HERIOT WATT UNIVERSITY

Genetic Programming I

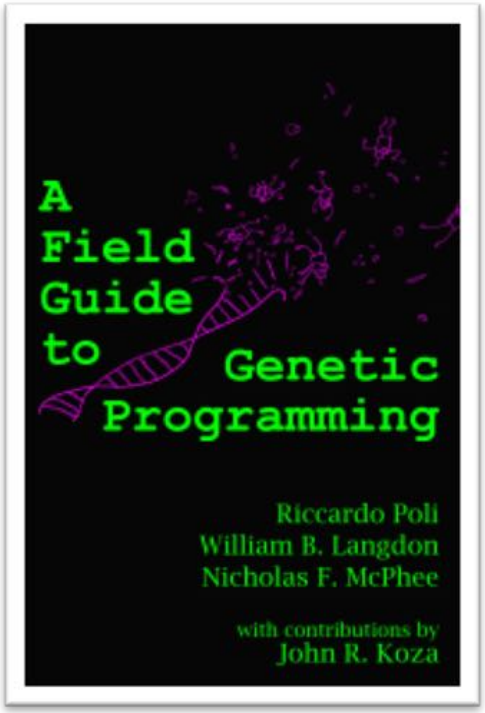
Dr. Michael Lones
Room EM.G31
M.Lones@hw.ac.uk

Free books on Vision

HERIOT WATT UNIVERSITY

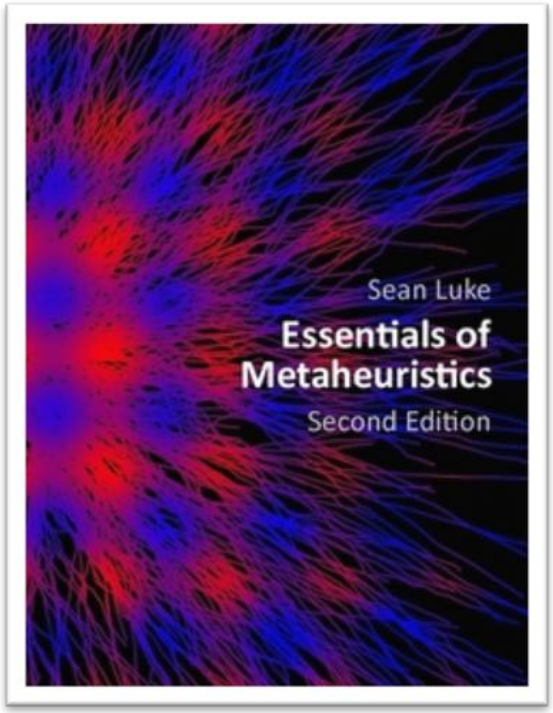
- A reminder of these books I mentioned last week:

R. Poli et al, A Field Guide to Genetic Programming



www.gp-field-guide.org.uk

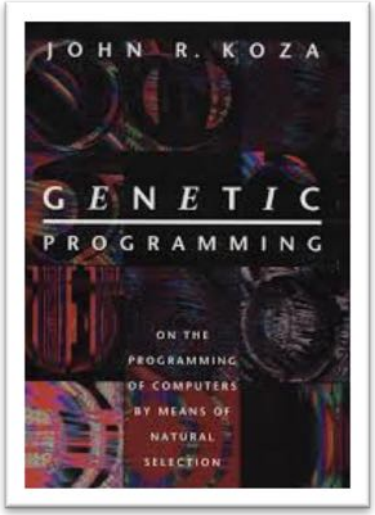
S. Luke, Essentials of Metaheuristics

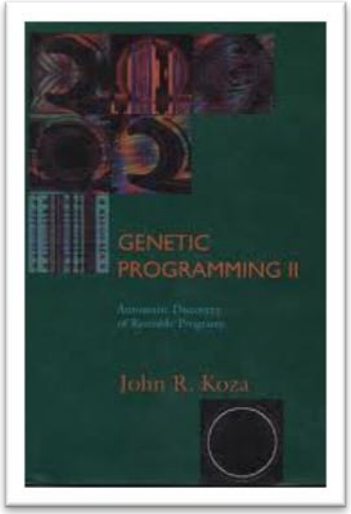


http://cs.gmu.edu/~sean/book/metaheuristics/

Other useful books

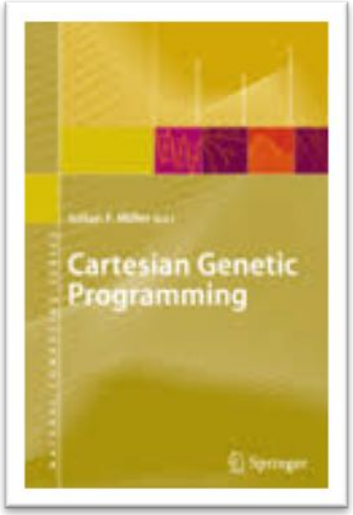
- Other books that are worth taking a look at:





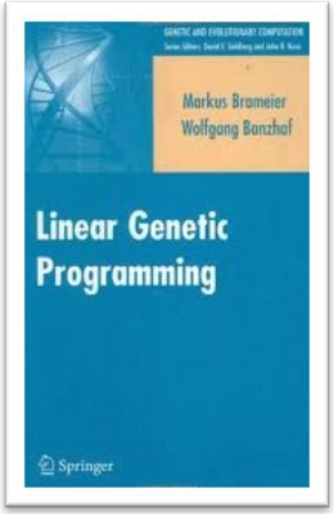
John Koza, Genetic Programming & Genetic Programming II

Both in the library



Julian Miller Cartesian Genetic Programming

http://link.springer.com/book/10.1007/978-3-642-17310-3



Brameier&Banzhaf Linear Genetic Programming

http://link.springer.com/book/10.1007%2F978-0-387-31030-5

Genetic Programming (GP)

- In a nutshell, GP is about using evolutionary algorithms to design computer programs
  - Or other 'executable structures', e.g. circuits, equations
  - Generally small programs that do specific things
  - So we wouldn't expect to evolve Microsoft Office

http://www.genetic-programming.com

Genetic Algorithm (GA)

- In a nutshell…

  - Create a population of random vectors
  - Then repeat:
    - Evaluate them
    - Kill off the (really) bad ones
    - Keep the (relatively) good ones
    - Use them to breed the next generation (by using mutation and recombination operators)
  - Until the problem is (hopefully!) solved

Genetic Programming (GP)

- In a nutshell…

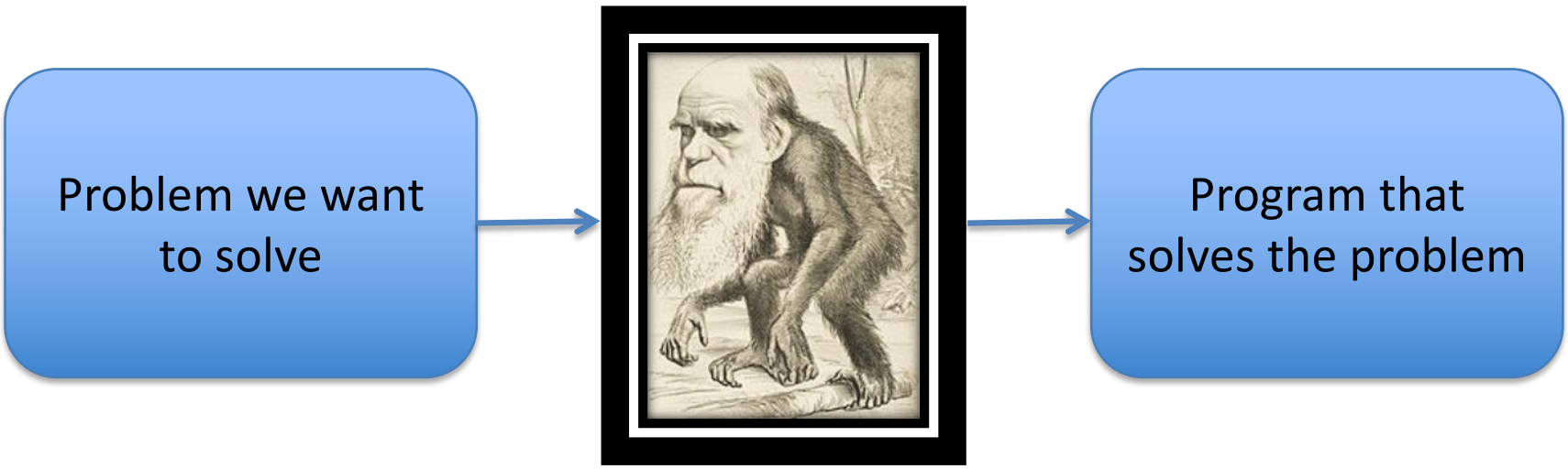
  - Create a population of random **programs**
  - Then repeat:
    - Evaluate them
    - Kill off the (really) bad ones
    - Keep the (relatively) good ones
    - Use them to breed the next generation (by using mutation and recombination operators)
  - Until the problem is (hopefully!) solved

Genetic Programming (GP)

- Why use evolutionary algorithms?
  - Program landscapes tend to be complex, and EAs are relatively good at solving difficult optimisation problems
  - They are flexible in how solutions are represented; programs are not just lists of numbers
  - However, the focus on EAs is in part historical; other optimisers may, in principle, be used

Genetic Programming (GP)

- Why do we want to evolve programs?
  - Sometimes because we're lazy!
  - More often because we don't know how to write a program to solve a particular problem
  - Or we want to do better than an existing solution

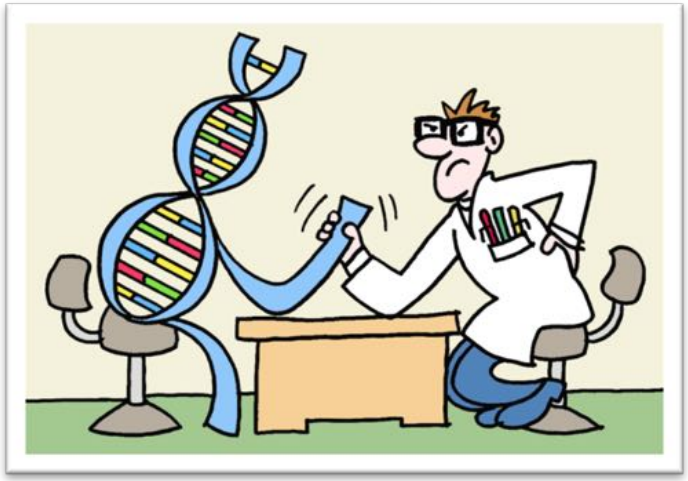


Evolutionary 'black box'

Genetic Programming (GP)

- Often portrayed as a form of automatic innovation
  - http://www.human-competitive.org/
  - "Humies" is an annual contest for human-beating results
  - $10,000 in prizes every year

- Previous Humies winners include:
  - Games controllers
  - Circuit designs/designers
  - Image analysis algorithms
  - Software engineering tools
  - Medical diagnostics tools



Particularly interesting this year ;)

Genetic Programming (GP)

- There are many different kinds of GP
- They differ in how they represent programs
  - Syntax trees, program graphs, list of instructions, …
  - The languages that programs can be evolved in
  - Also their degree of bio-inspiration
- Representation is important
  - The programs we write are fragile
  - Imagine "mutating" one ➔
  - Can we remove this fragility??
    (this is a big research question)

http://en.wikipedia.org/wiki/File:Windows_NT_BSOD_at_GVA_baggage_claim,_1999-10-03.jpg

Evolvability

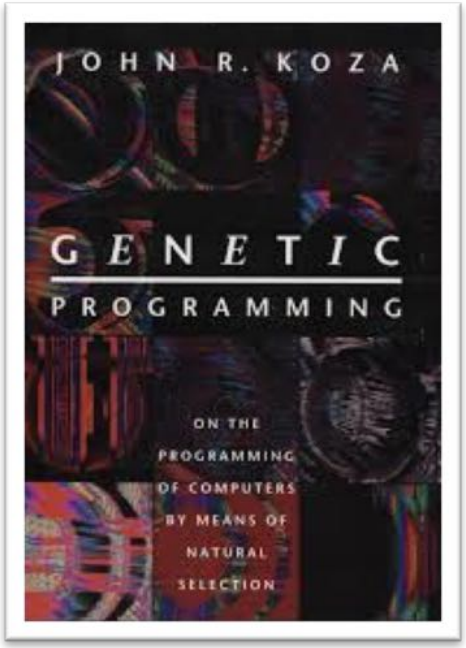
This is the capacity for a program to improve its fitness as a result of an evolutionary process (i.e. mutation and recombination).

For genetic programming, there's little value in being theoretically able to express a program if it can not be discovered by evolution.

Any Questions?

Koza Tree-Based GP

- This is the best known form of GP
  - Invented by **John Koza** in the 1980s
  - The earliest successful system for evolving programs
  - And still what most people think of when they hear the term "genetic programming"



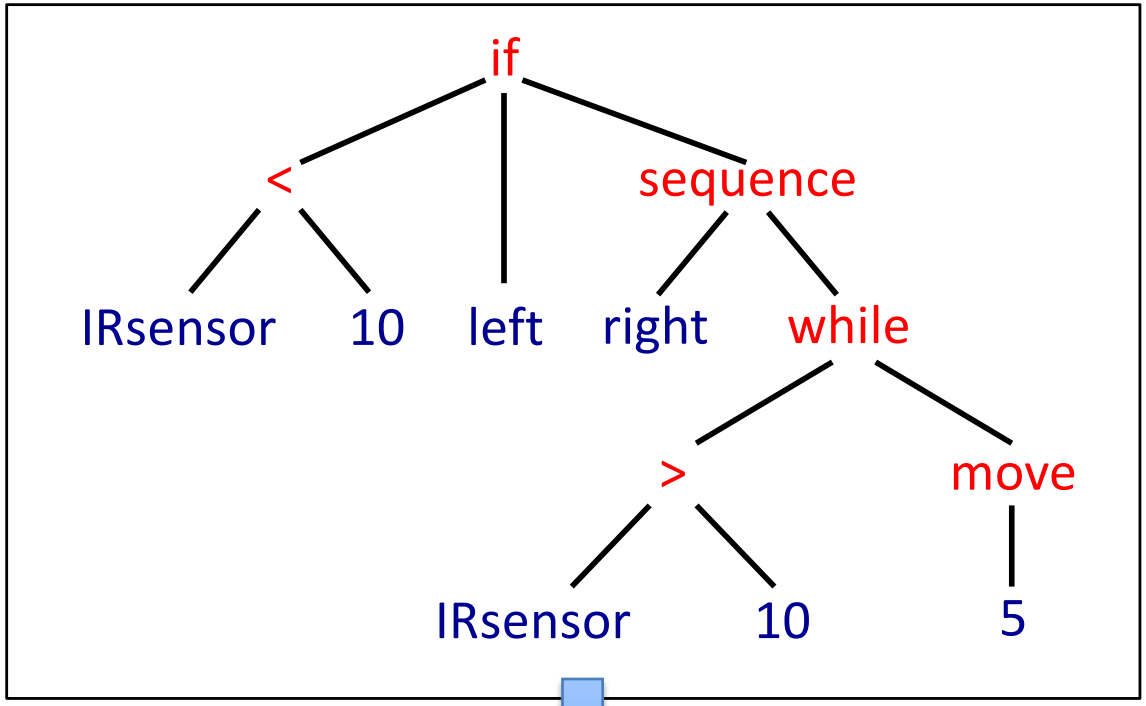
- Today's lecture focuses on Koza's tree-based GP
  - I'll go through the main points and issues
  - Next lecture: newer (better?) forms of GP

Koza Tree-Based GP

- Why is it called tree-based GP?
  - Because programs are represented by trees
  - More specifically *abstract syntax trees* or *parse trees*
  - These are also used by many computer language tools (e.g. compilers) to represent computer programs

- A parse tree comprises:
  - A set of internal nodes, which each represent a function instance selected from a **function set**
  - A set of leaf nodes, which each represent a terminal (e.g. an input or a constant), taken from a **terminal set**

Parse Tree Example




```
if(IRsensor()<10) left();
else {
    right();
    while(IRsensor()>10) move(5);
}
```

Function set =
{ if, <, >, sequence,
while, move, … }

Terminal set =
{ IRsensor, 10, left,
right, 5, … }

Koza Tree-Based GP

- GP is an evolutionary algorithm, and uses all the same mechanisms and processes, including:
  - An **initialisation** process, where solutions in the initial population are sampled from the solution space
  - **Mutation** operators, which make changes to existing solutions, and hence move through the solution space
  - **Crossover** operators, which take two existing parent solutions and generate one or more child solutions
  - However, in Koza GP, these have to be adapted to operate on trees rather than lists of numbers

Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }

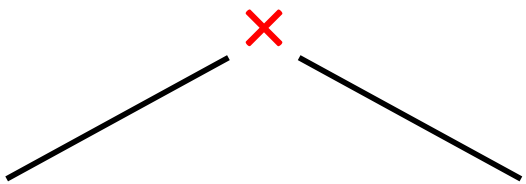
Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }

×

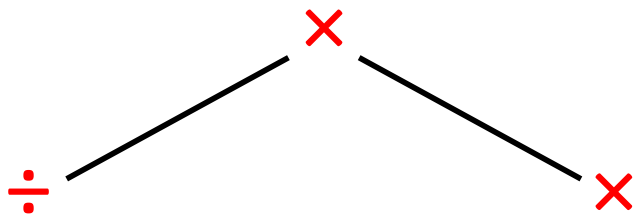
Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }



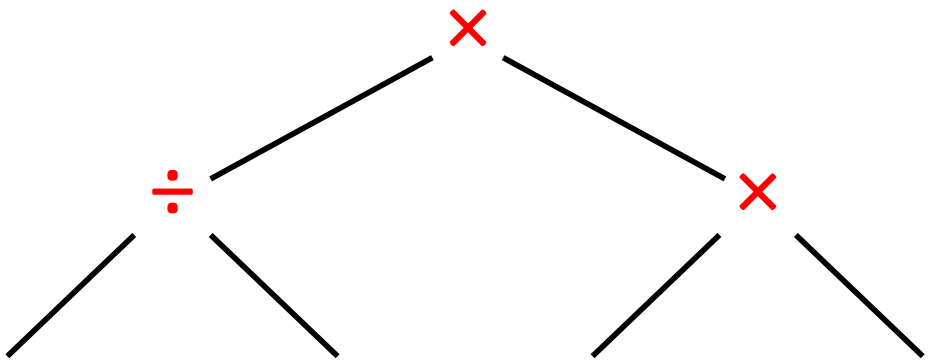
Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }



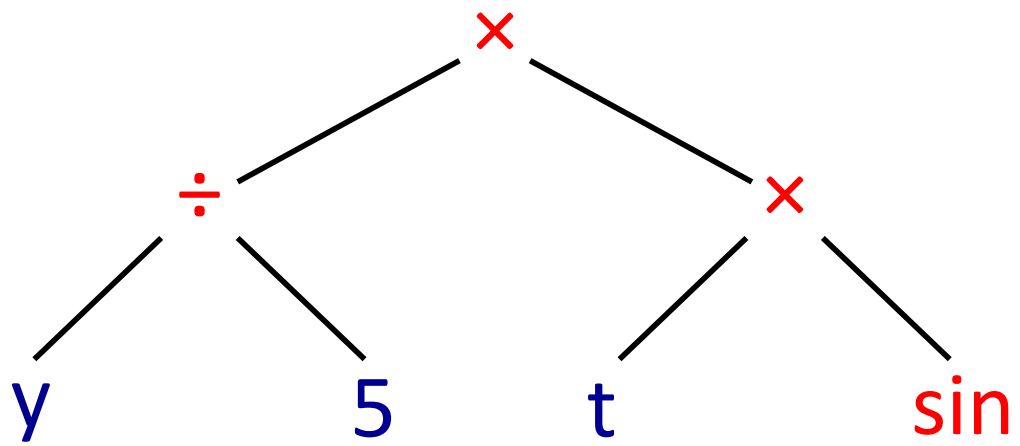
Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }



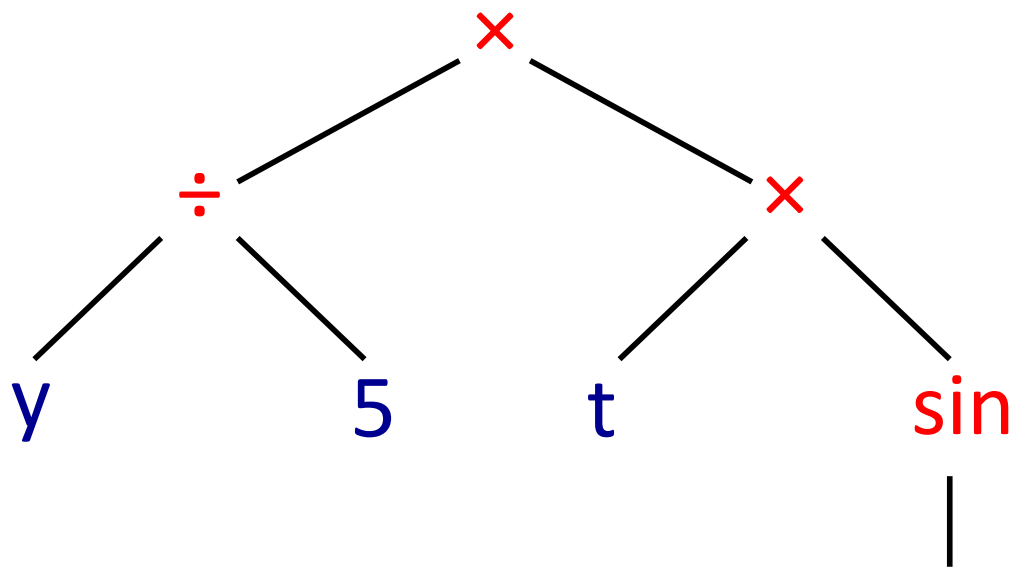
Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }



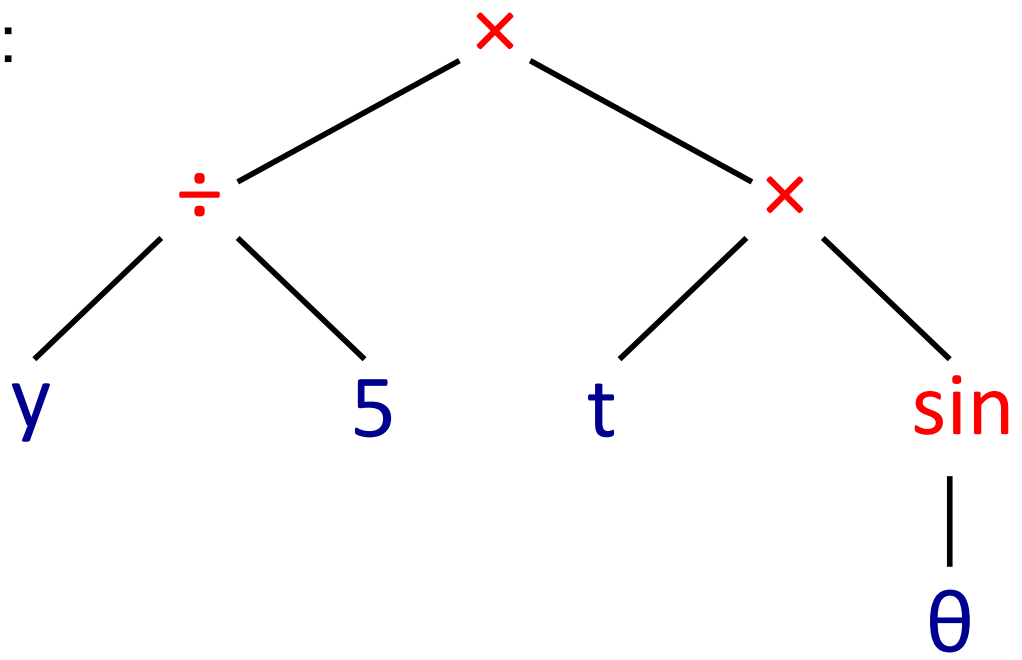
Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }



Initialisation

- E.g. Creating a random mathematical expression
  - Function set = { +, -, ×, ÷, sin, cos }
  - Terminal set = { y, t, θ, constant }
  - e.g. (y/5)*(t sin θ) :



Initialisation

- Here we created a random initial solution by randomly sampling the function and terminal sets
  - This works, but isn't used because it tends to restrict the range of tree shapes present in the initial population
  - A more common method is "ramped half-and-half", which explicitly generates both long-and-thin and bushy (i.e. wide-and-short) trees
  - It's important to have this initial diversity, because we don't know what shape an optimal program will be
  - I don't expect you to know about this, but you can find out more in Sections 2.2 & 5.1 of the Field Guide.

Implementation Code

- Where can I see some GP code?
  - "Essentials of Metaheuristics" sec. 3.3.3 & 4.3 (or ECJ)

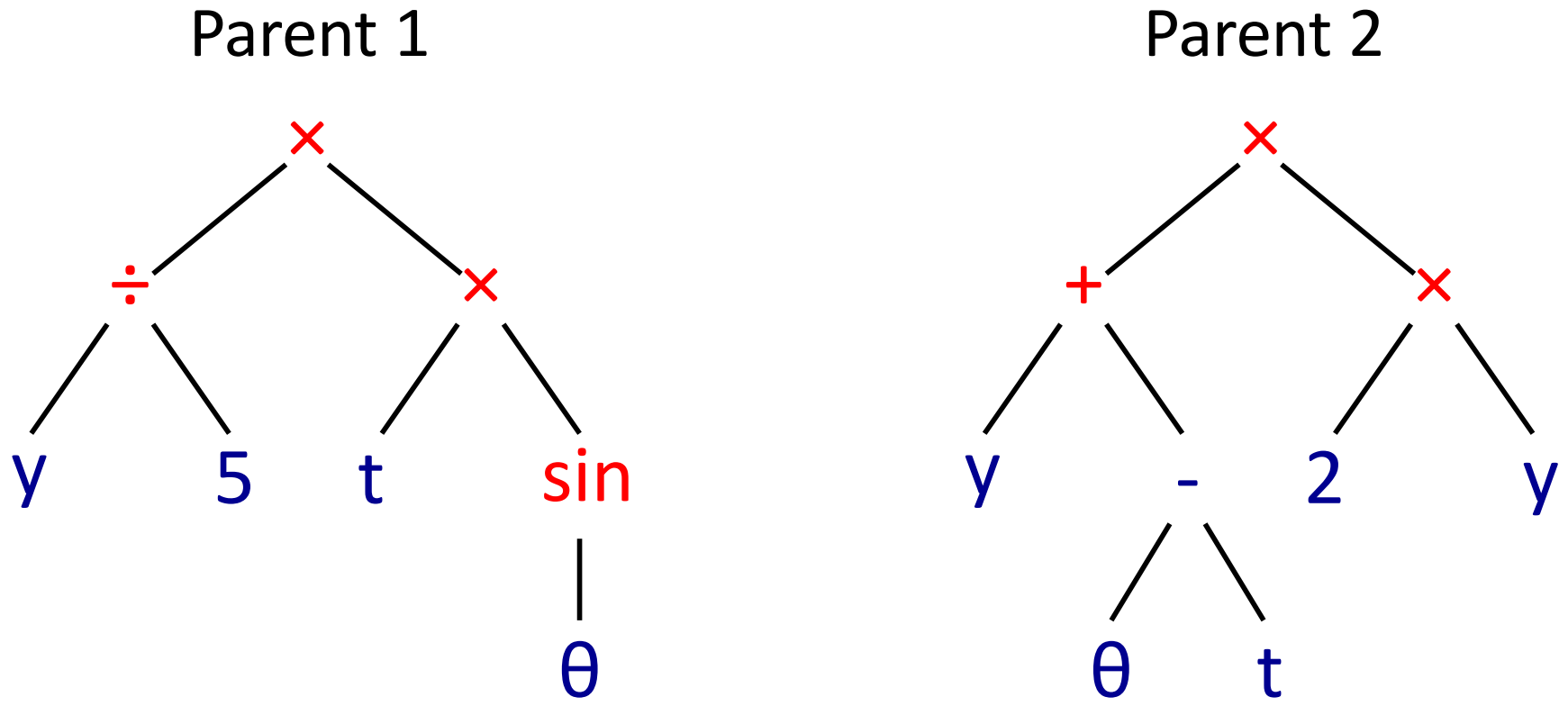
Algorithm 55 *The Ramped Half-and-Half Algorithm*

```
1: minMax ← minimum allowed maximum depth
2: maxMax ← maximum allowed maximum depth
3: FunctionSet ← function set

4: d ← random integer chosen uniformly from minMax to maxMax inclusive
5: if 0.5 < a random real value chosen uniformly from 0.0 to 1.0 then
6:     return DoGrow(1, d, FunctionSet)
7: else
8:     return DoFull(1, d, FunctionSet)
```

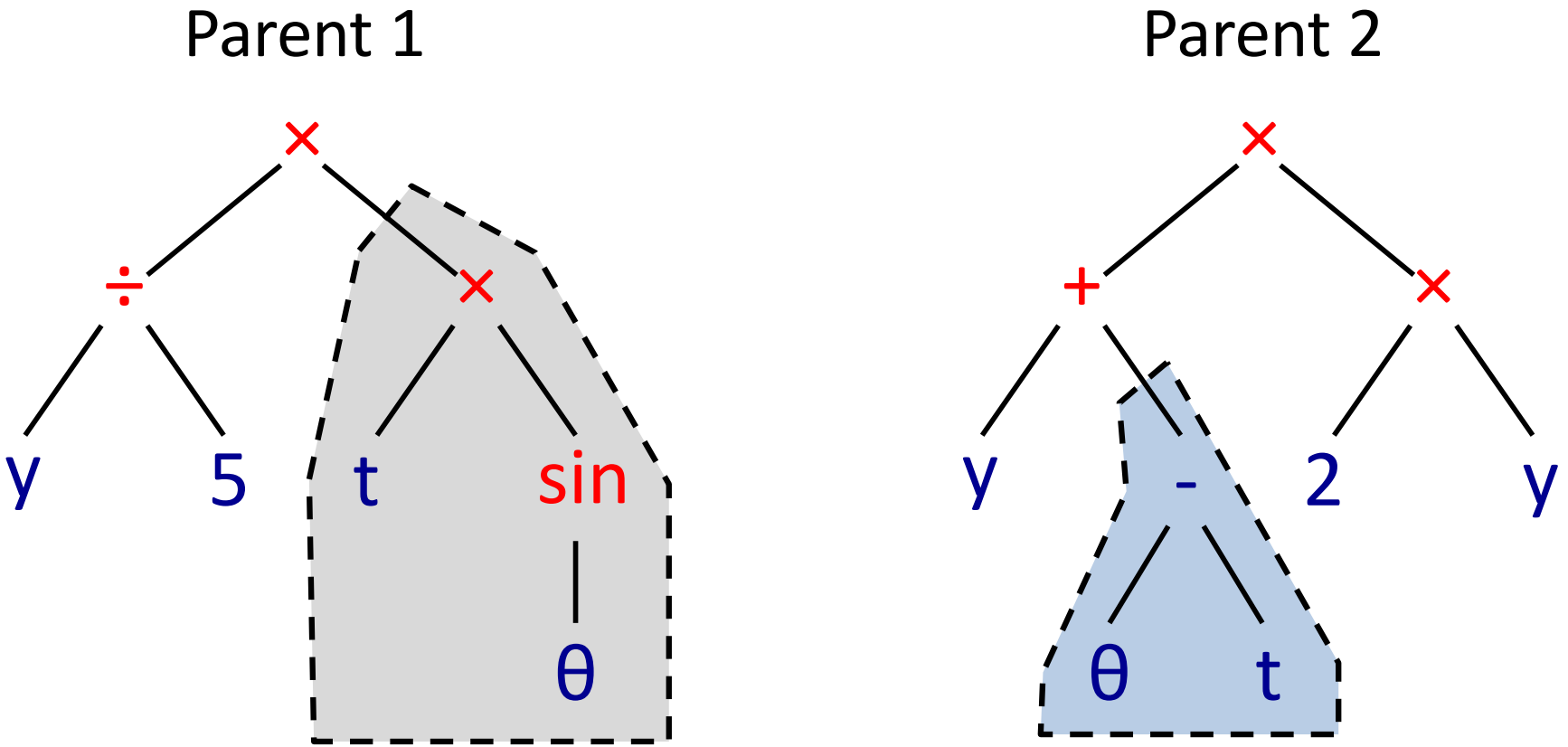
Recombination

- Sub-tree crossover:



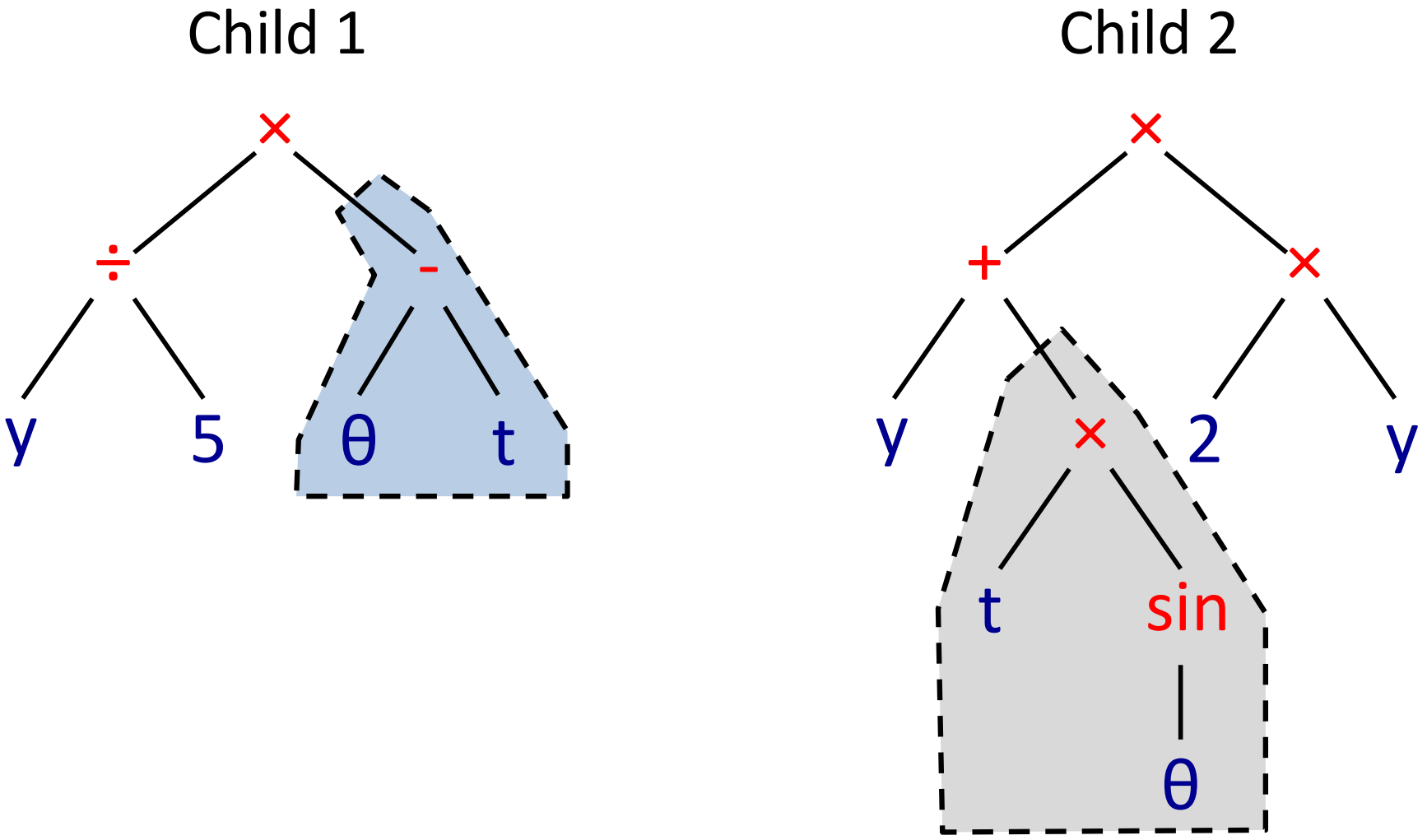
Recombination

- Sub-tree crossover:



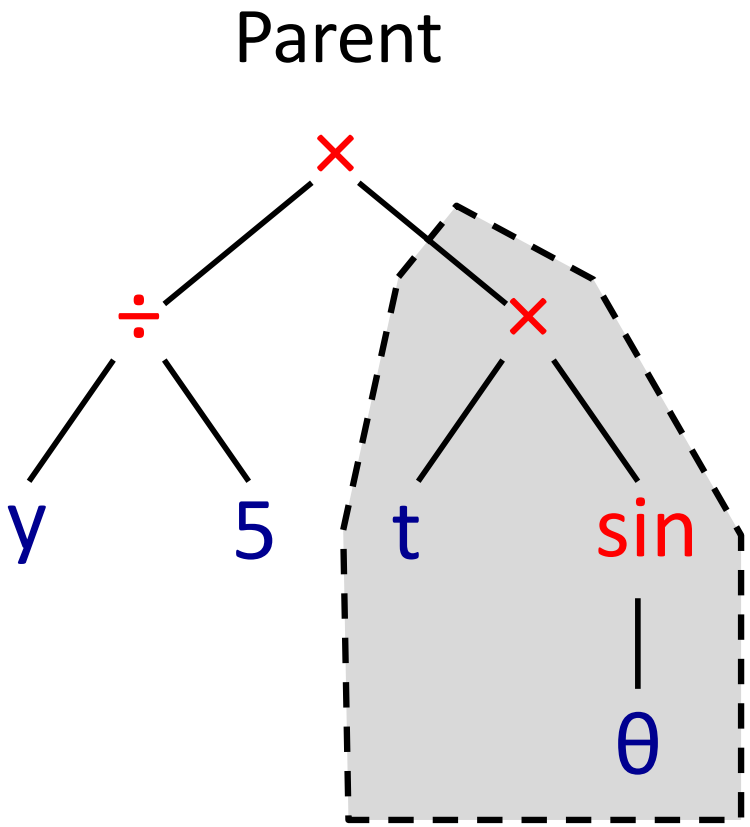
Recombination

- Sub-tree crossover:



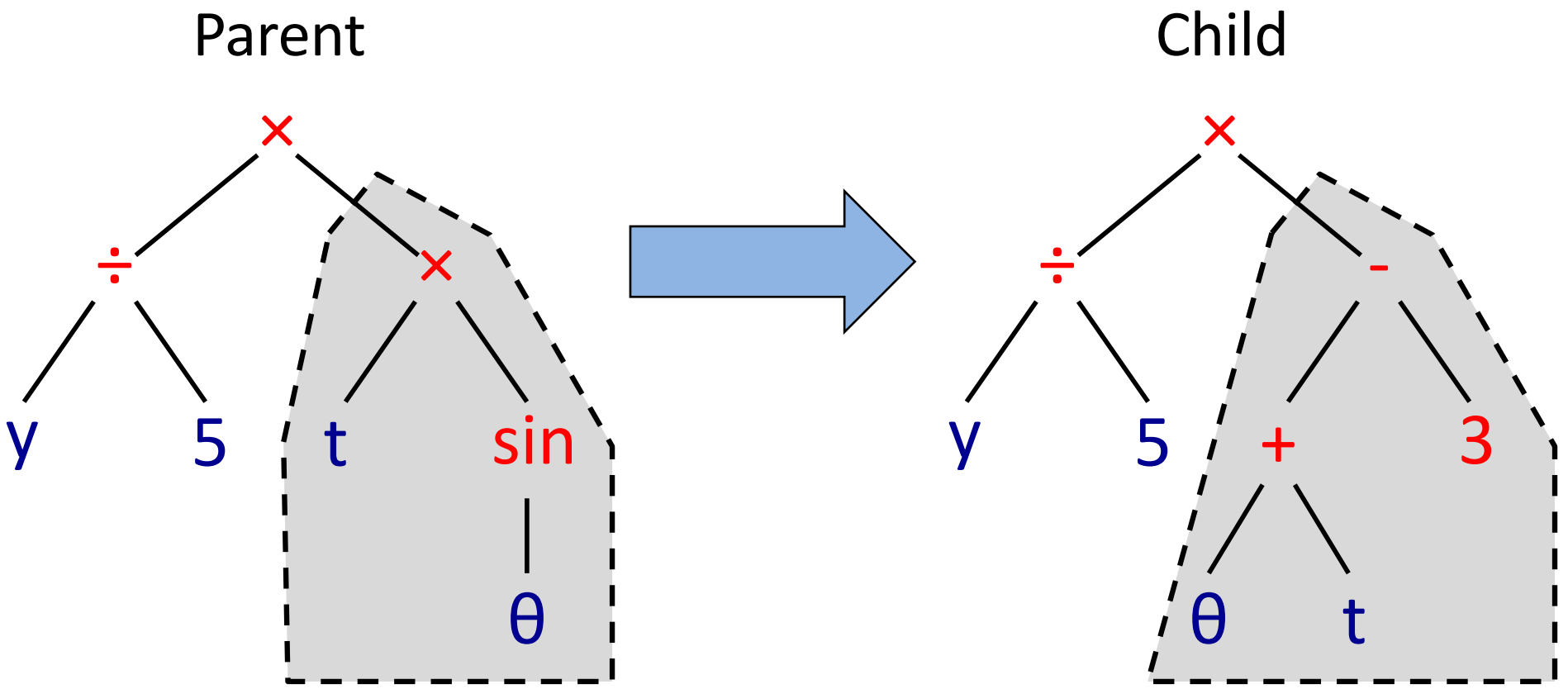
Mutation

- Sub-tree mutation:



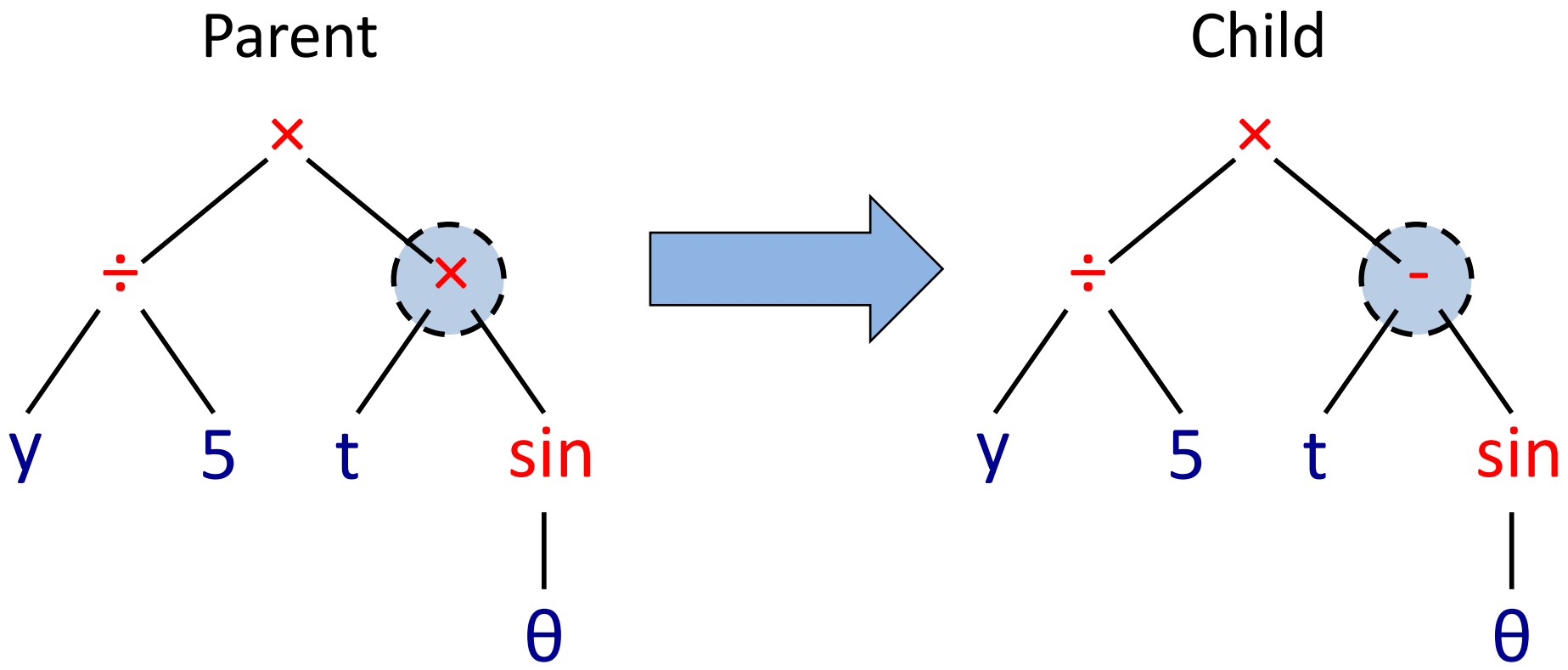
Mutation

- Sub-tree mutation:



Mutation

- Point mutation (less disruptive):



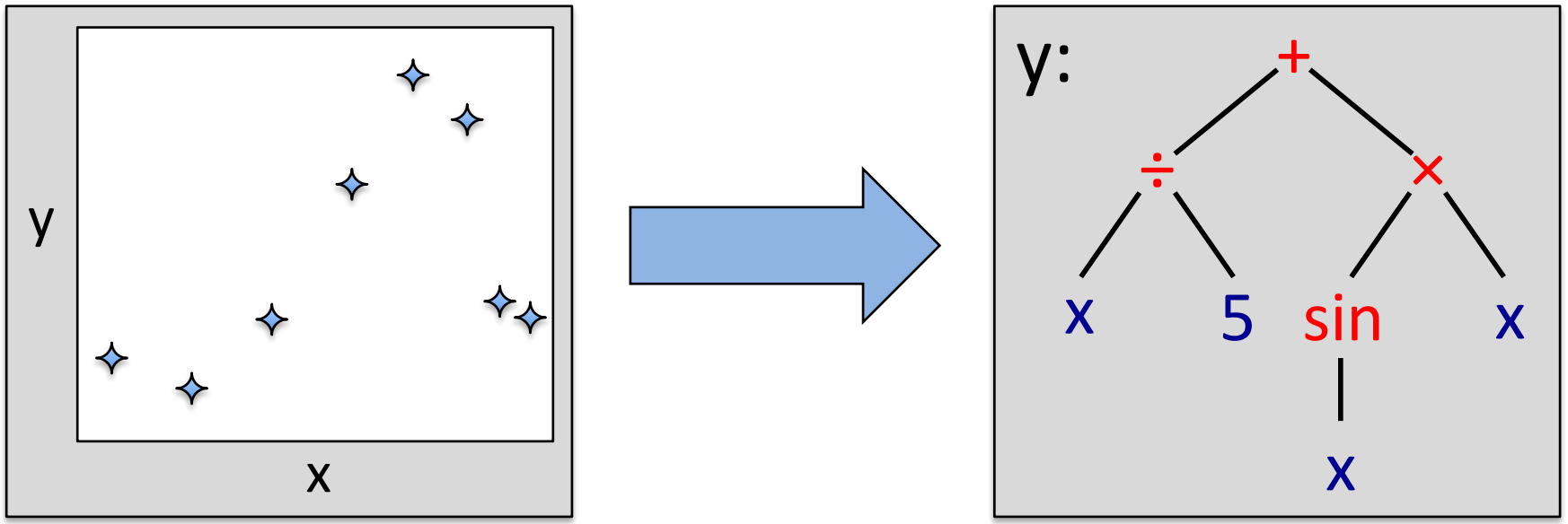
Variation Operators

- These are the most common operators used in GP
  - There are lots of variants; many are concerned with where crossover points or mutations can occur in a tree
  - These are designed to control how the search process explores the solution space
  - e.g. to avoid large trees (see "bloat" later), or to avoid making large behavioural changes to a program
  - You're not expected to know about these variants, but you can find out more in Sections 2.4, 5.2 & 5.4 of the Field Guide

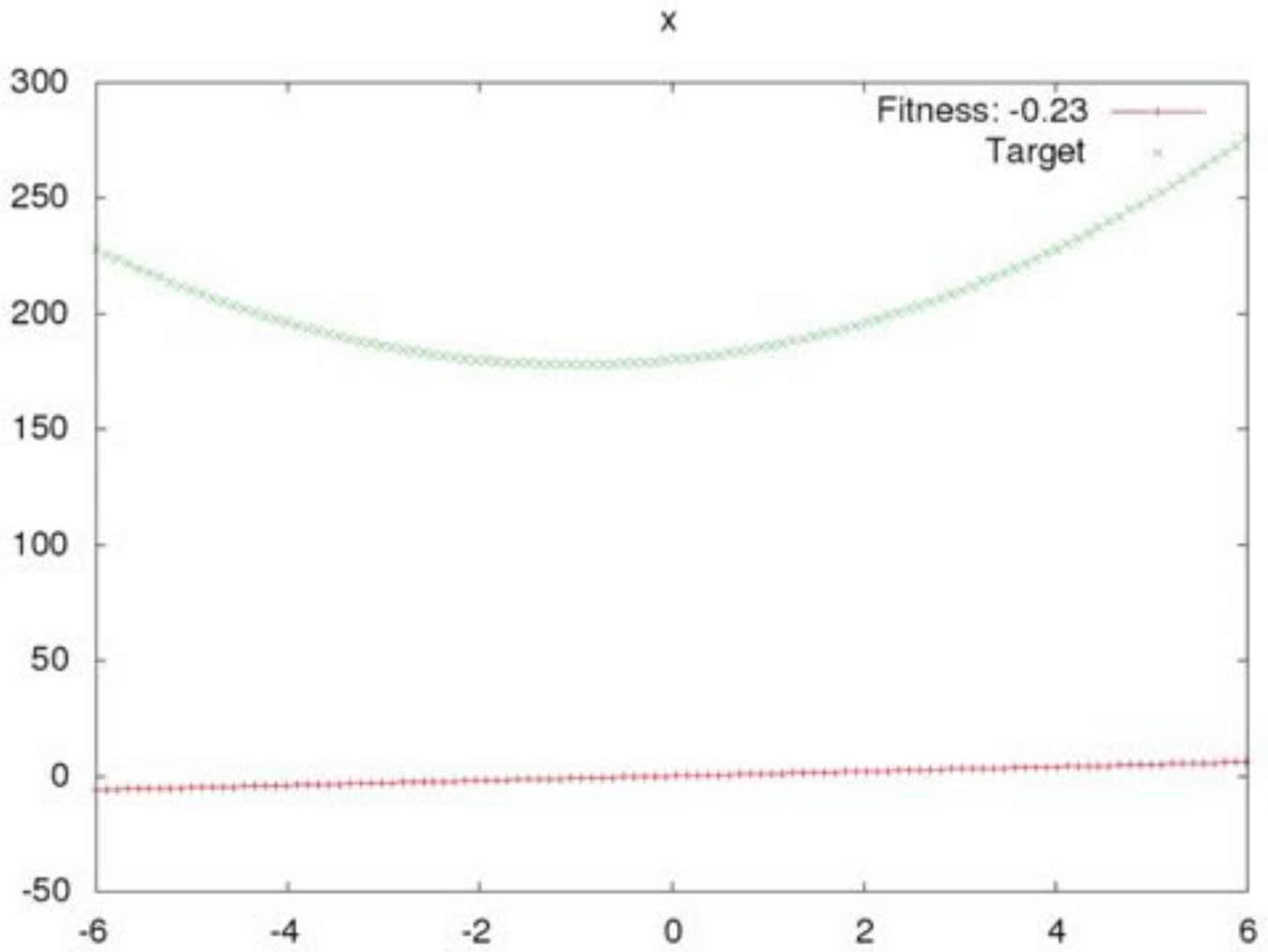
Any Questions?

Symbolic Regression

- Evolving mathematical expressions is one of the most popular applications of Koza tree-based GP
  - Finding an expression that explains a relationship in a data set; this is known as **symbolic regression**
  - Useful when little is known about the generating function



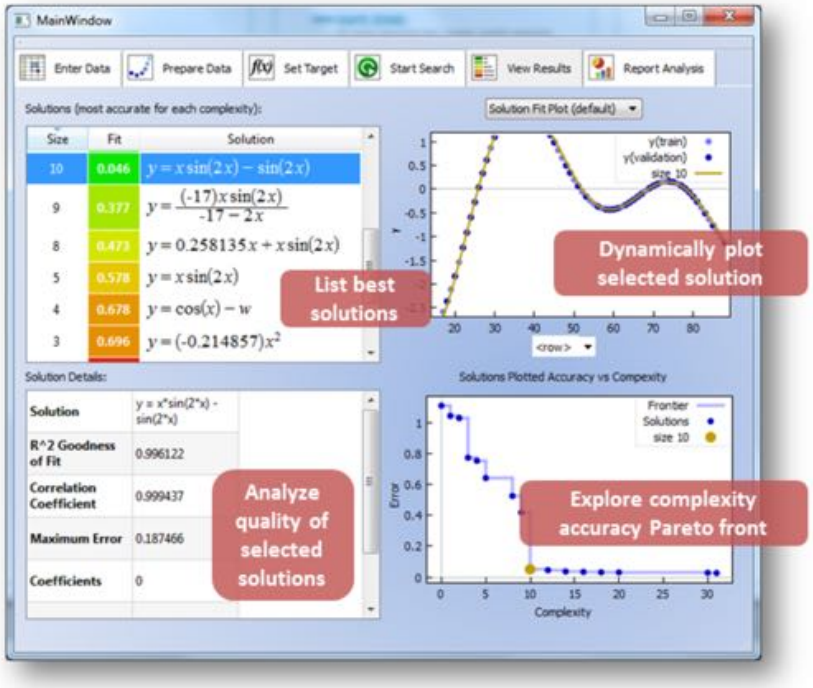
Curve Fitting Example



https://www.youtube.com/watch?v=37D3QpFvrgs

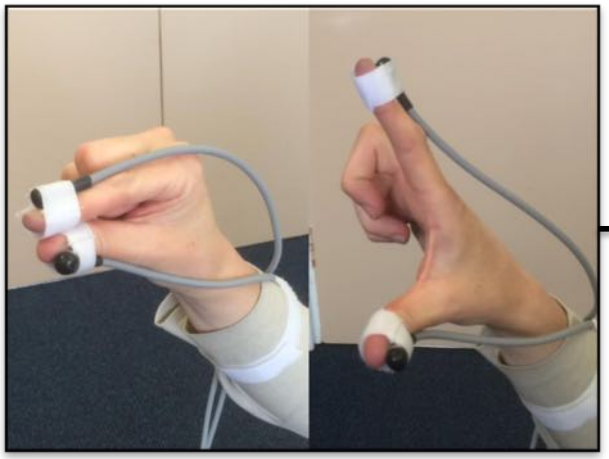
Notable Success Story

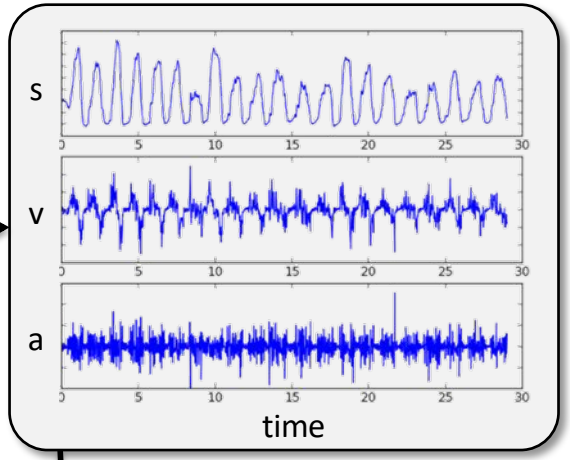
- Lipson's group at Cornell have done a lot of work in this area, mostly using their **eureqa** software
  - see http://www.nutonian.com/products/eureqa/
  - This software is used by NASA, Amazon, eHarmony, GM, Intel, Pfizer, BP, Thomson Reuters…
  - Their most interesting work involves finding invariants (i.e. laws defined as equations) that explain patterns seen in physics and biology data

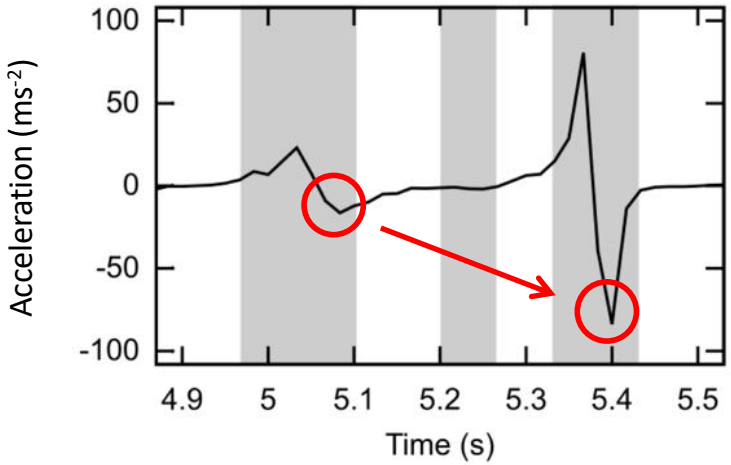


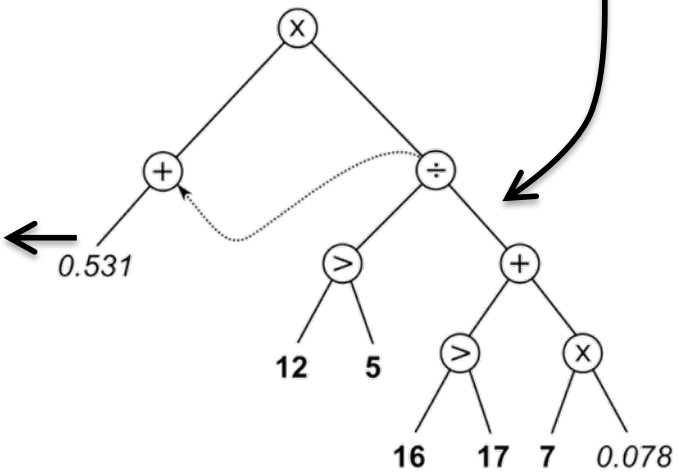
- **M Schmidt, H Lipson**, Distilling free-form natural laws from experimental data, *Science* 324:81-85, 2009.

Real World Example









This work used GP to evolve mathematical expressions that could identify the abnormal movements of Parkinson's patients

- **MA Lones, SL Smith, J Alty, S Lacy, K Possin, S Jamieson, AM Tyrrell**, Evolving Classifiers to Recognise the Movement Characteristics of Parkinson's Disease Patients, *IEEE Trans. Evolutionary Computation,* 2014.

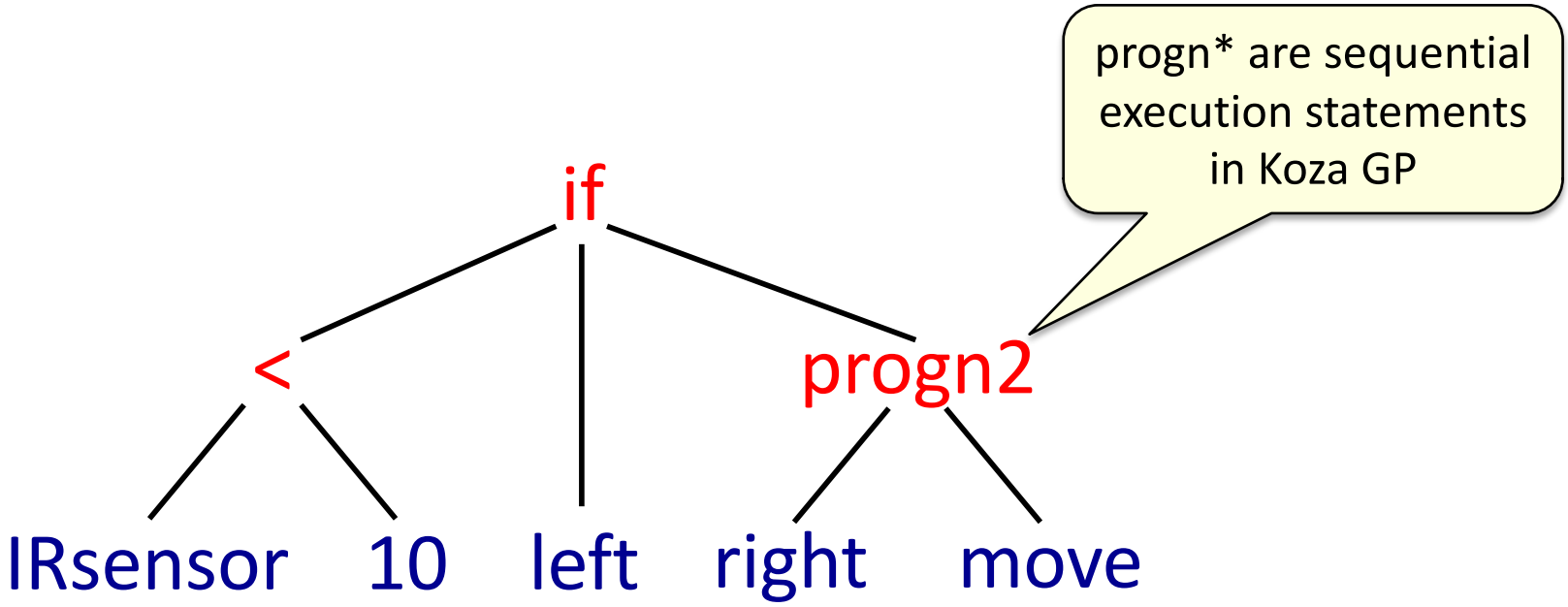
Programmatic Expressions

- Symbolic regression is a popular application of GP
  - But mathematical expressions aren't programs
  - Or, at least, not very exciting programs!

- Programmatic expressions also typically have:
  - Command sequences: `command; command; …`
  - Conditional execution: `if … then … else`
  - Iteration: `for …, do … while`
  - Memory, variables: `int i = 0 …`
  - Functions, modules: `foo = bar(x, y)`

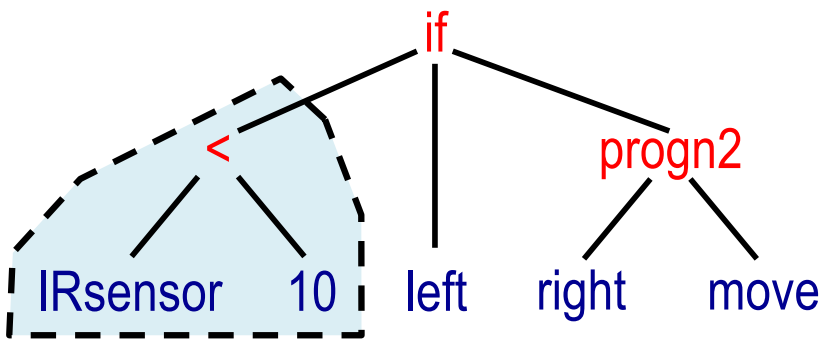
Conditional Execution

- Let's consider conditional execution (*if* statements)
- We might like to evolve something like this:

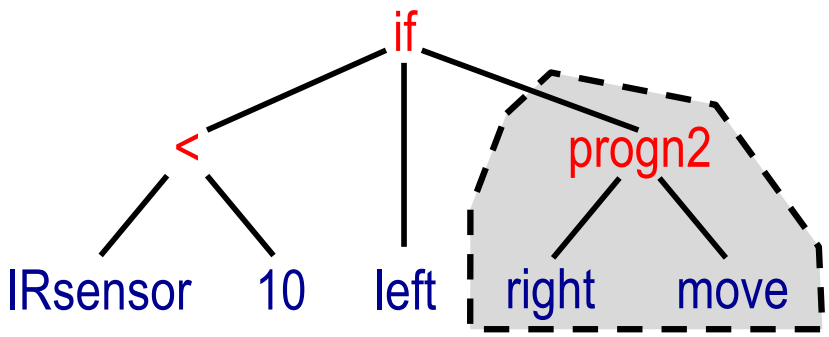


Crossover Problem

Parent 1

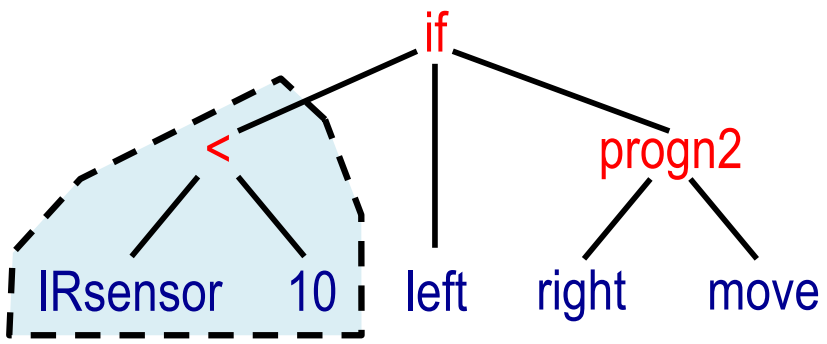


Parent 2

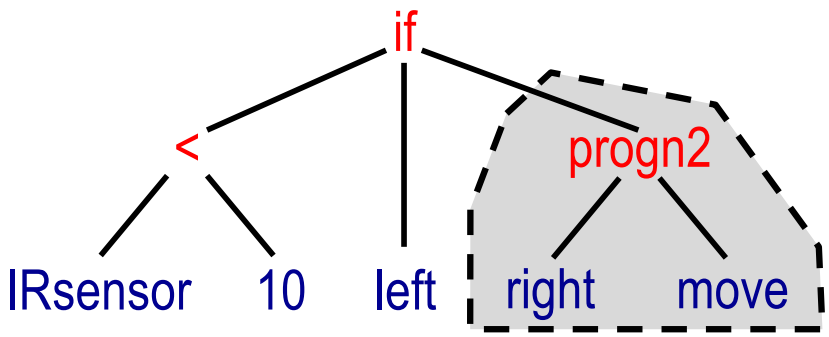


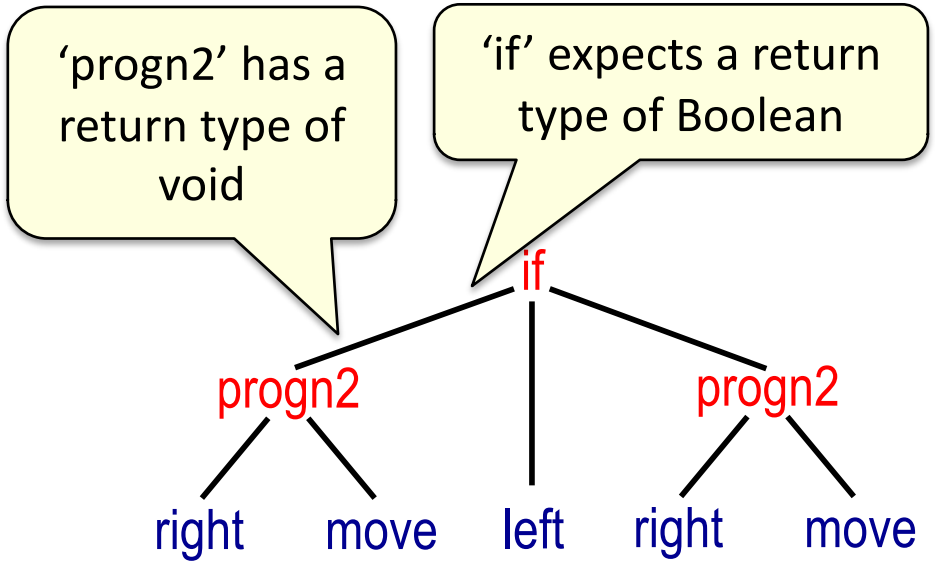
Crossover Problem

Parent 1

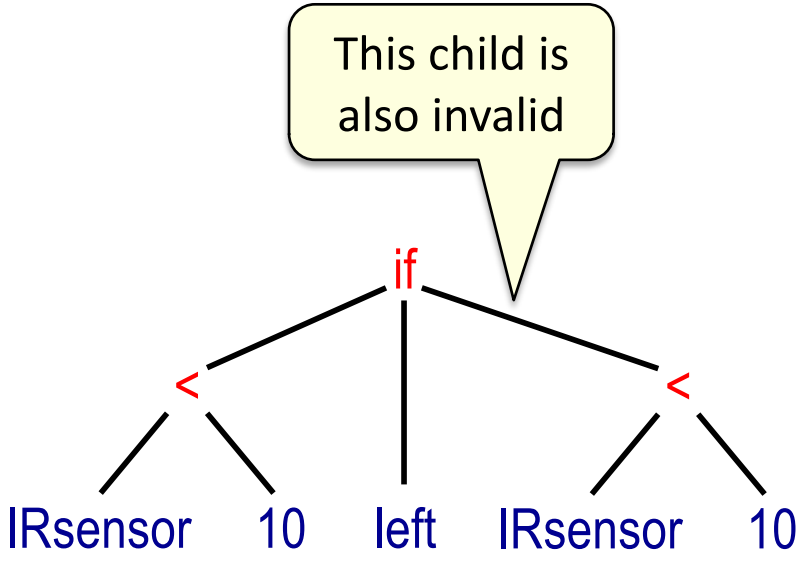


Parent 2





Child 1



Child 2

Closure

- Traditional tree-based GP requires **closure**
  - All functions must be able to do something with whatever input they may receive
  - i.e. their input types must be more general than any other function or terminal's output type

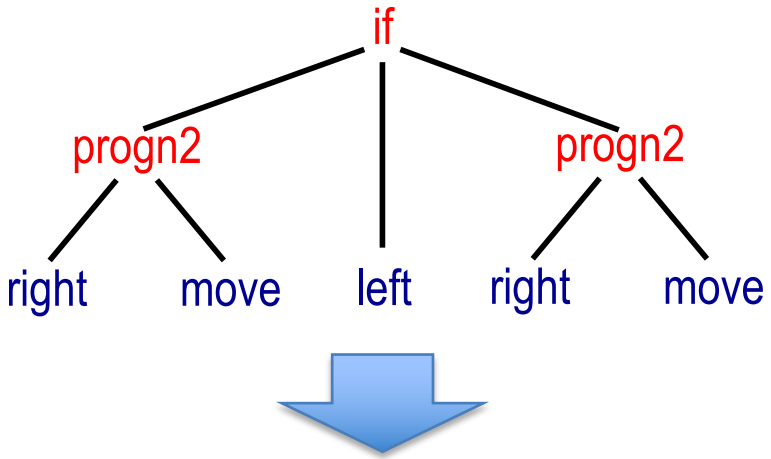
- Function set with closure – good ☺
  - { AND, OR, NAND, NOR, NOT }

- Function set without closure – bad ☹
  - { +, -, AND, OR, progn2, sin, cos }

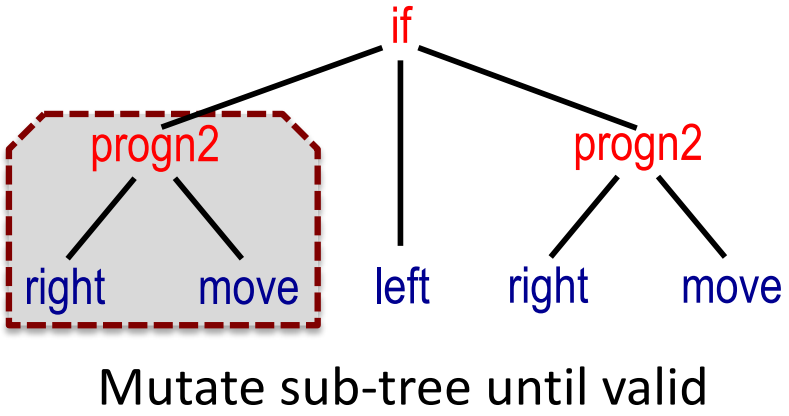
Can we avoid closure?

HERIOT WATT UNIVERSITY

- Penalise invalid solutions
  - A common approach in EAs
  - Easy to implement
  - Can lead to search space bias
  - Inefficient use of population if invalidity occurs often

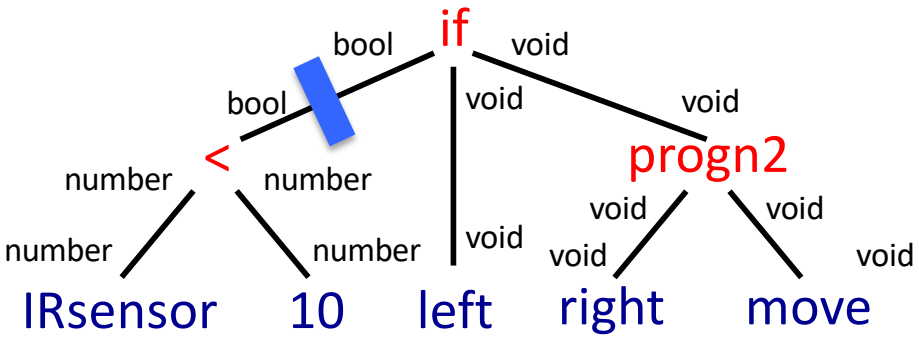


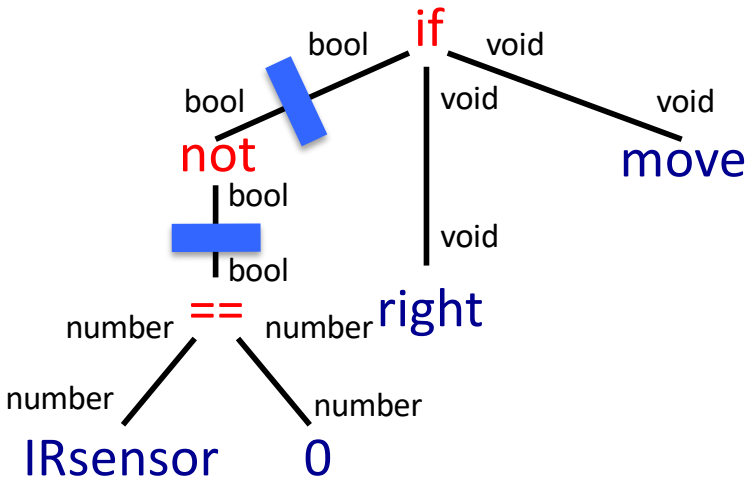
- Repair invalid solutions
  - Another common EA approach
  - Maintains population efficiency
  - Can be time consuming



Type-Constrained Operators

- Constrain initialisation and variation operators
  - By taking into account the return types of branches
  - e.g. only allow crossover points at type-compatible points
  - The preferred approach to handling mixed types in GP

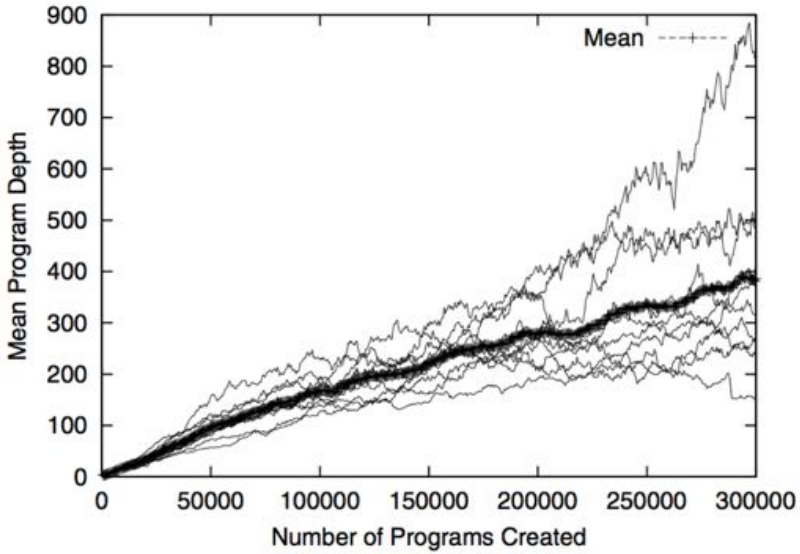


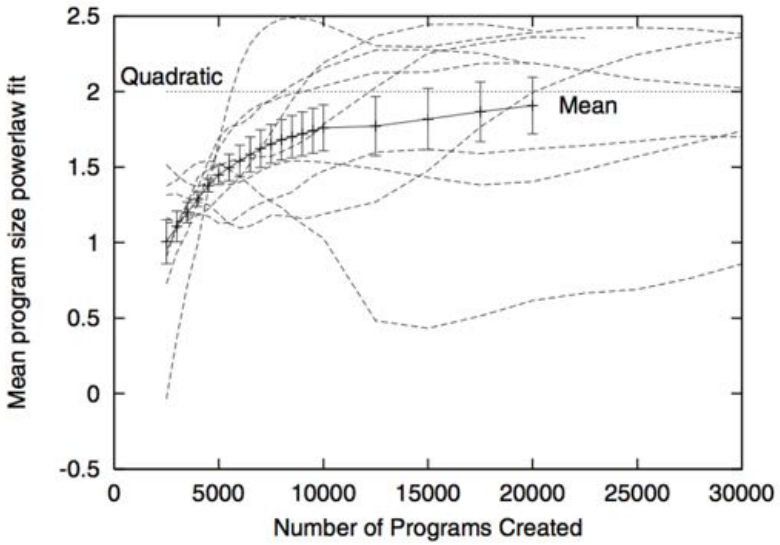


compatible crossover points

Bloat

- Bloat is another problem for genetic programming
  - Tendency for trees to grow large during evolution
  - In part, because there are more big programs than small
  - Leads to inefficient uninterpretable programs
  - Theories of why bloat occurs: See §11.3 of Field Guide





From Langdon, 2000, Quadratic Bloat in Genetic Programming

Bloat

HERIOT WATT UNIVERSITY

- But there are various ways to control bloat
  - Easiest way is to apply **depth constraints**
    e.g. only pick crossover points below depth *N* from top
  - **Parsimony pressure** involves penalising large programs
    e.g. subtract a term from their fitness in proportion to size
  - **Code editing** involves removing parts of large programs
    e.g. remove the bits that don't do anything

- If you want to know more, take a look at:
  - S. Luke and L. Panait, A Comparison of Bloat Control Methods for Genetic Programming, *Evolutionary Computation* 14(3):309-344, 2006
    http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.1580

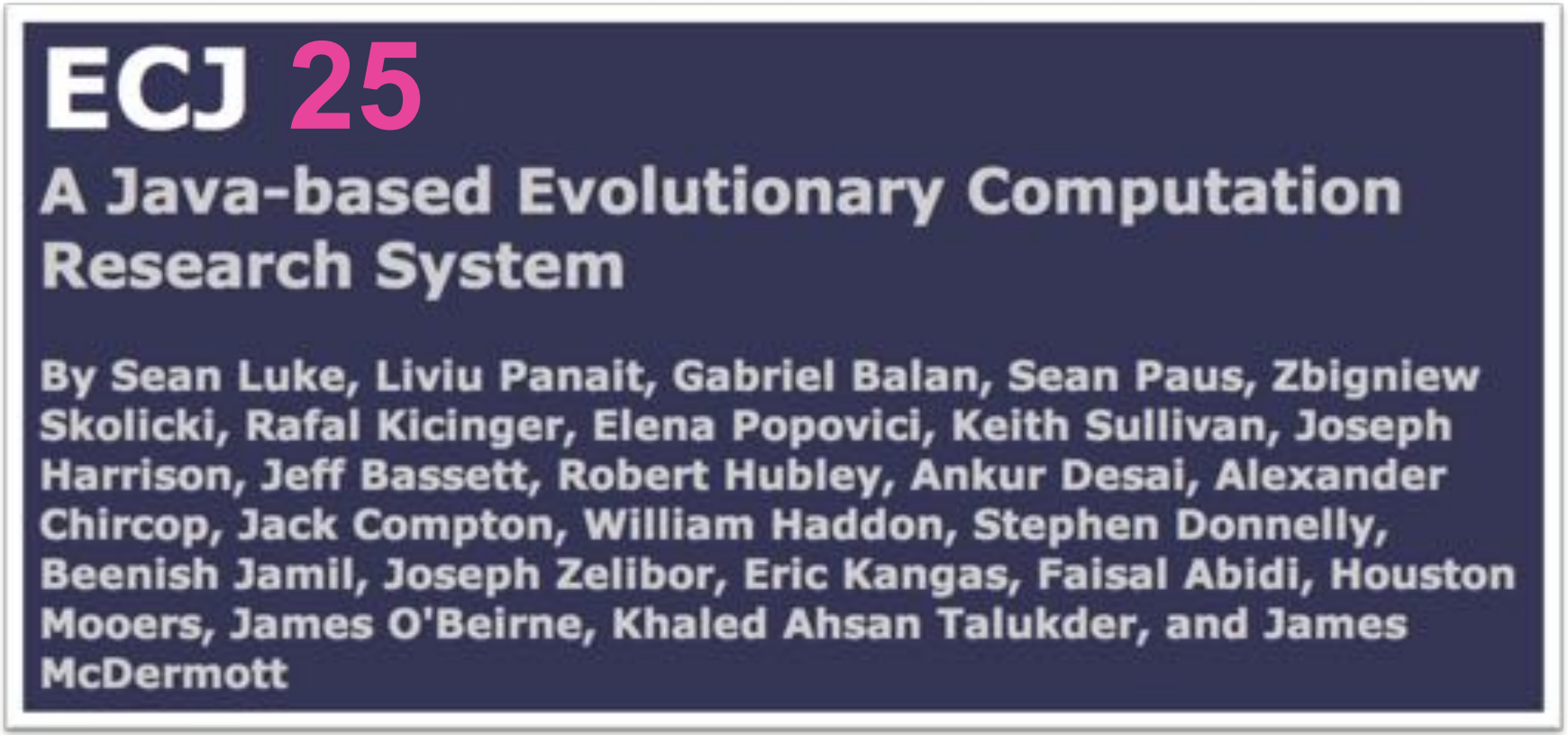
Any Questions?

Tools

- Here are some implementations of GP:
  - ECJ for Java, https://cs.gmu.edu/~eclab/projects/ecj/
  - GPLAB for Matlab, http://gplab.sourceforge.net
  - DEAP for Python, https://github.com/deap/deap
  - RGP for R (no longer supported, unfortunately): https://www.rdocumentation.org/packages/rgp/versions/0.4-1

GP Demo

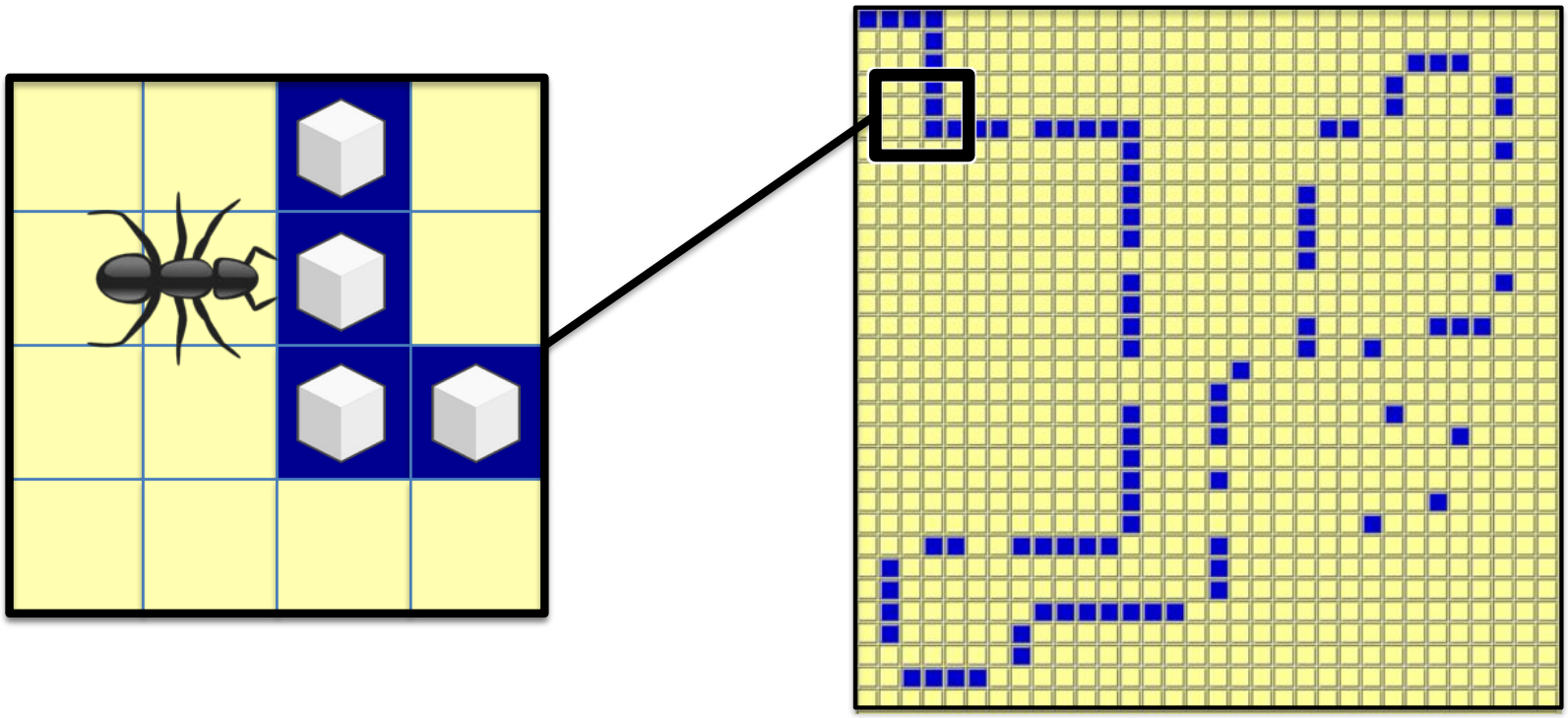
- ECJ is commonly used for GP (and other EAs):



  - Download from http://cs.gmu.edu/~eclab/projects/ecj/

GP Demo

- Santa Fe Trail Problem
  - A control problem commonly used to benchmark GP
  - Guide an ‘ant’ to ‘eat’ all the ‘food’ in minimum time

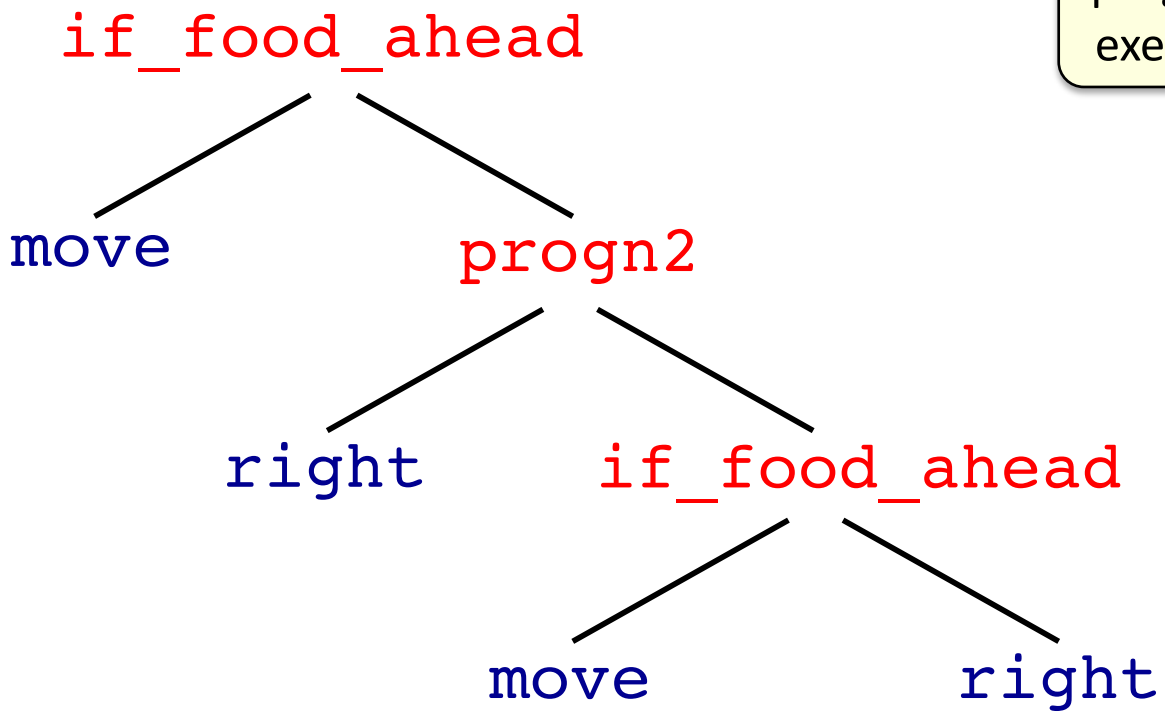


http://http://en.wikipedia.org/wiki/Santa_Fe_Trail_problem

GP Demo

- Function and terminal sets
  - Functions: { `if-food-ahead`, `progn2`, `progn3` }
  - Terminals: { `left`, `right`, `move` }

progn* are sequential execution statements



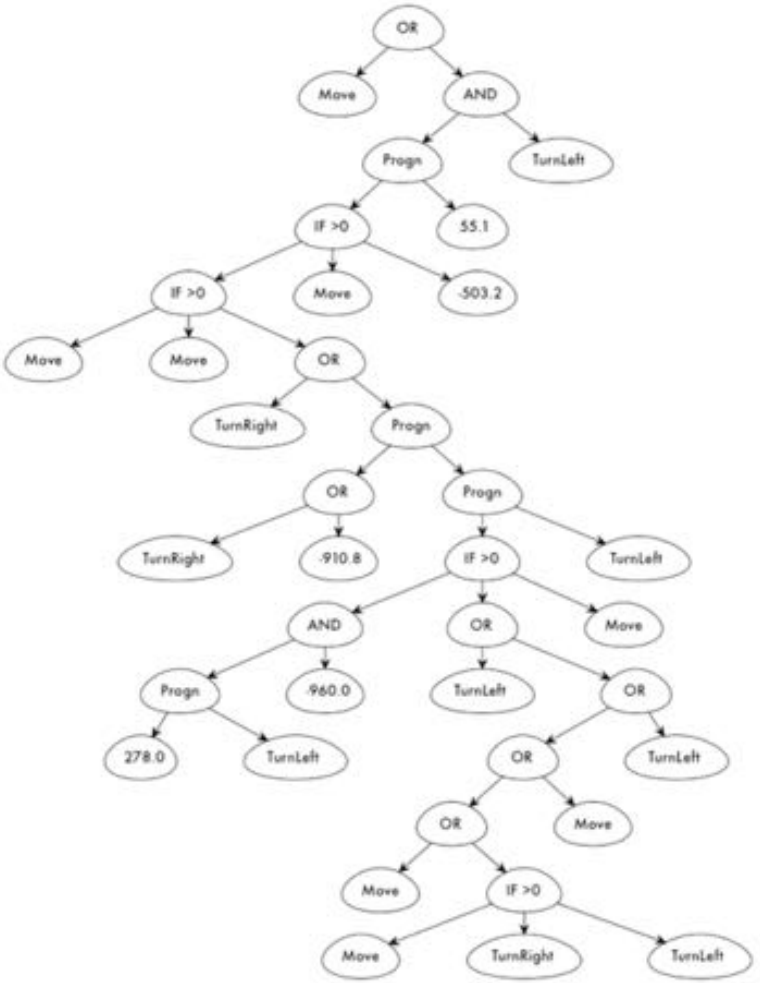
GP Demo

- Santa Fe Trail using ECJ

```
▸ java ec.Evolve -from app/ant/ant.params
▸ Generation: 50
  Fitness: Standardized=2 Hits=87
  Tree 0:
  (if-food-ahead (if-food-ahead (progn2 (if-food-ahead
       move left) move) left) (progn3 (if-food-ahead
       move left) (if-food-ahead (if-food-ahead
       (progn3 (if-food-ahead move (if-food-ahead
           left left)) (if-food-ahead move (progn3 left
            (if-food-ahead move left) (progn2 (progn3
           left move move) move)) (if-food-ahead (if-
            move move) (progn3 left move move))))
            move right)) (if-food-ahead move (progn3
  ...
```

This one 'ate' 87/89 pieces of 'food' – pretty good!

Santa Fe Solution Evolution



Generation: 1

https://www.youtube.com/watch?v=6cMXN5rGLCs

GP Demo

- Regression using ECJ
  - Target expression is the quartic polynomial $x^4+x^3+x^2+x$

```
▸ java ec.Evolve -from app/regression/erc.params
▸ Generation: 1
  Fitness: Adjusted=0.25664273 Hits=1
  Tree 0:
   (- (* (* x x) (+ (cos -0.315)
        (- x -0.870))) (* (rlog -0.707)
        (+ x x)))
  ...
  Generation: 10
  Fitness: Adjusted=1.0 Hits=20
  Tree 0:
   (+ x (* (+ (* (+ x (* x x)) x) x) x))
```

"erc" = ephemeral random constant, i.e. expressions can contain random numbers

Note the prefix notation commonly used by GP systems

Any Questions?

Things you should know

- What GP is and when you should use it
- Basics of tree-based GP:
  - Sub-tree crossover and mutation operators
  - Closure, why types can be a problem
  - Bloat: why it is a problem, methods for avoiding it
- I don’t expect you to know:
  - Details of initialisation methods
  - About the causes of bloat

Suggested Reading

- Read chapters 2-4 of "Field Guide to GP"
  - Available on Vision alongside today's lecture slides
  - Make sure you understand the basics of GP

Other things you could try out

- Get to know ECJ:
  - Install it: http://cs.gmu.edu/~eclab/projects/ecj/
  - Read the tutorials, browse the documentation
  - Play around with it
- Get to know GP:
  - Check out the GP facilities in ECJ
  - Have a look at the example problems
  - Play around with parameter files

Next Lecture

- More recent forms of GP
  - GP with graphs
  - GP with machine languages
  - GP with high level programming languages
- Genetic improvement
  - Applying GP to *existing* programs
  - A promising direction for GP