

# Genetic Algorithms and Genetic Programming

*Pavia University and INFN*  
*Second Lecture*

Eric W. Vaandering

ewv@fnal.gov

Vanderbilt University

# Course Outline

- Machine learning techniques
- Genetic Algorithms
  - Review
- Genetic Programming
  - Basic concepts and differences from Genetic Algorithms
  - Genetic operators for Genetic Programming
  - Example
  - Generating constants
  - Applications

# Review of Genetic Algorithms

Let's recall the key points of what we learned last time:

- Genotype to phenotype matching
  - (Binary) string codes to model in determined way
- Subsequent generation created from previous generation according to biological models
  - Cloning
  - Crossover (sexual recombination)
  - Mutation
- Each individual has a fitness
- Which individuals reproduce determined by fitness relative to others
- All choices are made randomly

# Genetic Programming

Almost all of what we learned with Genetic Algorithms can be applied to genetic programs. But, instead of evolving a genotype (a binary string) which maps to a phenotype (a predetermined model) we will directly evolve the phenotype (a computer program). This program will (we hope) solve the problem we pose. Remember our supposition from last time:

Since we will use computer programs to implement our solutions, maybe the *form* of our solution should *be* a computer program.

This has the advantage of not specifying, in advance, the form our solution will take.

# Genetic Programming

“How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?”

— Attributed to Arthur Samuel, 1959  
(Pioneer of Artificial Intelligence,  
coined term “machine learning”)

“Genetic programming *is* automatic programming. For the first time since the idea of automatic programming was first discussed in the late 40’s and early 50’s, we have a set of non-trivial, non-tailored, computer-generated programs that satisfy Samuel’s exhortation: ‘Tell the computer what to do, not how to do it.’ ”

— John Holland, University of Michigan, 1997  
(Pioneer of Genetic Algorithms)

# Programming Assumptions

When we normally program, we assume a number of guidelines:

- **Correctness:** The solution works perfectly
- **Consistency:** The problem has one preferred solution
- **Justifiability:** It is apparent why the solution works
- **Certainty:** A solution exists
- **Orderliness:** The solution proceeds in a orderly way
- **Brevity:** Every part of the solution is necessary, a shorter solution is better (Occam's Razor)
- **Decisiveness:** We know when the solution is complete

Genetic Programming *requires* that *all* of these assumptions be discarded

# GP principles

In fact, in Genetic Programming:

- **Correctness:** A solution may be “good enough”
- **Consistency:** Many very different solutions may be found
- **Justifiability:** It may be very unclear how or why a solution works
- **Certainty:** A perfect solution may never be found
- **Orderliness:** A solution may be very disorganized
- **Brevity:** Large parts of the solution may do nothing
- **Decisiveness:** We may never know if the best solution has been found

# GP Representations

In Genetic programming, instead of a binary string, we build, mutate, and cross-over *programs*. Internally, this is often (at least originally) done in LISP because these types of operations are easy in LISP's representation (everything is a function).

Let's look at a simple function:

C code:	LISP Code:
<pre>float myfunc(float x, float y) {     float val;     if (x &gt; y) {         val = x*x + y;     } else {         val = y*y + x;     }     return val; }</pre>	<pre>(IF (&gt; x y)     (+ (* x x) y)     (+ (* y y) x) )</pre>

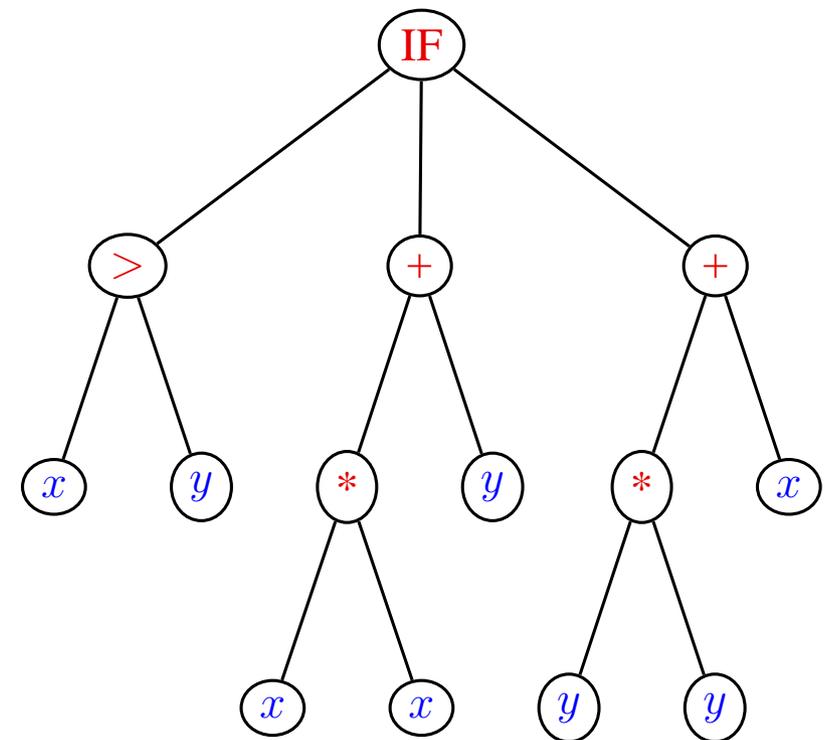
# Tree Representation

GP operations are even easier to illustrate if we adopt a “Tree” representation of a program. In this representation our example becomes:

C code:

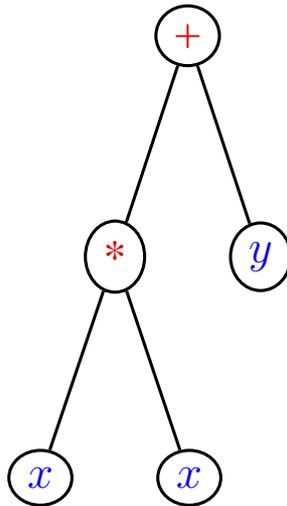
```
float myfunc(float x, float y) {  
    float val;  
    if (x > y) {  
        val = x*x + y;  
    } else {  
        val = y*y + x;  
    }  
    return val;  
}
```

Program tree



# Tree Representation, cont.

From a fraction of our tree, we can see a few things:



Two kinds of “nodes”

- There are functions (IF,  $>$ ,  $+$ ,  $*$ )
- There are “terminals” ( $x$ ,  $y$ )
- A function can have any number of arguments (IF has three,  $\sin x$  has one)

If we allow *any* function or terminal at any position, then all operations must be allowed:

- IF (float)
- $x + (y > x)$
- Divide by zero (if we use division)

# Preparatory Steps

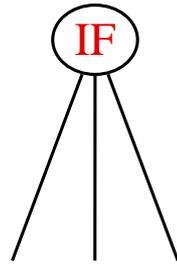
In the Genetic Algorithm, we began by defining a representation. (A binary string that mapped to the “physical” model).

In Genetic programming, we do something a little different. We choose a set of terminals (variables, for instance) and a set of functions (which take these variables as parameters). These “atoms” or “nodes” will form the basis of our programs (solutions).

As in the Genetic Algorithm, we have to define the fitness of a program for solving the problem.

# Building a tree

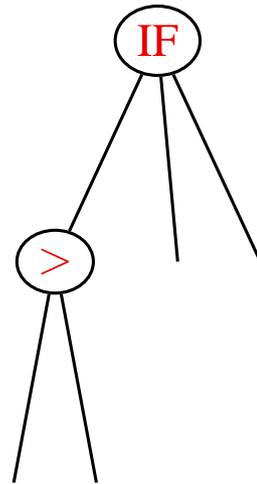
Trees are randomly built up one node at a time.



Root node 'IF' has 3 args.

# Building a tree

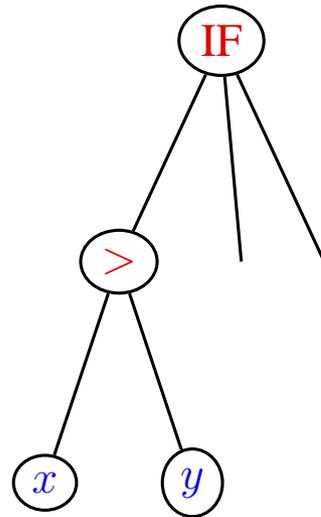
Trees are randomly built up one node at a time.



Root node 'IF' has 3 args.  
'>' chosen for 1st arg.

# Building a tree

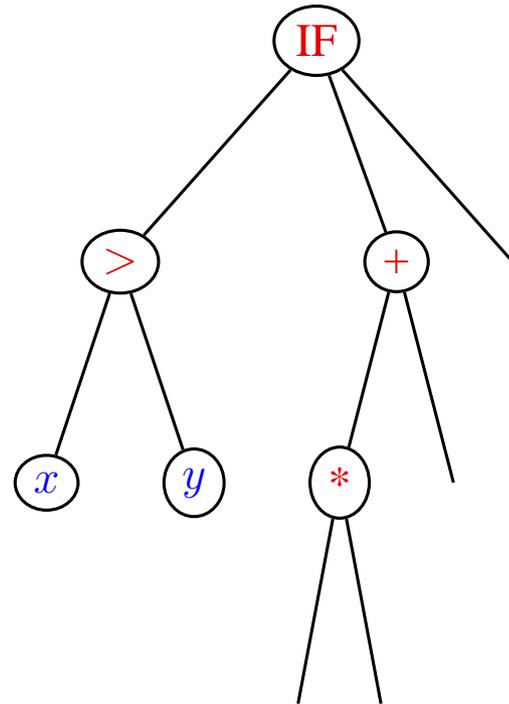
Trees are randomly built up one node at a time.



Root node 'IF' has 3 args.  
'>' chosen for 1st arg.  
*x* and *y* terminate '>'

# Building a tree

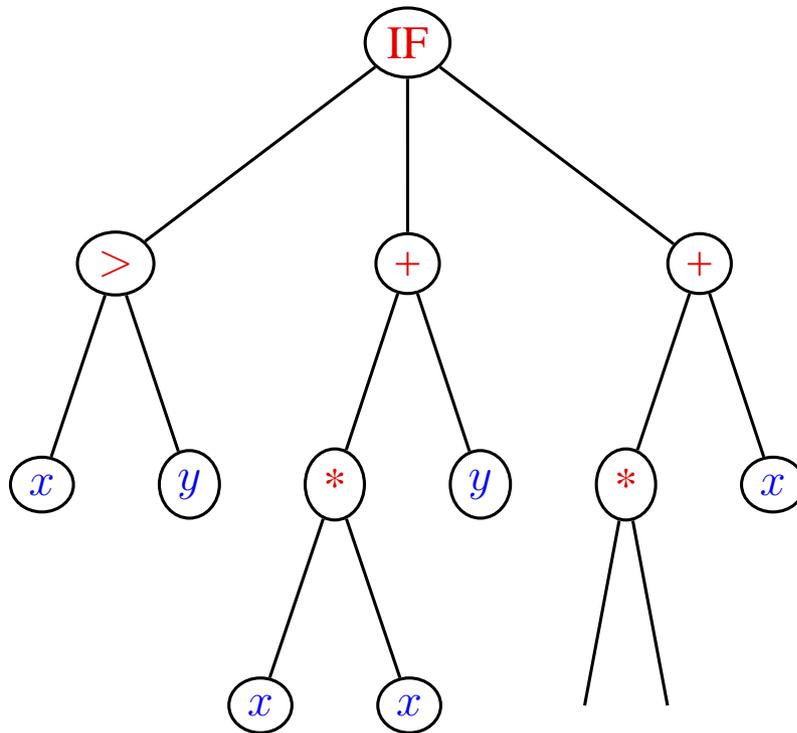
Trees are randomly built up one node at a time.



Root node 'IF' has 3 args.  
'>' chosen for 1st arg.  
 $x$  and  $y$  terminate '>'  
Next branch started

# Building a tree

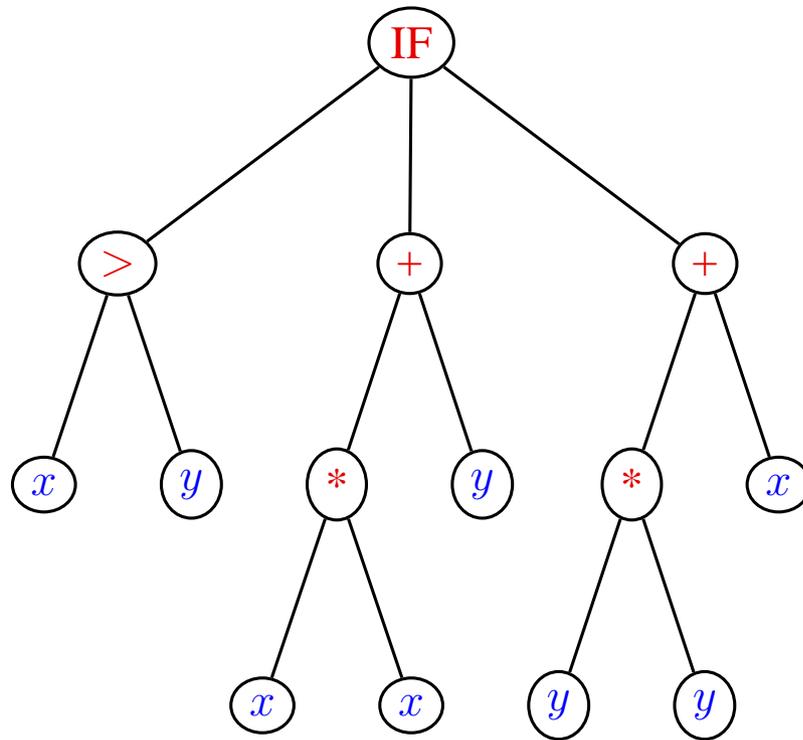
Trees are randomly built up one node at a time.



Root node 'IF' has 3 args.  
'>' chosen for 1st arg.  
 $x$  and  $y$  terminate '>'  
Next branch started  
Final branch started

# Building a tree

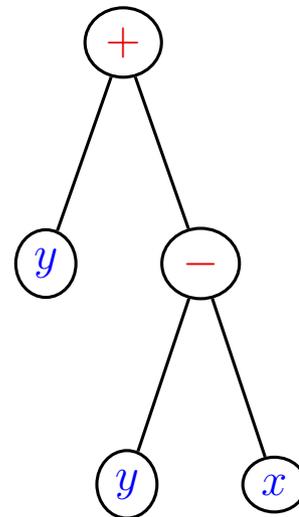
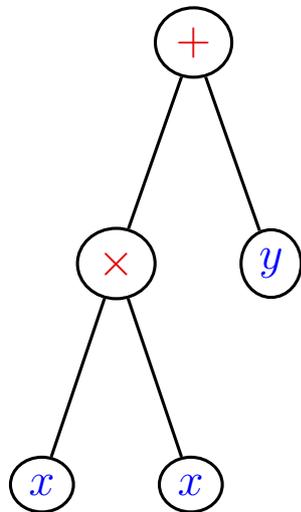
Trees are randomly built up one node at a time.



Root node 'IF' has 3 args.  
'>' chosen for 1st arg.  
 $x$  and  $y$  terminate '>'  
Next branch started  
Final branch started  
Tree is complete  
(all branches terminated)

# Crossover

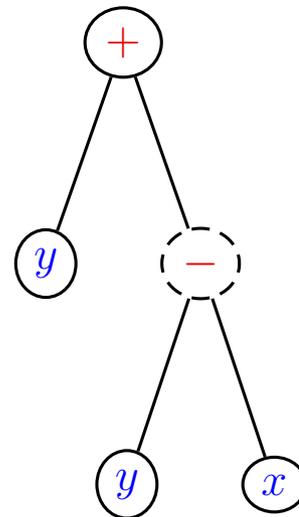
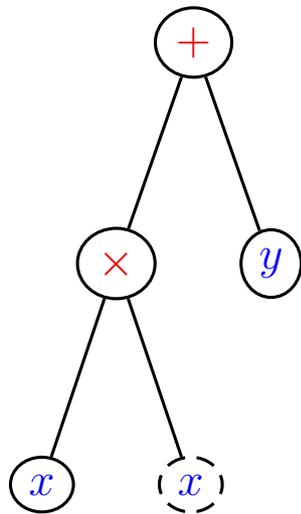
Recall in GA, crossover was a swap of DNA at a chosen point. Here's how we do crossover in GP:



Pick two parents

# Crossover

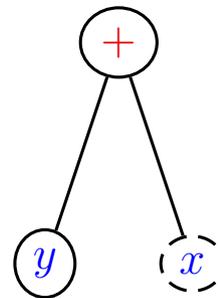
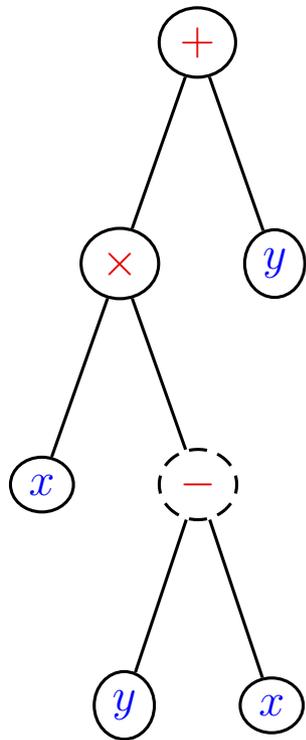
Recall in GA, crossover was a swap of DNA at a chosen point.  
Here's how we do crossover in GP:



Pick two parents  
Pick break point on each

# Crossover

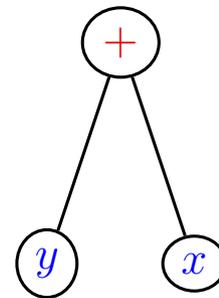
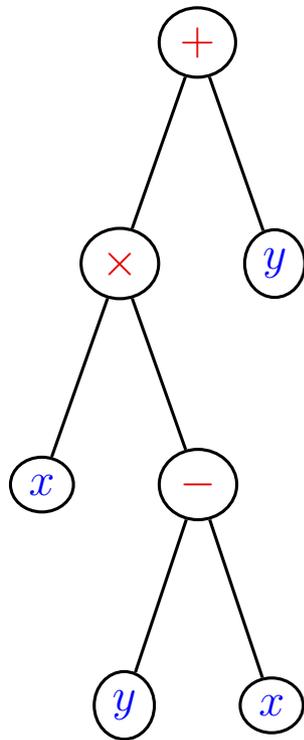
Recall in GA, crossover was a swap of DNA at a chosen point.  
Here's how we do crossover in GP:



Pick two parents  
Pick break point on each  
Swap sub-trees

# Crossover

Recall in GA, crossover was a swap of DNA at a chosen point. Here's how we do crossover in GP:

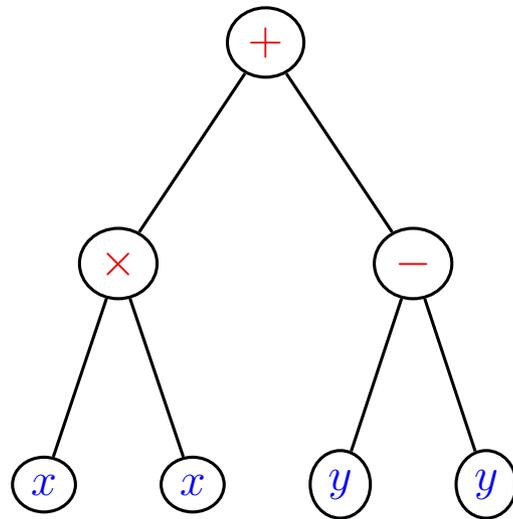


Pick two parents  
Pick break point on each  
Swap sub-trees

We have two new individuals for our next generation

# Mutation

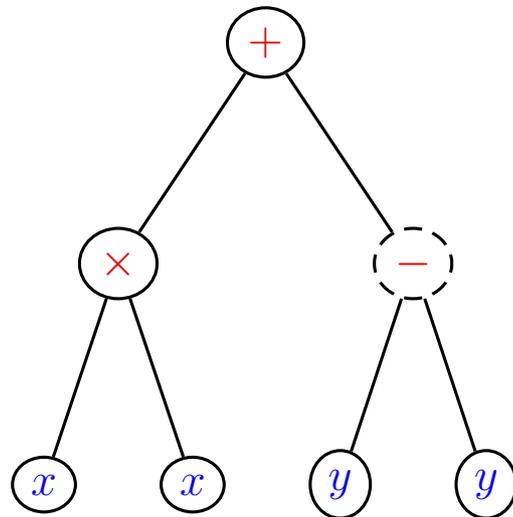
In GA, mutation was a bit flip at a single location.  
Here's how mutation works in GP:



Pick a parent

# Mutation

In GA, mutation was a bit flip at a single location.  
Here's how mutation works in GP:

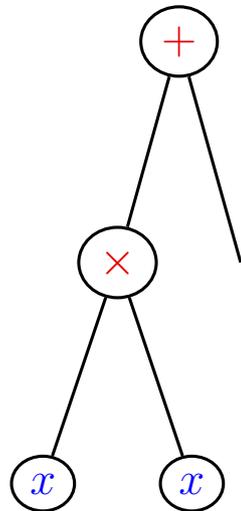


Pick a parent

Pick a mutation point

# Mutation

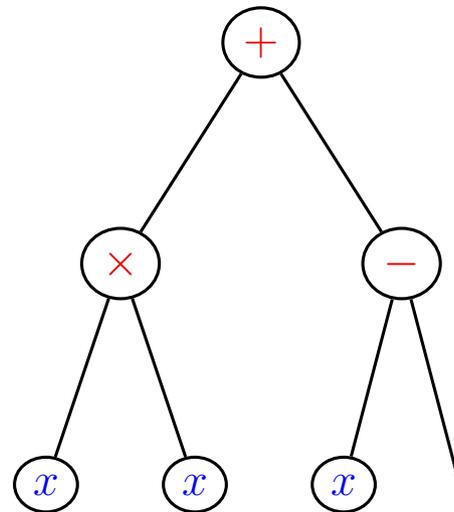
In GA, mutation was a bit flip at a single location.  
Here's how mutation works in GP:



Pick a parent  
Pick a mutation point  
Remove the subtree

# Mutation

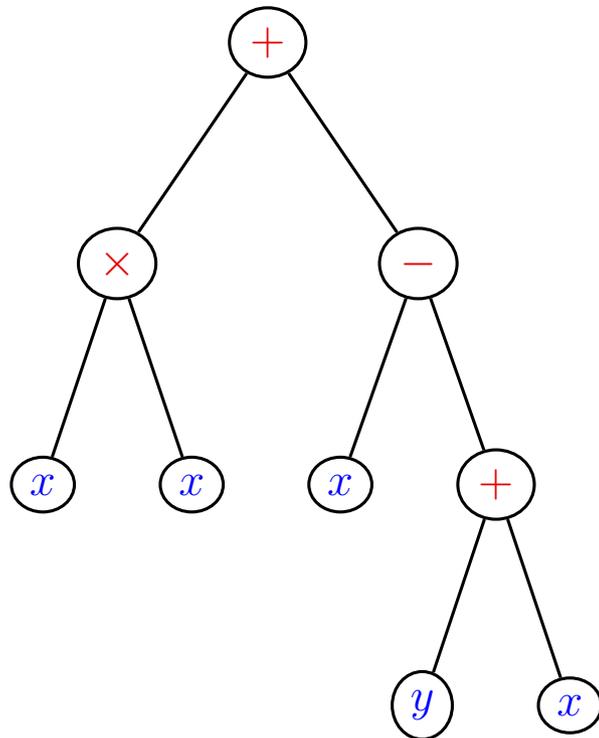
In GA, mutation was a bit flip at a single location.  
Here's how mutation works in GP:



Pick a parent  
Pick a mutation point  
Remove the subtree  
Start a new subtree

# Mutation

In GA, mutation was a bit flip at a single location.  
Here's how mutation works in GP:



Pick a parent

Pick a mutation point

Remove the subtree

Start a new subtree

Finish the new subtree just as if it were a “root” tree

Mutation can often be very destructive in Genetic Programming (as opposed to bit-flips in GA).

As in GA, both crossover and mutation are random processes.

# Practical considerations

Obviously, a tree can grow nearly infinite in size. This is usually undesirable. There are ways to control this:

- Set limits on number of nodes
- Set limits on depth of nodes
- Create initial topologies of specified depth

A common approach is to allow half of the initial population to grow completely randomly and to create the other half at a range of (shallow) depths. In the latter case, pick functions for all nodes  $<$  desired depth, pick terminals for all nodes at desired depth.

# Representations

In the simplest form of tree representation, input values and output values must be of the same type.

- Have to assume a representation when mixing type of functions
  - Logical and real? Maybe  $> 0$  is TRUE (1)
- Must protect against divide-by-zero,  $\sqrt{< 0}$ ,  $\log(< 0)$ , etc.
- Type needn't be simple (integer, float, etc.). Can be any “object.”

There are also representations of GP which are not “tree” representations. There are also strongly-type representations.

# Genetic Programming Example

Let's take a look at an example of a simple Genetic Programming exercise. This example is taken from

<http://www.genetic-programming.com/gpquadraticexample.html>

which goes into more detail.

The goal: create a computer program which outputs the same values as  $x^2 + x + 1$  over the interval  $(-1, +1)$

Notice the goal is *not* to create a program that is *explicitly*  $x^2 + x + 1$ . (Remember the 7 assumptions of programming we had to discard.)

# GP Example Setup

Define functions:

We'll keep this simple and pick  $+$ ,  $-$ ,  $\times$ , and  $/$ .

Remember  $/$  has to work for all values, so  $x/0 \equiv 1$ .

Define terminals:

We want the variable  $x$ , of course.

Also put in integers from  $(-5, +5)$ .

Define fitness:

In theory, we'd like to use the integral of the area (absolute value) between the program and  $x^2 + x + 1$  from  $(-1, +1)$

In practice, we'll do this numerically for maybe 100 points

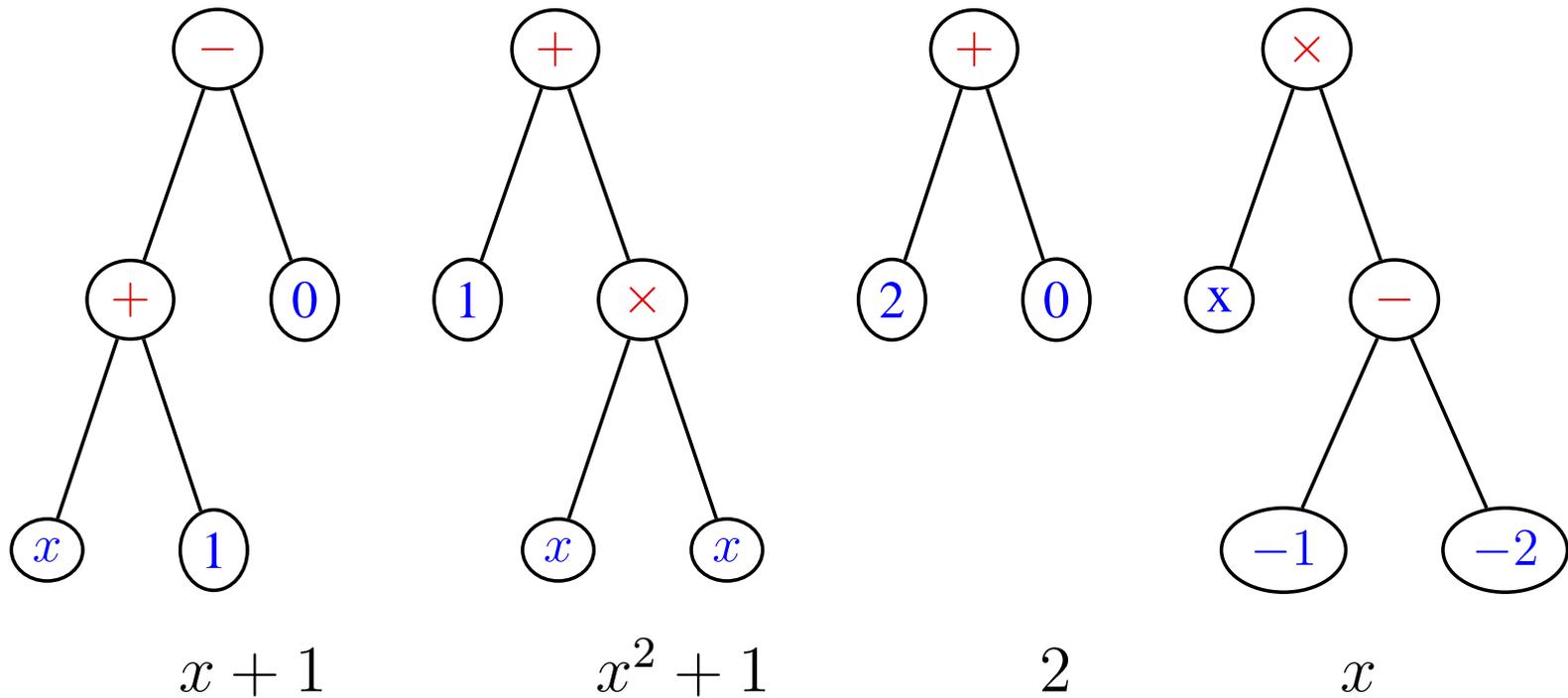
Termination criterion:

Require the fitness of best individual  $< 0.01$

With these few definitions, we are ready to run.

# GP Example, Generation 0

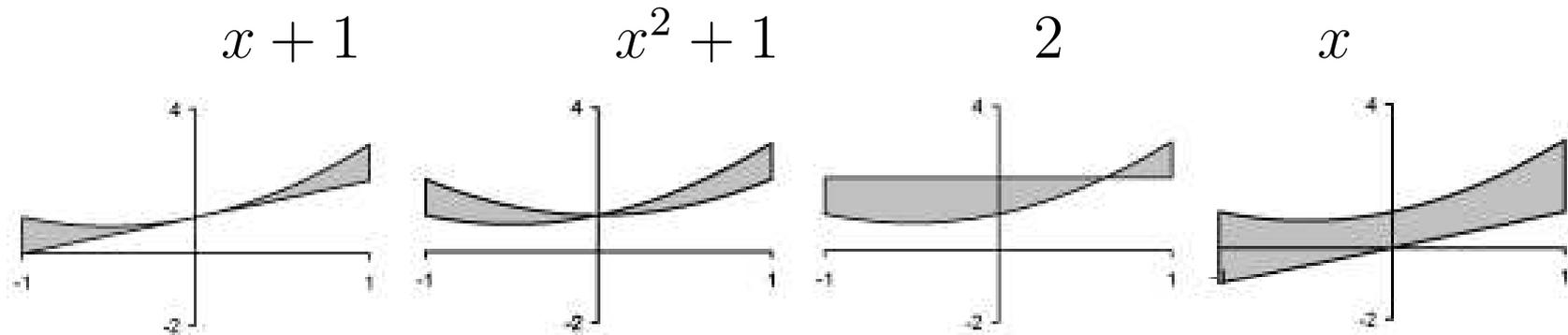
In our simple example, we will generate just 4 programs:



These are our four starting trees and their algebraic equivalents.

# Generation 0 Fitness

Now let's look at the four examples in our population



Here are the four programs from the previous slide and their differences from  $x^2 + x + 1$  shown graphically. (The difference, or fitness, is the area of the gray curve.)

$x^2 + 1$  and  $x + 1$  have the lowest (best) fitness.

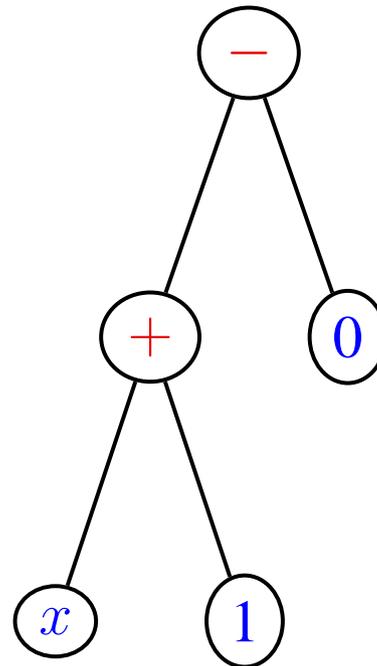
As expected, randomly generated programs will not usually be very efficient at solving even the simplest problems. But, *some will be better than others.*

# GP Example, Generation 1

Now it's time to “breed” a new generation of programs.

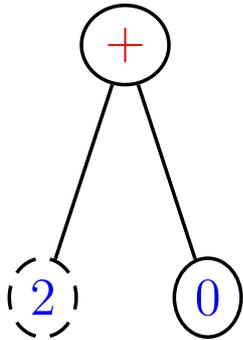
- Usually  $\sim 90\%$  crossover,  $\sim 10\%$  cloning,  $\sim 1\%$  mutation
- Our next generation: 2 crossover children, 1 clone, 1 mutation

First the clone, which happens to be the program with the best fitness (likely to participate, but not necessarily as a clone).

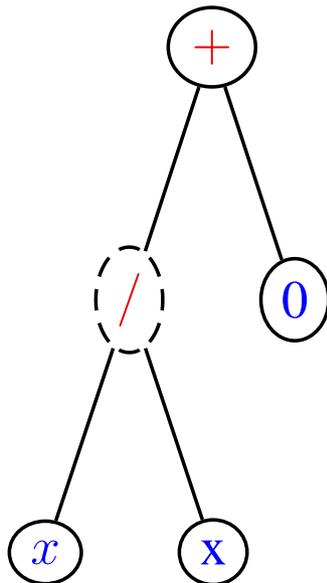


# Generation 1, Mutation

Let's assume the 3<sup>rd</sup> tree is chosen for mutation:



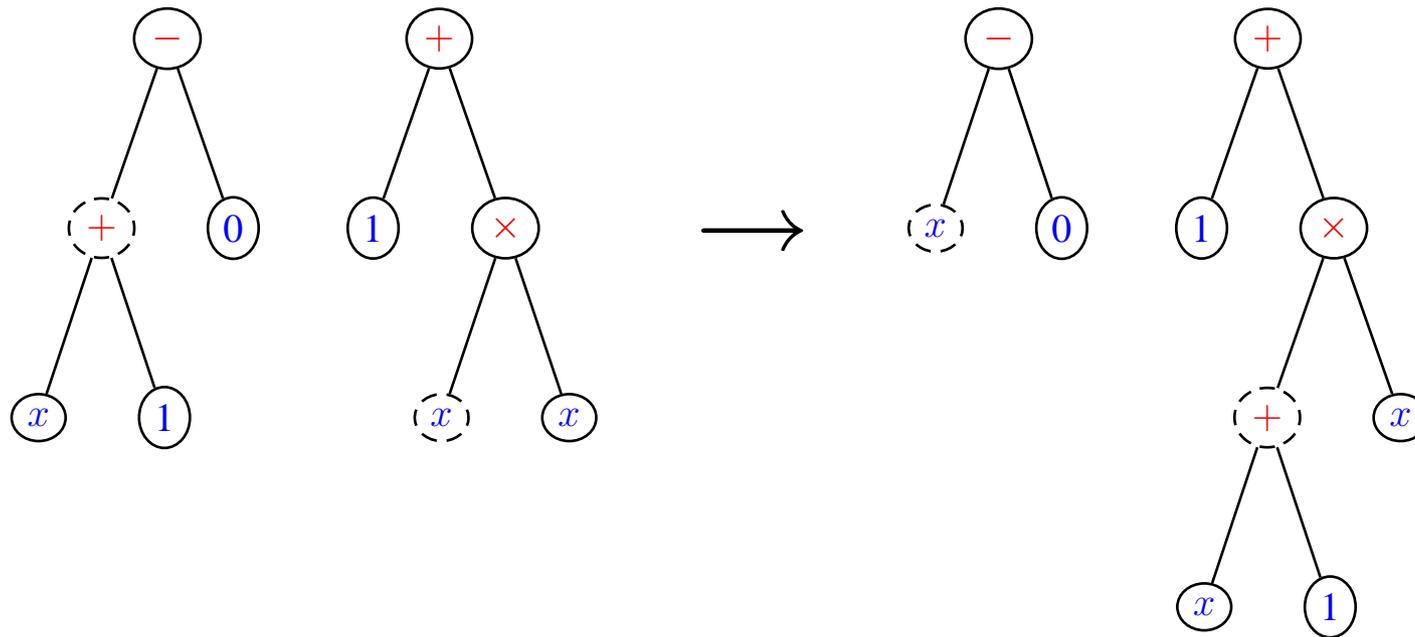
Location of “2” is randomly chosen for mutation and is removed.



A new tree ( $x/x$  with protection) is inserted in its place.

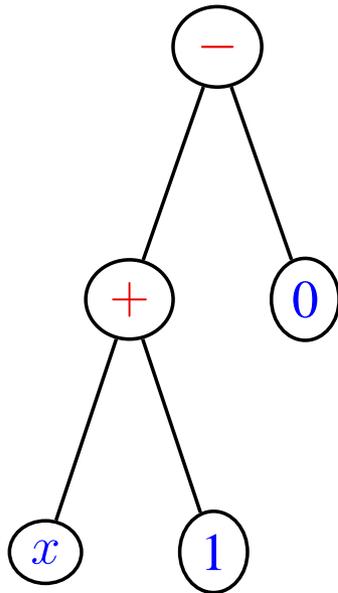
# Generation 1, Crossover

Lastly, two programs are chosen for crossover giving two new programs:

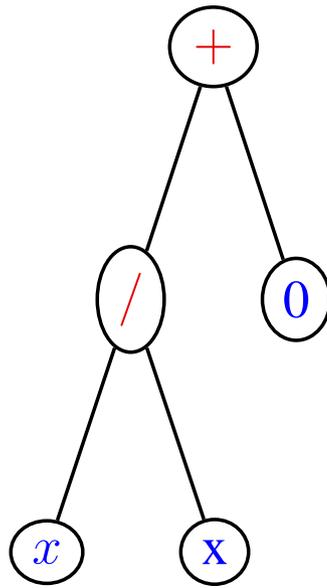


# GP Example, Generation 1

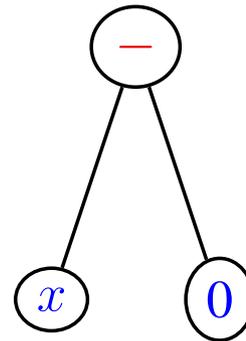
Here are our four trees for Generation 1



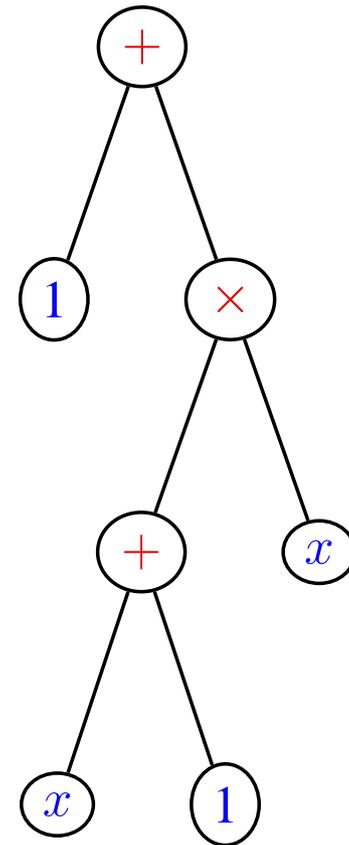
$$x + 1$$



$$1$$



$$x$$

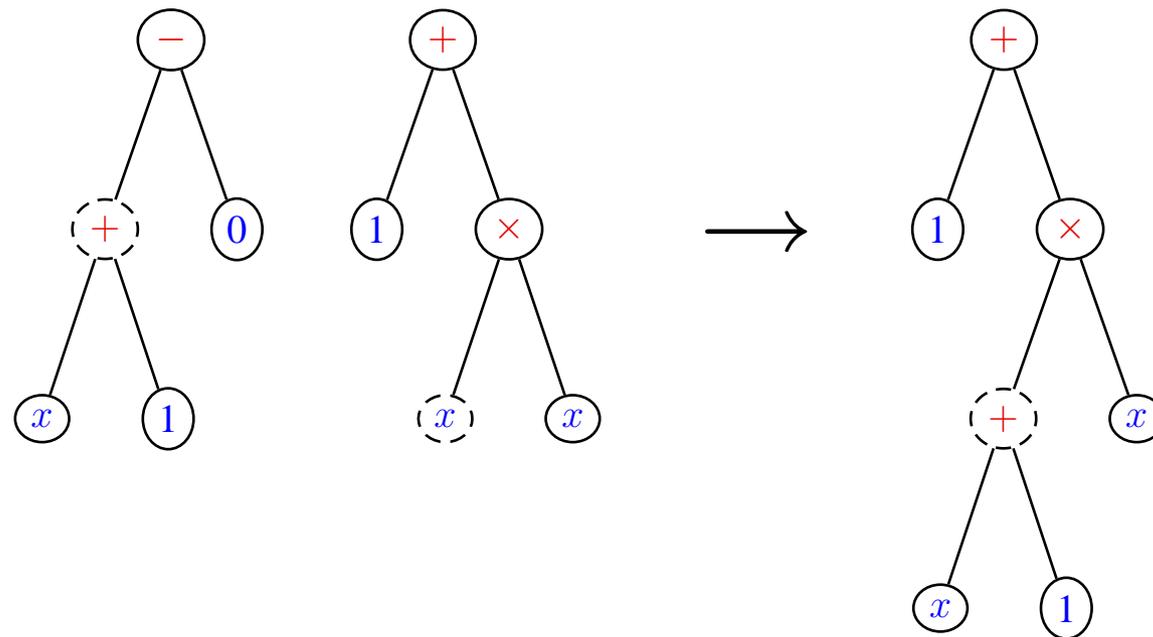


$$x^2 + x + 1$$

The final program is the perfect solution; the run is ended.

# Generation 1, Crossover

First, let's look at the best solution and why finding it is not completely random. It is the combination of two good traits from its two parents (who were "better" programs than their neighbors).

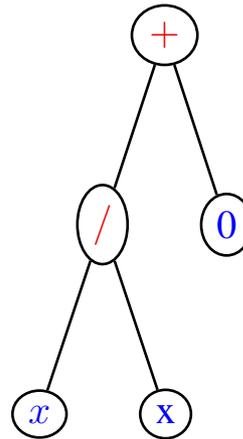


Of course, the computer had no way of *knowing* which traits were good, just which individuals had them.

# GP Example, Continued

Also, our simple example illustrates a few other points:

- Brevity: Many of the programs in our example could be simplified:



- 1<sup>st</sup> individual from generation 0 picked twice, 4<sup>th</sup> not picked (lowest fitness)
- But, a perfect solution has been found, which is *not* guaranteed in Genetic Programming

# Further points

Ok, so finding a program to “fit”  $x^2 + x + 1$  was easy. What if our target function was  $2.718x^2 + 3.1416x$  or something similar?

There are two ways Genetic Programs accomplish this, one implicit, one explicit:

- The GP can manufacture constants from functions and operators
- Introduce the concept of an *ephemeral random constant* (ERC). This is function that when first called, returns a random number. Thereafter, it returns the *same* number. We add this to our set of terminals. We denote this in our terminal set by  $\mathcal{R}$ .

We’ll look at both of these cases.

# Implicit Constant Generation

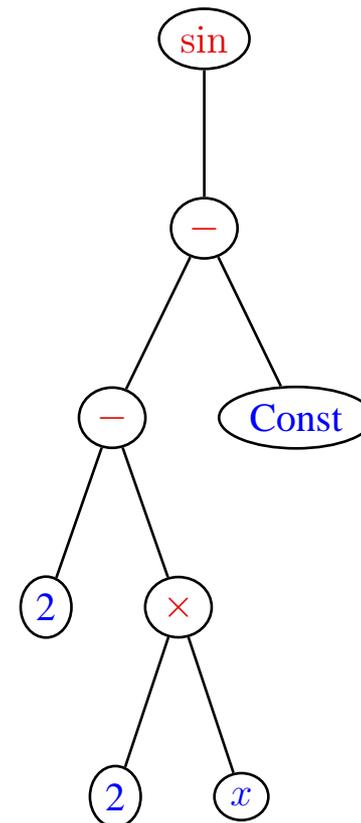
Let's look for a function that is equal to  $\cos 2x$  where  $x$  ranges from 0 to  $2\pi$ . (Example from Koza, Vol. I).

For terminals, we choose  $[x, 1.0, 2.0]$ . For functions,  $+$ ,  $-$ ,  $\times$ ,  $/$ , and  $\sin$ . (We leave out  $\cos$  to avoid trivial solutions).

In generation 30 of such a run, what was found was this:

Where “Const” is a generated constant defined on the next page.

This simplifies to  $\sin(2 - \text{Const} - 2x)$ .



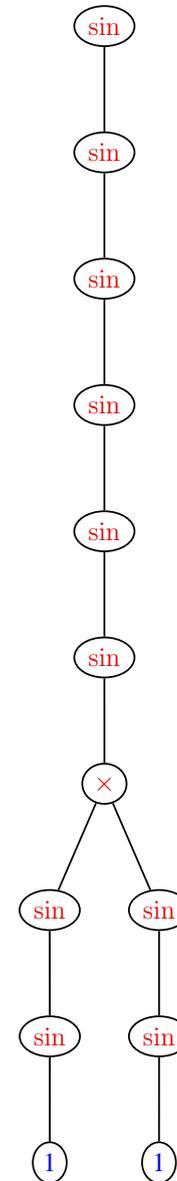
# Implicit Constant Generation

The constant that is evolved is this:

This expression evaluates to 0.433, so  $\sin(2 - \text{Const} - 2x)$  becomes  $\sin(1.567 - 2x)$ .  
 $\pi/2 = 1.571$ .

The identity “discovered” is

$$\sin\left(\frac{\pi}{2} - 2x\right) = \cos 2x.$$



# Implicit Constant Generation

How did this happen?

- Constant evolved over time
- First,  $\sin 1.0 = 0.841$
- Then,  $\sin(\sin 1.0) = 0.746$
- Then,  $\sin^2(\sin 1.0) = 0.556$
- Then, successive applications of  $\sin(x)$  to give 0.528, 0.504, 0.483, 0.464, 0.448, 0.433

# Explicit Constant Generation

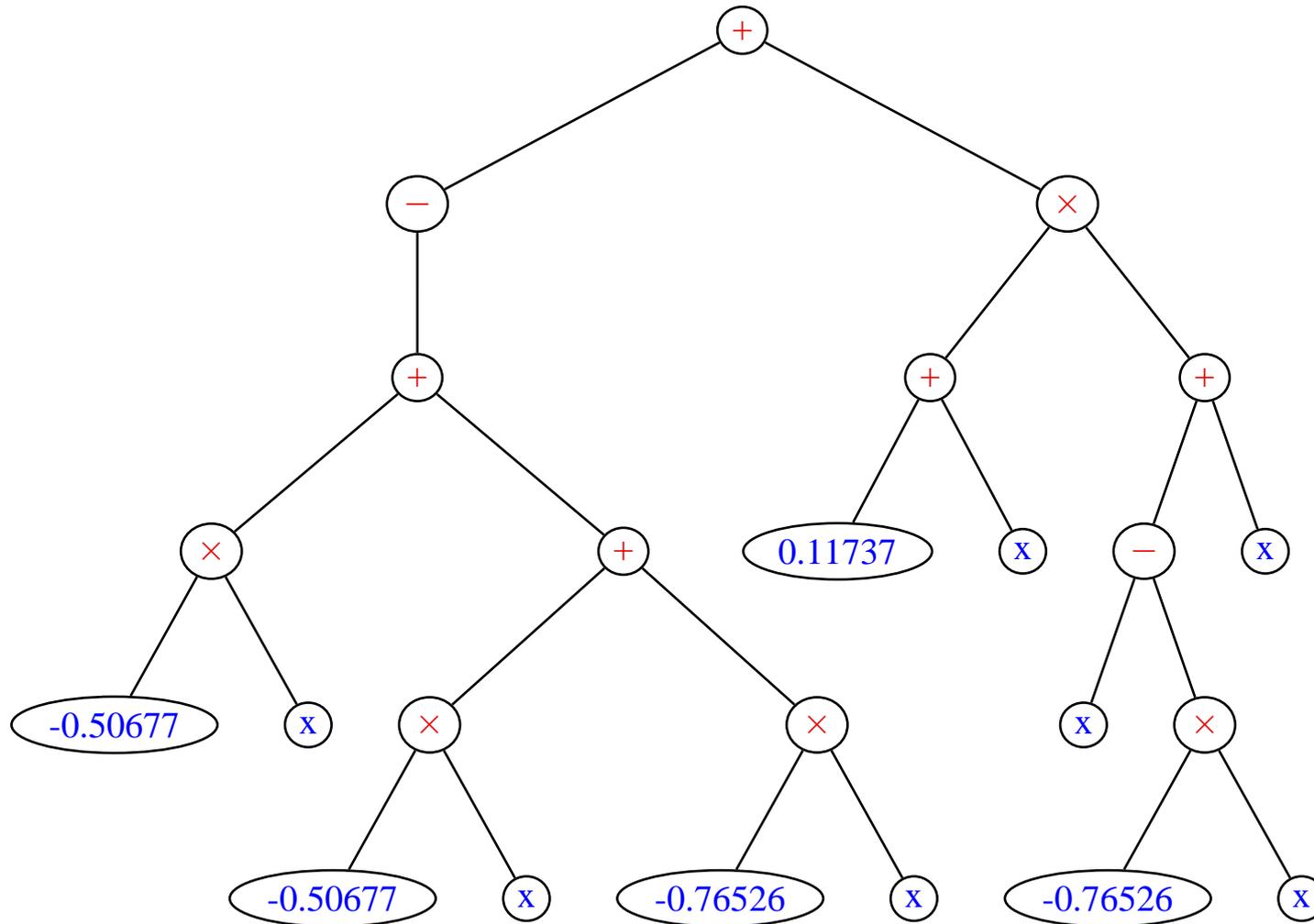
We can also introduce the concept of an *ephemeral random constant* (ERC). This is function that when first called, returns a random number. Thereafter, it returns the *same* number. We add this to our set of terminals. We denote this in our terminal set by  $\mathcal{R}$ .

This gives the GP a group of constants which can be combined to create new constants.

For instance, let's return to the problem of evolving  $2.718x^2 + 3.1416x$  with terminals of  $x$  and  $\mathcal{R}$ .

# Explicit Constant Generation

After 41 generations, we can evolve the tree:



# Explicit Constant Generation

This tree simplifies to the form  $2.76x^2 + 3.15x$ .

Recall we were looking for  $2.72x^2 + 3.14x$ .

Final point: The examples we've shown involve functions that return a single number. This need not be the case. If your framework is in C, your return value can be a structure. In C++ it can be an object. If you want, you can even generate programs in assembly code or even high level languages. The simplest models follow this function-tree representation, but fully specified grammars are possible with existing software.

# GP Applications

We can see that this might be applicable to a wide variety of applications. Here are some proven examples:

- Symbolic regression (finding a functional form for a set of data)
- Determining which combinations of 10,000 genes cause cancer
- A filter to classify high energy physics events

This last example is the use I've been studying, so we'll look at it in more detail next time.

# GA & GP Resources

There is a lot of information on the web about Genetic Algorithms and Programming:

- <http://www.aic.nrl.navy.mil/galist/> — Genetic Algorithms
- <http://www.genetic-programming.org/> — John Koza

Software frameworks for both GA and GP exist in almost every language (most have several)

- [http://www.genetic-programming.com/coursemainpage.html#\\_Sources\\_of\\_Software](http://www.genetic-programming.com/coursemainpage.html#_Sources_of_Software)
- <http://zhar.net/gnu-linux/howto/> – GNU AI HowTo (GA/GP/Neural nets, etc.)
- <http://www.grammatical-evolution.org> — GA–GP Translator