

# Control structures in linear and stack-based Genetic Programming

Elko B. Tchernev and Dhananjay S. Phatak

CSEE Dept., University of Maryland Baltimore County (UMBC)  
1000 Hilltop Circle, Baltimore, MD 21250, USA  
{etcher1, [Phatak](mailto:Phatak@umbc.edu)}@umbc.edu

Genetic Programming, or GP, has traditionally used prefix trees for representation and reproduction, with implicit flow control. The different clauses (the evaluation condition, the *if* and *else* sections, etc.) are all subtrees of the flow-control node. Linear and stack-based representations, however, require explicit nodes to define the extent of the control structures. This paper introduces a stack-based technique for correct control structure creation and crossover, and discusses its implementation issues in linear and stack-based GP. A set of flow-control nodes is presented, and examples given for evolving an artificial ant on the Santa Fe and Los Altos Trails, with and without looping constructs.

## Stack-based genetic programming

Stack based Genetic programming, introduced by Perkis in [4], represents programs as lists of nodes of functions or terminals that consume their inputs from a stack and place their outputs on a stack. These implementations, including the early work of Bruce, Stoffel and Spector, [1], [6] and later [5], do not try to preserve the stack correctness of the individuals in the population, but rather rely on the evaluation framework to identify any stack underflow or overflow. In contrast, GP with stack-correct (Forth) crossover was introduced by Tchernev in [7] and [8], with the EPROF (Evolutionary PROgramming in Forth) system. In EPROF, the crossover operators manipulate the post order representation of the program tree. Because the crossover points are chosen to have compatible stack depths, no malformation is possible. If the initial population is stack-correct (no individuals have underflow, and the final stack depth equals the desired number of outputs), it is guaranteed that all individuals produced by using stack-correct crossover will be stack-correct.

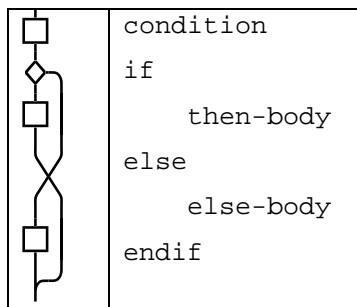
## Control structures in GP

Traditional LISP-based GP has control structures that are tree-based; i.e. they are delimited not by separate keywords *if*, *else*, *then* etc, but by the trees formed by the parentheses of the single control-flow node. For example, the *if* control structure is represented by the subtree `(if (condition) (then-part) (else-`

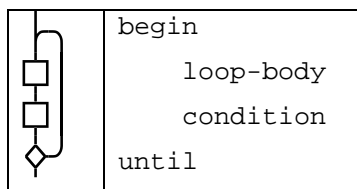
part)), and the while control structure – by the tree (while (condition) (body)). This approach has the advantage that no special handling is required of control structures during crossover – they are just another subtree.

Linear and stack-based GP, where the individuals are represented as linear lists (or arrays) of nodes, cannot use this approach. This, however, does not mean that nothing can be done, or is very difficult, as Keith and Martin mistakenly conclude in [2]. The key to having correct control structures in linear GP is to realize that the nodes are bound in sequential execution not by trees, but by traces of control flow. In a sequential program, the if statement introduces a bifurcation in the control flow, and the endif statement merges the two branches. The branch condition in sequential implementations is usually computed before the branch switching node, and is available either in a status register or variable, or on the stack; therefore it is not part of the control structure proper. The control flow picture of the if control structure can be seen below in Table 1, and that of the begin...until control structure can be seen in Table 2.

**Table 1.** Flow control of the if control structure.



**Table 2.** Flow control of the begin...until control structure.



In structured programming, there is no goto statement; therefore, any control structures inside the body of another control structure are local, i.e. do not interfere with the enclosing structure's control flow. That's why a stack holding the control flow levels is a natural mechanism for handling the information that is needed to resolve the backward and forward branches and to ensure correctness when crossover is performed between linear individuals having control structures. The stack depth increases when control structures are nested, and the top of the stack corresponds to the origin/destination of the closest enclosing control structure. **Note:** this control flow stack is needed and used at *compile time*.

It is easy to see, that crossover that does not change the stack depth of the control flow stack at the crossover points will always produce individuals with syntactically correct control structures. An example for such crossover is given in Table 3.

**Table 3.** Crossover that preserves the control stack depth in the crossover points

Parent 1 flow	Parent 1 text	Parent 2 flow	Parent 2 text	Offspring flow	Offspring text
	if xx if yy if zz else uu endif endif vv else ww endif		if aa if bb endif cc else if dd else ee endif ff endif		if aa if zz else uu endif cc else if dd else if dd else ee endif ff endif









As shown in [8], the most universal condition for ensuring stack correctness in crossover is the delta condition. That is, the node sections being exchanged must have the same control stack depth difference from beginning to end, and the stack usage of both must be compatible. Therefore, in order to have control structures in linear and stack-based GP, the crossover operation must be implemented such that it ensures correctness with respect to the control flow stack in addition to the other conditions it must satisfy.

### Control-flow in Forth and GP

The compile-time control-flow stack described above is the mechanism that ANS Forth uses for its control structures. Since Forth has a very fast single-pass compiler, it is optimized for simple handling of forward and backward branch resolution. Each

of the control structure keywords in Forth acts immediately upon scanning in the source by placing on the control flow stack a branch origin/destination or by consuming one and resolving the branch. The standard Forth control-flow words are given in Table 4, with an explanation of their action.

**Table 4.** Standard Forth control flow words

Picture	Name	Consumes	Supplies	Definition
	if		origin	Marks the origin of a forward conditional branch and compiles its code
	then	origin		Resolves a forward branch by patching its origin
	else	origin1	origin2	Compiles a forward unconditional branch whose origin2 is placed on the stack, and resolves the forward branch origin1
	begin		destination	Marks the destination of a backward branch
	until	destination		Compiles a backward conditional branch to destination
	again	destination		Compiles a backward unconditional branch to destination
	while	destination	origin destination	Compiles a forward conditional branch and places its origin on the control stack such that it becomes the second stack item, below the destination that must have been on the stack
	repeat	origin destination		Compiles a backward unconditional branch to destination (top of the control stack) and resolves the forward branch to current position by patching the origin

In addition to the above, ANS Forth defines the control stack operations `cs-pick` and `cs-roll` which can be used to copy a stack item to the top of stack, and to rearrange the stack items. With the help of these operations, any control structure can be portably custom-designed.




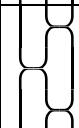



It can be seen, that as they stand, the Forth control flow words are not straightforwardly usable in GP. They place on and consume from the control stack items of two different types, namely origins and destinations. Any GP that uses these primitive Forth constructs would have to ensure that the crossover points match not only the stack depths, but the stack content types as well. A much more practical approach is to make sure that any control flow primitive can mark/resolve any other, provided its stack depth requirements are met, by making it supply and/or resolve *both* an origin and a destination. In this way the only condition a crossover operation needs to ensure

is that the control stack usage between the crossover points in both individuals satisfies the delta requirements. **Note:** in this case, the control stack exists and is used at crossover time.

## Control structures for linear GP

In linear GP, each node is a self-contained entity (a machine language instruction, or a VM language statement). These nodes can be mixed and executed in any order, so there are no additional constraints to the choice of crossover points, except the ones imposed by the control flow stack correctness requirements. The branch conditions are stored in a status register/accumulator/variable, or may be results of atomic execution of an embedded function, e.g. the *if-food-ahead* control of the artificial ant problem. Thus, the set of control flow nodes could include *begin*, *end*, *if*, *else*, *until*, *while*, *repeat*, with *if*, *until* and *while* being defined to use the system's status register/condition. Alternatively, they could be custom defined according to the problem, like *if-true*, *while-food-ahead*, *until-empty*, etc. Example implementation of these control nodes in Forth (assuming the VM for the particular Linear GP has been written in Forth) can be seen in Table 5.

**Table 5.** Control flow nodes for linear GP

Picture	Node	Control stack usage	Definition in Forth
	<code>begin</code>	( -- orig dest )	<code>true if begin</code>
	<code>end</code>	( orig dest -- )	<code>true until then</code>
	<code>if</code>	( -- orig dest )	<code>condition if begin</code>
	<code>else</code>	( orig1 dest – orig2 dest )	<code>1 cs-roll else 1 cs-roll</code>
	<code>until</code>	( orig dest -- )	<code>condition until then</code>
	<code>while</code>	( orig1 dest1 -- orig dest orig1 dest1 )	<code>condition if begin 3 cs-roll 3 cs-roll</code>
	<code>repeat</code>	( orig1 dest1 orig2 dest2 -- )	<code>again then true until then</code>

## Control structures for stack-based GP

Stack-based GP is a superset of tree-based GP, as the individuals might represent arbitrary digraphs, not just trees. For stack-correct crossover to operate, each primitive used by the problem (function or terminal) must have a defined stack usage (number of stack items it requires as inputs, and number of stack items it leaves on the stack). The branches that the control flow nodes compile, usually adhere to this rule and use the top stack items as flags to determine whether to execute the branch or not. In this respect they are just like regular nodes, and participate in the formation of the overall data stack picture. Crossover in stack-correct GP uses this stack picture to choose the crossover points as usual, and then checks whether they conform to the delta rule with respect to the control flow stack. This makes crossover in stack-based GP with control structures more constrained than without, because some choices of points will be rejected. The control flow nodes for stack based GP are essentially the same as these for linear GP, except that the branching primitives *if*, *while* and *until* consume one stack item from the data stack each.

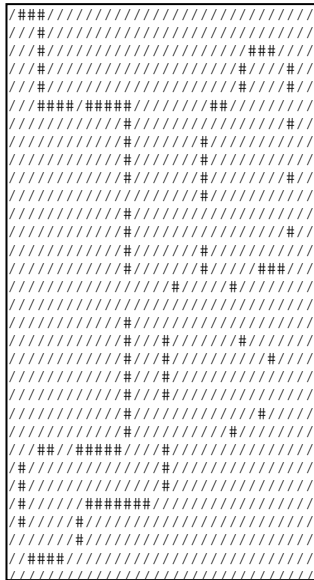
## Initial population with control structures

The stack-correct crossover rules outlined so far guarantee stack-correct offspring *iff* the initial population is stack-correct. Creating stack correct individuals involves keeping track of the running stack depth, and restricting the choice of nodes to be added as next node to those that will not underflow. The decision when to stop adding nodes and fix the final size of the individual being created can be taken only when the running stack depth is equal to the desired output depth. In addition, in order to guarantee a finite size, the selection of nodes to be added can be biased towards changing the running stack depth in the direction of the desired output depth.

The same procedure can be applied when creating individuals with control structures. Since control structures must be properly delimited, the end of an individual cannot occur at non-zero control stack depth (i.e. in the middle of a control structure), and the control stack must not underflow. When dealing with linear GP, these are the only conditions that must be observed.

Stack-based GP, however, places additional restrictions on the choice of control flow nodes. Since they create splits and joins in the execution path, but there is only one data stack picture, it must be guaranteed that both execution paths result in the same data stack depth after the closing node. Therefore, a follow-up node for a control structure (one of *else*, *end*, *until*, *while* or *repeat*) can be inserted only when the data stack depth of the active branch (the running stack depth) plus the data stack change effected by the node itself, results in the same stack depth as the one immediately after the opening node (*begin* or *if*). A simple method of keeping track is to push the current stack depth on the control stack together with the origin/destination information when adding an opening node, and pop it when the closing node is added. **Note:** now, the control stack exists and is used at initialization time.

## Experiments



**Figure 1.** The Santa Fe Trail in the grass

The artificial ant problem, introduced by Koza in [3], is used to demonstrate the use of control structures in linear GP. The problem is to evolve a program that will scan a toroidal field of 32 by 32 cells, on Figure 1, and “eat” the “food pellets” represented by the hash symbol ‘#’.

The set of operations is Move, Left and Right, where Move takes the ant one step forward in the direction it is facing, and if the field it lands on contains a “pellet”, “eats” it. Left and Right change the direction the ant is facing; initially the ant is positioned at the top left field at the beginning of the trail and faces the food. The ant is run until 400 steps (each of Move, Left and Right is 1 step), or until all food is eaten, whichever happens first. The fitness is equal to the number of remaining food pellets (minimization), the population is steady-state with elitism, and the selection uniform across the non-replaced individuals.

**Table 6.** Tableau for the Ant Problem, no looping constructs

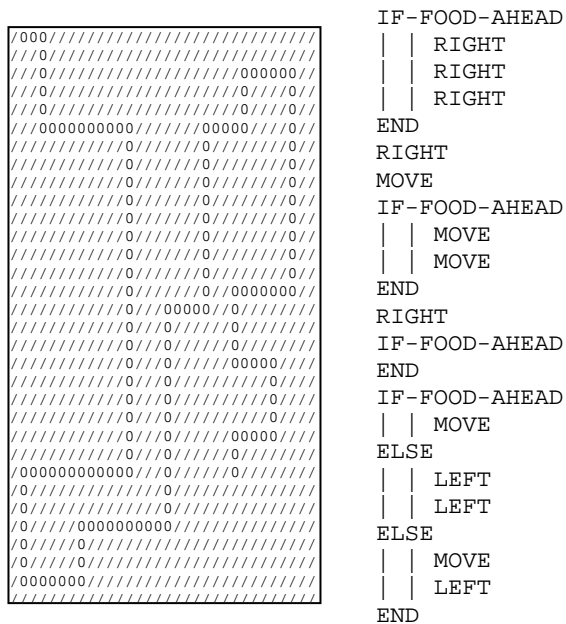
Terminals:	Left, Right, Move
Control Flow:	If-food-ahead, Else, End
Fitness case:	The Santa Fe Trail
Fitness:	Food remaining
Wrapper:	400 steps

The problem is run with two sets of control structure nodes – with conditional branching only, shown on Table 6, and with a full set of looping constructs, shown on Table 7. To prevent infinite loops, the wrapper was set to limit the total number of looping control executions to 400 steps as well.

**Table 7.** Tableau for the Ant Problem, with looping constructs

Terminals:	Left, Right, Move
Control Flow:	If-food-ahead, Else, End, Begin, While-food-ahead, Until-food-ahead, Repeat
Fitness case:	The Santa Fe Trail
Fitness:	Food remaining
Wrapper:	400 steps and 400 loops

Numerous solutions were found with both sets of control structures; usually the solutions were over fit to the trail, like on Figure 2: the 0 represents the number of times the ant returns onto a particular field. If it had moved onto any field twice, the content of the field would have been 1, etc. The evolved individual with conditional branching is shown next to the trail.



**Figure 2.** Ant trail through the grass with no loops, and evolved solution

When using the full set of control structure nodes, the obtained individuals exhibited significant bloat and were larger; here is one solution:

```

IF-FOOD-AHEAD MOVE MOVE IF-FOOD-AHEAD END MOVE IF-FOOD-
AHEAD END IF-FOOD-AHEAD WHILE-FOOD-AHEAD MOVE MOVE WHILE-
FOOD-AHEAD MOVE MOVE IF-FOOD-AHEAD END MOVE IF-FOOD-AHEAD
END IF-FOOD-AHEAD WHILE-FOOD-AHEAD MOVE MOVE WHILE-FOOD-
AHEAD MOVE MOVE IF-FOOD-AHEAD END IF-FOOD-AHEAD WHILE-
FOOD-AHEAD MOVE REPEAT IF-FOOD-AHEAD IF-FOOD-AHEAD IF-
FOOD-AHEAD IF-FOOD-AHEAD MOVE UNTIL-FOOD-AHEAD BEGIN
UNTIL-FOOD-AHEAD END MOVE RIGHT LEFT IF-FOOD-AHEAD MOVE
END IF-FOOD-AHEAD MOVE END ELSE END LEFT REPEAT END ELSE
WHILE-FOOD-AHEAD MOVE REPEAT IF-FOOD-AHEAD IF-FOOD-AHEAD
IF-FOOD-AHEAD END END LEFT REPEAT END ELSE END LEFT
WHILE-FOOD-AHEAD MOVE LEFT LEFT END END RIGHT RIGHT IF-
FOOD-AHEAD ELSE BEGIN END LEFT BEGIN END MOVE LEFT UNTIL-
FOOD-AHEAD

```

### The Los Altos Trail

Runs were performed on the significantly larger Los Altos trail using the branching set of primitives (no loops). The obtained solution was not over fit to the trail; it can be seen by its trace that it employs an algorithm of actively scanning the vicinity when the trail is lost in order to re-acquire it. Its trace is shown on Figure 3, and the solution follows.



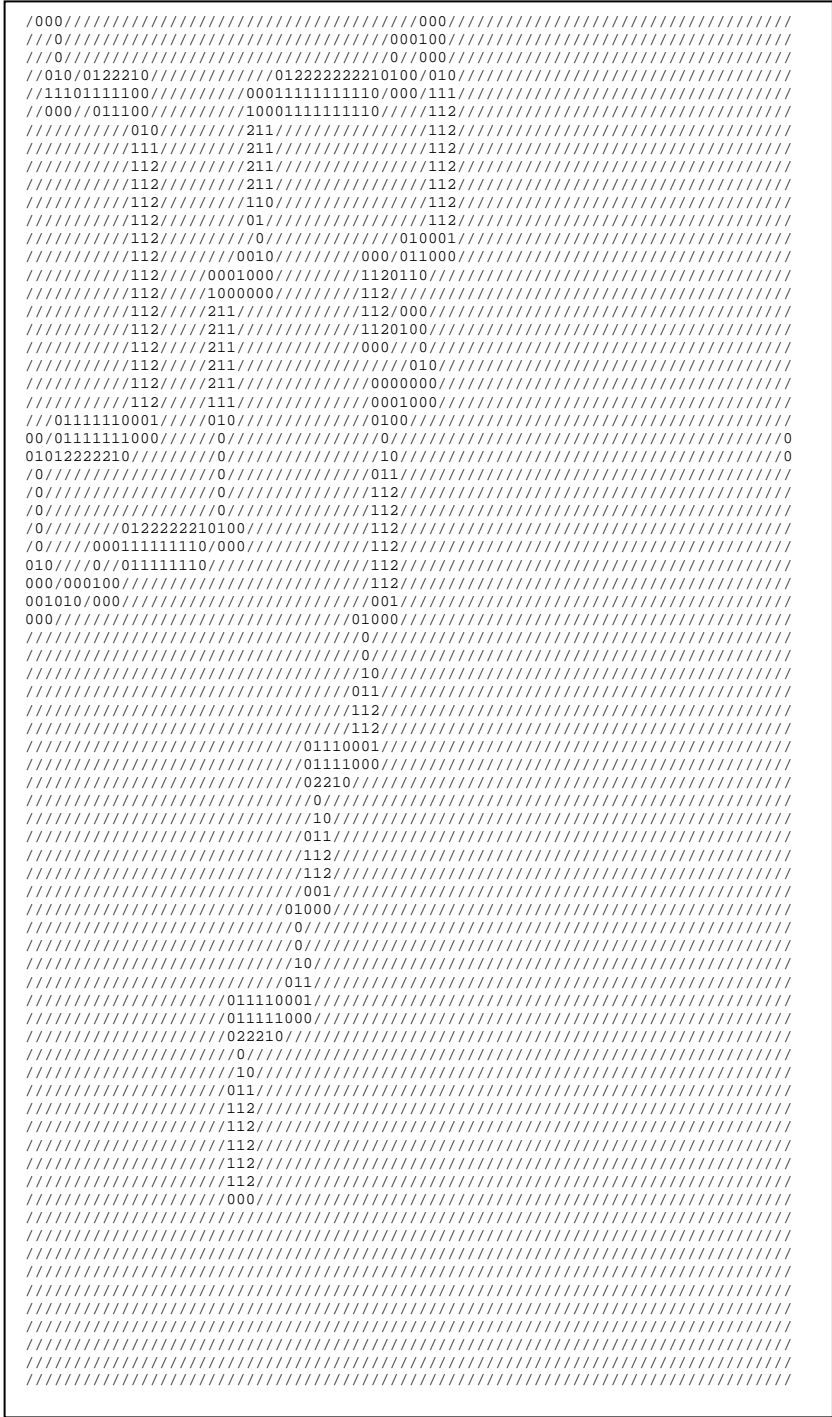


Figure 3. Ant movement on the Los Altos trail

Evolved solution to the Los Altos trail, no looping constructs:

```
IF-FOOD-AHEAD MOVE END MOVE IF-FOOD-AHEAD MOVE LEFT END
IF-FOOD-AHEAD ELSE END RIGHT MOVE IF-FOOD-AHEAD MOVE MOVE
MOVE RIGHT END IF-FOOD-AHEAD END MOVE IF-FOOD-AHEAD MOVE
MOVE RIGHT END IF-FOOD-AHEAD ELSE END RIGHT MOVE IF-
FOOD-AHEAD MOVE MOVE MOVE RIGHT IF-FOOD-AHEAD IF-FOOD-
AHEAD END END END IF-FOOD-AHEAD MOVE MOVE IF-FOOD-AHEAD
MOVE RIGHT MOVE END MOVE IF-FOOD-AHEAD IF-FOOD-AHEAD
RIGHT ELSE ELSE ELSE IF-FOOD-AHEAD END IF-FOOD-AHEAD ELSE
RIGHT END MOVE IF-FOOD-AHEAD MOVE END END MOVE RIGHT LEFT
RIGHT IF-FOOD-AHEAD RIGHT LEFT RIGHT IF-FOOD-AHEAD MOVE
RIGHT MOVE IF-FOOD-AHEAD MOVE ELSE END END MOVE RIGHT END
MOVE IF-FOOD-AHEAD LEFT RIGHT IF-FOOD-AHEAD RIGHT ELSE
END RIGHT ELSE IF-FOOD-AHEAD END ELSE RIGHT LEFT END
RIGHT END END RIGHT MOVE IF-FOOD-AHEAD MOVE MOVE MOVE
RIGHT IF-FOOD-AHEAD END RIGHT MOVE IF-FOOD-AHEAD ELSE
RIGHT END IF-FOOD-AHEAD LEFT RIGHT IF-FOOD-AHEAD RIGHT
ELSE ELSE ELSE RIGHT IF-FOOD-AHEAD END RIGHT MOVE IF-
FOOD-AHEAD END RIGHT IF-FOOD-AHEAD ELSE MOVE IF-FOOD-
AHEAD MOVE ELSE END END MOVE RIGHT MOVE IF-FOOD-AHEAD
MOVE END MOVE IF-FOOD-AHEAD MOVE LEFT END MOVE RIGHT IF-
FOOD-AHEAD IF-FOOD-AHEAD ELSE END RIGHT MOVE IF-FOOD-
AHEAD END END END IF-FOOD-AHEAD MOVE MOVE IF-FOOD-AHEAD
MOVE MOVE MOVE IF-FOOD-AHEAD MOVE MOVE RIGHT IF-FOOD-
AHEAD IF-FOOD-AHEAD END END END IF-FOOD-AHEAD MOVE RIGHT
END IF-FOOD-AHEAD ELSE RIGHT END ELSE ELSE RIGHT IF-FOOD-
AHEAD MOVE END MOVE END END MOVE END END MOVE IF-FOOD-
AHEAD MOVE MOVE MOVE RIGHT IF-FOOD-AHEAD END RIGHT MOVE
IF-FOOD-AHEAD ELSE RIGHT END MOVE END RIGHT
```

## Conclusion

It was shown how to efficiently implement universal control structures in linear and stack-based GP, and the results they produce when used to solve the artificial ant problem. The tools that make it possible are the well-understood stack-correct cross-over methods, and the usage of the control flow stack established in ANS Forth. It is hoped that this will encourage alternative approaches and increase the proportion of high-speed structured linear GP and stack-based GP as opposed to traditional tree-based.

## Acknowledgments

This work was supported in part by NSF grants ECS-9875705 and ECS-0196362.

## References

1. Wilker Shane Bruce. "The Lawnmower Problem Revisited: Stack-Based Genetic Programming and Automatically Defined Functions", In Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, USA, San Francisco, CA, USA, pp. 52-57, 1997.
2. Mike J. Keith and Martin C. Martin. "Genetic Programming in C++: Implementation Issues", In Advances in Genetic Programming, MIT Press, pp. 285-310, 1994.
3. John R. Koza. "Genetic Programming: On the Programming of Computers by Means of Natural Selection", Cambridge, MA, USA, MIT Press, 1992.
4. Tim Perkis. "Stack-Based Genetic Programming", In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, IEEE Press, pp. 148-153, 1994.
5. Lee Spector and Alan Robinson. "Genetic Programming and Autoconstructive Evolution with the Push Programming Language", In *Genetic Programming and Evolvable Machines*, pp. 7-40, 2002.
6. Kilian Stoffel and Lee Spector. "High-Performance, Parallel, Stack-Based Genetic Programming", In Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press, pp. 224-229, 1996.
7. Elko Tchernev. "Forth Crossover Is Not a Macromutation?", In Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, San Francisco, CA, USA, pp. 381-386, 1998.
8. Elko Tchernev. "Stack-Correct Crossover Methods in Genetic Programming", In Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002), New York, NY, AAAI, pp. 443-449, 2002.