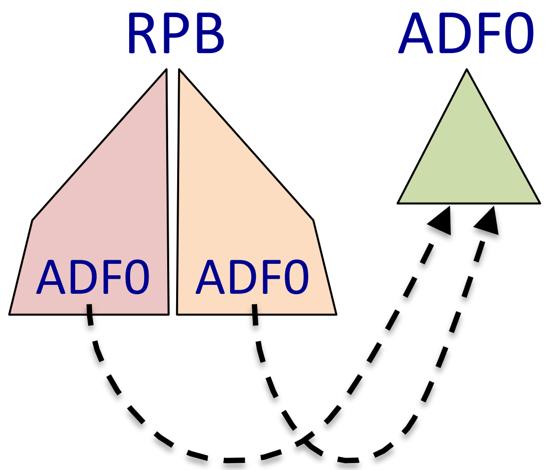
Genetic Programming: Language and Representation

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

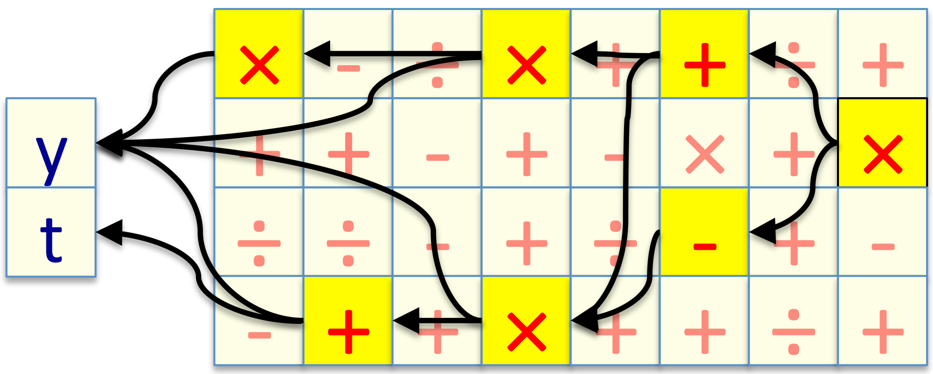
Previous Lecture

- How to make tree GP more expressive
  - Memory, iteration and modularity



- Approaches to modularity in GP
  - ADFs
  - Refactoring mutations
  - Analytical approaches

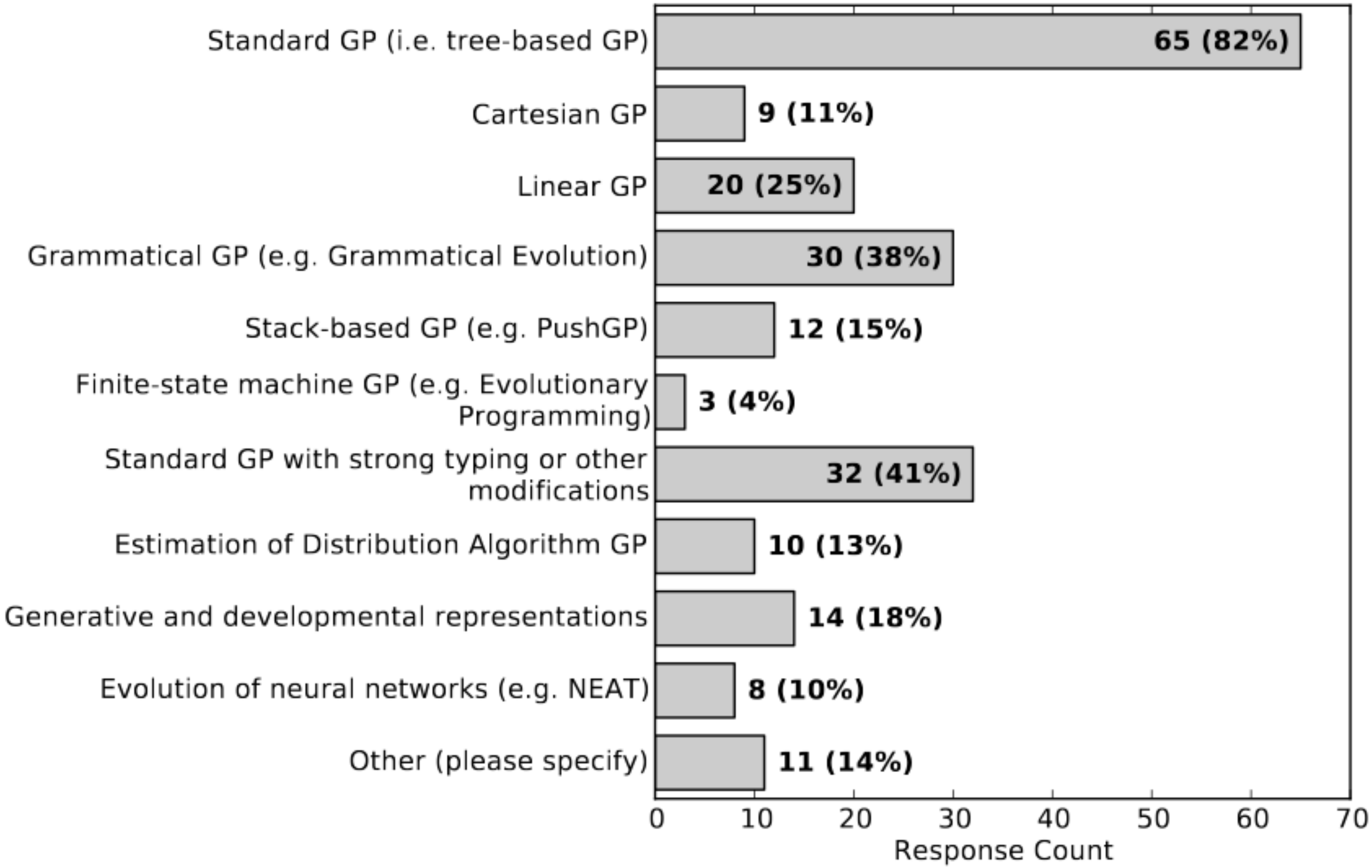
- Cartesian GP
  - Graph-based GP
  - Role of redundancy



Today's Lecture

- Other ways of representing programs
  - Linear GP: lists of instructions
  - Grammatical evolution: lists of grammar transitions
- Developmental genetic programming
  - Growing programs and other structures
  - Transitioning between representations
  - Addressing scalability
  - Computing more like biology

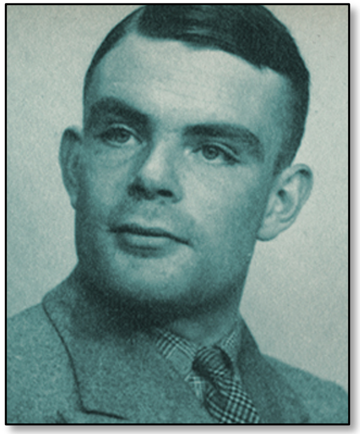
GP Community Survey



http://link.springer.com/article/10.1007%2Fs10710-012-9177-2

Real Languages?

- Tree GP doesn't use standard languages
  - Requires special interpreters
  - Challenging to integrate with existing code
  - Typically not Turing-complete
  - Language features appear a little *ad hoc*



- Other approaches do use conventional languages
  - Machine languages (e.g. x86 assembler): **Linear GP**
  - Imperative languages (e.g. C): **Grammatical evolution**
  - Functional languages (e.g. Haskell): PolyGP
  - Logic languages (e.g. Prolog): DCTG-GP

Linear GP

- Evolves lists of machine language instructions
  - So, works a lot like a genetic algorithm
  - Variation operators less likely to break syntax
  - Often no need for an interpreter or a compiler
  - However, semantic (e.g. runtime) errors are possible

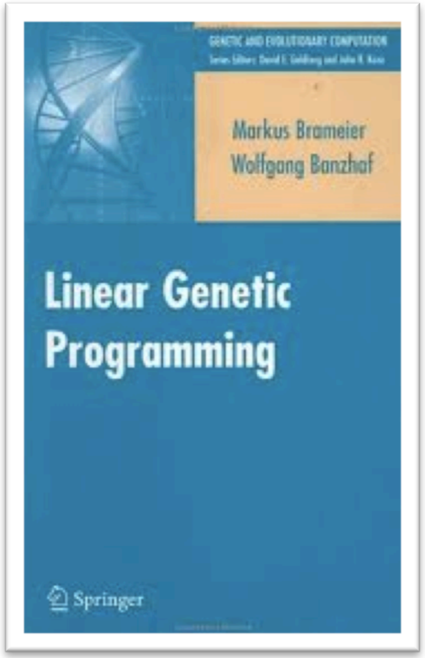
push r1	**mov r1, r2**	**mov 2, r1**	**mov r1, r2**	**pop r2**	**...**

mutate

push r1	**mov r1, r2**	**mov 2, r2**	**mov r1, r2**	**pop r2**	**...**

Linear GP

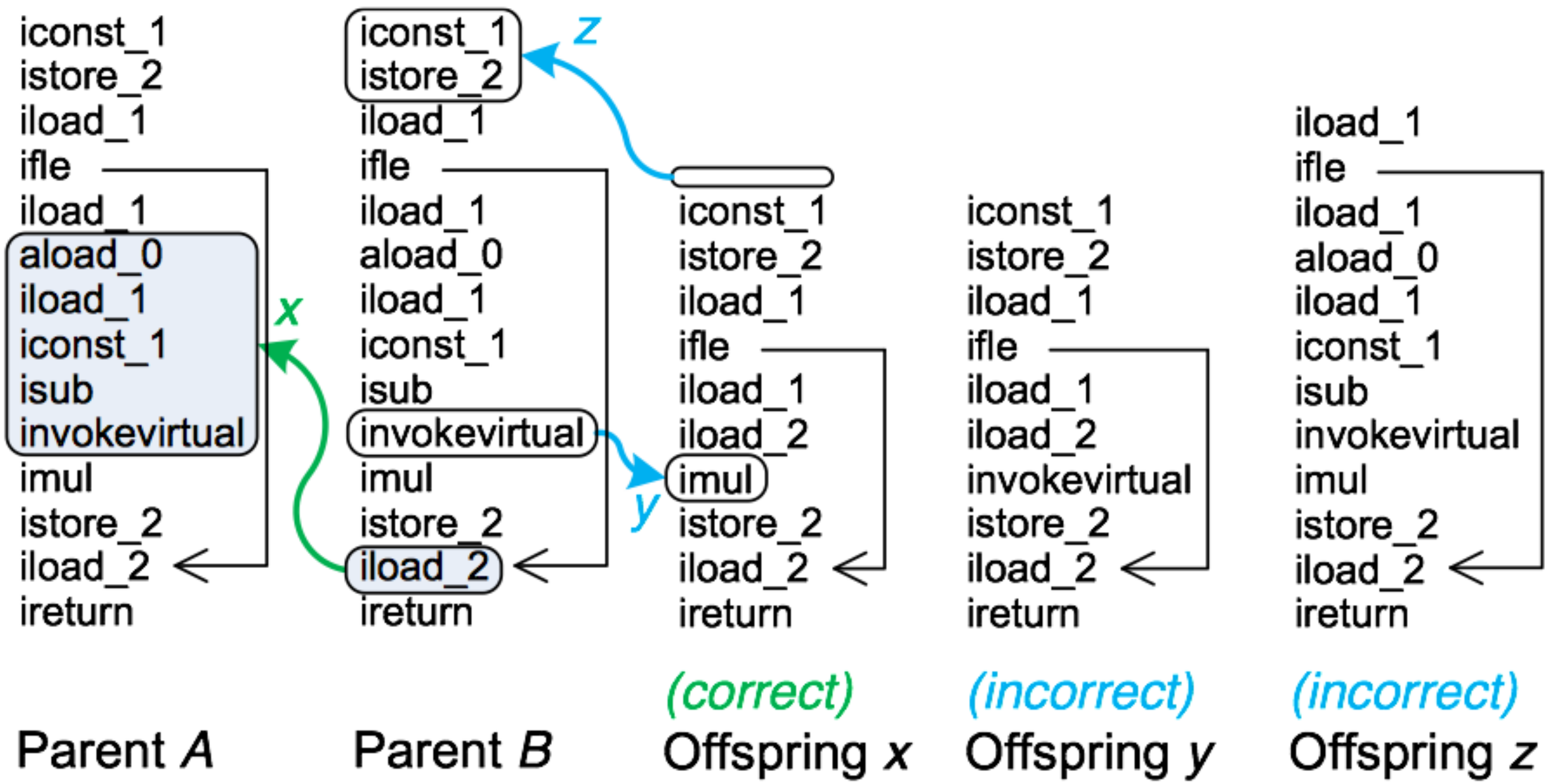
- Languages that have been targeted include:
  - Sun SPARC assembler (RISC)
  - Intel i386 assembler (CISC)
  - Java bytecode
  - CUDA instructions
  - Novel hardware languages
  - GP-specific languages, e.g. Push



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

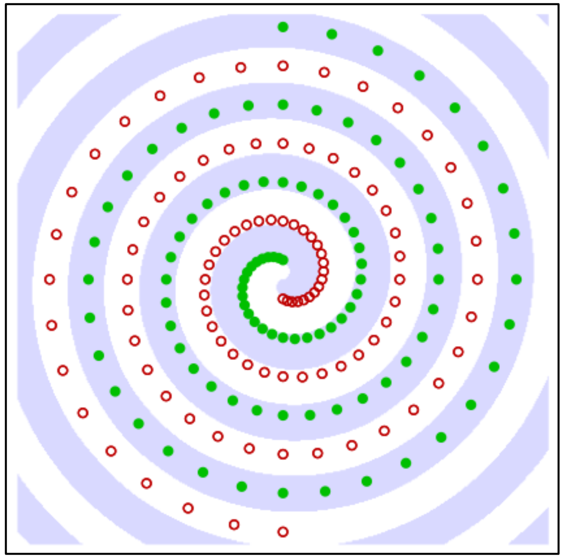
Java Bytecode Example

- FINCH [Orlov & Sipper 2010]
  - Special crossover/mutation operators preserve type-compatibility and stack depth-compatibility:



Java Bytecode Example

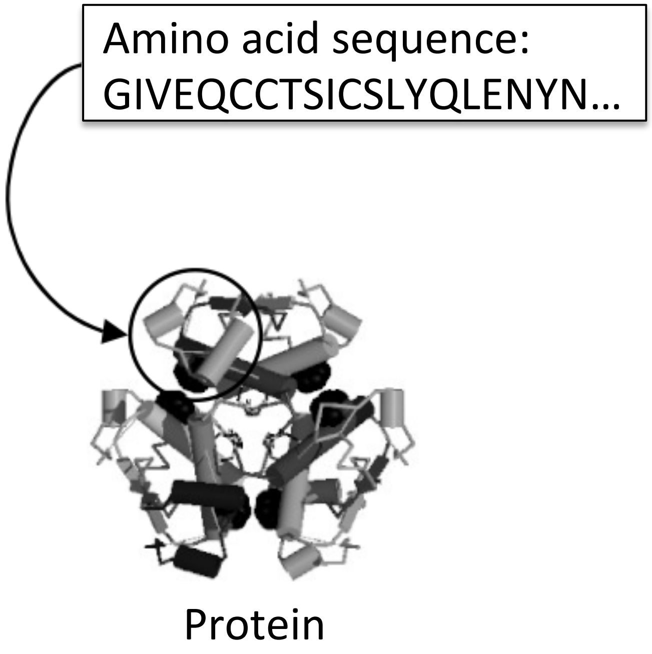
- Evolved bytecode programs can be converted to Java
  - e.g. Koza's intertwined spirals problem, which involves finding a classifier that can distinguish the points from two spirals:



```
boolean isFirst(double x, double y) {
  double a, b, c, e;
  a = Math.hypot(x, y);  e = y;
  c = Math.atan2(y, b = x) +
    -(b = Math.atan2(a, -a))
    * (c = a + a) * (b + (c = b));
  e = -b * Math.sin(c);
  if (e < -0.0056126487018762772) {
    b = Math.atan2(a, -a);
    b = Math.atan2(a * c + b, x);  b = x;
    return false;
  }
  else
    return true;
}
```

Real World Example

- Protein localisation [Heddad et al. 2004]
  - Predicting which part of a cell a protein will be located in based upon patterns in its amino acid sequence



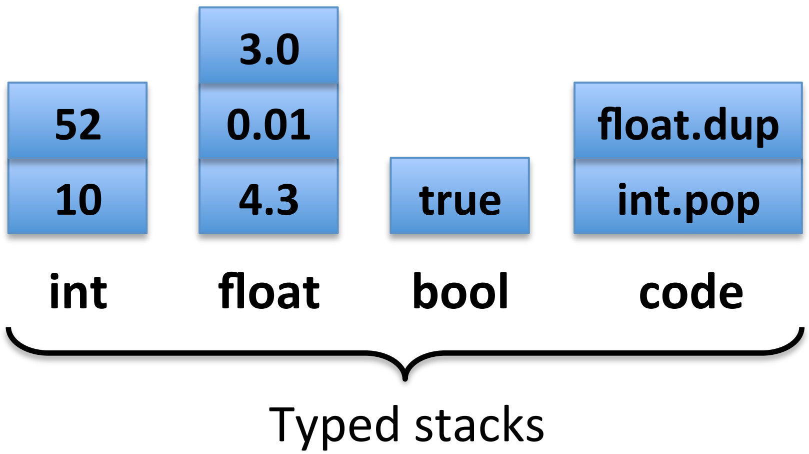

```
void gp(double *r, char *seq)
{
  r[3] = match("SA", seq);
  r[4] = match("[RK]R", seq);
  r[5] = match("[P]GP", seq);
  r[6] = match("[FWV]", seq);
  r[10] = match("[QS]", seq);

  r[3] = r[3] + r[5];
  r[8] = r[6] + r[3];
  r[0] = r[4] - r[8];
  r[0] = r[10] + r[0];
  r[0] = r[4] * r[0];
}
```

Push

- A language designed for GP [Spector 2001]
  - http://faculty.hampshire.edu/lspector/push.html
  - A stack-based language, with one stack per type
  - Handles syntax and semantics in a flexible manner
  - Code stack can be used for modularity
  - Turing complete, relatively good evolvability

```
int.+  # 52 + 10, 62 goes on int stack
code.do  # calls float.dup (duplicate)
float.*  # 3.0 * 3.0, 9.0 on float stack
...
```

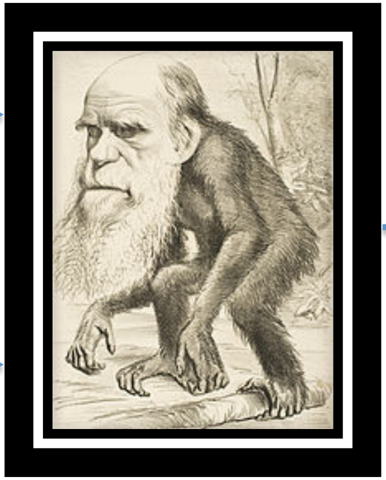


Grammatical Evolution (GE)

- Evolves lists of grammar transitions [Ryan 1998]
  - Programming language is expressed by a grammar
  - Described in Backus Naur form
  - GE can then evolve any program in that grammar

Problem we want to solve

Grammar to express solution in



List of grammar transitions that construct a program

Grammatical Evolution (GE)

<exp> ::= ^{0}<exp><op><exp> | ^{1}(<exp><op><exp>) | ^{2}<pre-op>(<exp>) | ^{3}<var>

<op> ::= ^{0}AND | ^{1}OR | ^{2}XOR

<pre-op> ::= ^{0}NOT

<var> ::= ^{0}IN1 | ^{1}IN2 | ^{2}IN3

Evolved chromosome:

10	18	41	27	6	39	21	47	13	31	19	8

Grammatical Evolution (GE)

<exp> ::= ^{0}**<exp><op><exp> | ^{1}(<exp><op><exp>) | ^{2}<pre-op>(<exp>) | ^{3}<var>**

<op> ::= ^{0}AND | ^{1}OR | ^{2}XOR

<pre-op> ::= ^{0}NOT

<var> ::= ^{0}IN1 | ^{1}IN2 | ^{2}IN3

Evolved

<exp>

For the first step, start with the most general term

10	18	41	27	6	39	21	47	13	31	19	8

10 mod 4 = 2

Number of transitions in current grammar rule

Apply transition 2 to the current term

Grammatical Evolution (GE)

<exp> ::= ^{0}<exp><op><exp> | ^{1}(<exp><op><exp>) | **^{2}<pre-op>(<exp>)** | ^{3}<var>

<op> ::= ^{0}AND | ^{1}OR | ^{2}XOR

<pre-op> ::= ^{0}NOT

<var> ::= ^{0}IN1 | ^{1}IN2 | ^{2}IN3

Evolved

For the first step, start with the most general term

<exp>

<pre-op>(<exp>)

10	18	41	27	6	39	21	47	13	31	19	8

10 mod 4 = 2

Number of transitions in current grammar rule

Apply transition 2 to the current term

Grammatical Evolution (GE)

<exp> ::= ^{0}<exp><op><exp> | ^{1}(<exp><op><exp>) | ^{2}<pre-op>(<exp>) | ^{3}<var>
<op> ::= ^{0}AND | ^{1}OR | ^{2}XOR
<pre-op> ::= ^{0}NOT
<var> ::= ^{0}IN1 | ^{1}IN2 | ^{2}IN3

Evolved

At each step, match the left-most unmatched term

<exp>
<pre-op>(<exp>)

10	18	41	27	6	39	21	47	13	31	19	8

18 mod 1 = 0

Number of transitions in current grammar rule

Apply transition 0 to the current term

Grammatical Evolution (GE)

<exp> ::= ^{0}<exp><op><exp> | ^{1}(<exp><op><exp>) | ^{2}<pre-op>(<exp>) | ^{3}<var>
<op> ::= ^{0}AND | ^{1}OR | ^{2}XOR
<pre-op> ::= ^{0}**NOT**
<var> ::= ^{0}IN1 | ^{1}IN2 | ^{2}IN3

Evolved

At each step, match the left-most unmatched term

10	18	41	27	6	39	21	47	13	31	19	8

<exp>
<pre-op>(<exp>)
NOT(<exp>)

18 mod 1 = 0

Number of transitions in current grammar rule

Apply transition 0 to the current term

Grammatical Evolution (GE)

<exp> ::= ^{0}<exp><op><exp> | **^{1}(<exp><op><exp>)** |
^{2}<pre-op>(<exp>) | ^{3}<var>
<op> ::= ^{0}AND | ^{1}OR | ^{2}XOR
<pre-op> ::= ^{0}NOT
<var> ::= ^{0}IN1 | ^{1}IN2 | ^{2}IN3

Evolved chromosome:

10	18	41	27	6	39	21	47	13	31	19	8

41 mod 4 = 1

<exp>
<pre-op>(<exp>)
NOT(**<exp>**)
NOT(**<exp><op><exp>**)
...
NOT(IN1 AND (IN2 OR IN3)

Real World Example

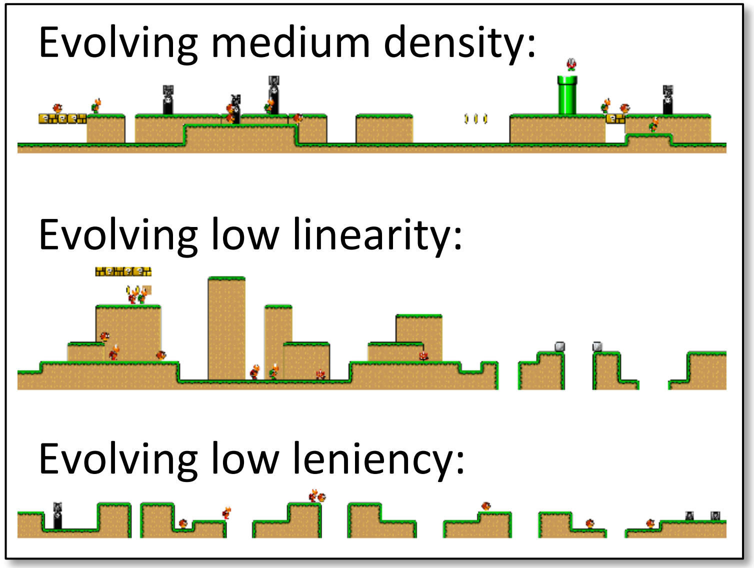
- Evolving Super Mario levels [Shaker et al. 2012]

```
<level> ::= <chunks>  <enemy>
<chunks> ::= <chunk> |<chunk> <chunks>
<chunk> ::= gap(<x>,<y>, <w_g>,<w_before>,<w_after>)
   | platform(<x>,<y>,<w>)
   | hill(<x>,<y>,<w>)
   | cannon_hill(<x>,<y>,<h>,<w_before>,<w_after>)
   | tube_hill(<x>,<y>,<h>,<w_before>,<w_after>)
   | coin(<x>,<y>,<w_c>)
   | cannon(<x>,<y>,<h>,<w_before>,<w_after>)
   | tube(<x>,<y>,<h>,<w_before>,<w_after>)
   | <boxes>

<boxes> ::=  <box_type> (<x>,<y>)^2 | ...
   | <box_type> (<x>,<y>)^6

<box_type> ::= block_coin | block_powerup
   | rock_coin   | rock_empty

<enemy> ::= (koopa | goompa)(<x>) ^2 | ...
   | (koopa | goompa)(<x>) ^10
<x> :: = [5..95] <y> ::= [3..5]
```

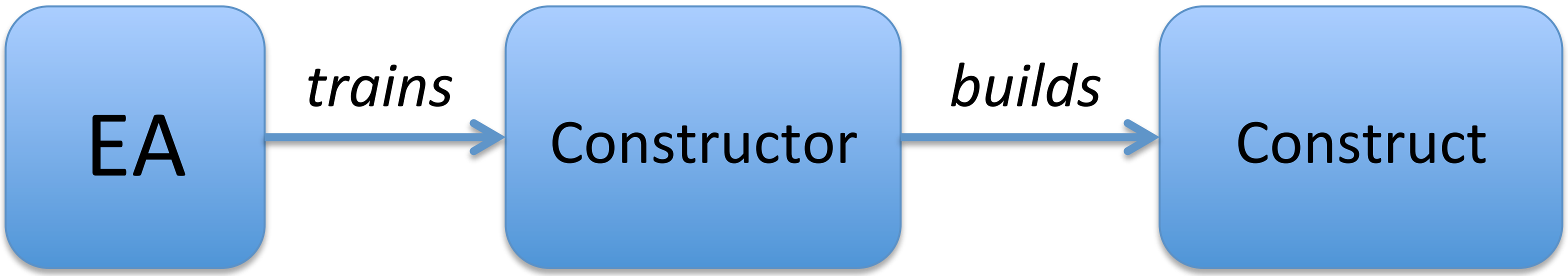


Grammatical Evolution (GE)

- A flexible and expressive approach
  - Since grammars can be defined for all languages
  - Programs have been evolved in C, for example
- Though modern languages are problematic
  - Complicated syntax and large APIs
  - Typically only a subset of a language is used
- Some concerns about evolvability
  - Sensitive to mutations at the left of a chromosome
  - These have a large effect upon the final expression

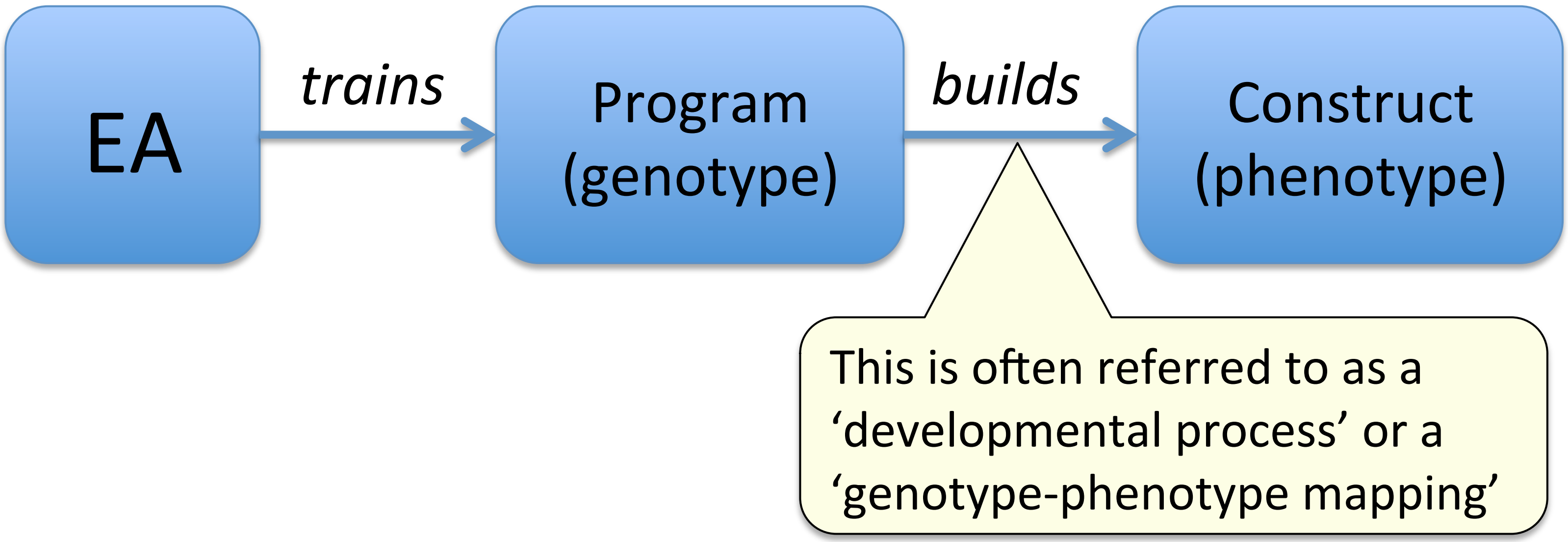
Indirect Encodings

- Evolve something that constructs something else
  - Grammatical evolution does this, in a limited sense
    i.e. evolve grammar transitions that construct a program
  - You also saw it with genetic algorithms
    e.g. rule sets that construct bin packing solutions



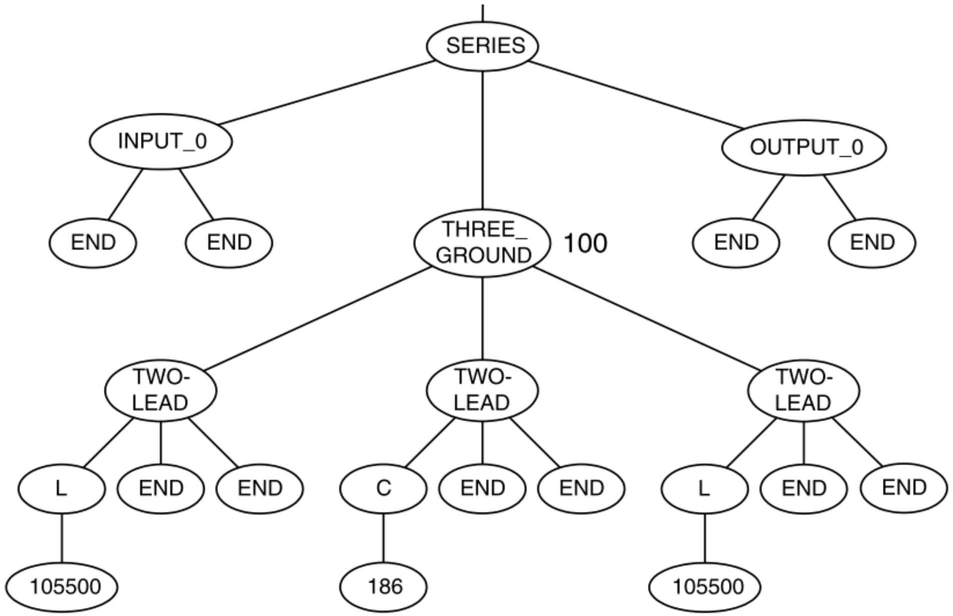
Developmental GP

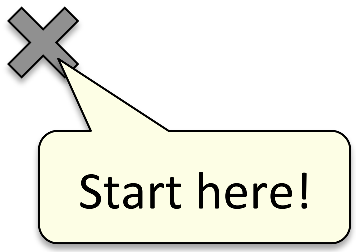
- Developmental GPs also use indirect encodings
  - Evolve programs that construct other entities, e.g. structures, circuits, or other programs
  - Often inspired by biological models of development
  - Often more scalable, able to solve bigger problems



Koza's Developmental GP

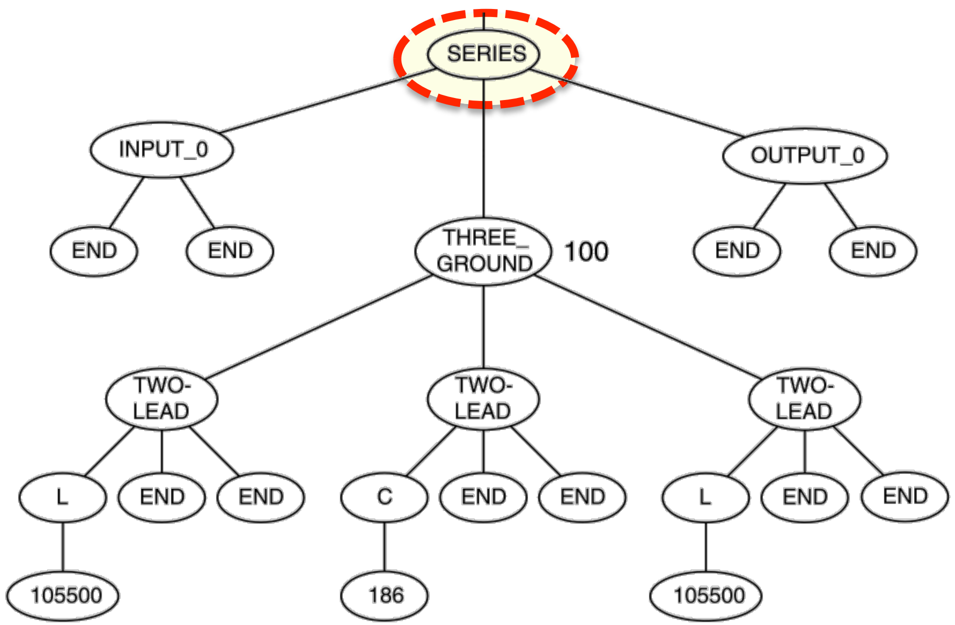
- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping



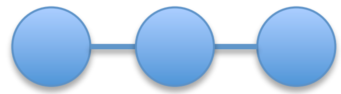


Koza's Developmental GP

- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping

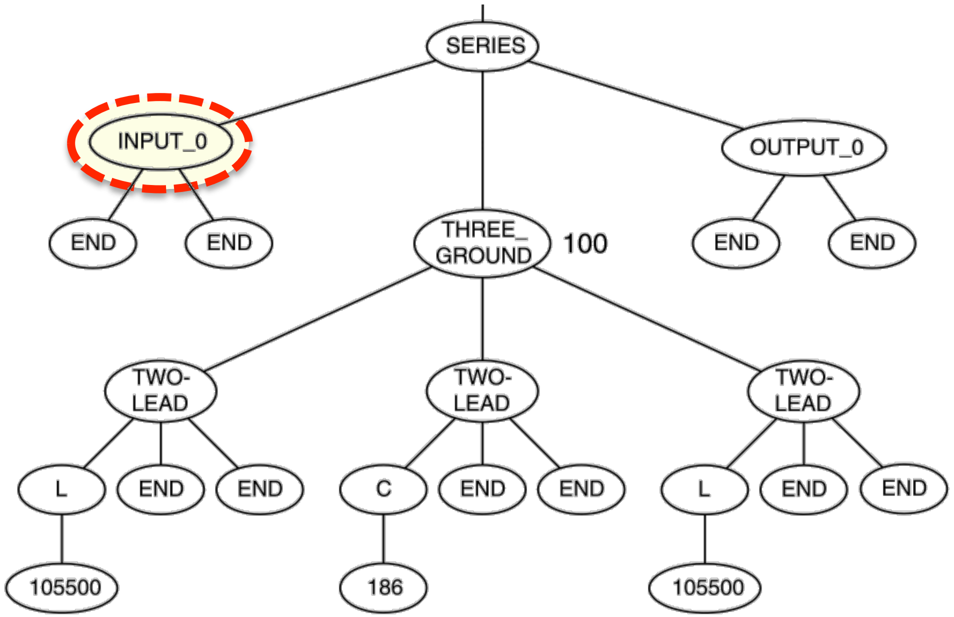


Create three nodes in series

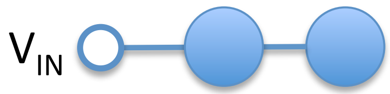


Koza's Developmental GP

- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping

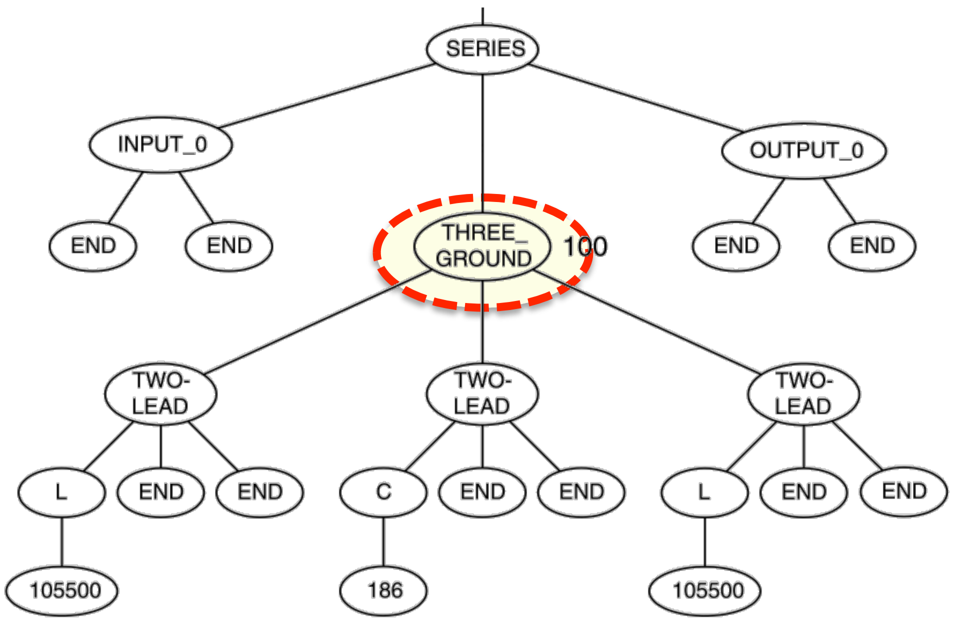


Instantiate first node

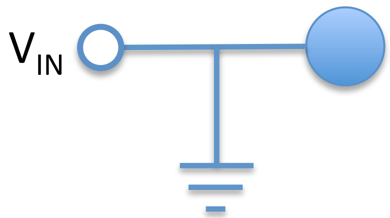


Koza's Developmental GP

- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping

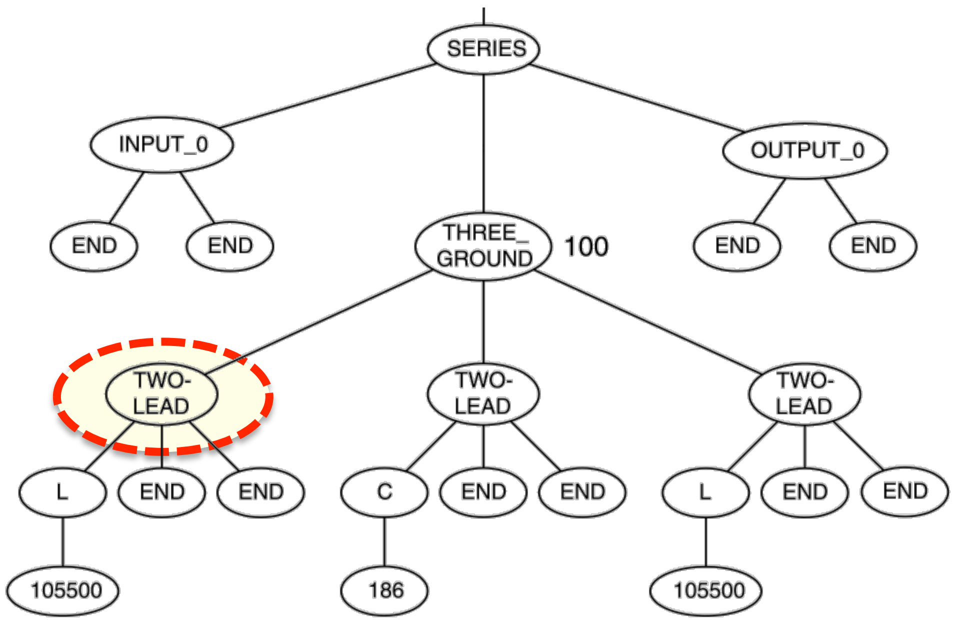


Add three-way link to ground

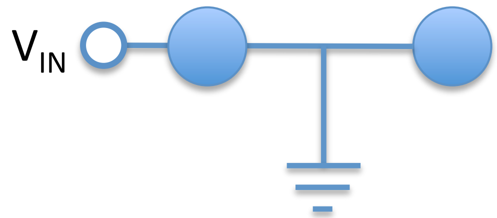


Koza's Developmental GP

- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping

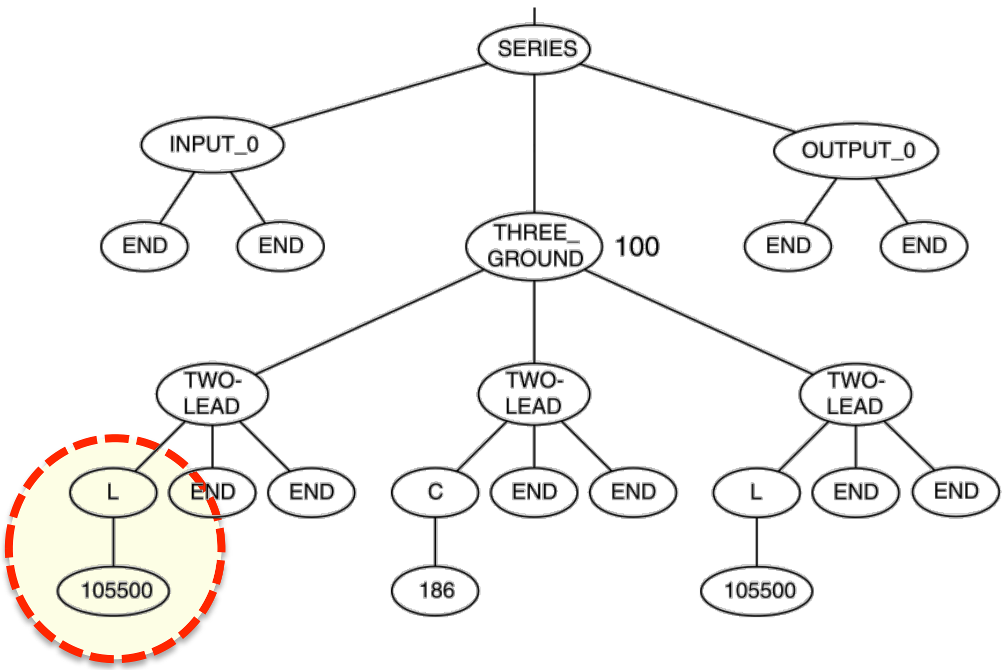


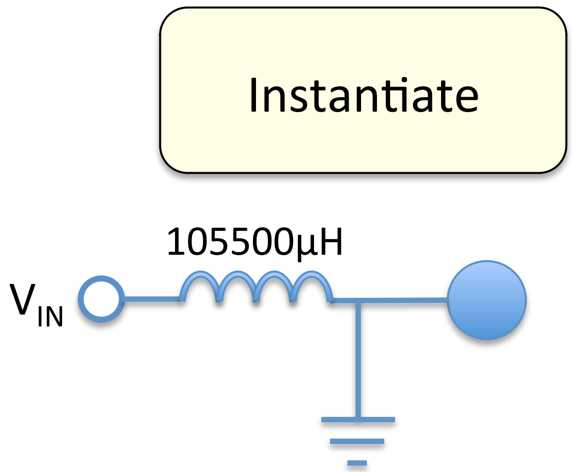
Add a node with two leads



Koza's Developmental GP

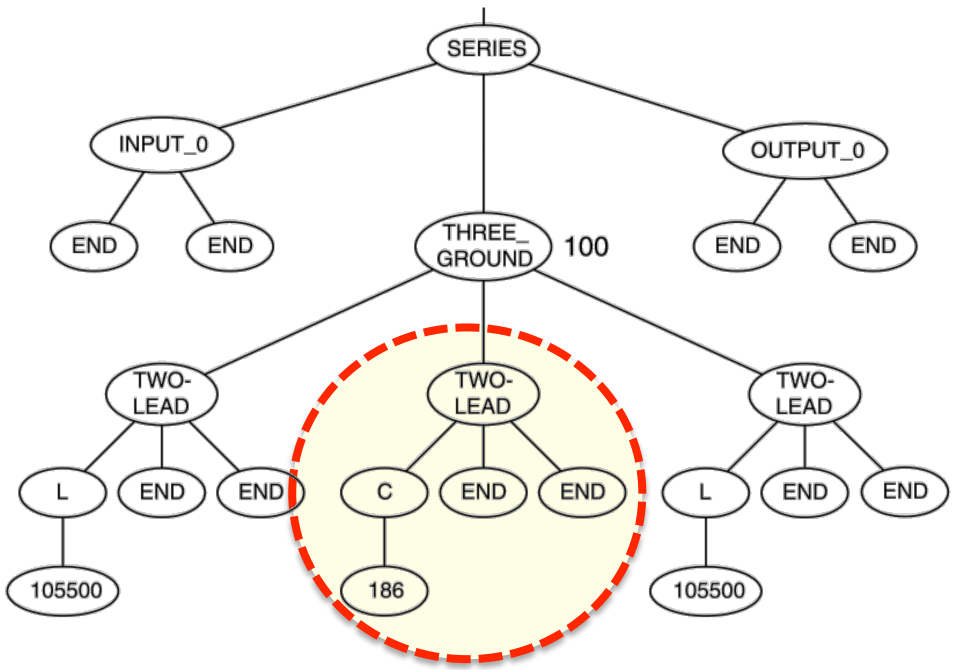
- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping



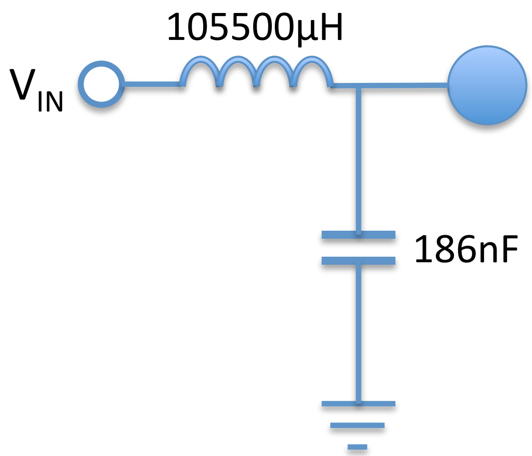


Koza's Developmental GP

- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping

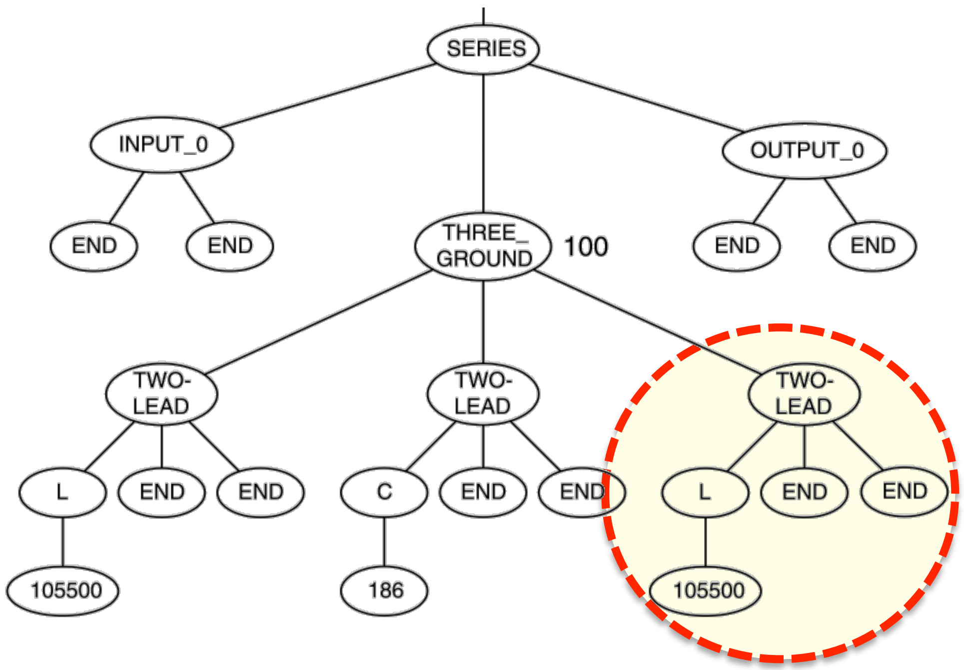


Insert and Instantiate

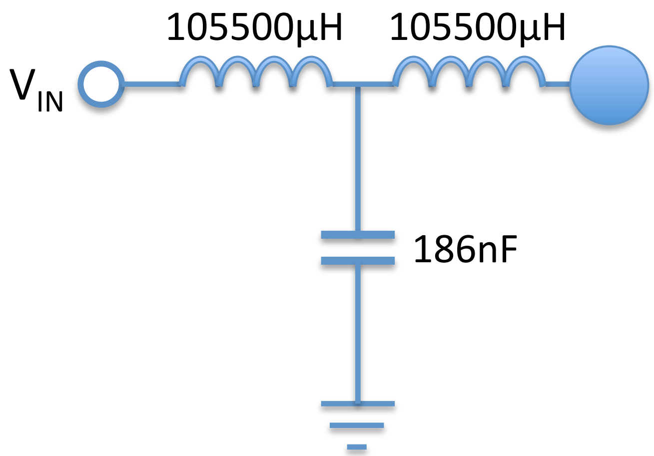


Koza's Developmental GP

- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping

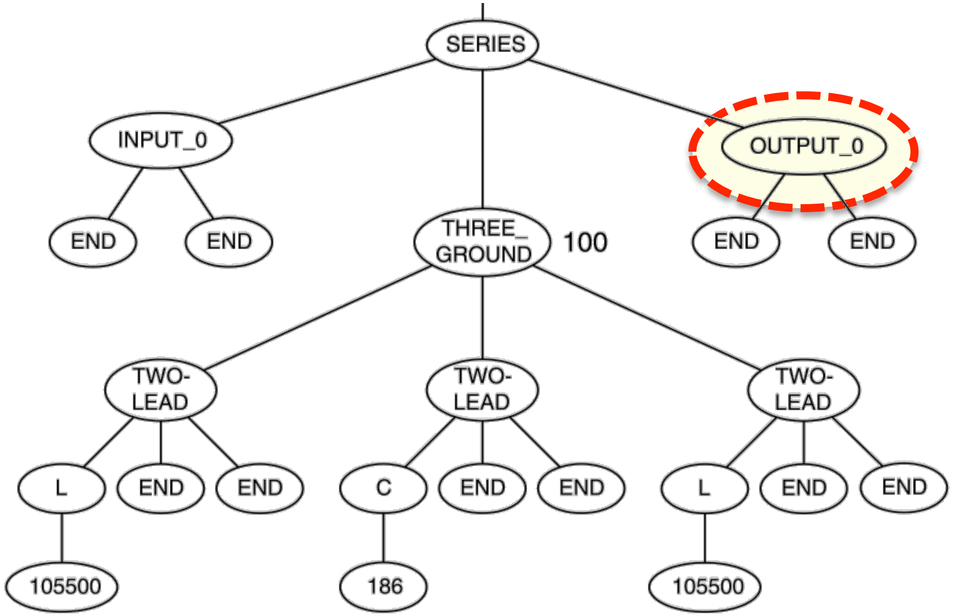


Insert and Instantiate

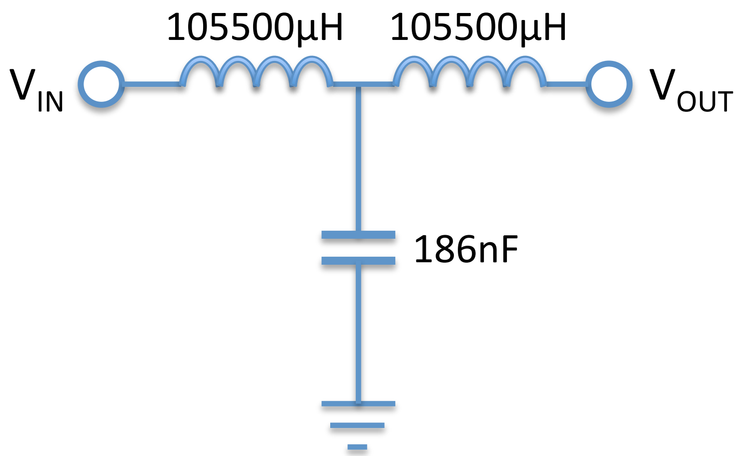


Koza's Developmental GP

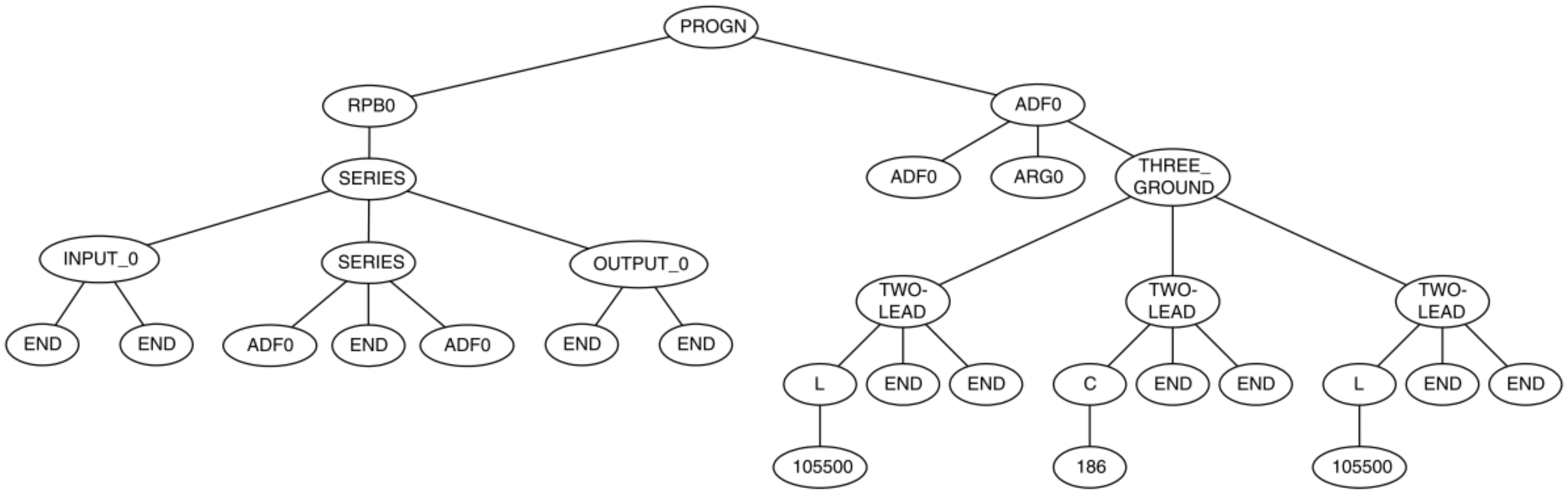
- Using tree-based GP to design circuits [Koza 2003]
  - This approach uses a GP tree to construct a graph
  - An early example of a developmental mapping

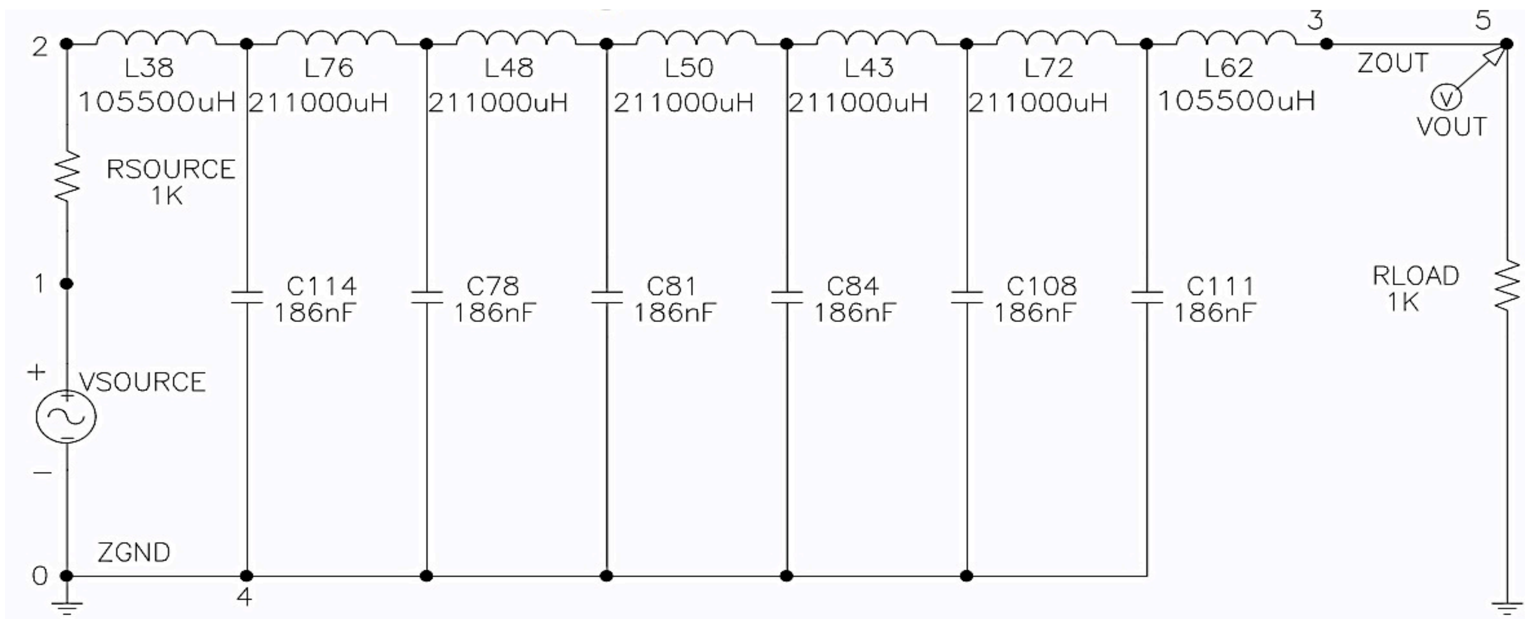


Instantiate

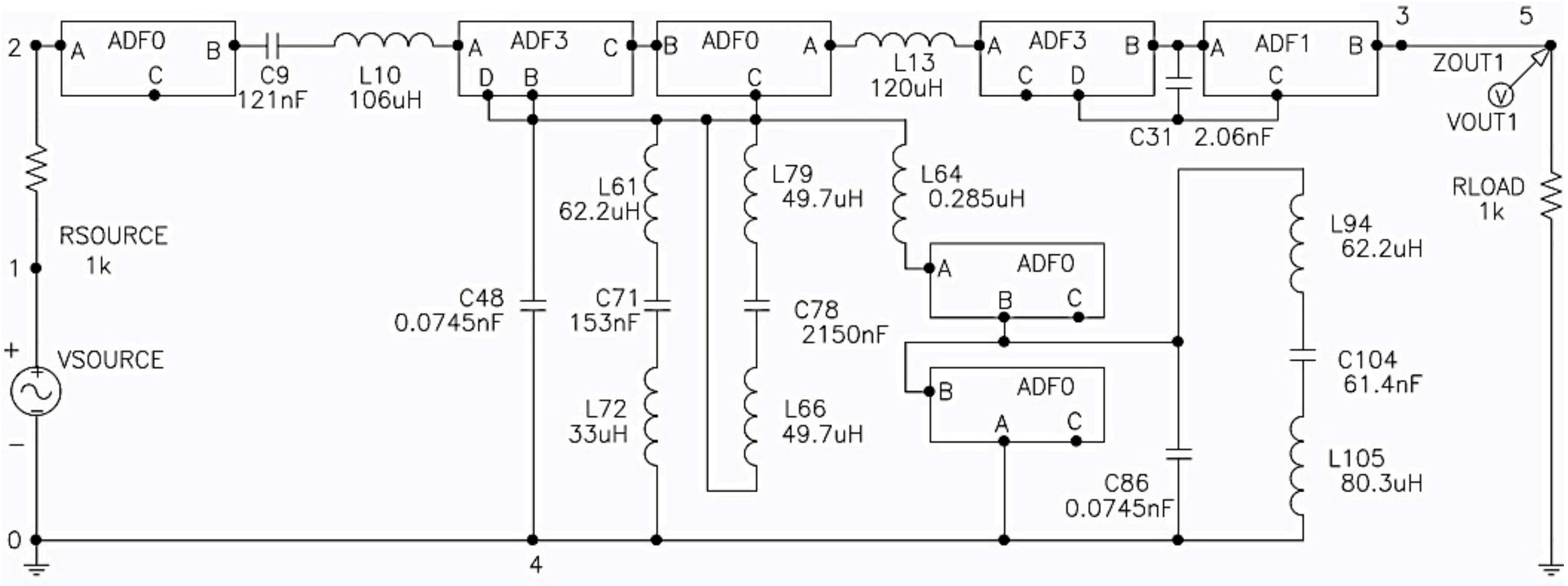


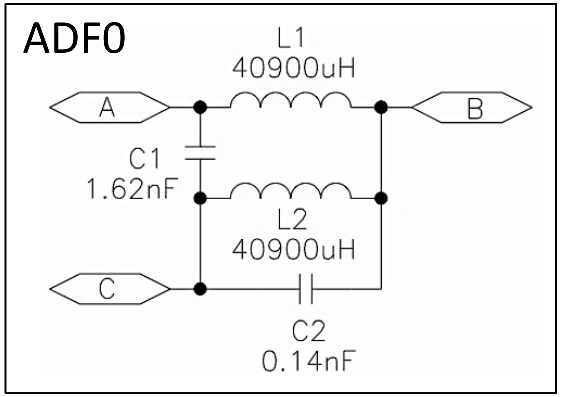
Modularity

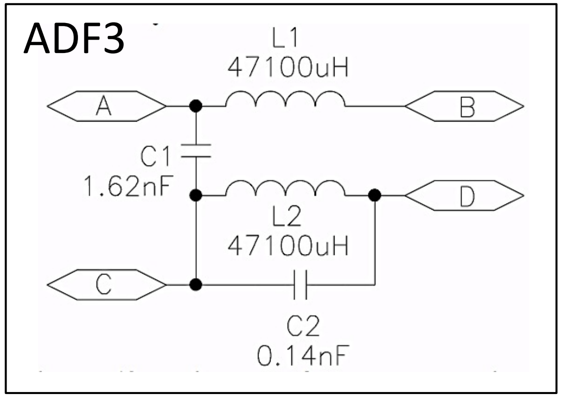

PROGN
RPB0
ADF0
SERIES
ADF0
ARG0
THREE_ GROUND
INPUT_0
SERIES
OUTPUT_0
TWO- LEAD
TWO- LEAD
TWO- LEAD
END
END
ADF0
END
ADF0
END
END
L
END
END
C
END
END
L
END
END
105500
186
105500


2
L38
105500uH
L76
211000uH
L48
211000uH
L50
211000uH
L43
211000uH
L72
211000uH
L62
105500uH
3
5
ZOUT
VOUT
RSOURCE
1K
1
C114
186nF
C78
186nF
C81
186nF
C84
186nF
C108
186nF
C111
186nF
RLOAD
1K
+
VSOURCE
–
ZGND
0
4

Koza's Developmental GP



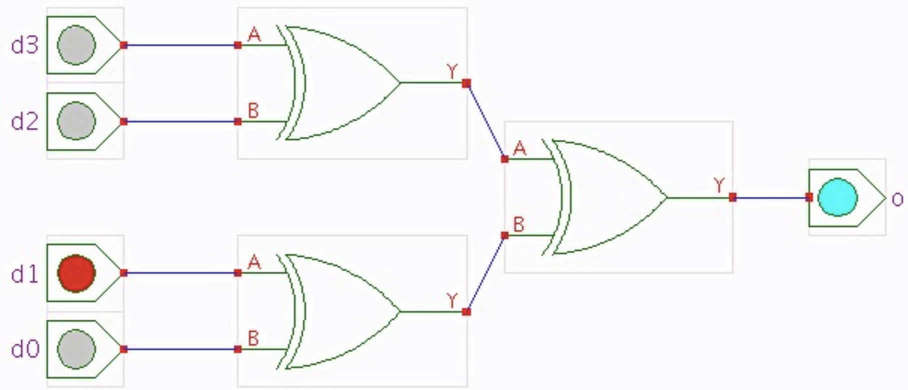




Scalability

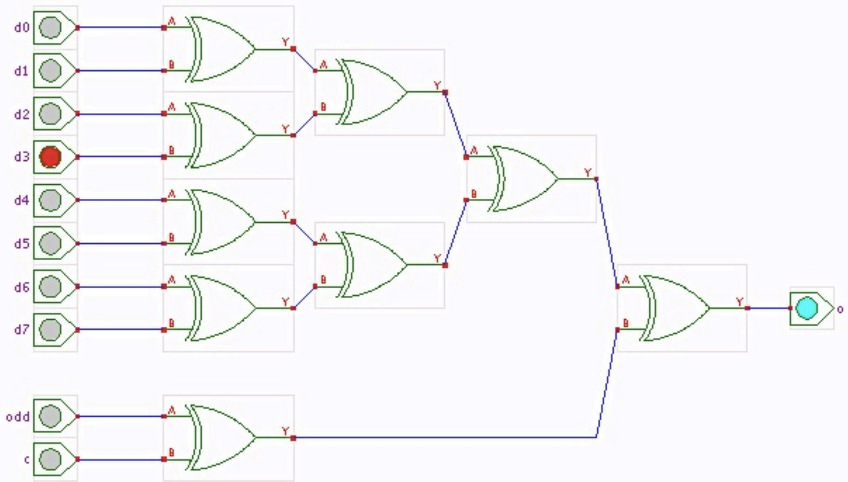
- Scalability is a problem for genetic programming
  - Search space grows exponentially with program size
  - Development allows programs to be compressed
  - Particularly when solutions have repetitive structure:

Even-4-parity circuit:



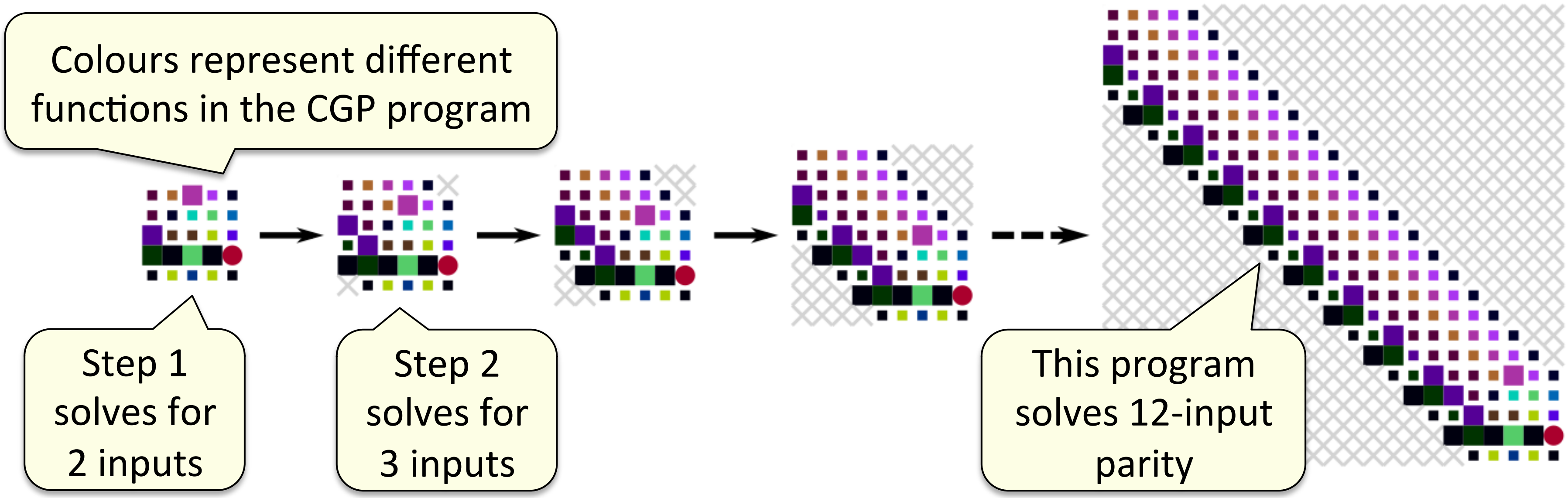
http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/index.html

Even-8-parity circuit:



Example: Self-Modifying CGP

- CGP programs that modify themselves [Harding'11]
  - Contains normal functions and self-modifying functions
    - E.g. delete nodes, duplicate nodes, change connection …
  - A new program is created at each program iteration
    - Can generate general solutions, e.g. even-n-parity circuit:



Autoconstructive Evolution

- Programs build their own offspring [Spector 2010]
  - i.e. they include code to copy and modify themselves
  - Based on the idea that programs can learn good patterns or regions of variation, for instance mutation hotspots
  - e.g. Autopush: autoconstructive evolution using Push
  - Modifies the program's "code" stack to construct a child

```
((integer.stackdepth (boolean.and code.map)) (integer.sub
(integer.stackdepth (integer.sub (in (code.wrap (code.if (code.noop)
Boolean.fromfloat (2) integer.fromfloat) (code.rand integer.rot)
exec.swap code.append integer.mult))))))
```

Append to the code stack

Generate some random code

Developmental GP

- A way of transitioning between representations
  - e.g. using a tree to represent a graph
  - Choose an evolvable representation, rather than one that is required by the problem domain
- Also a way of achieving scalability
  - Genotype space can be smaller than phenotype space
  - Human analogy: 30,000 genes encode 100 trillion cells
- And potentially for achieving complexity
  - Not limited to repetitive modular structures

Things you should know

- Linear GP and grammatical evolution
  - How programs are represented
  - When they should be used, strengths and weaknesses
- Developmental GP
  - The meaning of indirect/developmental representation
  - Advantages over non-developmental GPs
- I don’t expect you to know
  - Low-level details, e.g. translating a GE expression
  - Push, the different kinds of developmental GP

What's Next?

- Biological representations of computation
  - Most of the approaches discussed so far use conventional representations of computation
  - Conventional representations are fragile
  - Developmental GP hints at the potential of representing computation in unconventional ways
  - *This is the topic of the next few lectures…*
  - Next week: Cellular Automata
    - Complex behaviour from simple systems

Things to try out

- Have a look at Grammatical Evolution
  - `ec.gp.ge` package in ECJ
  - http://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf
    - See section 5.3
- Have a look at Push
  - `ec.gp.push` package in ECJ
  - http://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf
    - See section 5.4

Bibliography

- S. Harding et al, SMCGP2: self modifying Cartesian genetic programming in two dimensions, GECCO 2011 http://www.cartesiangp.co.uk/papers/gecco2011a-harding.pdf
- A. Heddad et al, Evolving Regular Expression-based Sequence Classifiers for Protein Nuclear Localisation, EvoBIO 2004 http://www.sbc.su.se/~maccallr/publications/heddad-evobio2004.pdf
- J. Koza et al., The Importance of Reuse and Development in Evolvable Hardware, 2003 NASA/DoD Conference on Evolvable Hardware, 2003 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.5365
- M. Orlov and M. Sipper, Flight of the FINCH Through the Java Wilderness, IEEE TEvC 15(2), 2011 http://finch.cs.bgu.ac.il/pubs/finch.pdf
- C. Ryan, Grammatical Evolution: Evolving Programs for an Arbitrary Language, EuroGP 1998, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.7697
- N. Shaker et al, Evolving Levels for Super Mario Bros Using Grammatical Evolution, CIG2012, http://noorshaker.com/docs/ELGE.pdf
- L. Spector, Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems, GPTP VIII, 2010. http://faculty.hampshire.edu/lspector/pubs/spector-gptp10-preprint.pdf