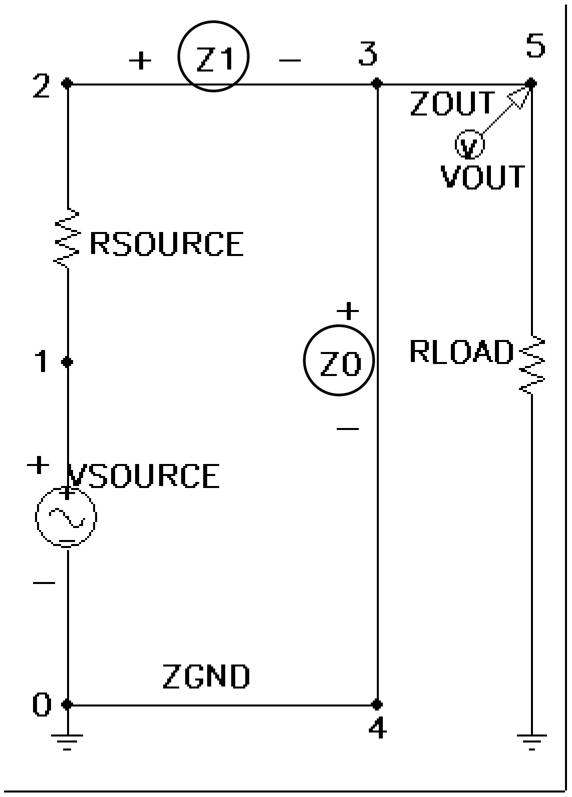
DEVELOPMENTAL GENETIC PROGRAMMING

- **Electrical circuits (introduction)**
- **Cellular encoding (neural networks)**
- **Auto-parallelization of serial computer programs**

AUTOMATIC SYNTHESIS OF BOTH THE TOPOLOGY AND SIZING OF ELECTRICAL CIRCUITS

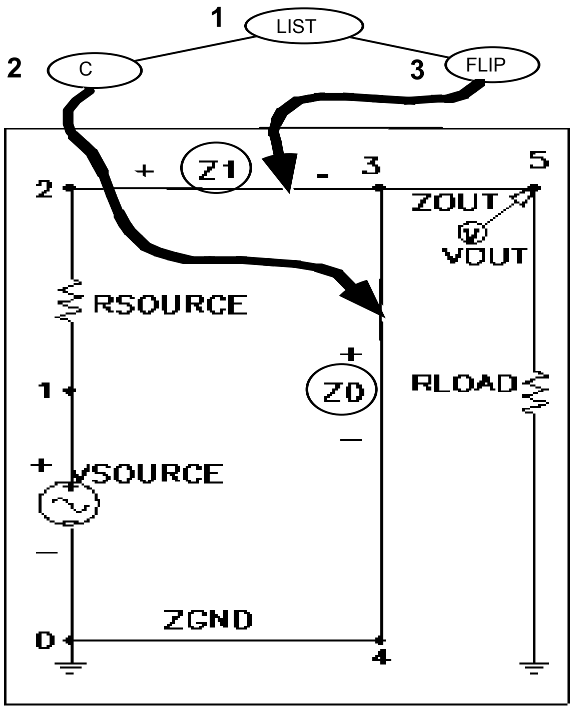
ONE-INPUT, ONE-OUTPUT INITIAL CIRCUIT

- **Initial circuit consists of embryo and test fixture**
- **Embryo has modifiable wires (e.g., Z0 AND Z1)**
- **Test fixture has input and output ports and usually has source resistor and load resistor. There are no modifiable wires (or modifiable components) in the test fixture.**



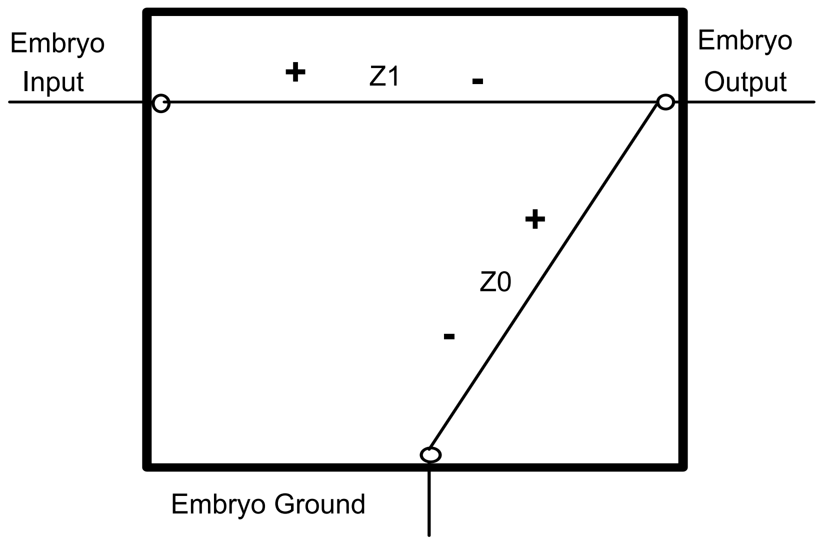
THE INITIAL CIRCUIT

- **Circuit-constructing program tree contains**
    - **Component-creating functions**
    - **Topology-modifying functions**
    - **Development-controlling functions**
- **Circuit-constructing program tree has one result-producing branch for each modifiable wire in embryo of the initial circuit**
- **There is a writing head linking each modifiable wire (or modifiable component) with one point of the program tree**



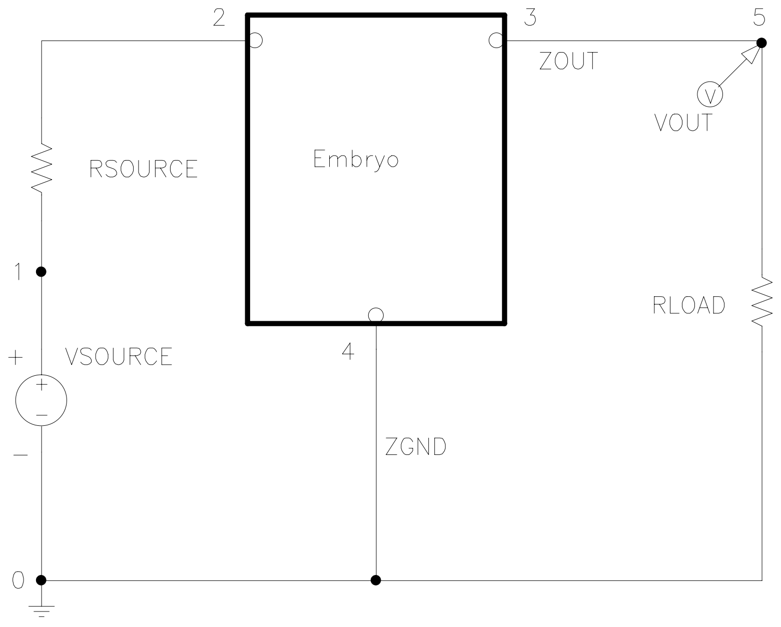
TWO PARTS OF AN INITIAL CIRCUIT

EMBRYO WITH TWO MODIFIABLE WIRES, Z0 AND Z1 AND THREE PORTS (EMBRYO_INPUT, EMBRYO_OUTPUT, AND EMBRYO_GROUND)



TWO PARTS OF AN INITIAL CIRCUIT

ONE-INPUT, ONE-OUTPUT TEST FIXTURE WITH THREE PORTS TO THE EMBRYO



TWO PARTS OF AN INITIAL CIRCUIT

THE EMBRYO

• An embryo contains at least one modifiable wire. A modifiable wire is a wire that is capable of being converted into electrical components, other modifiable wires, and nonmodifiable wires during the developmental process.
• The embryo is <u>very</u> simple - often as little as one (or a few) modifiable wires
• An embryo has one or more ports that enable it to be embedded into a test fixture.
• When the embryo is embedded in the test fixture, the result is typically a useless and degenerate circuit

TWO PARTS OF AN INITIAL CIRCUIT

THE EMBRYO - CONTINUED

- **Occasionally, an embryo may also contain non-modifiable wires, non-modifiable electrical components, or modifiable electrical components.**
- **The developmental process operates on the embryo (not the test fixture)**
- **The distinctive feature of the embryo is that it contains at least one modifiable wire.**

TWO PARTS OF AN INITIAL CIRCUIT

THE TEST FIXTURE

- **The test fixture is a fixed (hard-wired) substructure composed of nonmodifiable wires and nonmodifiable electrical components.**
- **Its purpose is to provide a means for testing another electrical substructure, namely the embryo.**
- **A test fixture has one or more ports that enable an embryo to be embedded into it.**
- **The test fixture provides access to the circuit's external input(s) and outputs and permits probing of the circuit's output.**

TWO PARTS OF AN INITIAL CIRCUIT

THE TEST FIXTURE - CONTINUED

• A test fixture typically incorporates certain fixed elements that are required to test the type of circuit being designed. For example, the test fixture often contains a source resistor reflecting the reality that all sources have resistance and a load resistor representing the load that must be driven by the output.

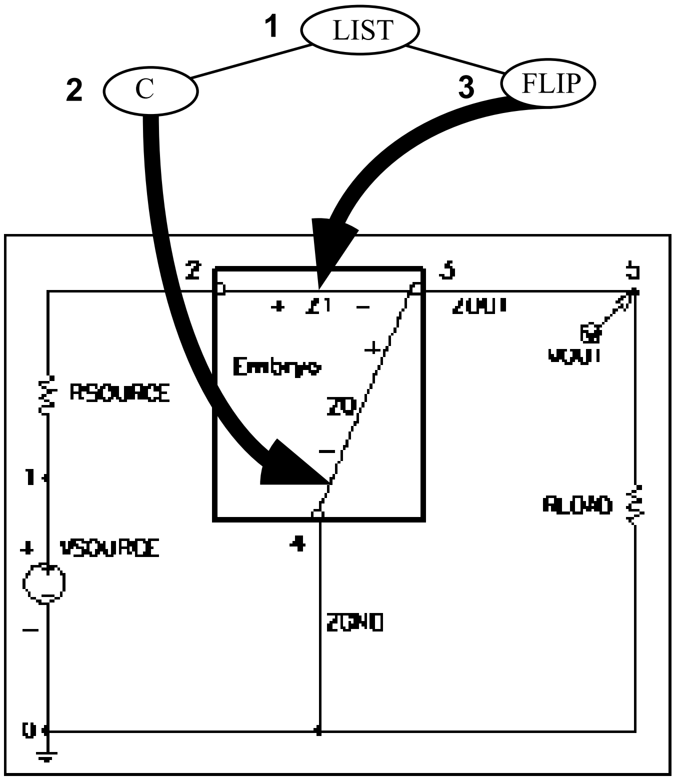
• The distinctive feature of the test fixture is that it contains no modifiable wires and no modifiable components.

DEVELOPMENTAL PROCESS

- **In the initial circuit, the test fixture encases only the embryo.**
- **The developmental process operates on the embryo (not the test fixture)**
- **The developmental process applies functions in the circuit-constructing program tree to certain designated elements of the embryo (and its successors).**
- **The functions in the program tree side-effect the embryo (and its successors during the developmental process).**
- **The developmental process ends when the program tree is fully executed.**
- **After the embryo is fully developed, the test fixture encases the nontrivial substructure that is developed from the embryo.**

WRITING HEADS

ONE-INPUT, ONE-OUTPUT INITIAL CIRCUIT WITH TWO WRITING HEADS ASSOCIATED WITH THE TWO MODIFIABLE WIRES (Z0 AND Z1) OF THE EMBRYO



COMPONENT-CREATING FUNCTIONS

TWO-LEADED

- **Resistor `R` function**
- **Capacitor `C` function**
- **Inductor `L` function**
- **Diode `D` function**
- **`TWO_LEAD_OPAMP` function**
- **Digital `NOT` function (inverter)**

THREE-LEADED

- **Transistor `QT` function**
- **`THREE_LEAD_OPAMP` function**
- **Digital `AND`, `OR`, `NAND`, `NOR` functions**

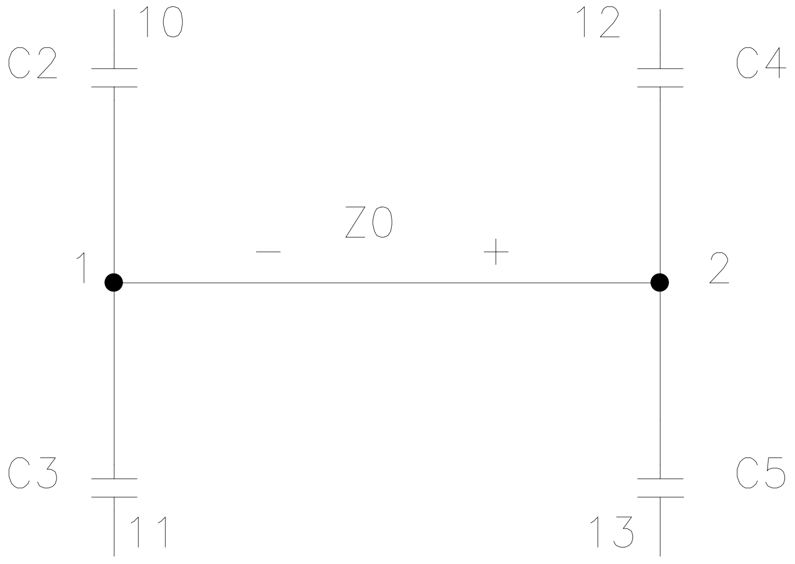
FOUR-LEADED

- **Transformer `TRANFORMER` function**

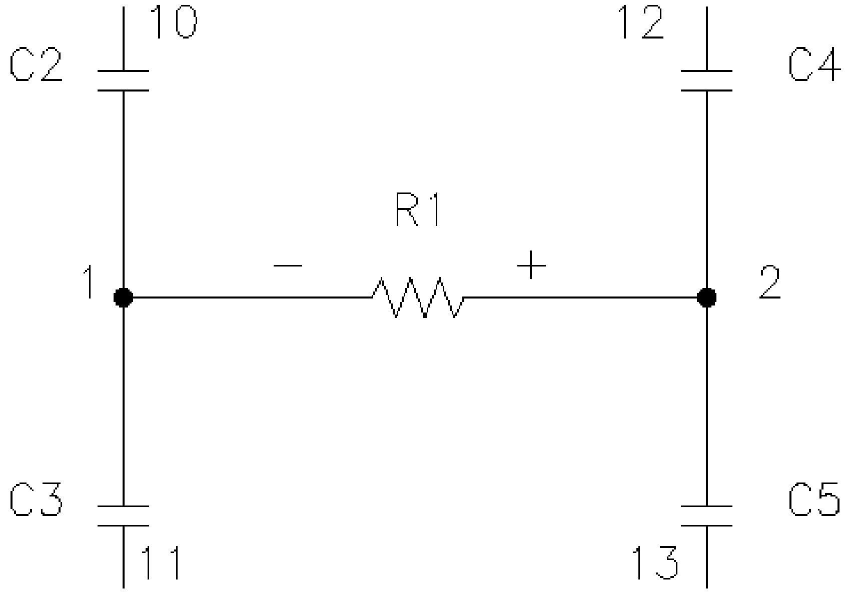
FIVE-LEADED

- **`FIVE_LEAD_OPAMP` function**

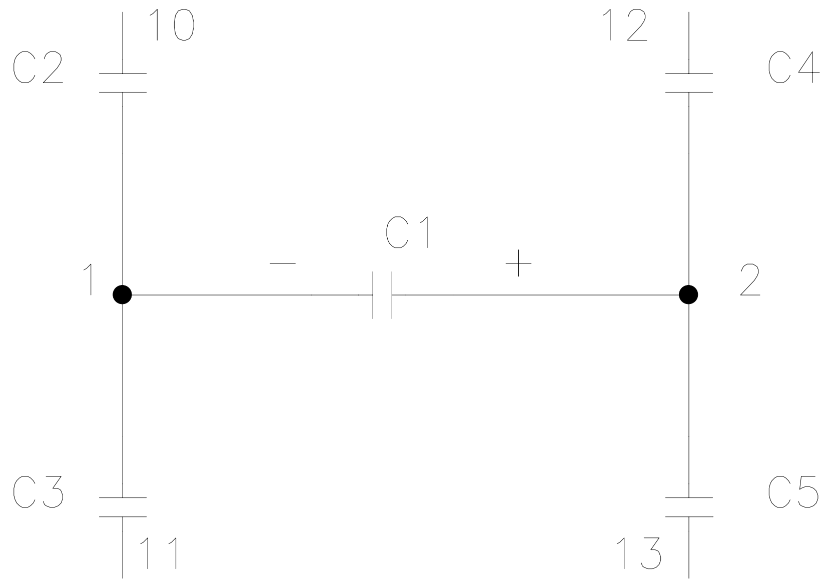
A PORTION OF A CIRCUIT CONTAINING A MODIFIABLE WIRE Z0



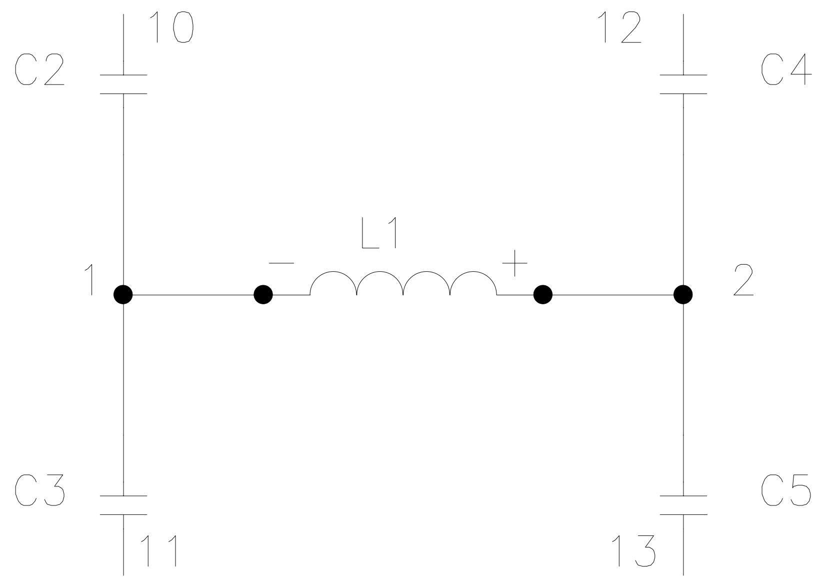
RESULT AFTER R FUNCTION



RESULT AFTER C FUNCTION



RESULT AFTER L FUNCTION



NETLIST BEFORE EXECUTION OF `R` FUNCTION

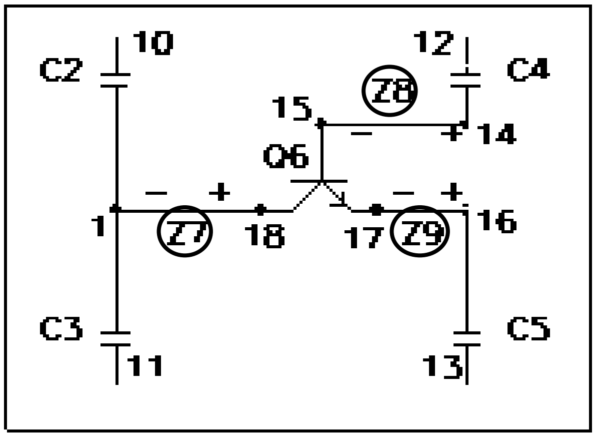
```
C2 1 10
C3 1 11
Z0 2 1
C4 2 12
C5 2 13
```

Note: By convention, the first-listed node is the node connected to the positive lead of a two-leaded component

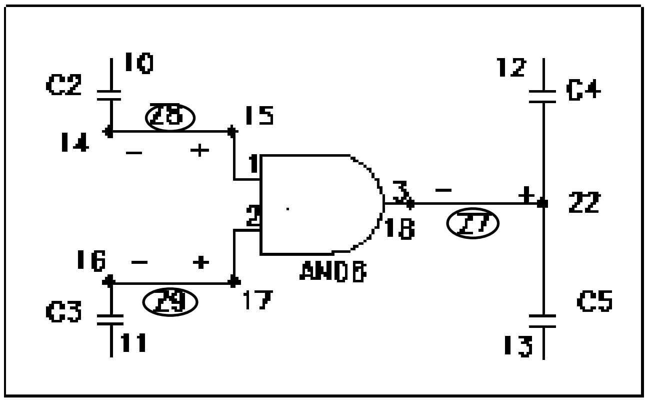
NETLIST AFTER EXECUTION OF `R` FUNCTION CREATING A 5-Ω RESISTOR

```
C2 1 10
C3 1 11
R1 2 1 5ohms
C4 2 12
C5 2 13
```

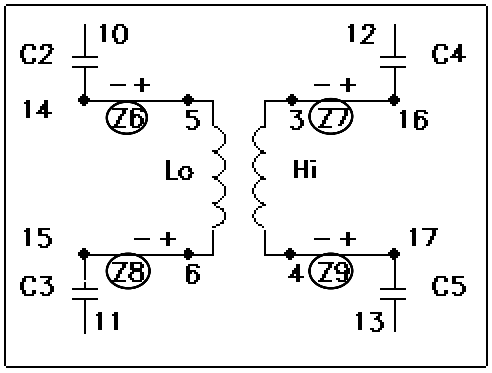
RESULT AFTER `QT0` FUNCTION



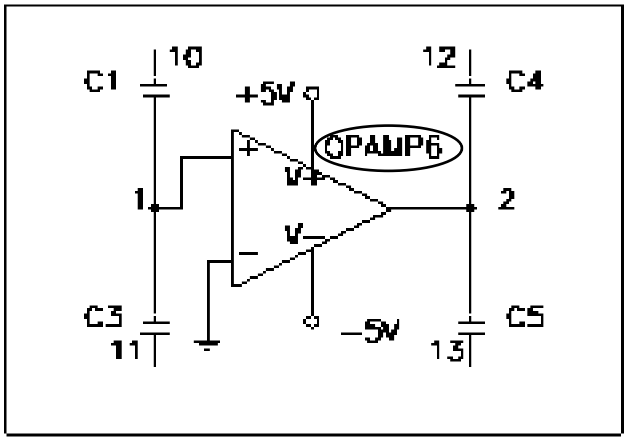
RESULT AFTER `AND0` DIGITAL GATE



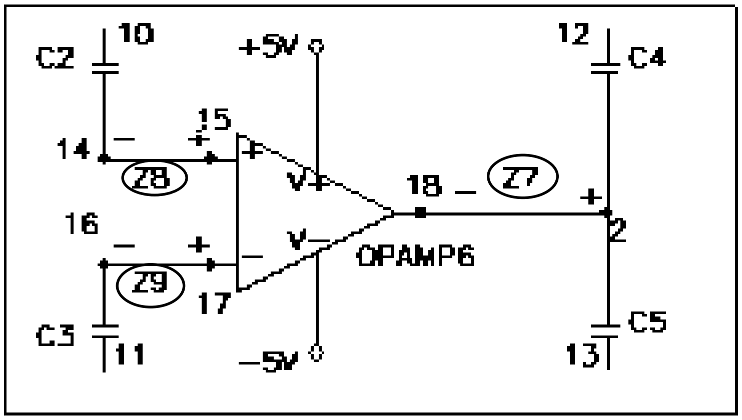
RESULT AFTER TRANFORMER0 FUNCTION



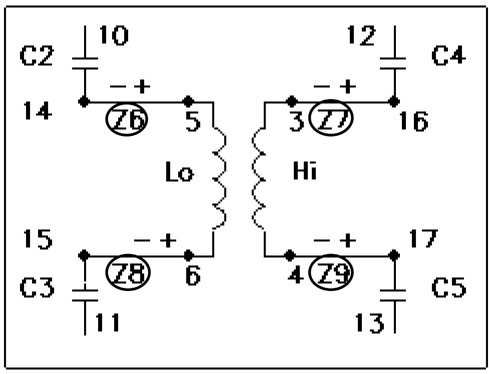
RESULT AFTER `TWO_LEAD_OPAMP1`



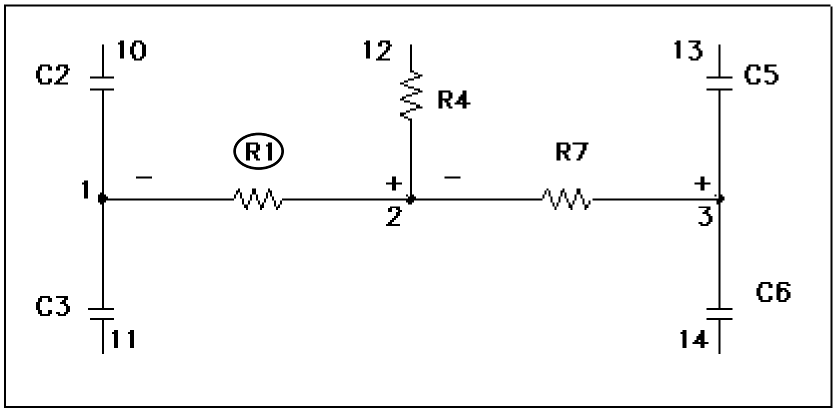
RESULT AFTER `THREE_LEAD_OPAMP1` FUNCTION



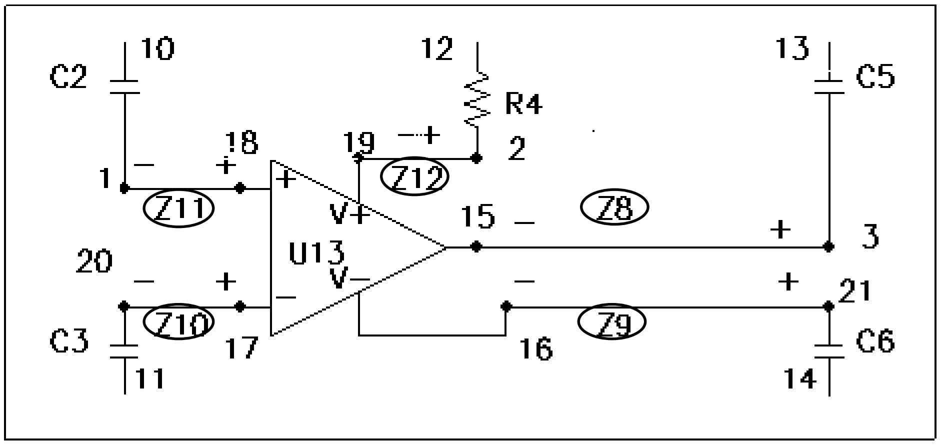
RESULT AFTER `TRANFORMER0` FUNCTION



A PORTION OF A CIRCUIT CONTAINING A RESISTOR R1



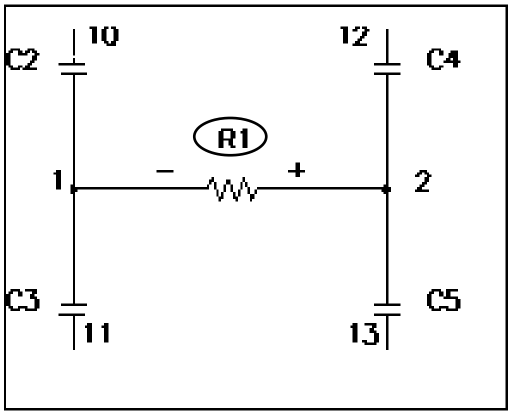
RESULT AFTER FIVE_LEAD_OPAMP3



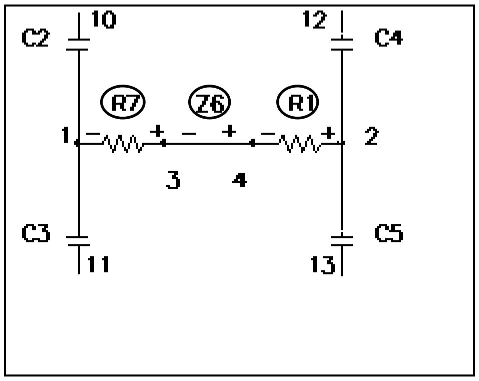
TOPOLOGY-MODIFYING FUNCTIONS

- **`SERIES` division function**
- **`PARALLEL0` and `PARALLEL1` parallel division functions**
- **`STAR` division function**
- **`TRIANGLE` division function**
- **`VIA` function**
- **`FLIP` function**

A CIRCUIT CONTAINING A RESISTOR R1



AFTER THE SERIES FUNCTION



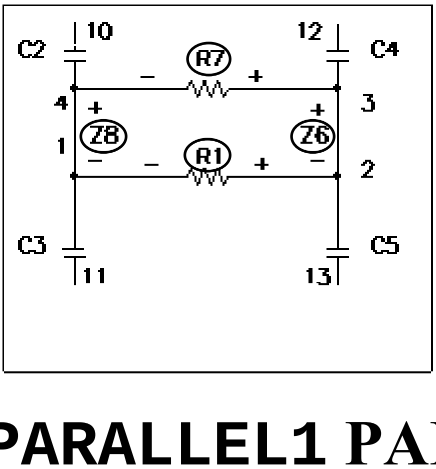
NETLIST WITH R1

```
C2 1 10
C3 1 11
R1 2 1 5ohms
C4 2 12
C5 2 13
```

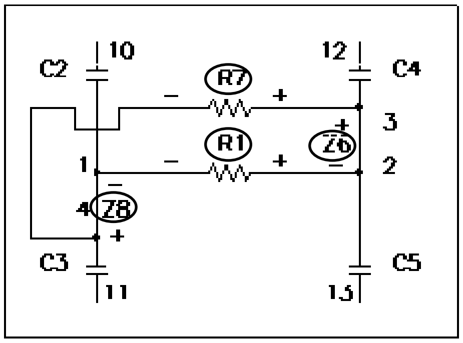
NETLIST AFTER SERIES FUNCTION

```
C2 1 10
C3 1 11
R1 2 4 5ohms
Z6 4 3
R7 3 1 5ohms
C4 2 12
C5 2 13
```

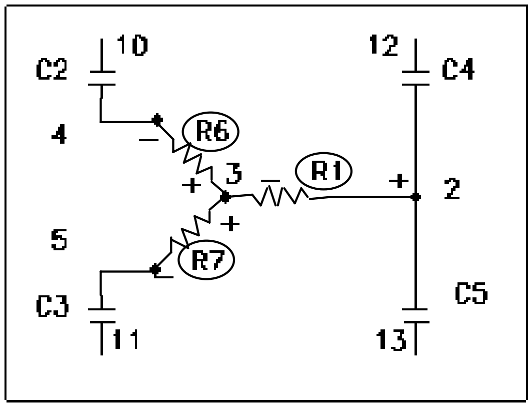
AFTER `PARALLEL0` PARALLEL DIVISION



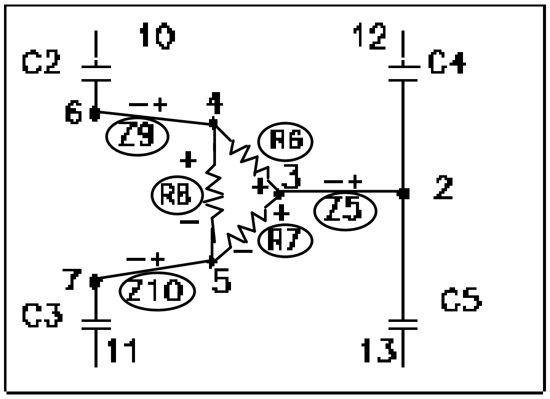
AFTER `PARALLEL1` PARALLEL DIVISION



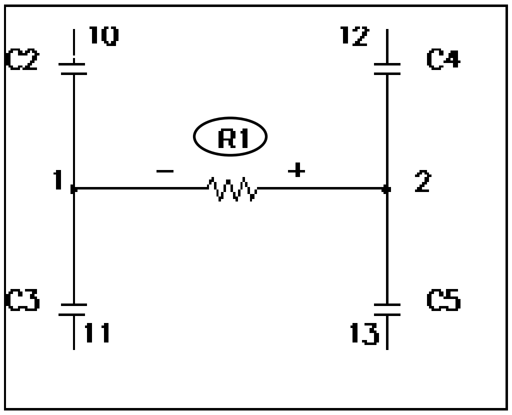
AFTER STAR1 FUNCTION



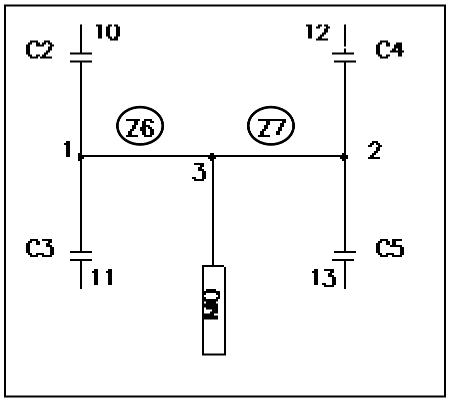
AFTER TRIANGLE1 FUNCTION



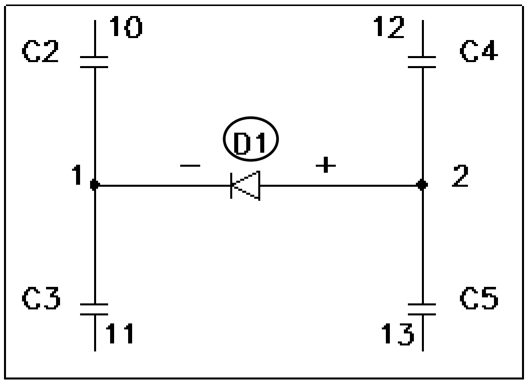
A CIRCUIT CONTAINING A RESISTOR R1



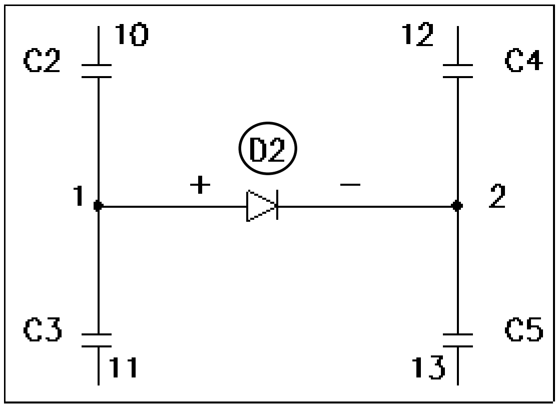
AFTER VIA0 FUNCTION



A CIRCUIT CONTAINING A DIODE D1



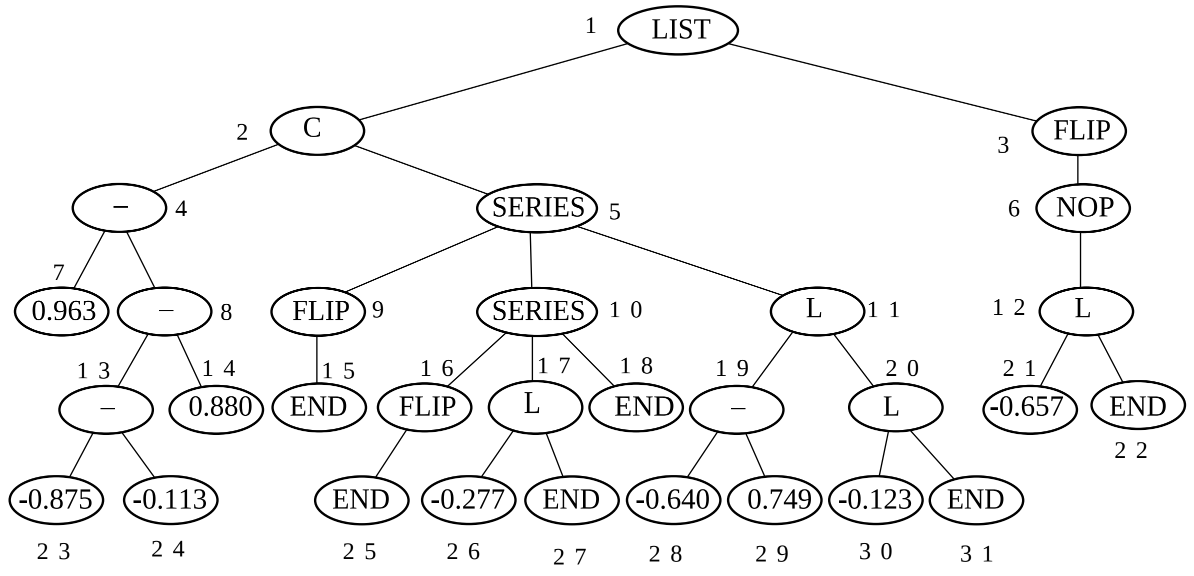
AFTER THE FLIP FUNCTION



DEVELOPMENT-CONTROLLING FUNCTIONS

- **`END` function**
- **`NOP` (No Operation) function**

A CIRCUIT FROM A CIRCUIT-CONSTRUCTING PROGRAM TREE



HIGHLIGHTED ARITHMETIC-PERFORMING SUBTREES

(LIST (C **(– 0.963 (– (– -0.875 -0.113) 0.880)**) (series (flip end) (series (flip end) (L **-0.277** end) end) (L **(– -0.640 0.749)** (L **-0.123** end))))

(flip (nop (L **-0.657** end))))

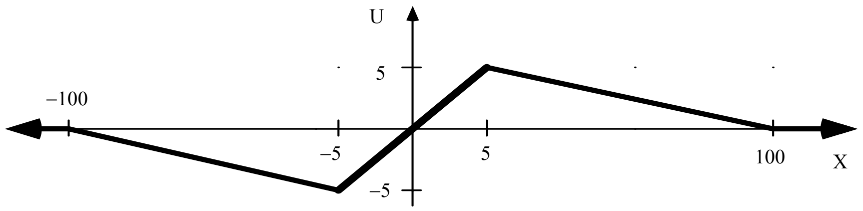
FUNCTIONS AND TERMINALS FOR ARITHMETIC-PERFORMING SUBTREES

Name	**Short Description**	**Arity**
$\Re$	Random constants	0
`+`	Addition	2
`-`	Subtraction	2
`ADF0`, `ADF1`,...	Automatically defined function 0, etc.	Vario us
`ARG0`, `ARG1`,...	Dummy argument 0 (formal parameter 0), etc.	0

LOGARITHMIC INTERPRETATION OF ARITHMETIC-PERFORMING SUBTREES

• The arithmetic-performing subtree produces floating-point number *X*.

• *X* is used to produce an intermediate value *U* in the range of –5 to +5.



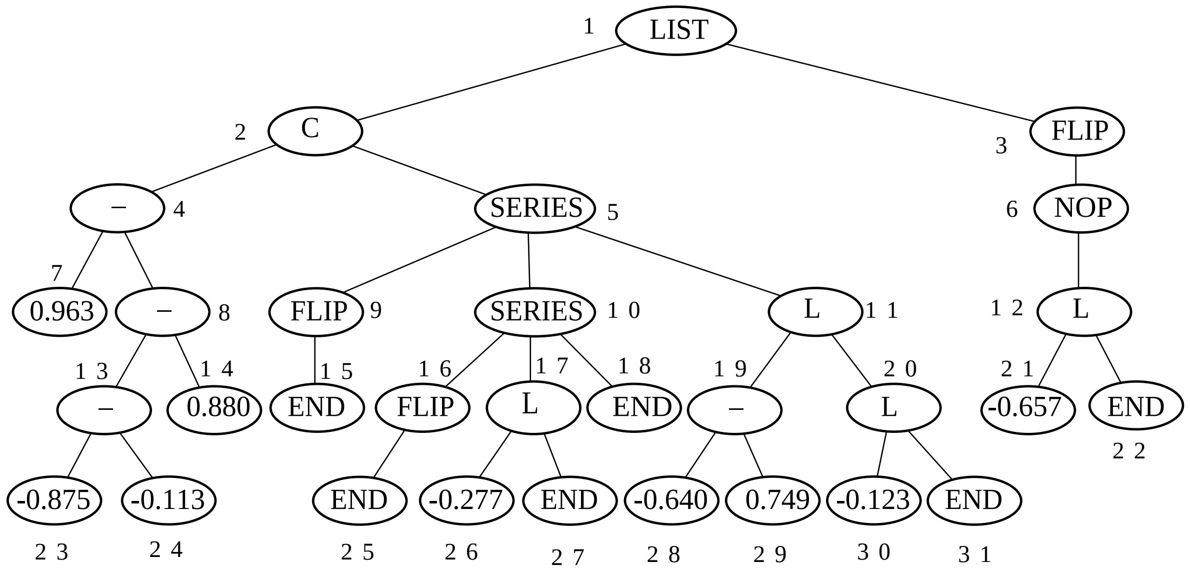
- **If the return value *X* is between –5.0 and +5.0, an intermediate value *U* is set to the value *X* returned by the subtree.**
- **If the return value *X* is less than –100 or greater than +100, *U* is set to a saturating value of zero.**
- **If the return value *X* is between –100 and –5.0, *U* is found from the straight line connecting the points (–100, 0) and (–5, -5).**
- **If the return value *X* is between +5.0 and +100, *U* is found from the straight line connecting (5, 5) and (100, 0).**

POTENTIAL FUNCTIONS AND TERMINALS FOR AUTOMATICALLY DEFINED FUNCTIONS

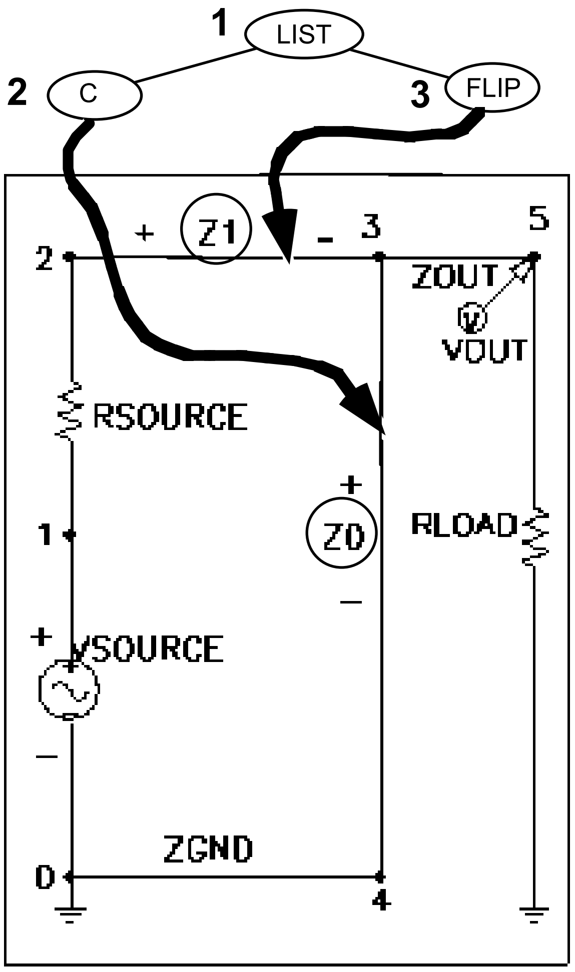
Name	**Short Description**	**Range of Arity**
`ADF-i`	Automatically defined function *i*	0 to MAX_{adf}
`ARG-i`	Dummy variable *i* (formal parameter) of automatically defined function(s)	0 to MAX_{arg}

DEVELOPMENT OF A CIRCUIT FROM A CIRCUIT-CONSTRUCTING PROGRAM TREE AND THE INITIAL CIRCUIT

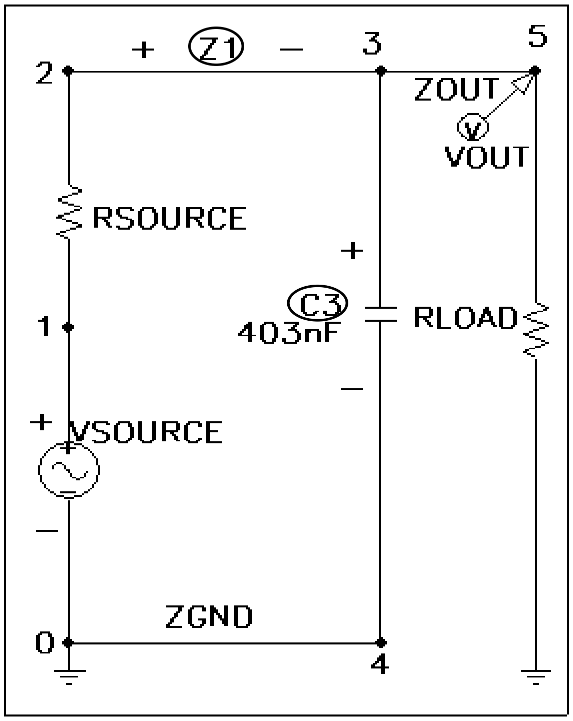
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```



INITIAL CIRCUIT WITH TWO WRITING HEADS



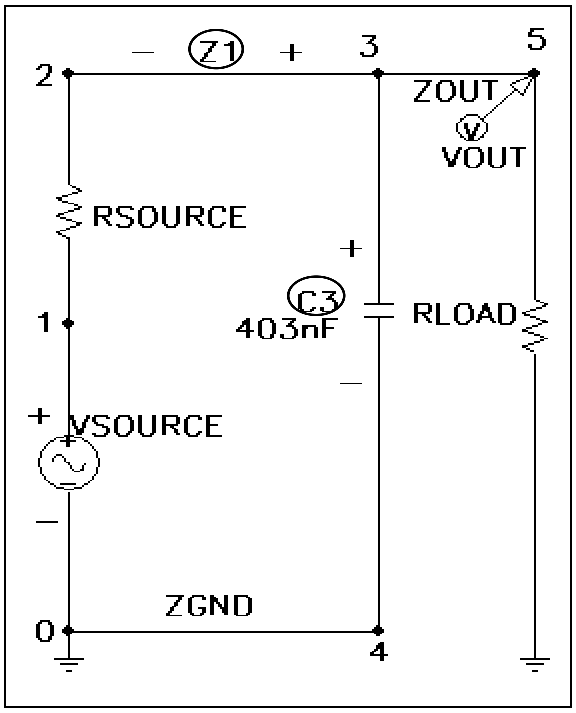
RESULT OF THE `C` (2) FUNCTION



```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

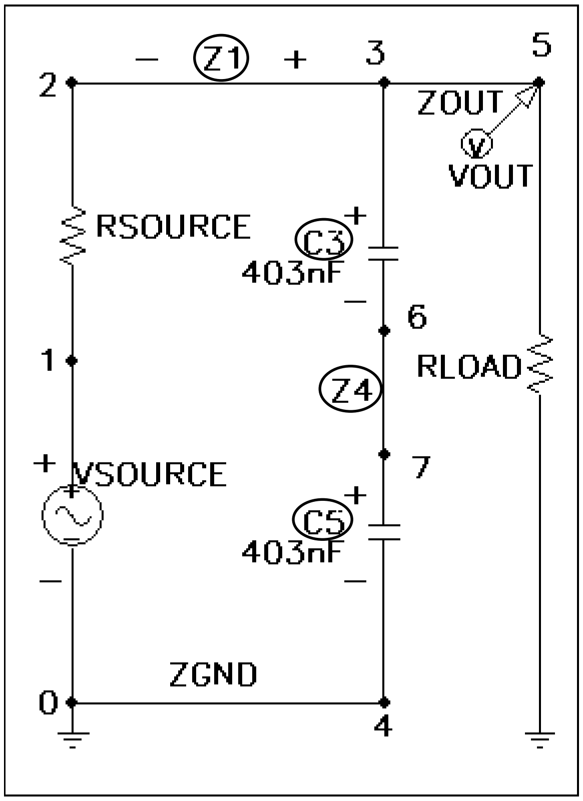
NOTE: Interpretation of arithmetic value

RESULT OF THE `FLIP` (3) – IN 2nd RESULT-PRODUCING BRANCH



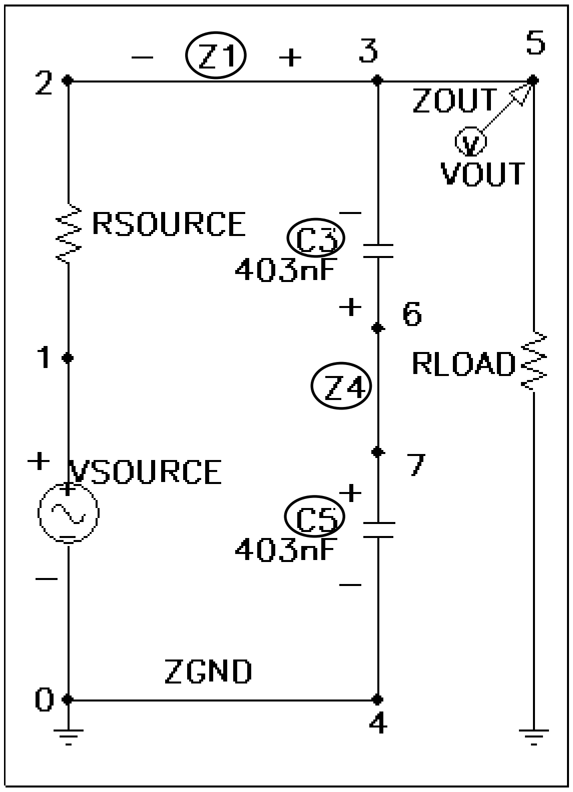
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

RESULT OF SERIES (5) FUNCTION



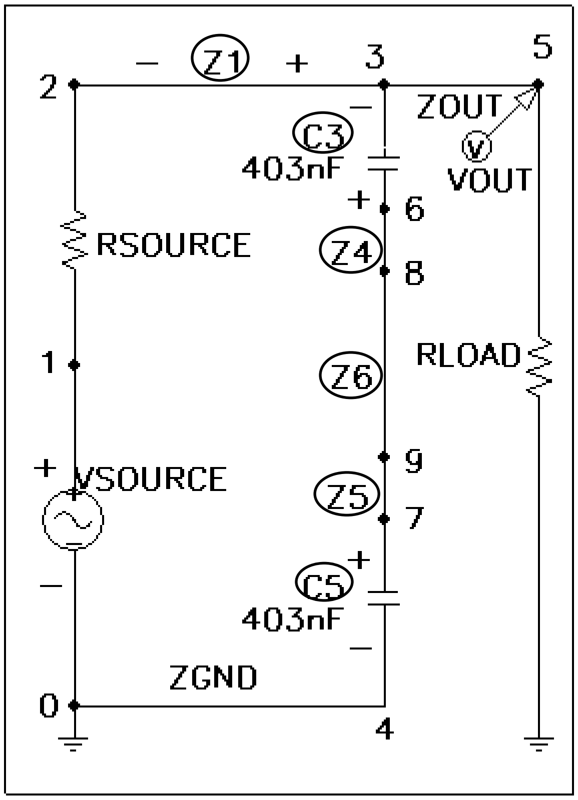
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

RESULT OF THE FLIP (9) FUNCTION



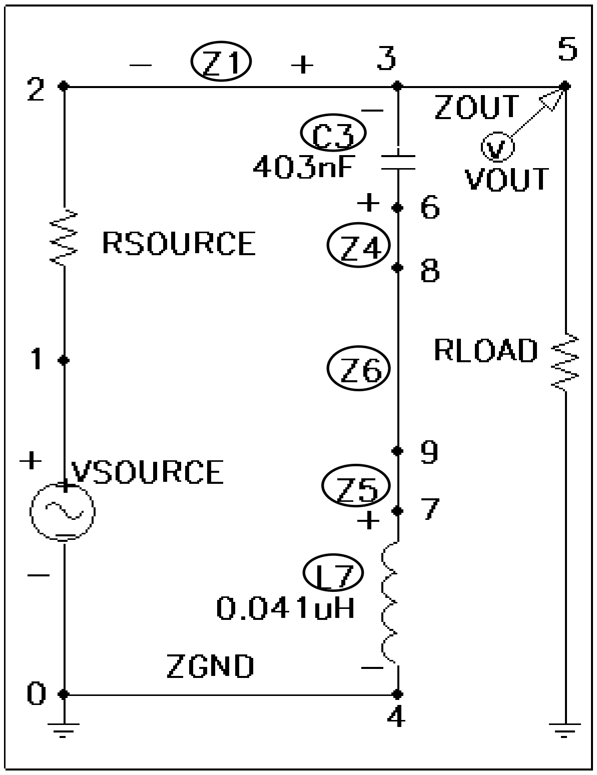
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

RESULT OF SERIES (10) FUNCTION



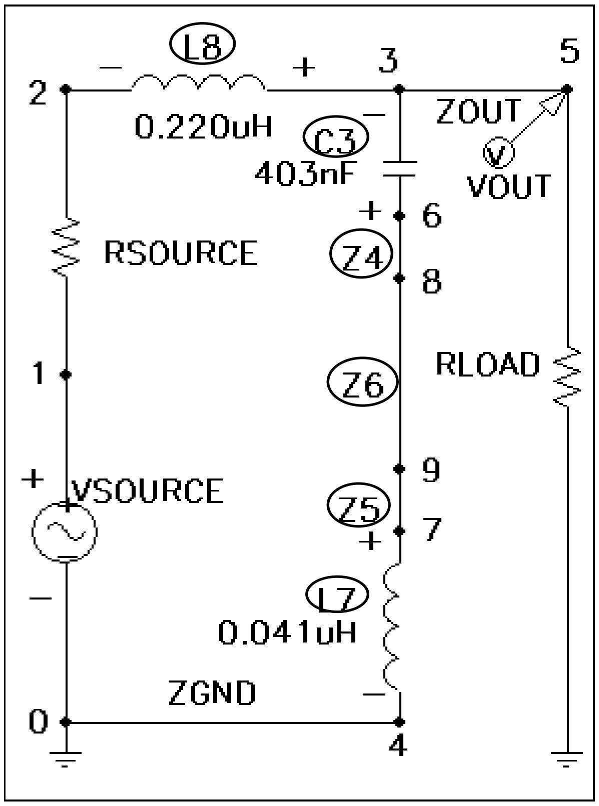
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

RESULT OF L (11) FUNCTION



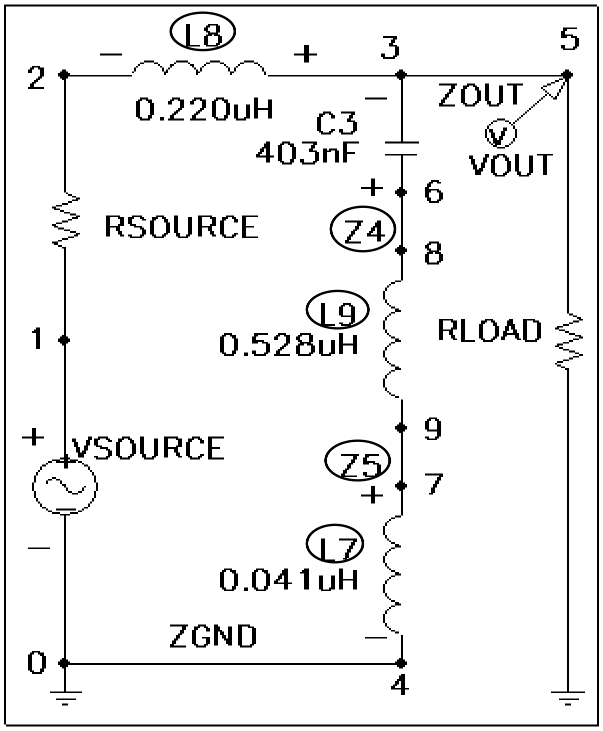
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

RESULT OF L (12) FUNCTION



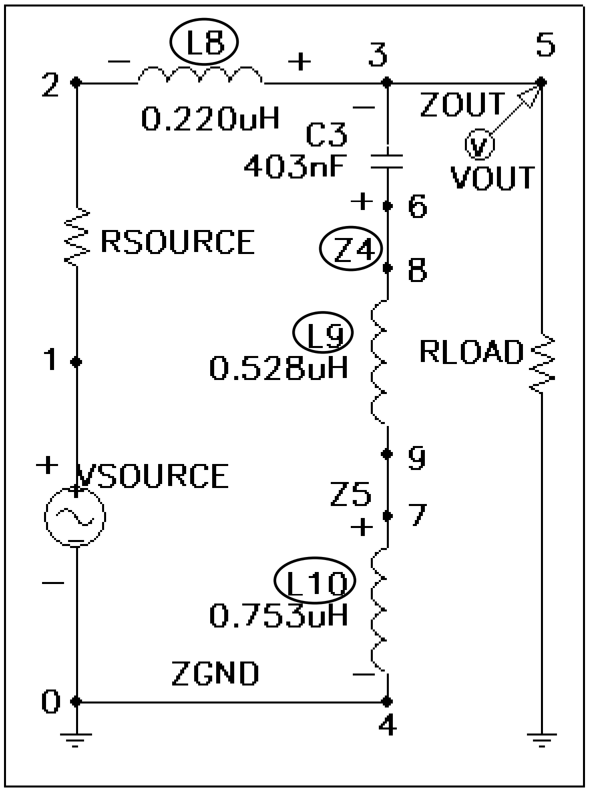
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

RESULT OF `L` (17) FUNCTION



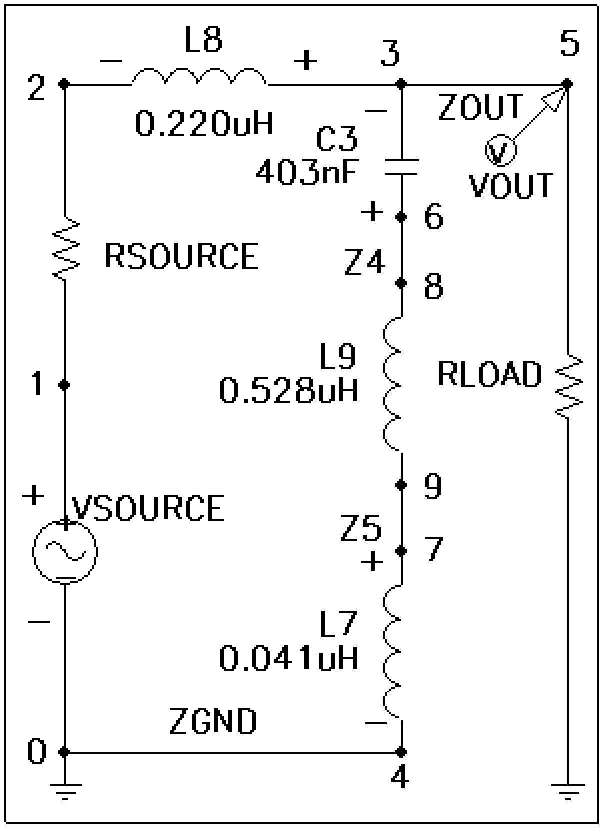
```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

RESULT OF L (20) (OVERWRITING)

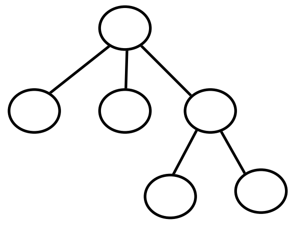


```
(LIST (C (– 0.963 (– (– -0.875
-0.113) 0.880)) (series (flip
end) (series (flip end) (L -
0.277 end) end) (L (– -0.640
0.749) (L -0.123 end)))) (flip
(nop (L -0.657 end)))))
```

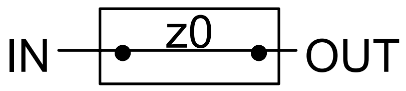
THE FULLY DEVELOPED EXAMPLE CIRCUIT (BOG 0 - LPF)



FITNESS MEASUREMENT PROCESS


Program Tree

+



Embryonic Circuit

$$\begin{tikzcd}
\text{Program Tree} + \text{Embryonic Circuit} \arrow[d] \\
\text{Fully Designed Circuit (NetGraph)} \arrow[d] \\
\text{Circuit Netlist (ascii)} \arrow[d] \\
\text{Circuit Simulator (SPICE)} \arrow[d] \\
\text{Circuit Behavior (Output)} \arrow[d] \\
\text{Fitness}
\end{tikzcd}$$

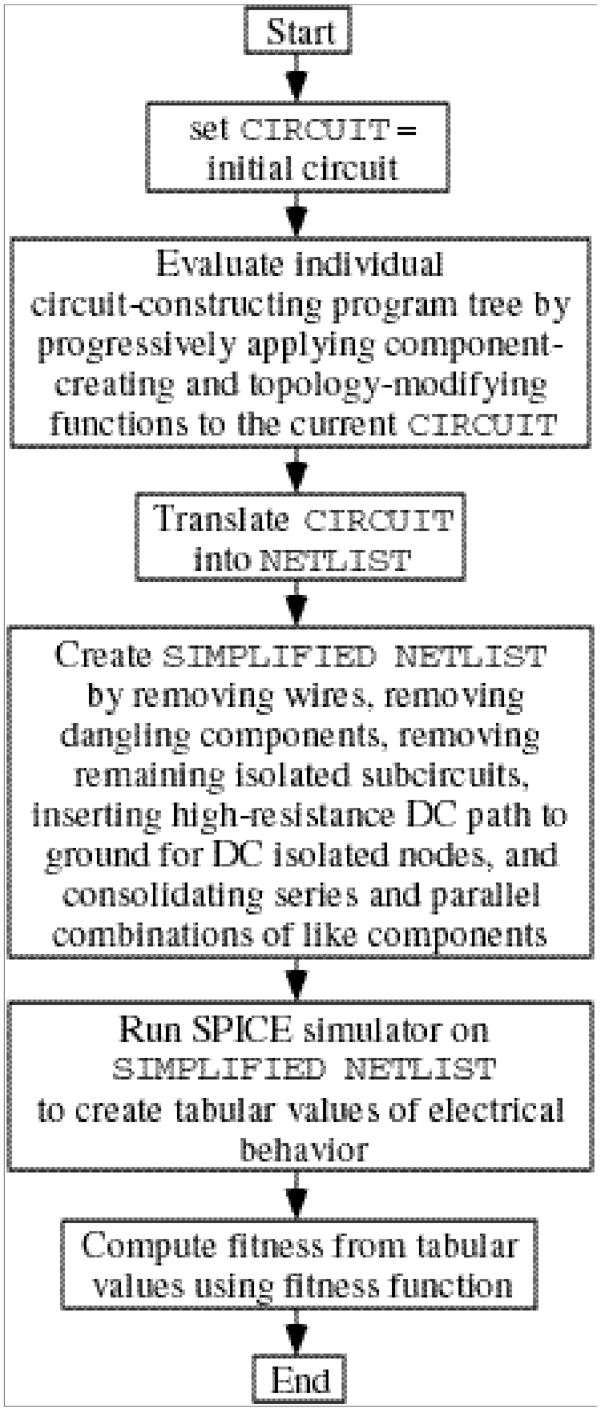
SPICE CIRCUIT SIMULATOR

- **Developed at Univ. of California - Berkeley**
- **217,000 lines of C code**
- **Hundreds of thousands of users**
- **Available in many forms from various sources (e.g., PSPICE, HSPICE)**
- **Input required by SPICE**
  - **Netlist**
  - **SPICE Commands**
  - **Models**
- **Types of analysis produced by SPICE**
  - **Fourier**
  - **AC Sweep**
  - **DC Sweep**
  - **Transient**
  - **Operating point**
  - **Sensitivity**
  - **Distortion**
  - **Noise**
  - **Pole-Zero**
  - **Transfer function**

SPICE NETLIST

```
Circuit Must Have a Title
V0        2  0    DC 0V
QD2       3  3  4 Q2P3906
R6        1  0    1.00e+00K
R7        2  1    1.00e+00K
QNC9      5  1  1 Q2P3906
QGE12     1  6  0 Q2N3904
QD13      1  1  4 Q2P3906
QNC15     5  3  7 Q2P3906
Q17       7  6  0 Q2N3904
VNEGQ999  5  0    DC -15V
.MODEL Q2P3906PNP Bf=180.7 Br=4.977
.MODEL Q2N3904NPN Bf=416.4 Br=0.7371
.DC V0 -0.25 0.251 0.025
.PLOT DC V(1)
.OPTIONS NOPAGE NOMOD
.END
```

FITNESS EVALUATION OF A CIRCUIT


Start
set CIRCUIT = initial circuit
Evaluate individual circuit-constructing program tree by progressively applying component-creating and topology-modifying functions to the current CIRCUIT
Translate CIRCUIT into NETLIST
Create SIMPLIFIED NETLIST by removing wires, removing dangling components, removing remaining isolated subcircuits, inserting high-resistance DC path to ground for DC isolated nodes, and consolidating series and parallel combinations of like components
Run SPICE simulator on SIMPLIFIED NETLIST to create tabular values of electrical behavior
Compute fitness from tabular values using fitness function
End

CELLULAR ENCODING – DEVELOPMENTAL GP (GRUAU 1992)

• GAs have been previously applied to chromosomes that directly represent the identifiable parts of an already-designed neural network.
• Cellular encoding applies GP to trees that specify how the neural net is to be constructed.
• Used to evolve, simultaneously, the architecture of the neural network AND all numerical weights, thresholds, and biases.
• In cellular encoding, each individual (constructing tree) in the population is a composition of network-constructing, neuron-creating, and neuron-adjusting functions and terminals.
• Each of these constructing trees in the population is one step removed from the actual neural network. The constructing tree is the genotype and the neural network constructed in accordance with the constructing tree's instructions is the phenotype.

CELLULAR ENCODING – CONTINUED

• The fitness of an individual constructing tree in the population is measured in terms of how well the neural network that is constructed in accordance with the instructions contained in the constructing tree performs the desired task.
• GP breeds a population of program trees in the usual manner using reproduction and crossover applied to the constructing trees.
• Given a constructing tree, the construction process for a neural network starts from an embryonic neural network consisting of a single neuron. This embryonic neuron has a threshold of 0; its input is connected to all of the network's input nodes with connections with weights of +1; its output is connected to all of the network's output nodes.

CELLULAR ENCODING – CONTINUED

• The network-constructing functions in the constructing tree then specify how to grow the embryonic neuron into the full neural network. Certain network-constructing functions permit a particular neuron to be subdivided in a parallel or sequential manner. Other neuron-adjusting functions can change the threshold of a neuron, the weight of a connection, or the bias on a neuron.

• Reusable neural sub-networks can be implemented (like ADFs).

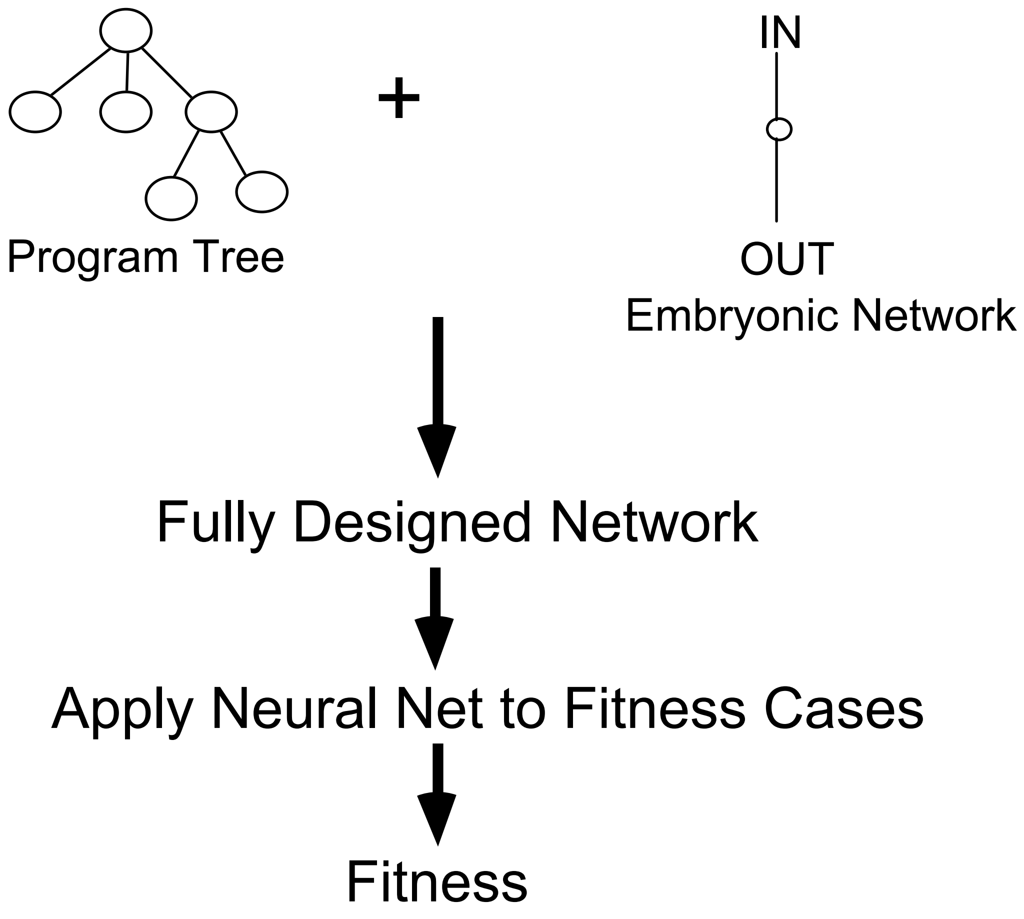
• Recursions can be implemented to create neural networks for high-order symmetry and high-order parity functions.

• Applied by Gruau and Whitley (1995) to 2-pole-balancing problem

• Applied by Gruau to six-legged walking creature

• Applied by Brave (1995, 1996) to Finite Automata

CELLULAR ENCODING – CONTINUED



CELLULAR ENCODING (DEVELOPMENTAL GENETIC PROGRAMMING)

• Gruau, Frederic. 1992b. *Cellular Encoding of Genetic Neural Networks*. Technical report 92-21. Laboratoire de l'Informatique du Parallélisme. Ecole Normale Supérieure de Lyon. May 1992.

• Also: Gruau 1992a 1992b 1993 1994a 1994b; Gruau and Whitley 1993; Esparcia-Alcazar and Sharman 1997)

• GAs have been previously applied to chromosomes that directly represent the identifiable parts of an already-designed neural network.

• Cellular encoding applies GP to trees that specify how the neural net is to be constructed.

• Used to evolve, simultaneously, the architecture of the neural network AND all numerical weights, thresholds, and biases.

CELLULAR ENCODING (DEVELOPMENTAL GENETIC PROGRAMMING) – CONTINUED

• In cellular encoding, each individual (constructing tree) in the population is a composition of network-constructing, neuron-creating, and neuron-adjusting functions and terminals.

• Each of these constructing trees in the population is one step removed from the actual neural network. The constructing tree is the genotype and the neural network constructed in accordance with the constructing tree's instructions is the phenotype.

• The fitness of an individual constructing tree in the population is measured in terms of how well the neural network that is constructed in accordance with the instructions contained in the constructing tree performs the desired task.

CELLULAR ENCODING (DEVELOPMENTAL GENETIC PROGRAMMING) – CONTINUED

- **GP breeds a population of program trees in the usual manner using reproduction and crossover applied to the constructing trees.**
- **Given a constructing tree, the construction process for a neural network starts from an embryonic neural network consisting of a single neuron. This embryonic neuron has a threshold of 0; its input is connected to all of the network's input nodes with connections with weights of +1; its output is connected to all of the network's output nodes.**

CELLULAR ENCODING (DEVELOPMENTAL GENETIC PROGRAMMING) – CONTINUED

• The network-constructing functions in the constructing tree then specify how to grow the embryonic neuron into the full neural network. Certain network-constructing functions permit a particular neuron to be subdivided in a parallel or sequential manner. Other neuron-adjusting functions can change the threshold of a neuron, the weight of a connection, or the bias on a neuron.

• Reusable neural sub-networks can be implemented (like ADFs).

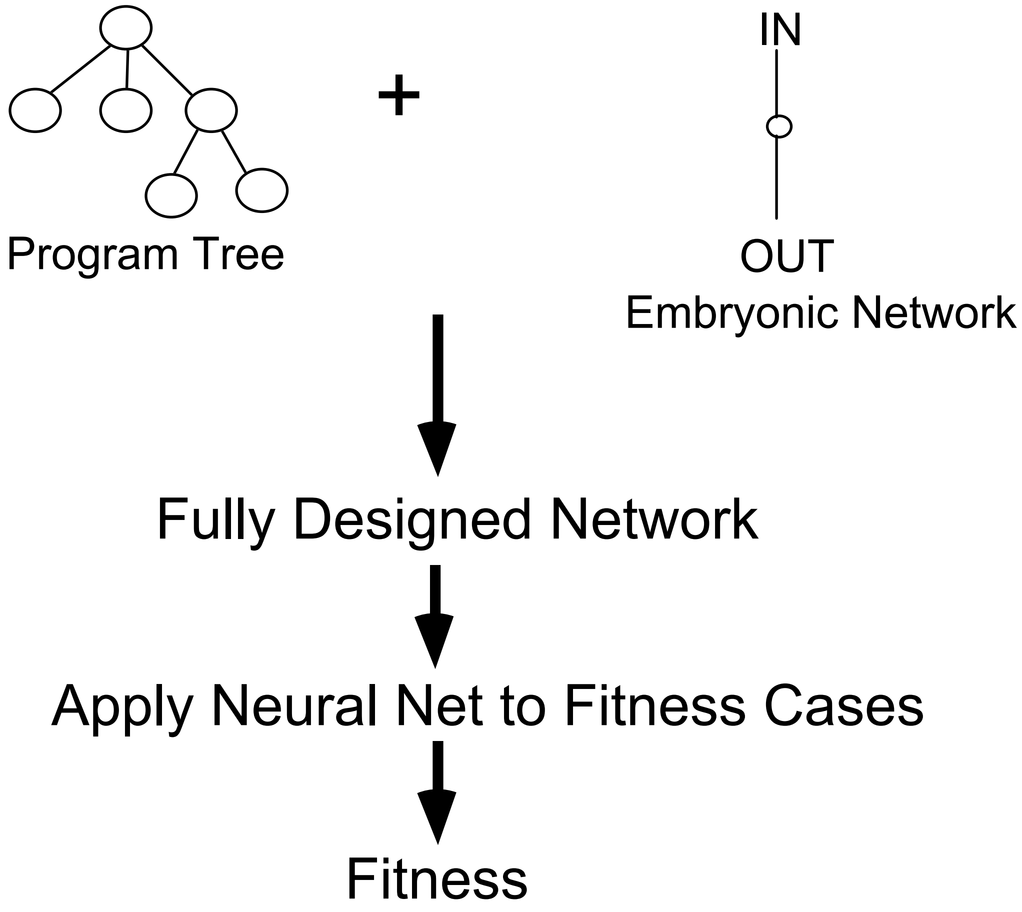
• Recursions can be implemented to create neural networks. Demonstrated by Gruau for high-order symmetry and high-order parity functions.

• Applied by Gruau and Whitley (1995) to 2-pole-balancing problem

CELLULAR ENCODING (DEVELOPMENTAL GENETIC PROGRAMMING) – CONTINUED

- **Applied by Gruau to six-legged walking creature**
- **Applied by Brave (1995, 1996) to Finite Automata**

CELLULAR ENCODING (DEVELOPMENTAL GENETIC PROGRAMMING) – CONTINUED



AUTOMATIC PARALLELIZATION OF SERIAL PROGRAMS USING GP

• Ryan, Conor. 1999. *Automatic Re-engineering of Software Using Genetic Programming*. Amsterdam: Kluwer Academic Publishers.

• Start with working serial computer program (embryo)

• GP program tree contains validity-preserving functions that modify the current program. That is, the functions in the program tree side-effect the current program.

• Execution of the complete GP program tree progressively modifies the current program

• Fitness is based on execution time on the parallel computer system