# INTRODUCTION TO GENETIC PROGRAMMING

Luca Manzoni

# OUTLINE

- How programs are represented: tree-based GP
- Initialisation, Crossover, and Mutation
- Overfitting and how to combat bloat → A PARTICULAR TYPE OF OVERFITTING

# WHAT'S GP?

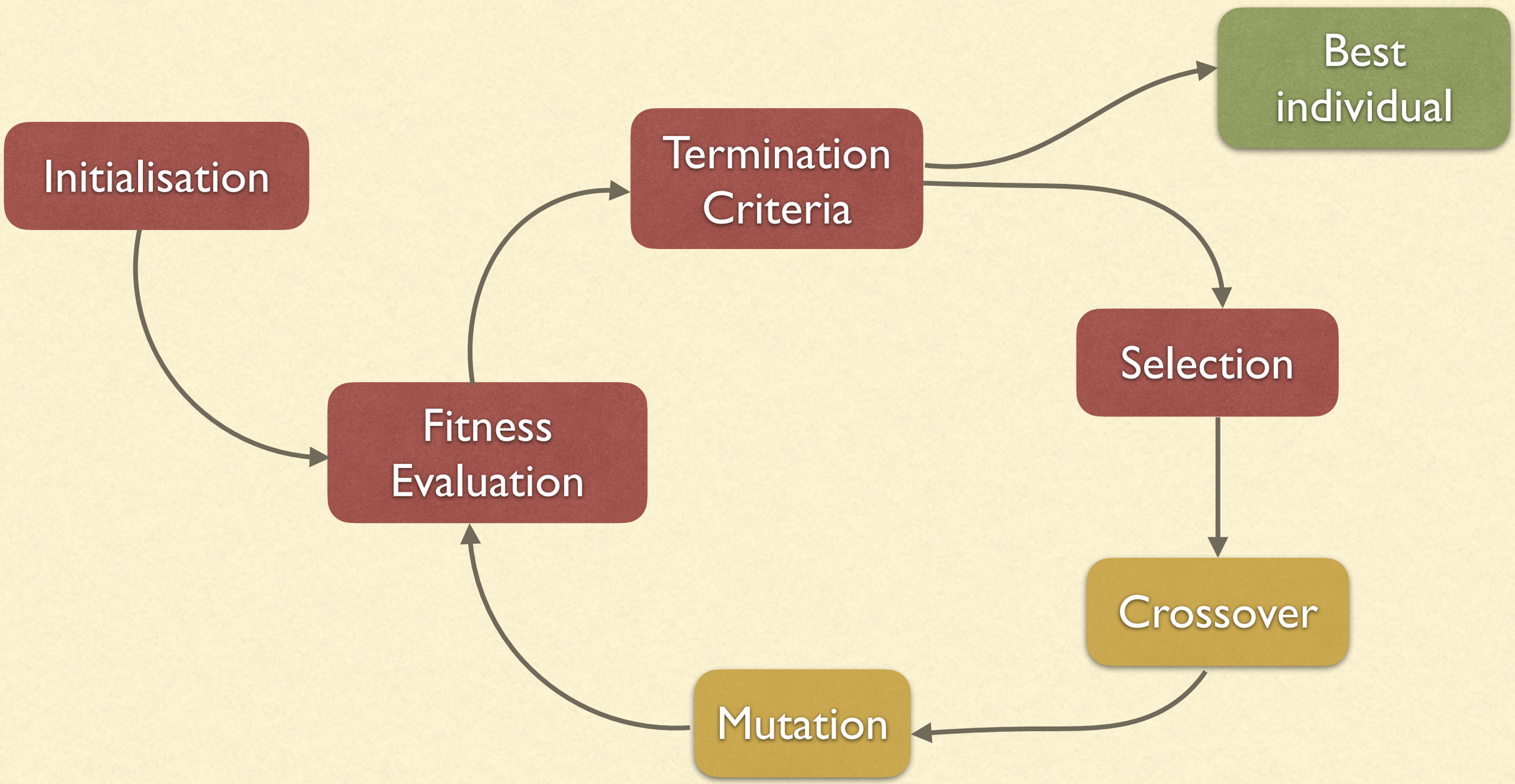
- GP is a technique to *stochastically* evolve a *population* (multiset) of individuals encoding *computer programs*

- John Koza was one of the firsts to talk about evolving computer programs in the late 80s → TIMELINE FROM GENETIC ALGORITHM TO EVOLUTION STRATEGIES AND THEN GENETIC PROGRAMMING
- While the main "evolutionary cycle" is like the one for GA…

- …GP has many peculiar properties, starting from the representation used.

# EVOLUTION CYCLE

THIS IS EXCLTLY LIKE GA, BUT...



# PROGRAM REPRESENTATION

WE PRATICALLY HAVE A STRING OF TEXT

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

—> INTRINSIC PROBLEM

Offspring 1

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

Offspring 2

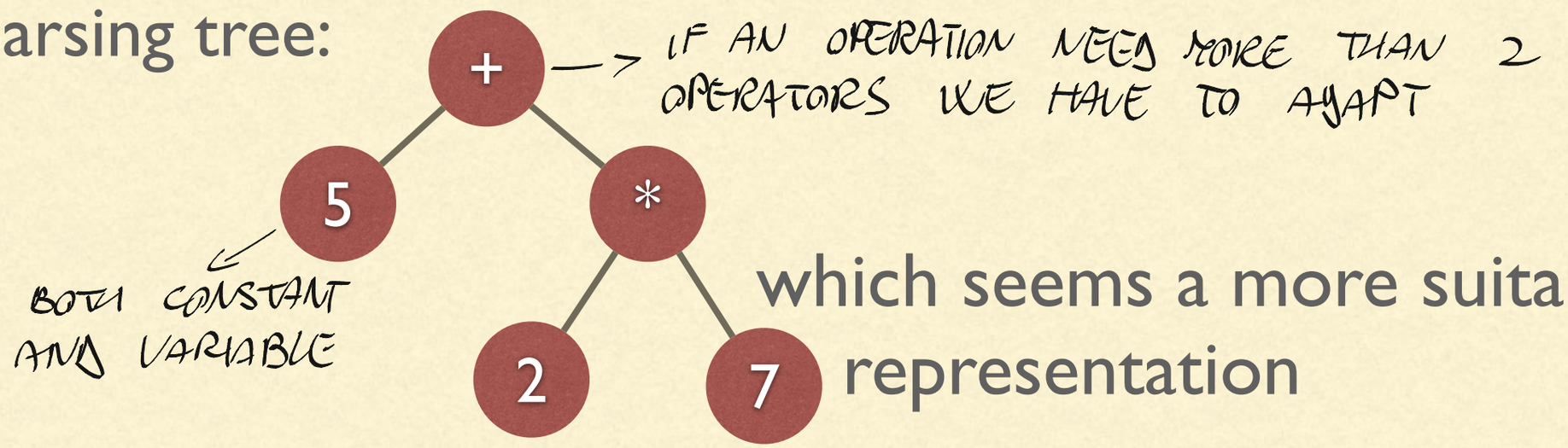
```
while (x != 4){
  do_);
}
```

WE RISK TO MAKE A VERY BIG ERROR
A STRING CAN NOT BE USED

=> WE NEED ANOTHER INTERPRETATION

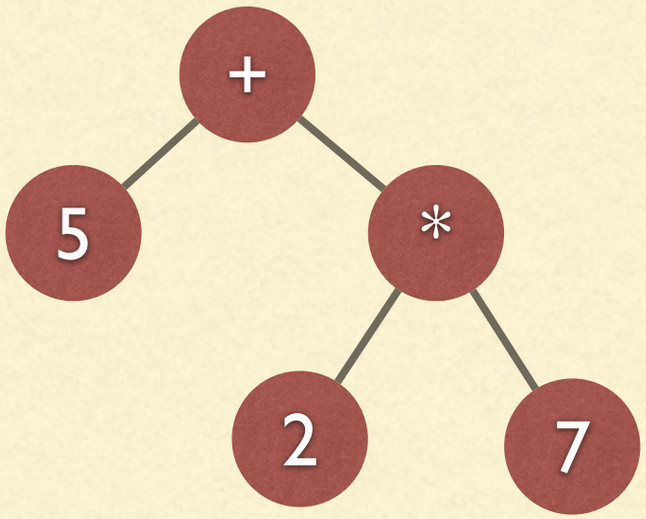
# PROGRAM REPRESENTATION

- 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

ONCE WE CHOOSE A REPRESENTATION WHAT WE ACTUALLY NEED:

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 |
| --- | --- |
| $\{0,1,2,3,4,\ldots\}$ | $\{x_0, x_1, \ldots\}$ |
| {true, false} | |
| $\{e, \pi, -1,\ldots\}$ | |

# FUNCTION SET

The function 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. => THE OUTPUT OF EVERY FUNCTION MUST BE USABLE FROM EVERY OTHER FUNCTION.

- *Type consistency*: the terminals and the output of any functional must be valid inputs to all functionals
  IT MEANS THAT IN WATHEVER PART OF THE TREE IS, IT CAN BE USED BY ANY OTHER FUNCTION
- *Evaluation safety*: primitives that can fail at runtime (e.g., division) should be "protected" to avoid runtime failures.
  EX: $N/0$ INSTEAD OF RETURN FAILURE YOU CAN RETURN A DEFAULT VALUE

# 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

WE HAVE TO MAKE PEACE WITH IT

# 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:

WICH MEANS THAT WE NEED A SPECIFIC FUNCTION

- "*Random numbers should not be generated with a method chosen at random.*" — Donald Knuth

- …also holds for randomly generating trees.

WE COULD END UP WITH ALL TREES WITH THE SAME EXPRESSION OR IN THE SAME SPACE THAT COMPUTE THE SAME FUNCTION

$(x \cdot (x \cdot (x \cdot (x \cdot x))))$

# GROW

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

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

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

NEED TO BE LARGE ENOUGH TO RAPRESENT THE GOOD SOLUTION BUT NOT TOO LARGE TO END UP WITH ENOURMOUS TREE.
BETWEEN 5 AND 20 SHOULD BE GOOD

EX: IF THE FUNCTION IS GRADE 5 WE NEED AT LEAST A DEPTH OF 3

max_depth = 2

# GROW

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

ALWAYS START WITH FUNCTIONS

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

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



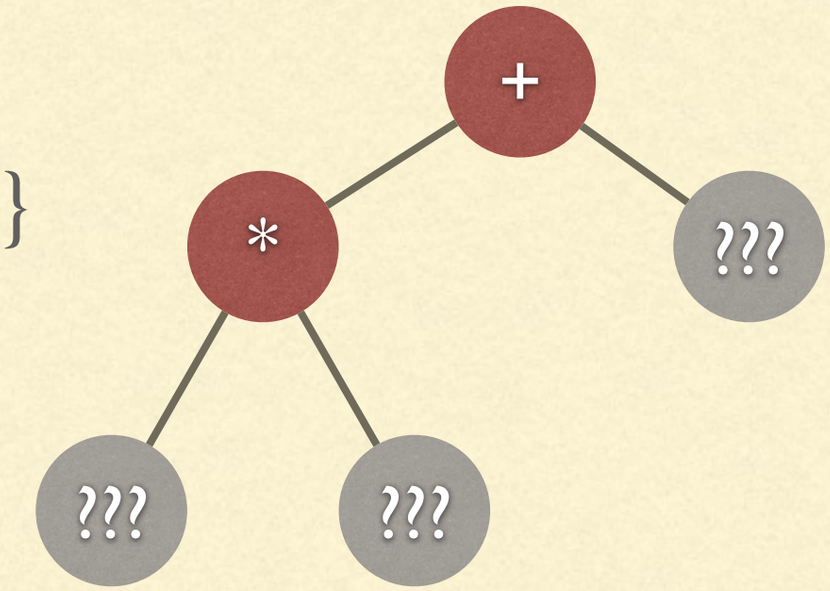

max_depth = 2

# GROW

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

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

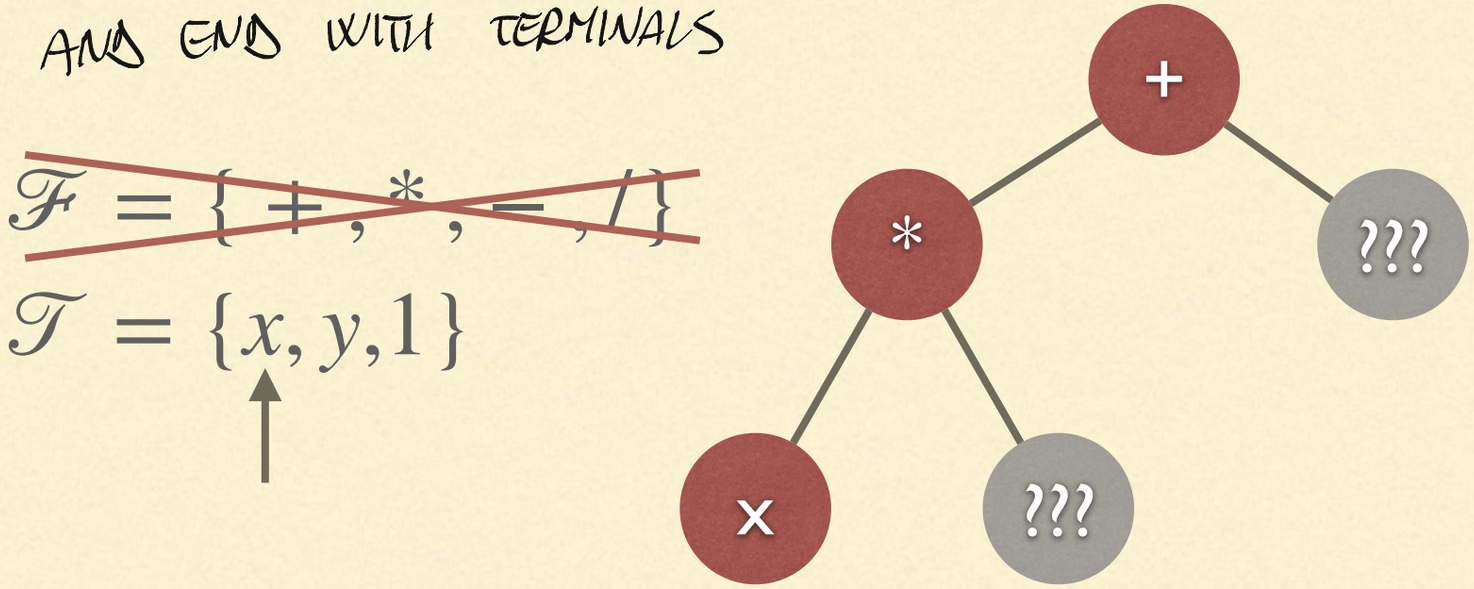
$\mathcal{T} = \{x, y, 1\}$




max_depth = 2

# GROW

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



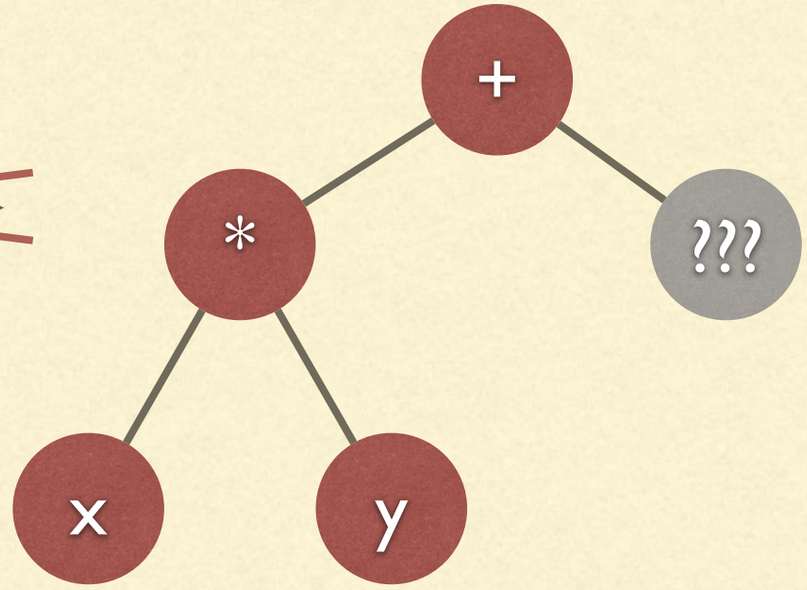

max_depth = 2

# 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

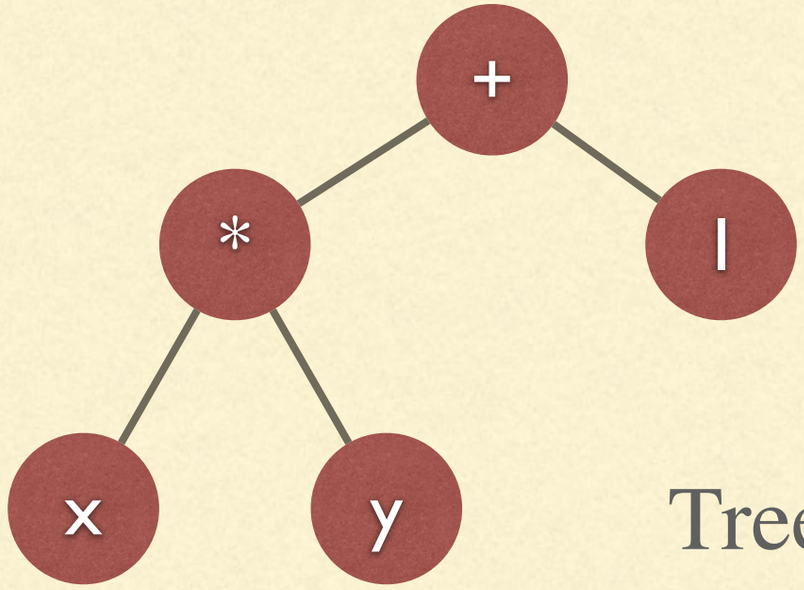
# GROW

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

THIS COULD BE A SIMPLE BUT FUNCTIONING INITIALIZATION

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

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




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)

THER ARE TREES THAT CANNOT GENERATE BY FULL THAT GROW CAN AND VICE VERSA.

- 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

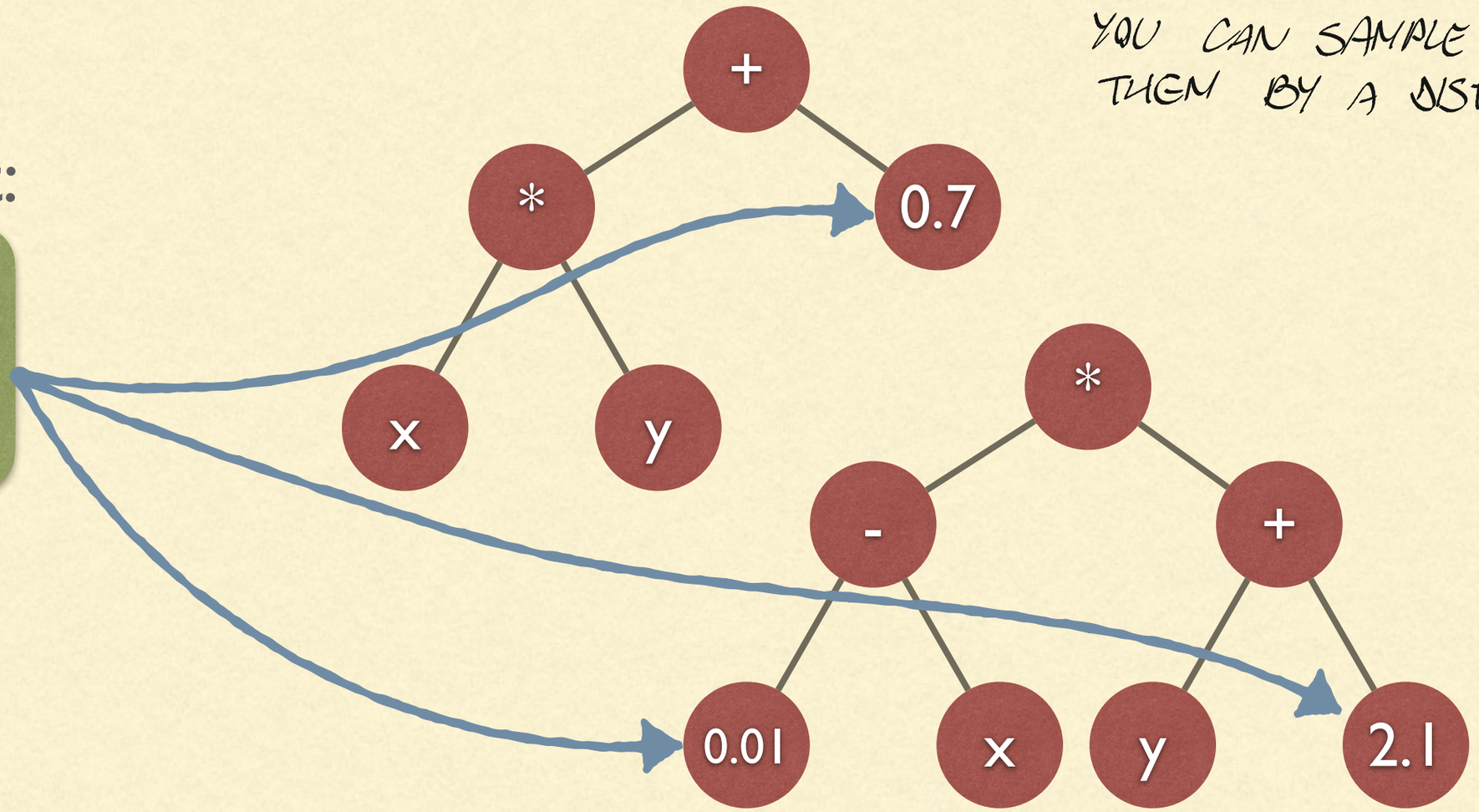
ONE ADDITIONAL STEP IN INITIALIZATION

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

In the terminal set:

A random constant in [0,3]

YOU CAN SAMPLE THEM BY A DISTRIBUTION



# CROSSOVER

# HOMOLOGOUS CROSSOVER

Lets move back to GA

$x =$ 


$x =$ 


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

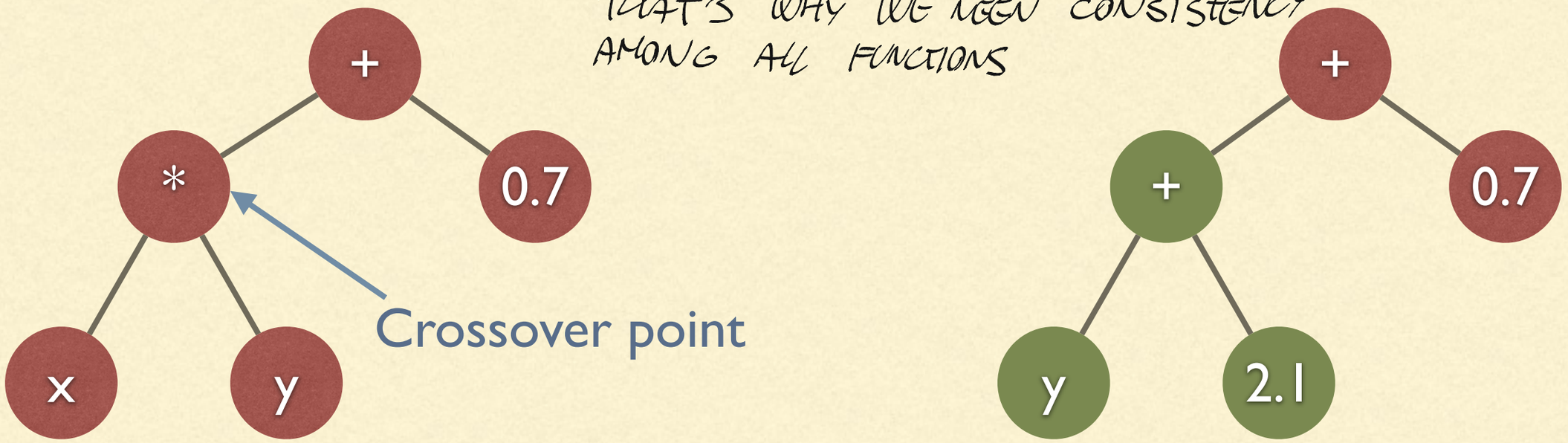
**NO**

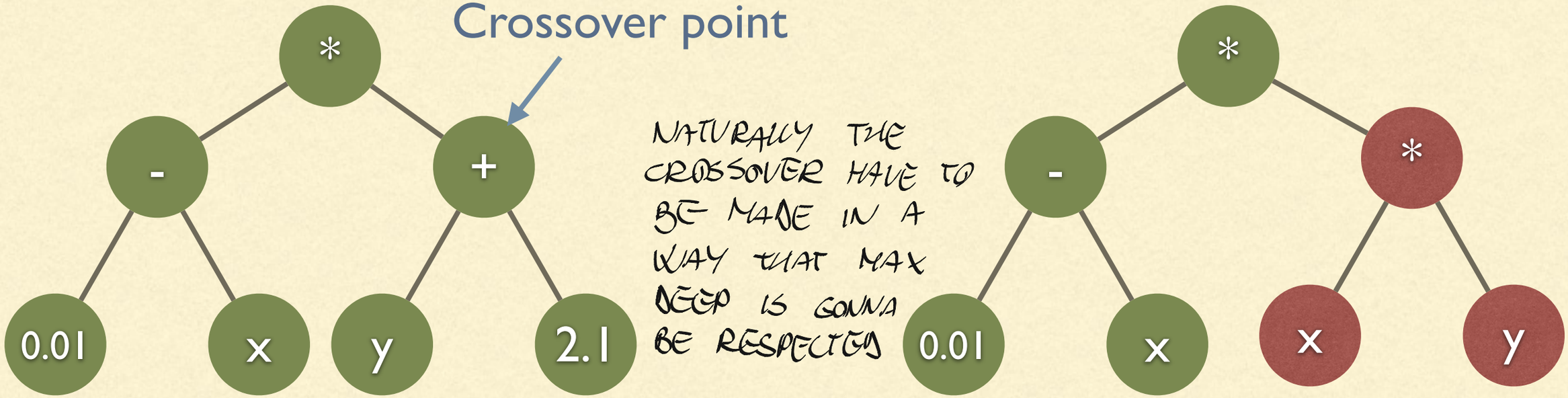
The classical GA crossovers are homologous

This is not usually the case for GP

# SUBTREE CROSSOVER

WE CAN SELECT RANDOM POINT
THAT'S WHY WE NEED CONSISTENCY
AMONG ALL FUNCTIONS



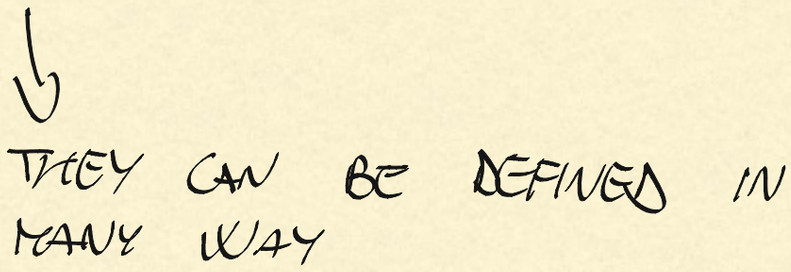



# SUBTREE CROSSOVER

- 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

WE ARE GONNA SE THEM LATER

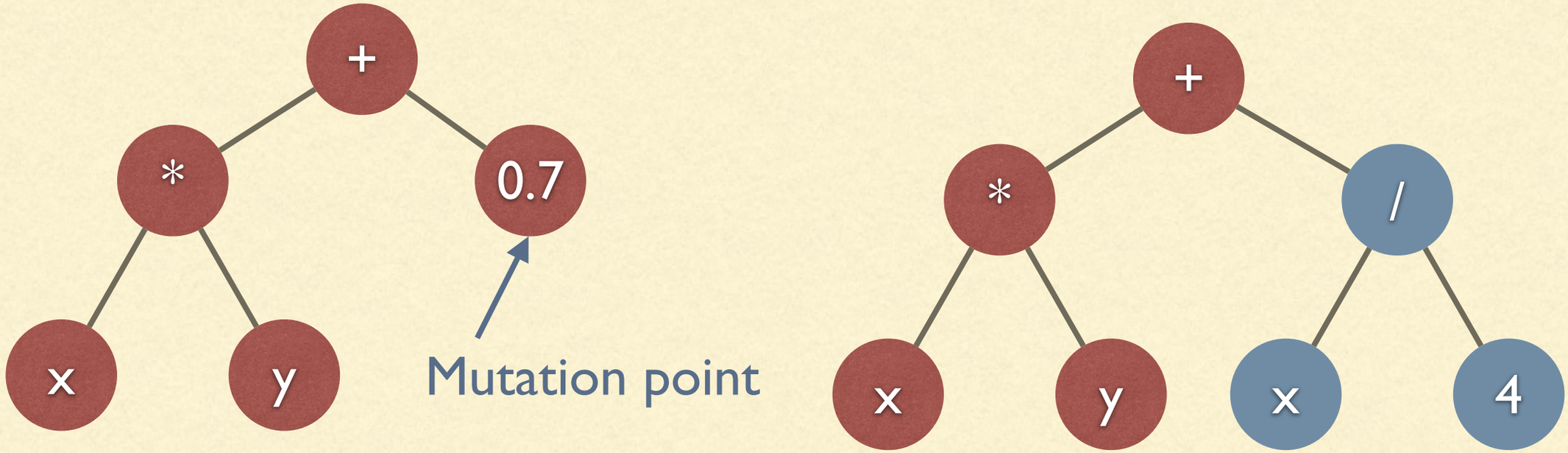
# MUTATION(S)



# SUBTREE MUTATION

**Replacement of a randomly selected subtree with a new random subtree**

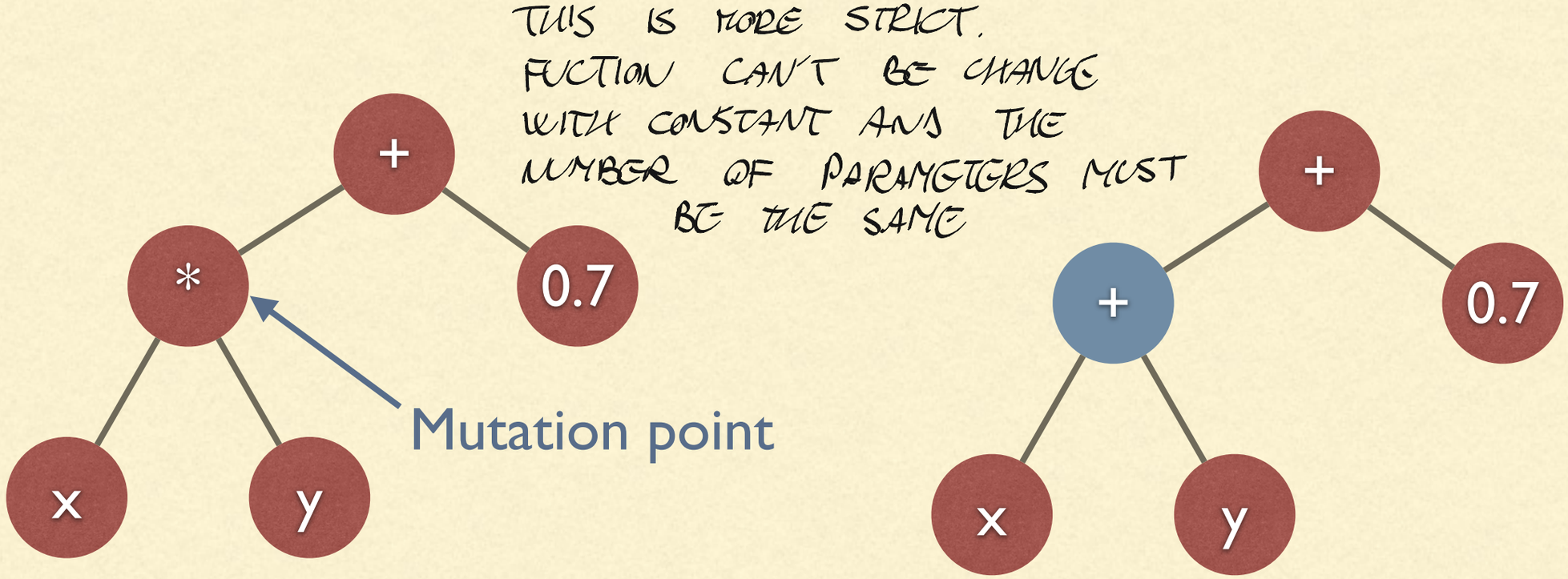
THE TREE CAN EITHER GROW OR REDUCE IN SIZE



# POINT MUTATION

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

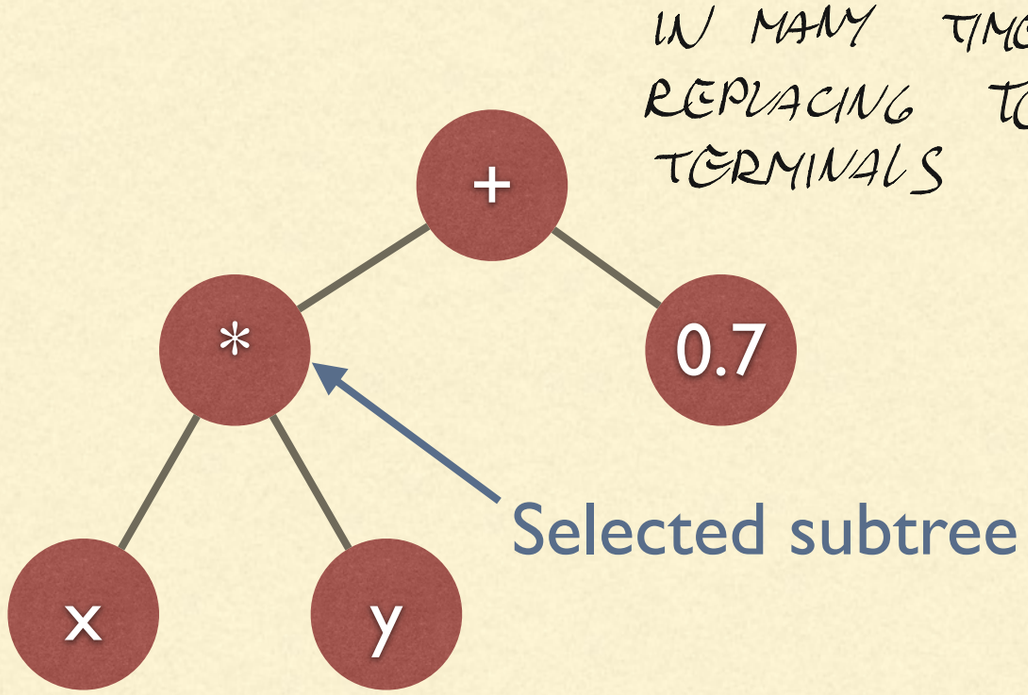
THIS IS MORE STRICT.
FUCTION CAN'T BE CHANGE
WITH CONSTANT AND THE
NUMBER OF PARAMETERS MUST
BE THE SAME

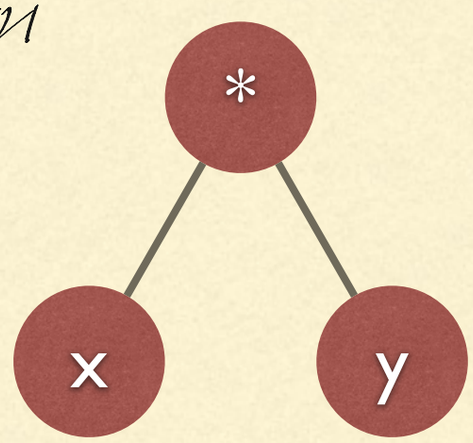


# HOIST MUTATION

Replacement of the entire tree
with one of its subtree

IN MANY TIMES YOU ARE REPLACING TERMINALS WITH TERMINALS

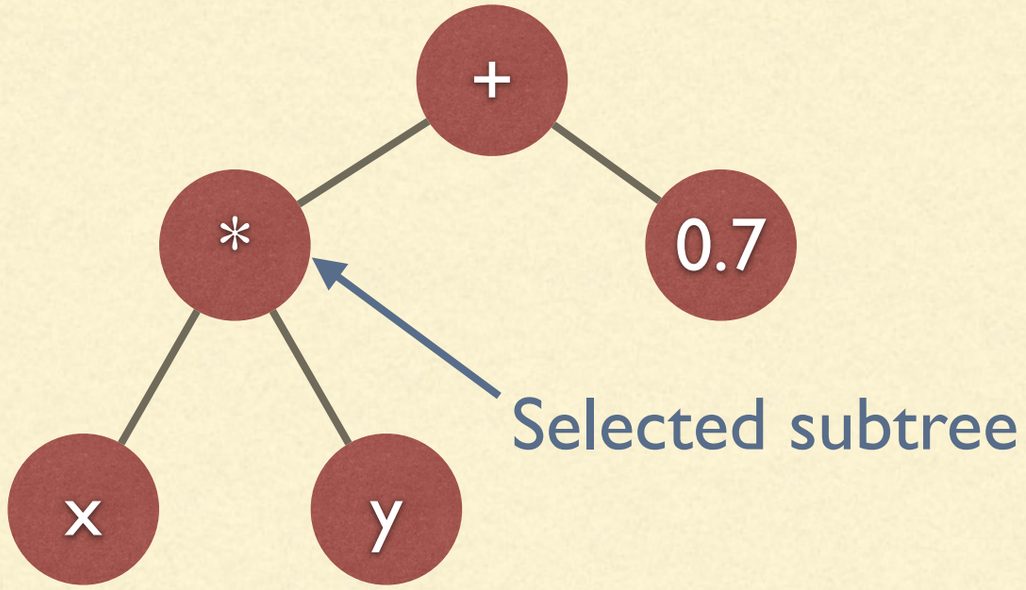


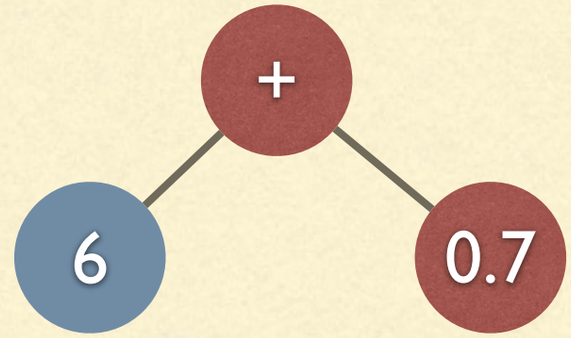




Used to reduce program size

# SHRINK MUTATION

Replacement a randomly selected subtree
with a randomly selected terminal

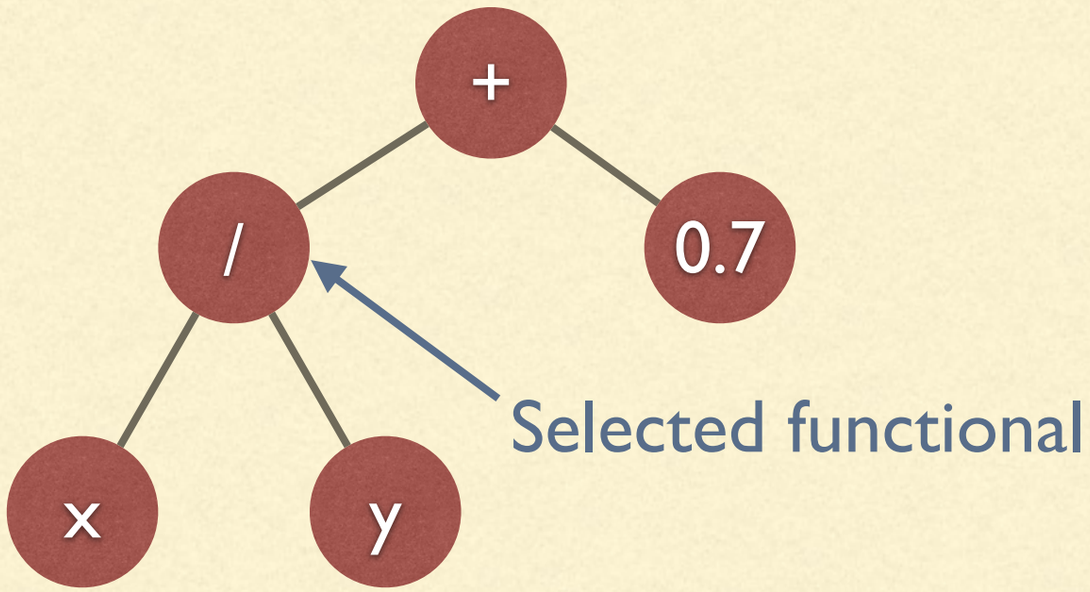


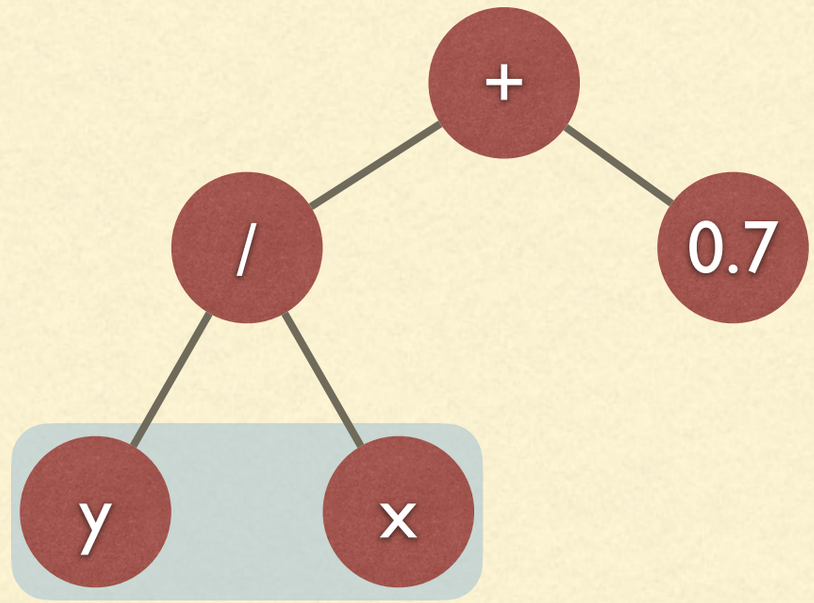




Used to reduce
program size

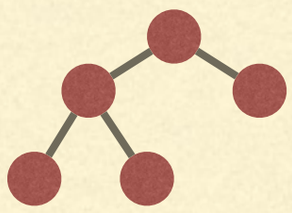
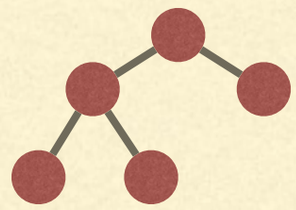
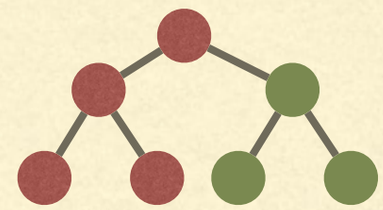
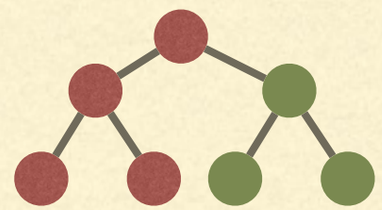
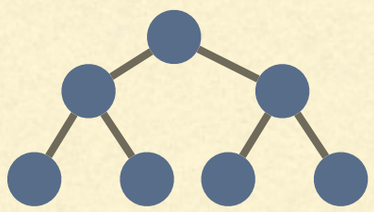
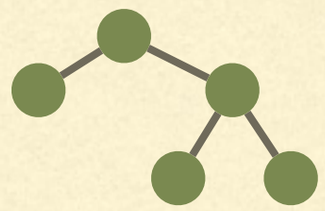
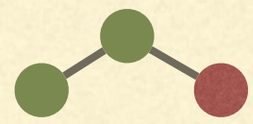
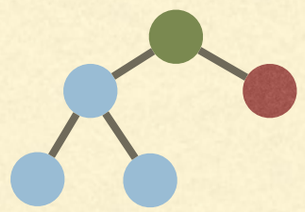
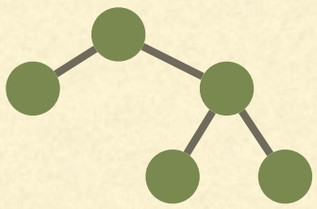
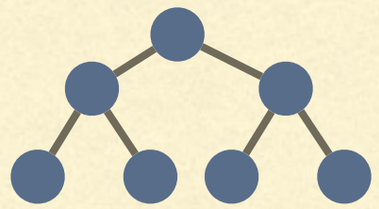
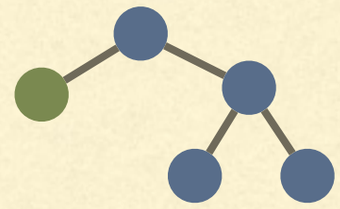
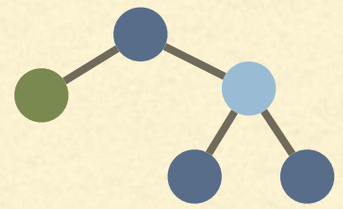
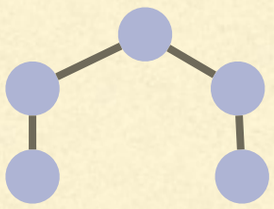
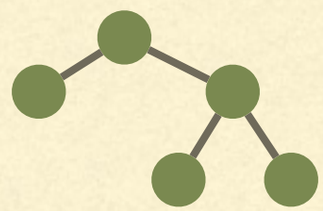
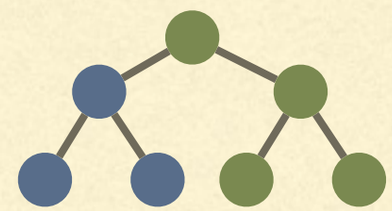
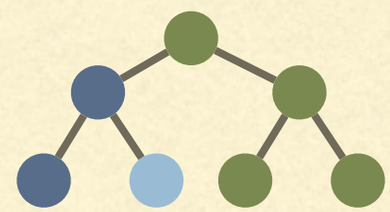
# PERMUTATION/SWAP MUTATION

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

| Initial population | Fitness evaluation | Selection | Crossover | Mutation |
|---|---|---|---|---|
|  | 2 |  |  |  |
|  | 3 |  |  |  |
|  | 7 |  |  |  |
|  | 1 |  |  |  |

# AUTOMATICALLY DEFINED FUNCTIONS - ADF

# MOTIVATIONS

- One of the most useful (and older) ideas in programming is the use of *subroutines*

→ COPY PASTE SOME PREMADE CODE

- 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

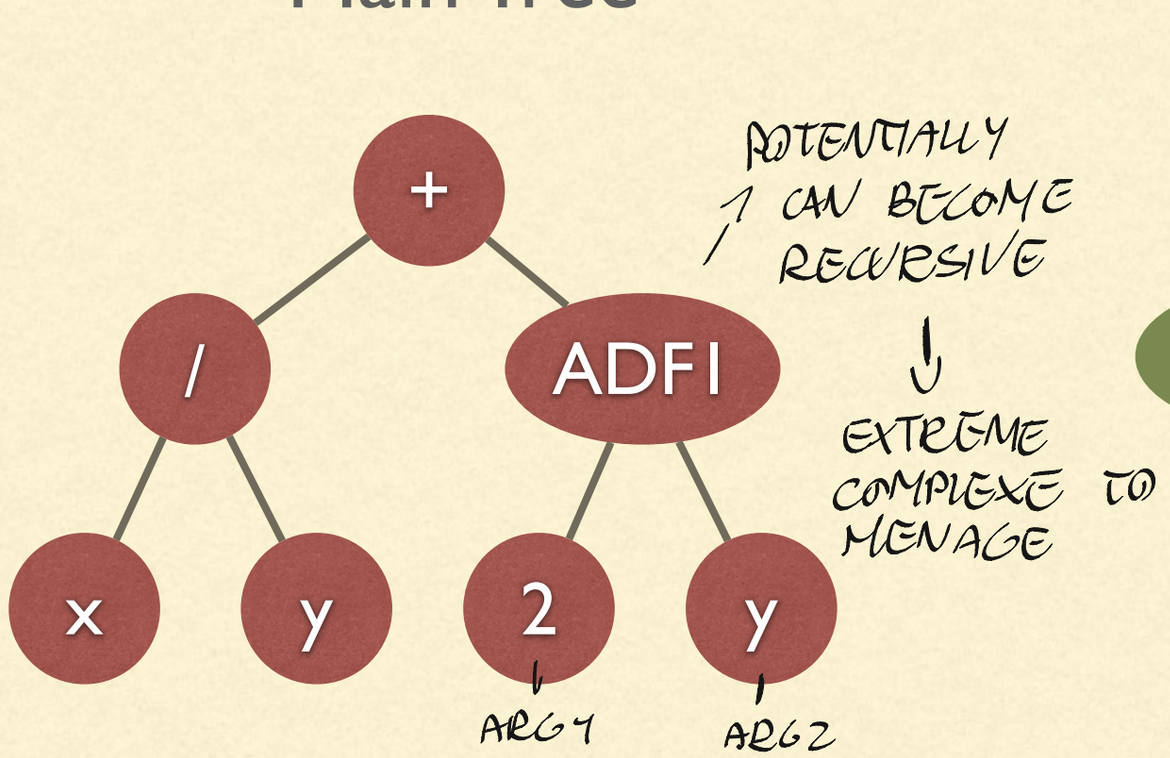
WHY? BECAUSE THEY ARE GONNA BE GENERATED AUTOMATICALLY

- In GP those are called Automatically Defined Functions or ADF

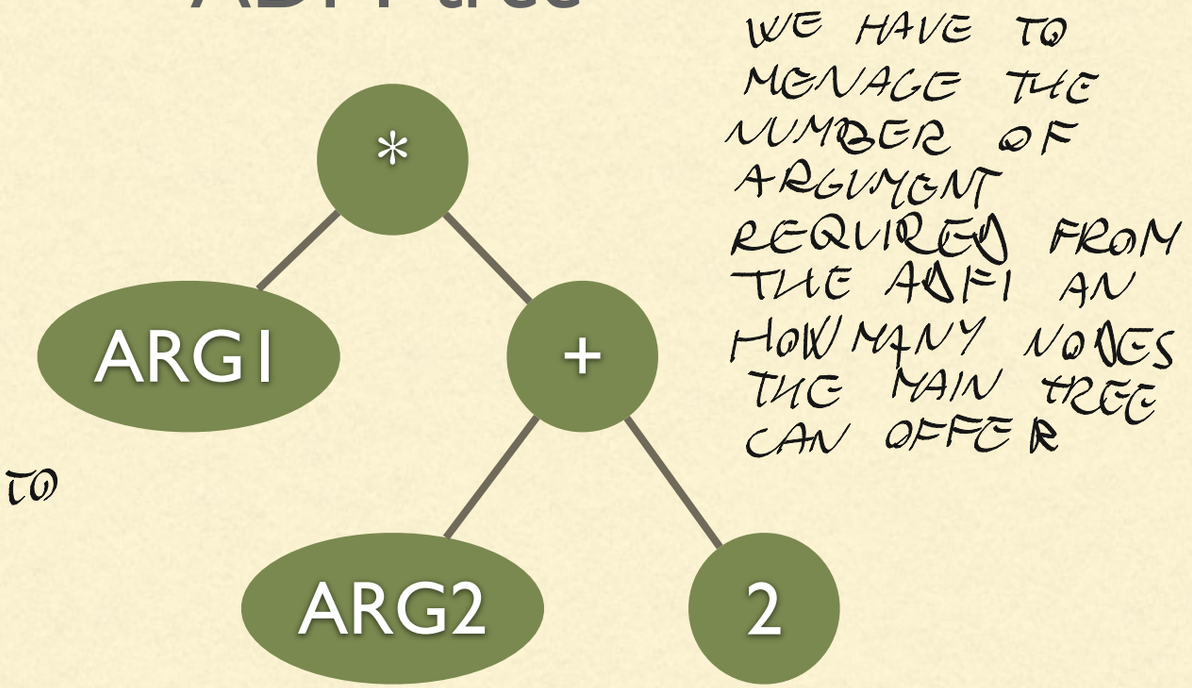
# AUTOMATICALLY DEFINED FUNCTIONS

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

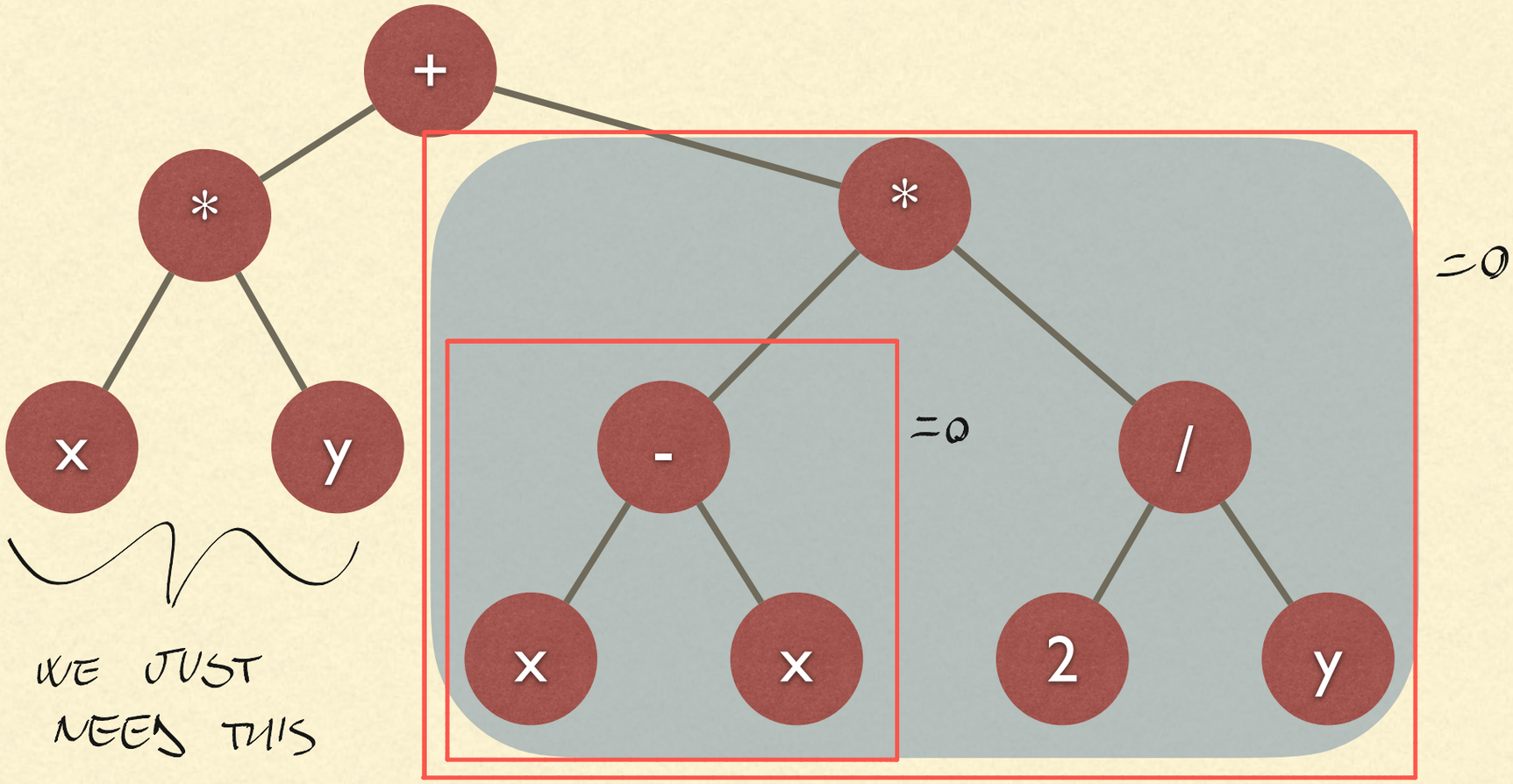
But each of them is an entire tree

# 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… UP TO A REASONABLE LIMITS
- …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?

IS A PARTICULAR FORM OF OVER FITING

- 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

  WE WANT THE SMALLEST TREE WITH THE BEST RESULT
- As a consequence, fitness evaluation is slower, slowing down the entire GP process

HOW CAN WE AVOID OR IDENTIFY THE PROBLEM?

# ATTACKING BLOAT

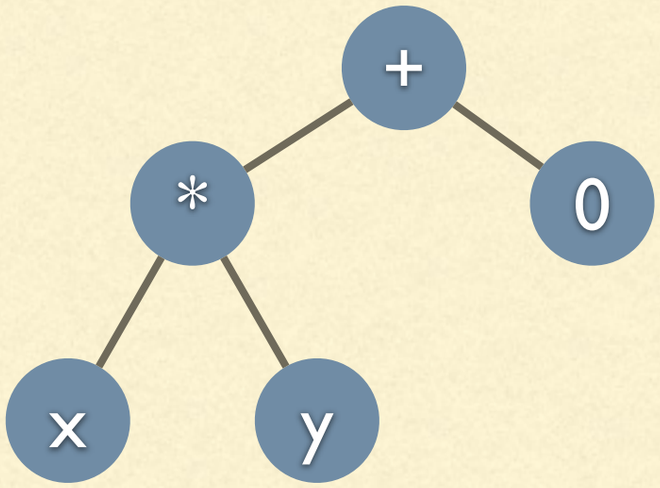
Limit individual size

- 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

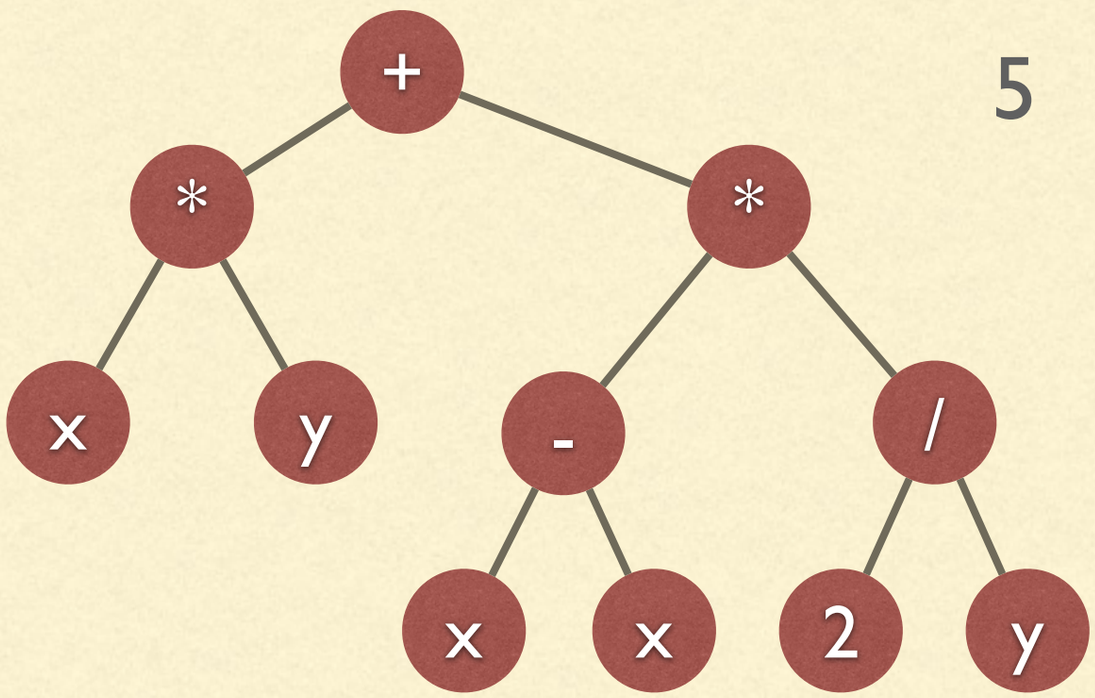



Real fitness

5

Adjusted fitness

$$0.9 \times 5 - 0.1 \times 5 = 4$$




5

$$0.9 \times 5 - 0.1 \times 11 = 3.4$$

Real fitness

$$\alpha f - (1 - \alpha) s$$

Size

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