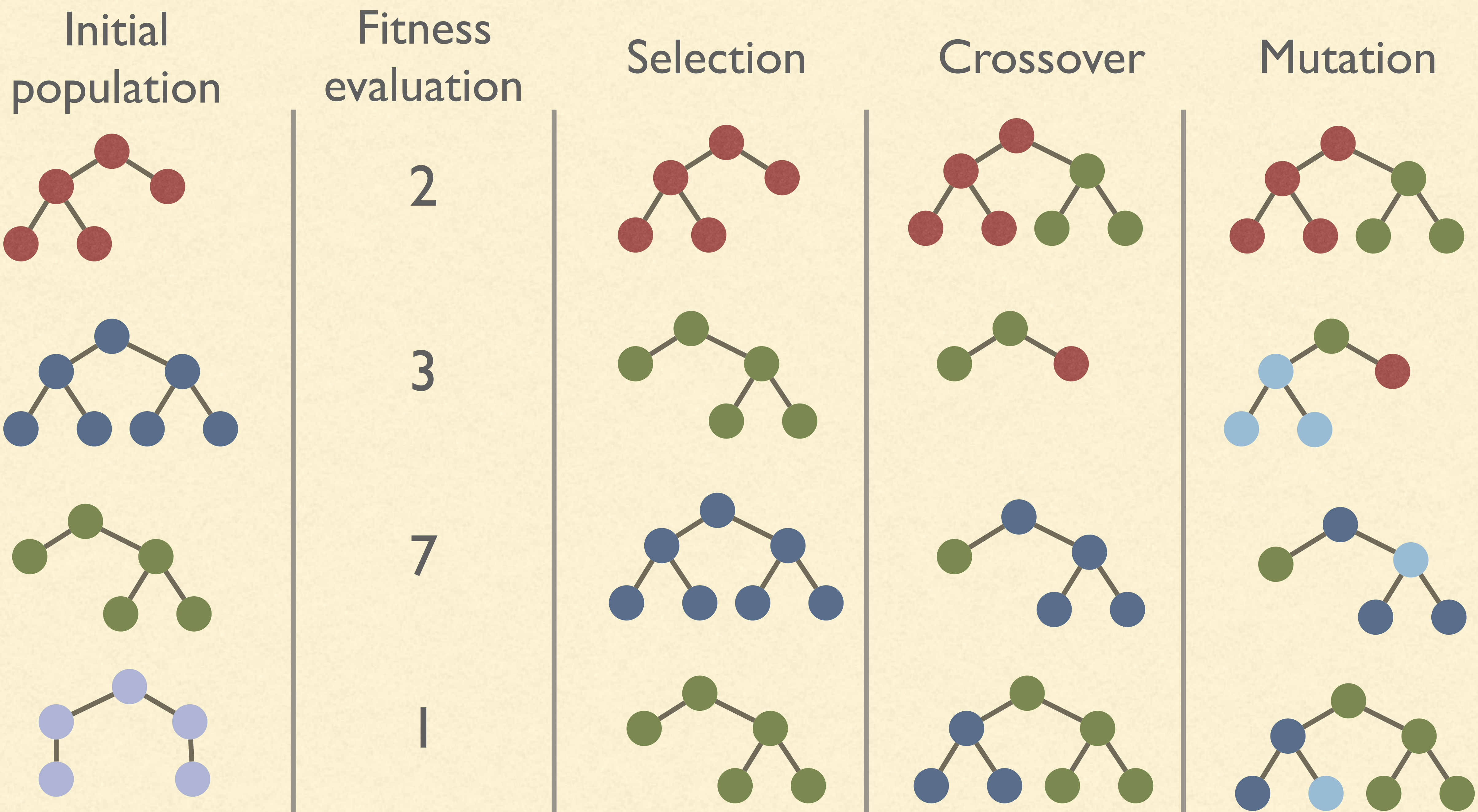


Used to reduce
program size

PERMUTATION/SWAP MUTATION

Apply a permutation to the arguments of a functional.
Swap mutation is a special case when only non-commutative binary operators have their arguments swapped

A GENERATION OF GP



AUTOMATICALLY DEFINED FUNCTIONS - ADF

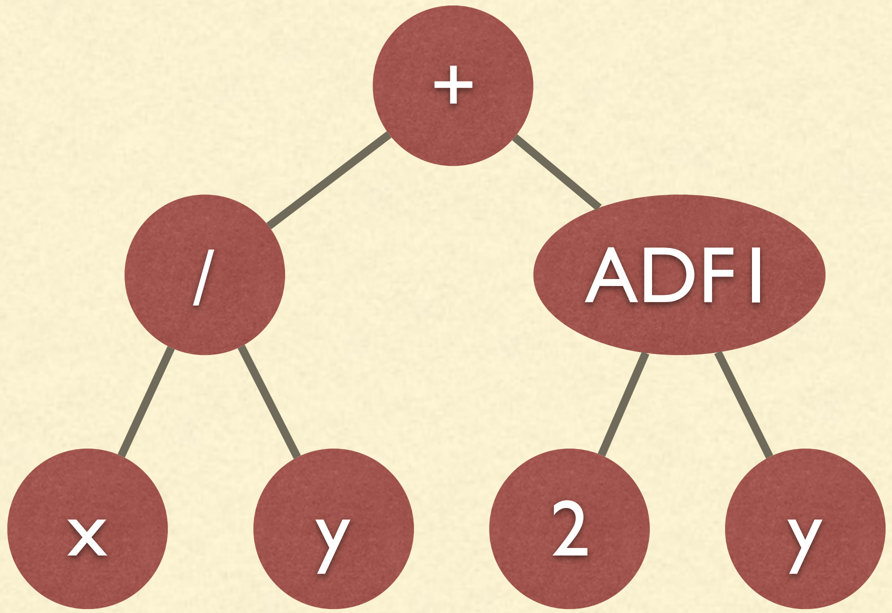
MOTIVATIONS

- One of the most useful (and older) ideas in programming is the use of *subroutines*
- However, instead of calling a function *k* times, a GP individual must evolve the same code *k* times
- Since this is unlikely we can add “subroutines” and calling subroutines to GP
- In GP those are called Automatically Defined Functions or ADF

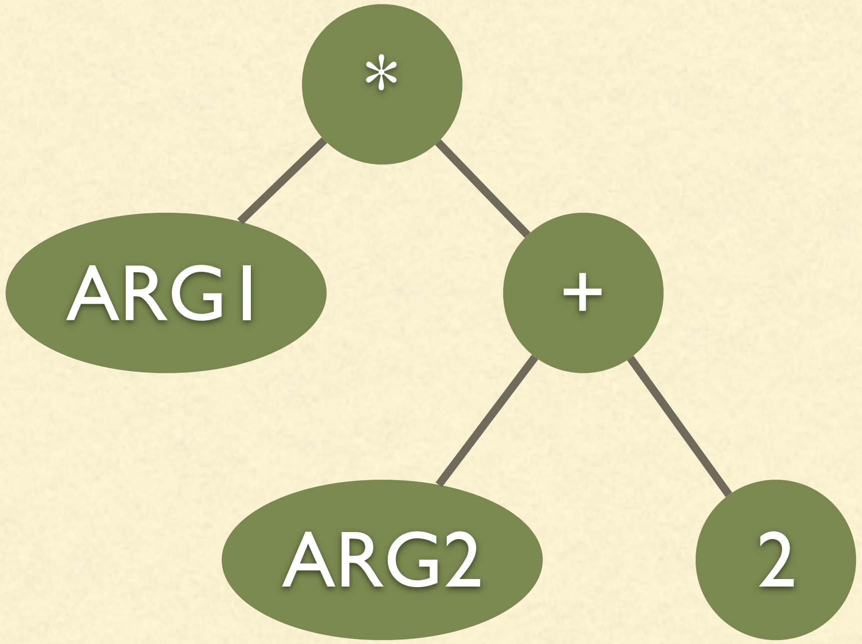
AUTOMATICALLY DEFINED FUNCTIONS

We select how many ADF we need and for each one the number of arguments that it accepts

Main Tree



ADF1 tree



ADFs are used as functionals

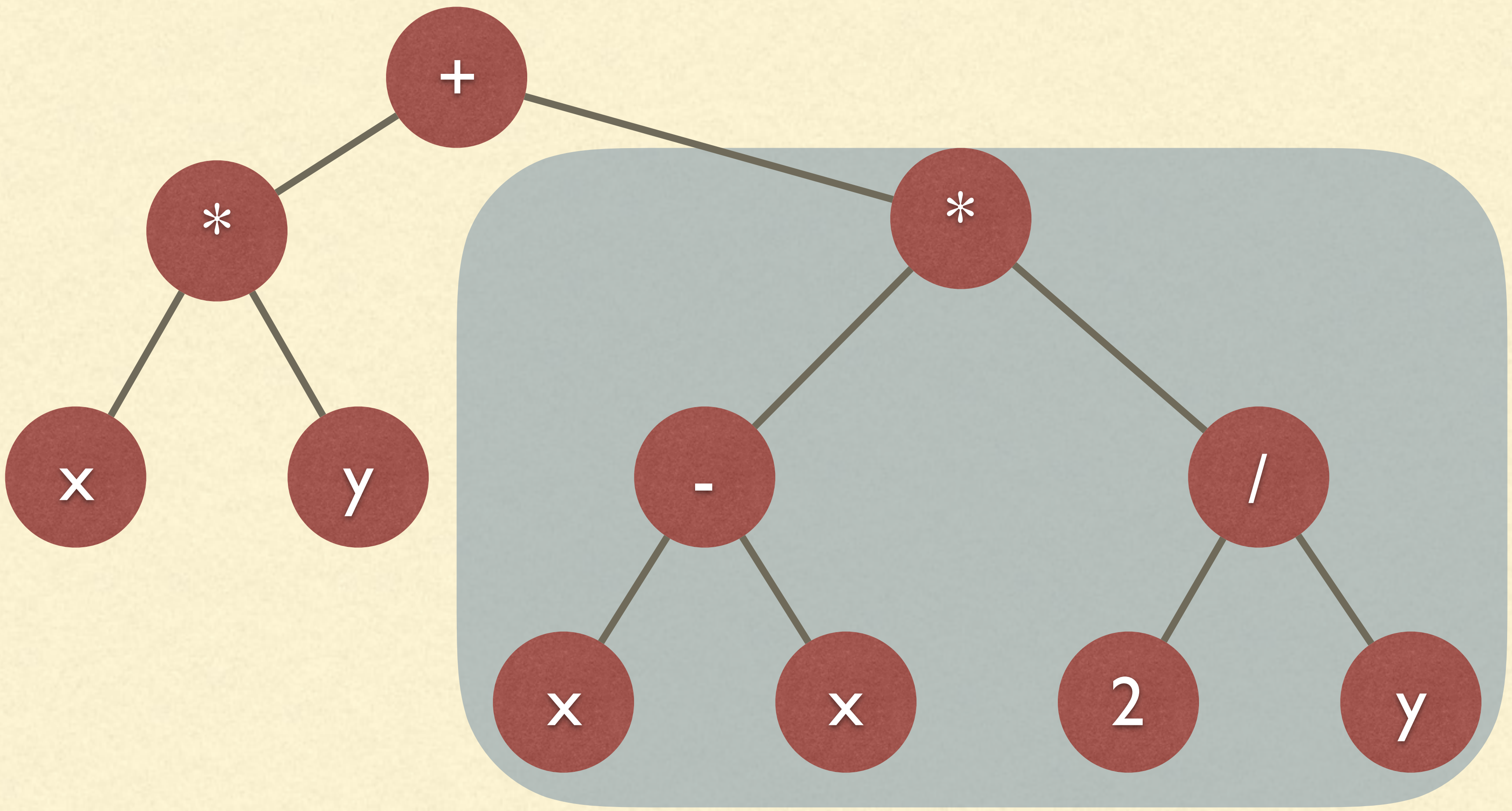
But each of them is an entire tree

AUTOMATICALLY DEFINED FUNCTIONS

- The individuals are now vectors of trees (forests)
- We can still apply the usual mutation and crossover to the elements of the vector
- We can have as many ADF as we want…
- …and ADF can call each other (nested subroutines/functions calls)
- recursion might be problematic (evolving the base case might not be easy)

BLOAT

SPOT THE PROBLEM



This entire region
is non-coding

WHAT IS BLOAT?

- Not a simple definition, lets focus on what can be observed:
- Large non-coding regions: many operations on the tree do not change the function that it represents
- Increase in the size of the trees without a noticeable increase in the fitness
- As a consequence, fitness evaluation is slower, slowing down the entire GP process

ATTACKING BLOAT

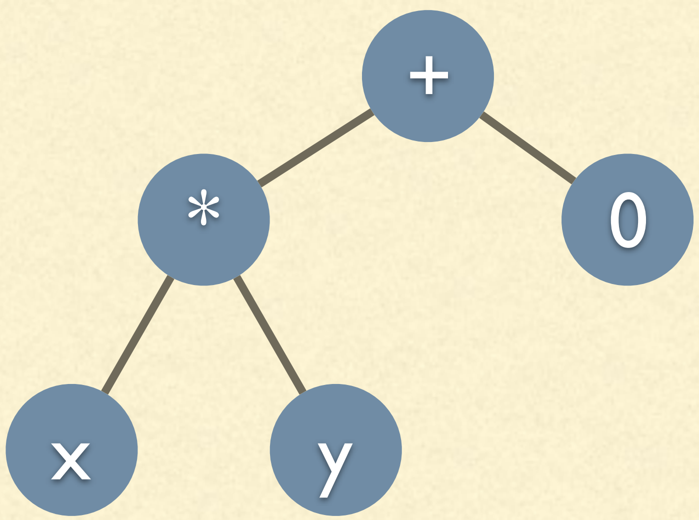
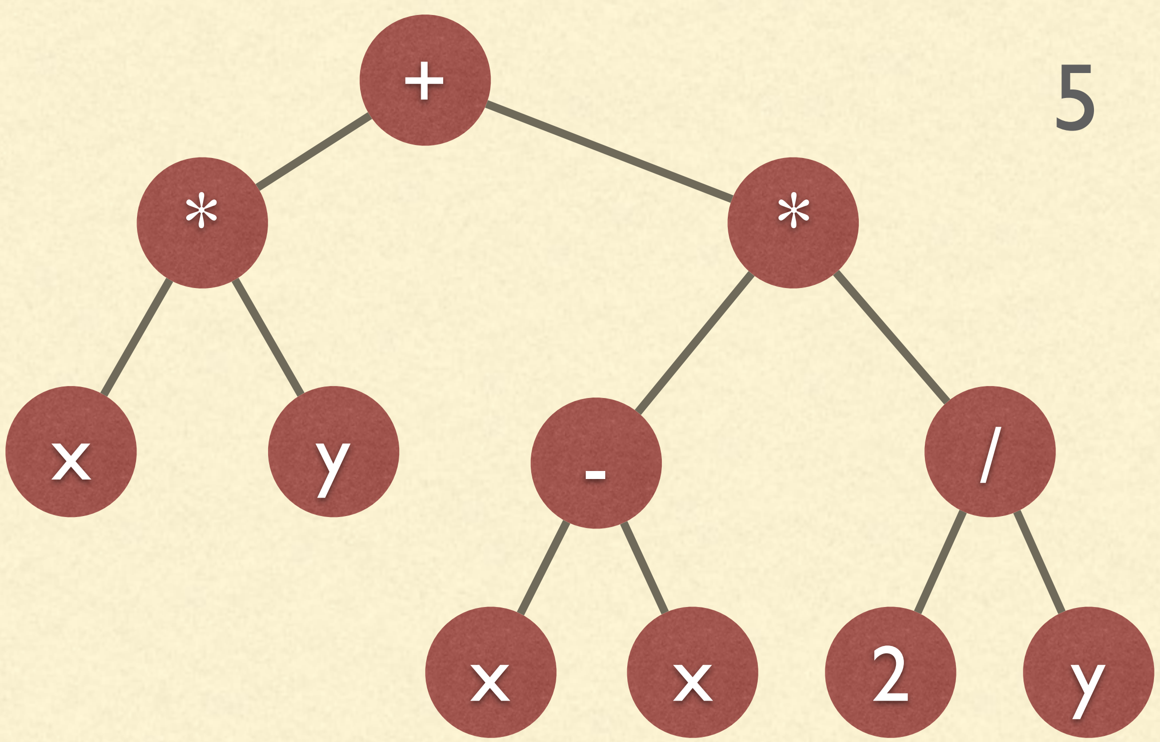
Limit individual size

- Maximum depth/size
- Operators that limit the size of the offsprings

Remove non-coding regions

Punish the individuals that are too big

LINEAR PARSIMONY PRESSURE

	Real fitness	Adjusted fitness
	5	$0.9 \times 5 - 0.1 \times 5 = 4$
	5	$0.9 \times 5 - 0.1 \times 11 = 3.4$

$$\overbrace{\alpha f}^{\text{Real fitness}} - (1 - \alpha)\underbrace{s}_{\text{Size}}$$

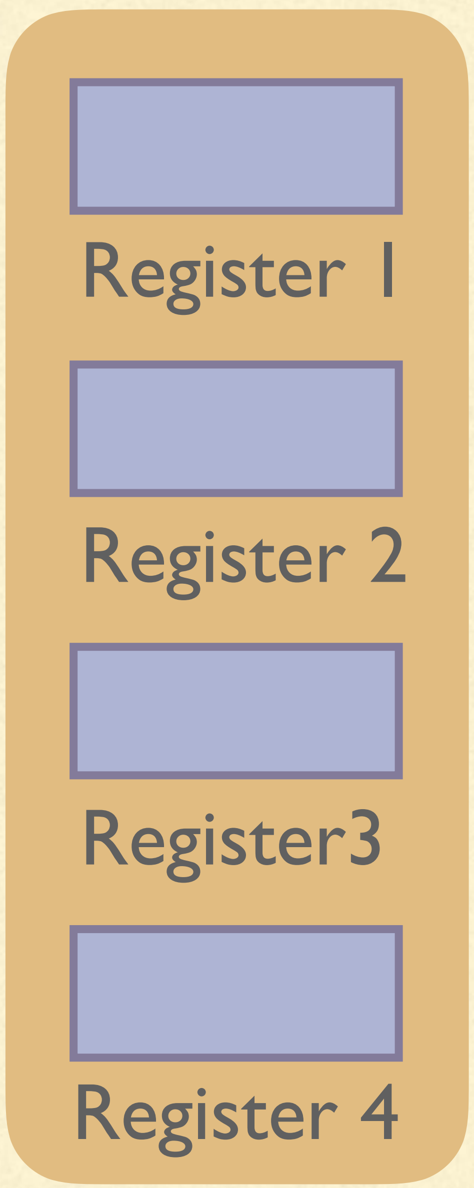
$$\alpha \in [0,1]$$

LINEAR GP

MOTIVATIONS

- Trees are not the only way of representing programs
- Also streams of instructions can represent programs
- Instead of LISP-like structure we now use assembly-like commands
- Linear GP: a linear stream of "assembly-like" instructions

AN EXAMPLE OF LINEAR GP



Registers of a virtual (or real!) machine

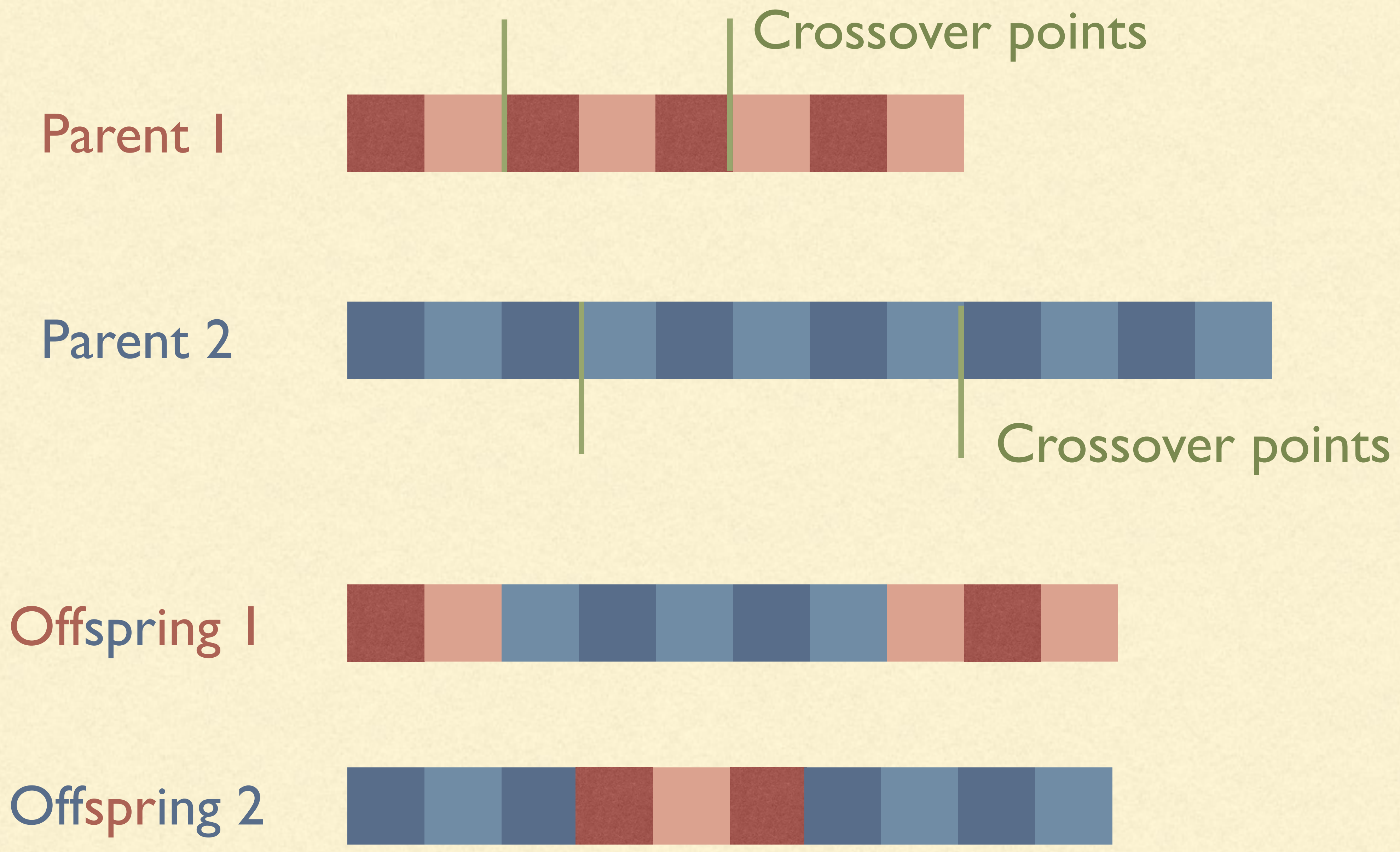
Add R1, R2,R1
Sub R3, R1,R4
Add R4,R3,R2
Mul R1,R1,R2

A Linear GP individual:
a list of instructions for the machine communicating via registers

LINEAR GP AND GA

- Linear GP seems pretty similar to standard GA
- Except that the individuals can be of non-fixed length
- An important difference is that we are evolving programs and we "execute" the individuals
- Most of the operators of GA can be used for Linear GP
- Two points crossover (with possibly different crossover points between the two individuals) is usually employed

TWO-POINTS CROSSOVER

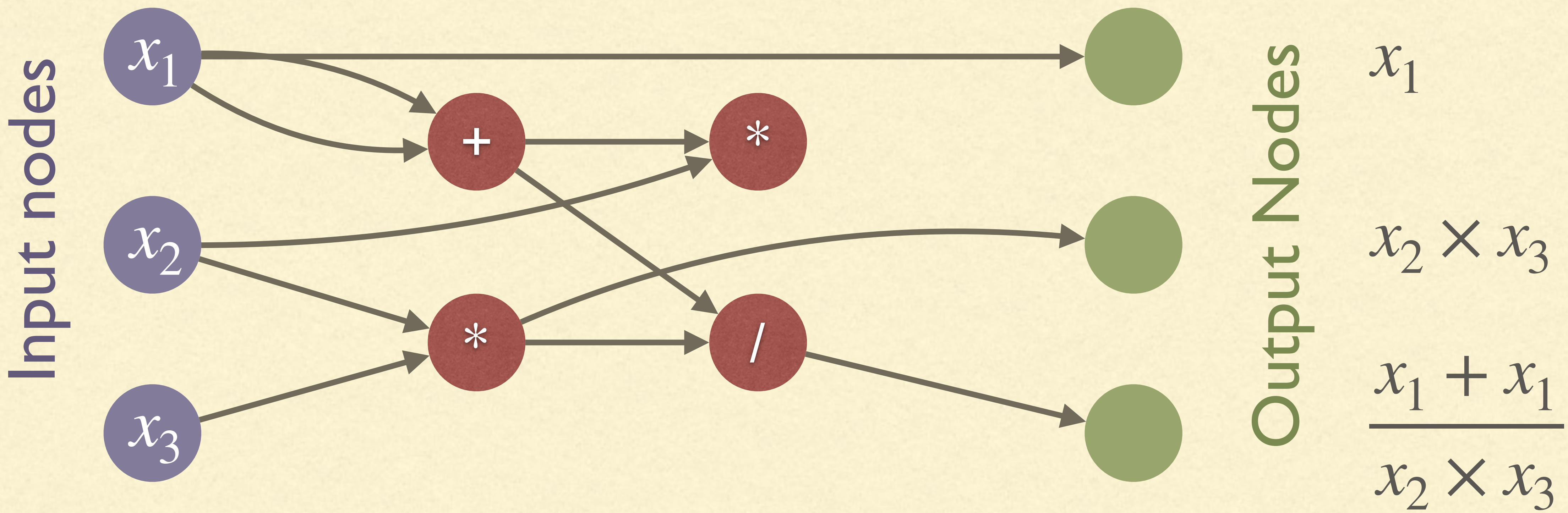


CARTESIAN GP

MOTIVATIONS

- It is possible to represent programs as circuits/graphs
- Cartesian GP represents individuals in this way
(reference person: Julian F. Miller)
- Naturally suited for problems with many inputs and many outputs (instead of using multiple trees)
- Has some tracts in common with linear GP (the representation for the circuit/graph is encoded in a linear way)

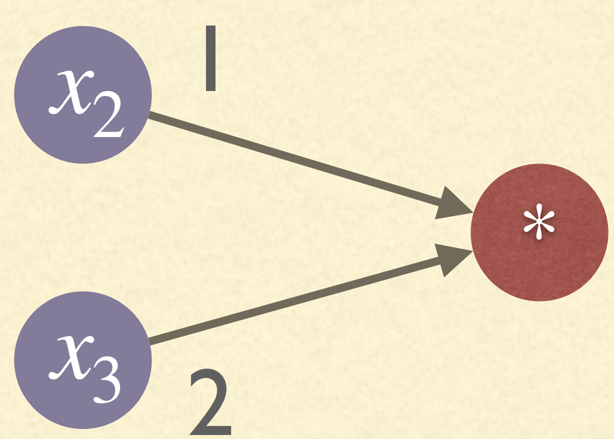
AN INDIVIDUAL OF CGP



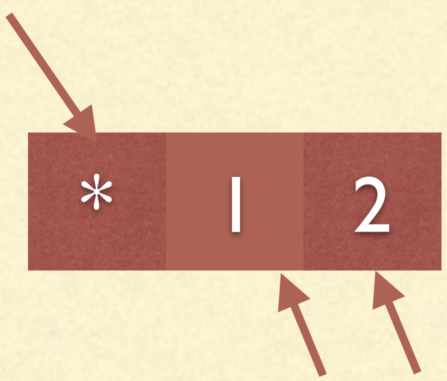
But how is the individual encoded?

ENCODING IN CGP

Internal nodes



Function

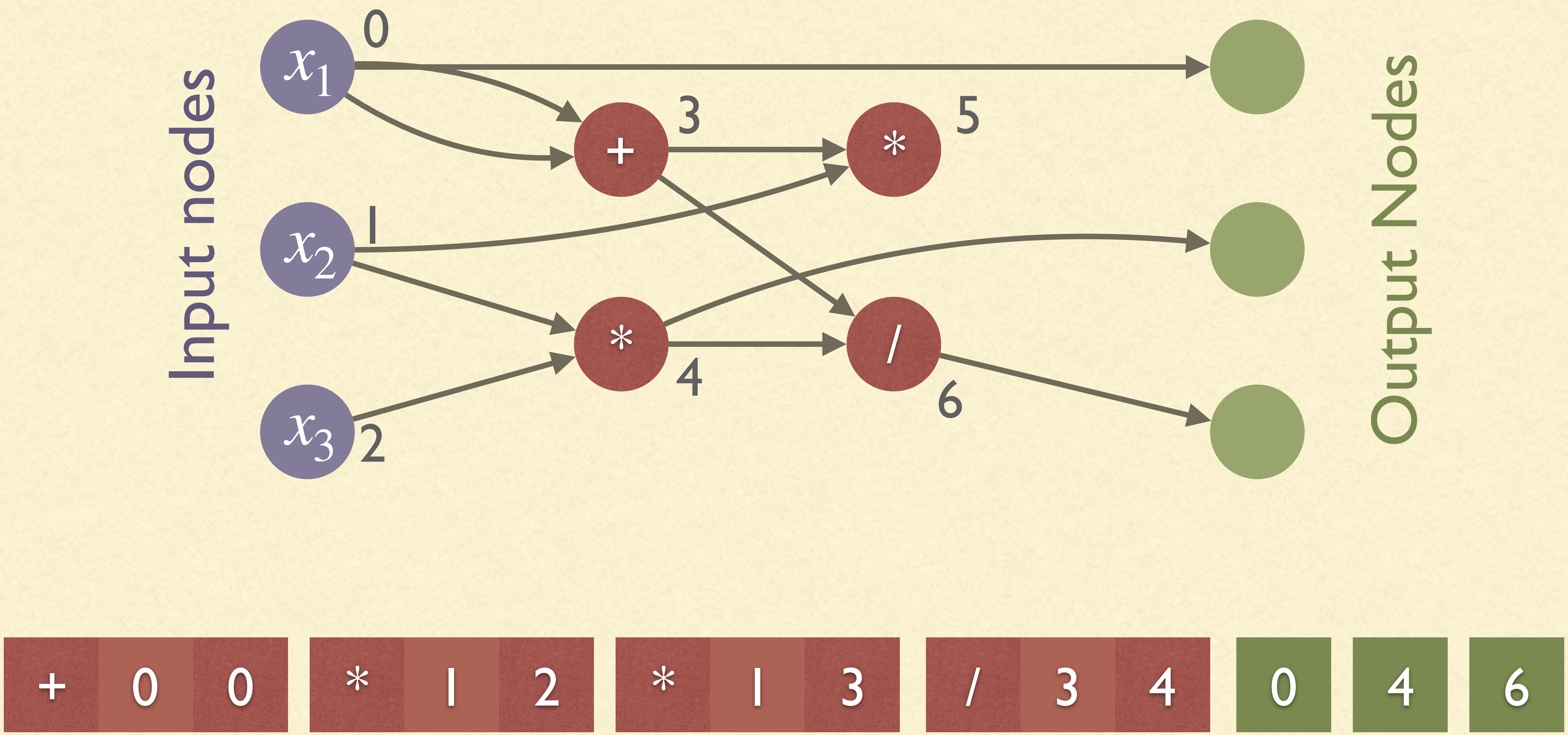


List of where
to get the inputs

Output nodes

Where to get
the output

AN INDIVIDUAL OF CGP



CGP MUTATION