Lecture 1:

2 books:
- R. Poli et al, A Field Guide to Genetic Programming
- S. Luke, Essentials of Metaheuristics

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

Genetic Programming (GP)

- In a nutshell, 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 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

Why use evolutionary algorithms?

- Good at solving global optimisation problems
- Flexible in how solutions are represented
- However, focus on EAs is in part historical
- Other optimisers may, in principle, be used

Also a slightly iffy bio-inspired argument

- Biological systems are evolved
- Biological systems are, in a sense, complex computers
- Therefore complex computations can be evolved
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

Problem we want to solve → Evolutionary ‘black box’ → Program that solves the problem

◊ Often portrayed as a form of automatic innovation
  ▶ “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
Genetic Programming (GP)

◊ There are a number of varieties of GP
  ▶ You’ll see lots of these over the coming lectures

◊ They differ in how they represent programs
  ▶ Syntax: control structures, modules, language
  ▶ 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)

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.
Invented by John Koza
- Also invented the scratch card
- Earliest successful form of GP
- (Though arguably not the first)
- Still the most widely used form

Programs are represented by trees
- Also known as syntax trees or parse trees
- Nodes are sampled from a function set
- Leaves are sampled from a terminal set

Parse Trees

\[
\begin{align*}
&\text{\(\frac{y}{5}\times t\)} \\
&\frac{\text{sin}\theta}{\text{IRsensor} < 10} \left\{ \begin{array}{ll}
\text{left} & \text{move} \\
\text{right} & \text{move}
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
&\text{progn3 WRITE 2} \\
&\text{READ 0} \\
&\text{progn2 READ 4} \\
&\text{WRITE 2} \\
&\text{WRITE 9} \\
&\text{WRITE i}
\end{align*}
\]
To create a mathematical expression

- Function set = \{ +, -, \times, \div, \sin, \cos \}
- Terminal set = \{ y, t, \theta, \text{constant} \}

- e.g. \((y/5) \times (t \sin \theta)\):

```
\begin{tikzpicture}
    \node (x) {$\times$} child { node {\div} child { node {y} } child { node {5} } } child { node {$\times$} child { node {t} } child { node {$\sin \theta$} } };
\end{tikzpicture}
```

- Other initialisation methods exist
  - E.g. ramped half-and-half: see Field Guide!

\[ \theta \]
Recombination

Sub-tree crossover:

Child 1

Child 2

Mutation

Sub-tree mutation:
Sub-tree mutation:

Parent

$\times$
$
\div$
$\times$

$y$
$5$
$\sin$
$	heta$

Child

$\times$
$
\div$
$-\ 
$
$y$
$5$
$\sin$
$	heta$
$3$
$\theta$

Point mutation (less disruptive):

Parent

$\times$
$
\div$
$\times$

$y$
$5$
$\sin$
$	heta$

Child

$\times$
$
\div$
$\times$

$y$
$5$
$\sin$
$	heta$
Symbolic Regression

- Fitting a mathematical expression to data
  - A common use of genetic programming
  - Useful when little is known about the generating function

Curve Fitting Example

https://www.youtube.com/watch?v=37D3QpFvrgs
Symbolic regression is a popular application of GP
- But mathematical expressions aren't programs
- Or at least, not very interesting 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)
Santa Fe Trail Problem
- A control problem commonly used to benchmark GP
- Guide an ‘ant’ to ‘eat’ all the ‘food’ in minimum time

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

progn* are sequential execution statements

```
if_foodAhead
   move
   progn2
       right
       if_foodAhead
           move
           right
```
Crossover Problem

Parent 1

if
    <
    IRsensor 10 left right move

progn2

Child 1

Child 2

Parent 2

if
    <
    IRsensor 10 left right move

progn2

‘progn2’ has a return type of void

‘if’ expects a return type of Boolean
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?

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

Mutate sub-tree until valid
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
Lecture 2:

- A variant of GP [Montana, 1995]
  - Builds upon the idea of type constraints
  - Every terminal and function is assigned a type
  - Provides scope for type hierarchies
  - Also supports generic functions with flexible types
  - Paper discusses mixing scalars, vectors and matrices:
    - http://dx.doi.org/10.1109/GECCO.2015.7248149

- Bloat is a big problem for genetic programming
  - Tendency for trees to grow large during evolution
  - In standard GP, growth has quadratic complexity
  - Leads to inefficient uninterpretable programs

  ![Graph showing bloat](image)

  From: Angeline, 2003. Quadratic Bloat in Genetic Programming

- Many theories for why bloat occurs:
  - There are more big programs than small programs
  - GP operators tend to explore larger trees (operator bias)
  - Programs protect themselves with non-functional code
  - [See §11.3 of “Field Guide”]

- There are various ways to control bloat
  - Easiest way is to apply depth constraints
    - e.g. only pick crossover points below depth N
  - Pare 笈 constraints involve penalizing 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
  - An extra objective can be added to a multiobjective EA
    - e.g. second objective of minimizing number of nodes

  For more info, read [Luke, 2006]
  - http://dx.doi.org/10.1109/GECCO.2015.7248149
A framework for evolutionary computing

- Supports common evolutionary algorithms
  - GAs, evolution strategies, GP, PSO, ...
  - You just need to implement a Problem subclass

- Individual components are configurable
  - Using parameter files
  - Representations, operators, selection mechanisms

- Relatively easy to evolve non-standard things
  - New representations subclass Individual and Species
  - New variation operators subclass BreedingPipeline

Expressiveness

This is the capacity for a program representation to express different kinds of behaviours.

For genetic programming, you can’t evolve a program if you can’t express it.

In practice, there is often a trade-off between expressiveness and evolvability.
Adding Memory

- There are various ways of adding memory to GP
  - However, in practice these are not widely used

- Consider the approach used by Astro Teller [1994]
  - This introduces a memory, of width M:
    
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | M |
    |
  - And 2 new functions:
    - `READ(X)` – read value from memory location X
    - `WRITE(Y, X)` – write value Y to memory location X

Consequence: Usual problems of accessing memory that is not there – crash!

Data Structures

- Using addressable memory is problematic
- An alternative is to use data structures
  - These are less expressive than ‘full memory’
  - But less susceptible to bad mutations
  - e.g. using a stack:

```
progn3
  |  |
PUSH  PUSH  POP
  |  |
  2  +  2  POP
  |
```
Other Memory Approaches

- “Soft memory” [Poli et al 2009]
  - Blends new assignments with old memories
  - Intended to be more evolvable
  - [http://cswww.essex.ac.uk/staff/poli/papers/PoliMcPheeCitiCraneJAEA2009.pdf](http://cswww.essex.ac.uk/staff/poli/papers/PoliMcPheeCitiCraneJAEA2009.pdf)

  - Memory is shared amongst the population
  - Allows programs to communicate

Adding Loops

- This is possible, but...
  - Requires caution!
  - Halting Problem: can’t predict termination
  - Need to use constraints to prevent infinite loops
  - (e.g. max iterations)
  - Or use a time-out during solution evaluation

[http://www.cooperbem.com](http://www.cooperbem.com)
Recap

◊ It is possible to add syntactic features to GP:
   ▶ Types ✓
   ▶ Memory ✓
   ▶ Loops ✓

◊ But this has consequences:
   ▶ More complex initialisation and variation operators
   ▶ More constraints during evolution
   ▶ Possible biases within the search landscape

▶ *So, only use them when necessary*
A program is defined as a forest of trees
- One "result producing branch" (RPB), i.e. main()
- One of more ADFs, each with one or more arguments

![Diagram]

ADF0 must be defined before a GP run
- Both the number of ADFs, and the number of arguments for each ADF must be specified in advance
- This is a prominent limitation of ADFs

Some heuristics for choosing these values:
- *A priori* knowledge of problem decomposition
- Over-specification, i.e. more than will ever be needed
- Affordable capacity, since ADFs tend to increase the complexity and hence the execution time of programs

Good things about ADFs
- They can reduce overall program size
- They make it easy to solve modular problems
- They have been used to solve hard problems
  
  See Koza's books!

Bad things about ADFs
- They can increase program complexity
- Incorrect parameter settings hinder evolution
- Modular dependencies may hinder evolution...
Modular Dependencies

- This is also seen in software engineering
  - Two routines make use of shared code
  - The routines’ requirements diverge
  - The shared code must be copied and modified
  - i.e. it is no longer shared code!

Removing Dependencies

- Koza introduced a similar mechanism to GP
  - Called ‘case splitting’
  - Part of a group of ‘architecture altering operators’
  - Basically mutation operators that refactor code
Another group of methods for achieving modularity
- These attempt to identify useful code blocks
- And then turn them into modules
- For potential use elsewhere in evolving programs

Adaptive Representation through Learning (ARL)
- A successful form of analytical modularity
- First, it identifies programs with improved fitness
- Then extracts the most-executed code blocks
- Placing these in a library shared between programs

Recap

- It is possible to add syntactic features to GP:
  - Types ✔
  - Memory ✔
  - Loops ✔
  - Modularity ✔

- But this has consequences:
  - More complex initialisation and variation operators
  - More constraints during evolution
  - Possible biases within the search landscape
  - So, only use them when necessary
Some limitations of GP are due to using trees:

There are advantages to using graphs instead

- Instant reuse!
There are a number of graph-based GPs
  > PADO, PDGP, GNP, CGP, ...

Cartesian GP (CGP) is the best known [Miller]
  > Functions are arranged on a Cartesian grid

Other notable properties of CGP
  > Constrained grid limits program size (no bloat!)
  > Mutation can connect/disconnect nodes
  > Disconnected nodes are a form of redundancy
  > Redundancy has evolutionary advantages
Other notable properties of CGP

- Constrained grid limits program size (no bloat!)
- Mutation can connect/disconnect nodes
- Disconnected nodes are a form of redundancy
- Redundancy has evolutionary advantages

Lecture 3:
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 ...
```
**FINCH** [Orlov & Sipper 2010]

- Special crossover/mutation operators preserve type-compatibility and stack depth-compatibility:

```
iconst_1
istore_2
ifle
iload_1
iload_1
aload_0
iload_1
iconst_1
isub
invokevirtual
imul
istore_2
iload_2
ireturn

iconst_1
istore_2
ifle
iload_1
iload_1
aload_0
iload_1
iconst_1
isub
invokevirtual
imul
istore_2
iload_2
ireturn
```

Parent A | Parent B | Offspring x | Offspring y | Offspring z
---------|---------|-------------|-------------|-------------

**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)

- 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

![Diagram showing EA, Constructor, and Construct]

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

![Diagram showing EA, Program (genotype), and Construct (phenotype) with a note about 'developmental process' or 'genotype-phenotype mapping']
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](http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/index.html)

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:

```
Colours represent different functions in the CGP program

Step 1 solves for 2 inputs
Step 2 solves for 3 inputs
This program solves 12-input parity
```
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

Lecture 4:
Conventional
- Centralised
- Top-down
- Halting
- Static
- Exact
- Fragile
- Synchronous

Biological
- Distributed
- Bottom-up (emergent)
- Ongoing
- Dynamical
- Inexact
- Robust
- Asynchronous

See Mitchell, “Biological Computation,” 2010
http://www.santafe.edu/media/workingpapers/10-09-021.pdf

What is a cellular automaton?
- A model of “emergence”
  - complex behaviour emerges from interactions between simple rules
  - Emergent behaviour occurs widely in biological systems
- Originally developed by Ulam & von Neumann in the 1940s/50s
- Popularised by John Conway’s work on the ‘Game of Life’ in the 1970s
- Significant later work by Stephen Wolfram from the 1980s onwards
Definition

- Computation takes place on a grid, which may have 1, 2 or more dimensions, e.g. a 2D CA:

![Grid Diagram]

Definition

- At each grid location is a **cell**
  - Which has a **state**
  - In many cases this is binary:

![Cell States Diagram]
Definition

◊ Each cell contains an **automaton**
  ▷ Which observes a **neighbourhood** around the cell

◊ Each cell contains an **automaton**
  ▷ And applies an **update rule** based on this neighbourhood
  ▷ Every automaton uses the same update rule

If one neighbour is on, turn on, else turn off
Definition

- The CA is run over a number of discrete time steps
  - At each time step, each automaton applies its update rule
  - Time = 0

If one neighbour is on, turn on, else turn off

---

Definition

- The CA is run over a number of discrete time steps
  - At each time step, each automaton applies its update rule
  - Time = 1

If one neighbour is on, turn on, else turn off
The CA is run over a number of discrete time steps
- At each time step, each automaton applies its update rule
- Time = 2

If one neighbour is on, turn on, else turn off

The CA is run over a number of discrete time steps
- At each time step, each automaton applies its update rule
- Time = 3

If one neighbour is on, turn on, else turn off

The CA is run over a number of discrete time steps
- At each time step, each automaton applies its update rule
- Time = 4

If one neighbour is on, turn on, else turn off
A number of different neighbourhoods are used in CAs
- This is called a **Moore** neighbourhood

A number of different neighbourhoods are used in CAs
- This is called a **von Neumann** neighbourhood
A number of different neighbourhoods are used in CAs

- This is called an extended von Neumann neighbourhood

At the edges, toroidal neighbourhoods are often used.

Also known as periodic boundary conditions.
Cellular Automata (CA)

◊ What are cellular automata used for?
  ▶ Modelling spatial processes
    • e.g. forest fires, disease spread
  ▶ Modelling physical processes
    • e.g. crystal formation, thermodynamics
  ▶ Modelling biological processes
    • e.g. pattern formation, self-replication
  ▶ Solving computational problems
    • e.g. random number generators, ciphers
  ▶ Parallel processing architectures
    • e.g. systolic arrays, Connection Machine

Conway’s Game of Life

◊ Developed by John Conway in the 1970s
  ▶ A simple model of self-replication
  ▶ Surprisingly complex behaviour
  ▶ Led to wider interest in CAs

◊ 2 states (live, dead), Moore neighbourhood, 4 rules:
  ▶ A live cell with <2 live neighbours dies (under-population)
  ▶ A live cell with 2-3 live neighbours remains alive
  ▶ A live cell with >3 live neighbours dies (over-crowding)
  ▶ A dead cell with 3 live neighbours becomes a live cell (reproduction)

Spaceships, Guns
Methuselahs

- Patterns that grow and take a long time to stabilise
  - Complexity emerges from simple rule and initial state
  - Can be seen as carrying out a complex computation

- Acorn: size 7, grows to 1057, lasts 5206 time steps
  - Stable pattern consists of 41 blinkers, 4 traffic lights, 34 blocks, 30 beehives, 1 honey farm, 13 gliders, 8 boats, 5 loaves, 3 ships, 2 barges, 2 ponds and 1 mango

http://www.conwaylife.com/wiki/Methuselah

Game of Life

- A computationally interesting cellular automata
  - Simple definition, complex behaviour
  - Unexpected emergent phenomena
  - i.e. spaceships, methuselah
  - Computationally universal
Various multi-valued state CAs have been studied
  - e.g. Langton’s loops model self-replication
    • Uses 8 states:
  - e.g. WireWorld models electron flow in circuits
    • Uses 4 states:
Elementary Cellular Automata

- 1D binary CAs that take place on a single grid row
  - Appear simple, but can be deceptively complex
  - Probably the most studied form of CA
  - Stephen Wolfram’s work on these is very well known

Space-time diagram

- They are based around a neighbourhood of size 3:
  - \( t=n \)
  - \( t=n+1 \)

- Hence, it maps \( 2^3 = 8 \) possible patterns to 0 or 1
  - Meaning there are \( 2^8 = 256 \) possible update rules

Rule 30, Rule 110
How do I compute using a CA?

- 1) Find a suitable rule
- 2) Encode the problem instance in the initial state
- 3) Execute the CA for a certain number of steps
- 4) Read the result from the final state of the CA

Lots of cryptography applications

- Google ‘cellular automata encryption’
- Lots of different CA models used
- Have a look!

* Koza's definition of an architecture of a program – architecture of a program is the total number of trees, the type of each tree, the number of arguments possessed by each tree and the hierarchical references allowed among the trees (ADF1, ADF2...)

  3 ways to determine the architecture of a program that will be evolved:
  1. User specifies in advance the structure of overall program
  2. Execution of GP may impose the design of the architecture
  3. Execution of GP may impose the **architecture-altering operations**, which can create new ADFs, remove ADFs and increase/decrease number of inputs in ADF. Note: new arrangements of ADFs make it easier for subsequent changes to evolve better programs later.

* Michael Lones:
  ADFs provide space (extra trees) in which modules can evolve
  Developmental GP uses an indirect encoding (cellular encoding) – uses a program for genotype, which, when executed, constructs phenotype (which may or may not be a program).

* ADFs (Automatically Defined Functions)
  Programmers organize reusable code in components and invoke these components/functions with different inputs.
  ADF provide a mechanism for evolving these kind of potentially reusable components

* Strongly typed GP
  In strongly typed GP, every terminal has a type, and every function has a type for each of its arguments and return type.
**Type Consistency** – outputs of any subtree can be used as one of the inputs to any node. The basic subtree crossover operator shuffles tree components entirely randomly. Crossover cannot lead to incompatibility between nodes. This also stops mutation from producing illegal programs. Use types and grammars to bias or constrain search with the primary aim of increasing the chance of finding a suitable program.

**Grammar-based Constraints**

Another way to express constraints – via grammars (define grammar using BNF notation)

Need to keep track not only of the program itself, but also of whether it is grammatically/syntactically correct

Programs must still be executed to evaluate their fitness

**Constraints and Bias**

Constraint systems: grammars & types – they limit search space by restricting kinds of structures that can be constructed. (has a price)

More difficult to generate type-correct individuals in the initial population or during mutation or crossover – all burden is placed on the initialization phase (only correct solutions can be produced)

No guarantee that such constraints will make evolutionary process easier

**Developmental GP**

Goes beyond production of computer programs by using appropriate terminals (from terminals set), functions (from functions set) as well as interpreters.

In **Cellular Encoding**, programs are interpreted as sequences of instructions, which modify initial structure (embryo)

When program is finished, quality of the structure it produced is evaluated and it is not considered as a fitness of a program

Cellular Encoding – encoding as grammar tree (indirect)

For cellular encoding to work, primitives of a language must be able to grow structures, appropriate to problem domain

Cellular encoding has been used to evolve NN and electric circuits

Tree-adjoining grammars to create a system, where at each stage, result is a working program that respects grammatical constraints

Advantage of indirect encoding (i.e. cellular encoding) – standard GP operators can be used to evolve structures, which have nothing in common with standard GP trees.

Advantage 2: structures resulting from developmental GP, often have some regularity, which other methods obtain from the use of ADF, constraints, types, etc.

Disadvantage: individuals require additional genotype-to-phenotype decoding