Genetic L-System Programming

Christian Jacob

Chair of Programming Languages, Department of Computer Science,
University of Erlangen-Nürnberg, Martens-Str. 3, D-91058 Erlangen, Germany
email: jacob@informatik.uni-erlangen.de

Abstract. We present the Genetic L-System Programming (GLP) paradigm for evolutionary creation and development of parallel rewrite systems (L-systems, Lindenmayer-systems) which provide a commonly used formalism to describe developmental processes of natural organisms. The L-system paradigm will be extended for the purpose of describing time- and context-dependent formation of formal data structures representing rewrite rules or computer programs (expressions).
With GLP two methods gleaned from nature are combined: simulated evolution and simulated structure formation. A prototypical GLP system implementation is described. Controlled evolution of complex structures is exemplified by the development of tree structures generated by the movement of a 3D-turtle.

1 L-Systems

The development of an organism may [...] be considered as the execution of a 'developmental program' present in the fertilized egg. The cellularity of higher organisms and their common DNA components force us to consider developing organisms as dynamic collections of appropriately programmed finite automata. A central task of developmental biology is to discover the underlying algorithm for the course of development.

Aristid Lindenmayer and Grzegorz Rozenberg [6]

Morphogenesis or formation of structures in nature are always the result of complex growth processes. The central idea of L-systems is that structure formation can be interpreted as the execution of 'programs' or rewrite rules. In nature there is no blue print for an organism, instead 'rule systems' tell how to build organels and how to combine these parts to form a complete and functioning organism. These programs are highly parametrized where the parameters are set by the environment in which development and interaction processes take place.

Parallel rewrite systems or L-systems [7] provide a useful formal model for the description of developmental processes in organisms. We will give some rudimentary definitions for context-free L-systems with stacking capability. As it is in general very difficult to create an L-system simulating some special growth process we will introduce an evolutionary method (GLP) supporting L-system inference.

Genetic L-System Programming, PPSN III - Parallel Problem Solving from Nature, International Conference on Evolutionary Computation, Lecture Notes in Computer Science 866, Springer-Verlag, Berlin, pp. 334 - 343, 1994.

1.1 DOL-systems

The context-free D0L-systems[1] are the simplest type of L-systems. Formally a D0L-system can be defined as a triple $G = (\Sigma, P, \omega)$ where $\Sigma = \{s_1, s_2, \ldots, s_n\}$ is an *alphabet*, P is an endomorphism defined on Σ^* , and ω, referred to as the *axiom*, is an element of Σ^* . P is defined by a *production map* $P\colon \Sigma \to \Sigma^*$ with $s \to P(s)$ for each $s \in \Sigma$. Whenever there is no explicit mapping for a symbol s the identity mapping $P(s) = s$ is assumed. In a deterministic L-system there is at most one production rule for each symbol $s \in \Sigma$. The word sequence $E(G)$ generated by G is defined as

$$\omega^{(0)} = P^0(\omega),\ \omega^{(1)} = P^1(\omega),\ \omega^{(2)} = P^2(\omega),\ \ldots$$

where P^i denotes i-fold iteration of P and each string $\omega^{(i+1)}$ is obtained from the preceding string $\omega^{(i)} = \omega_1^{(i)}\omega_2^{(i)}\ldots\omega_m^{(i)}$ by applying the production rules to all m symbols of the string $\omega^{(i)}$ simultaneously:

$$\omega^{(i+1)} = P(\omega_1^{(i)})P(\omega_2^{(i)})\ldots P(\omega_m^{(i)}).$$

The language of G is defined by $L(G) = \{P^i(\omega); i \geq 0\}$.

1.2 Turtle interpretation of bracketed parametric DOL-Systems

Let us consider the following L-system G:

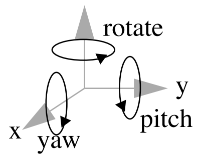
$$\begin{aligned}
&\Sigma = \{F, B, Rl, Rr, P, Pb, Yl, Yr\}\\
&\omega\colon \quad F\\
&P\colon \quad F \to [F[RlF]F[RrF]F]
\end{aligned} \tag{1}$$

which generates the following sequence of strings:

Axiom:	F
Iteration 1:	$[F[RlF]F[RrF]F]$
Iteration 2:	$[[F[RlF]F[RrF]F][Rl[F[RlF]F[RrF]F]][F[RlF]F[RrF]F]]$ $[Rr[F[RlOF]F[RrF]F]][F[RlF]F[RrF]F]$

These string sequences describe the fractal growth of an artificial structure. The structure formation process can be easily visualized if we define the following interpretation for the symbols $F, B, Rl, Rr, P, Pb, Yl, Yr$ and [...]. A common interpretation is to let these symbols control the movement of an artificial object (usually known as a 'turtle') which draws lines on its way in 2- or 3-dimensional space :

$F(s_1)$	:	move forward with a stepsize of s_1
$B(s_2)$	:	move backward with a stepsize of s_2
$Rl(\alpha_1)$	:	rotate left for an angle α_1
$Rr(\alpha_2)$	:	rotate right for an angle α_2



1. D0 stands for *deterministic* with *no context.*

$P(\alpha_3)$	:	pitch for an angle α_3
$Pb(\alpha_4)$	:	pitch back for an angle α_4
$Yl(\alpha_5)$	:	yaw left for an angle α_5
$Yr(\alpha_6)$	:	yaw right for an angle α_6.

In the example system above we have quietly assumed a fixed rotation angle $\alpha_1 = 90$, $\alpha_2 = 60$ and stepsize $s = 0.5$ so we do not have to include these parameters into the strings, however, this was only done in order to keep strings small.

Modular substrings can be marked by the bracket symbols [and]. For each string of the form $s_1[s_2]s_3$ the strings $s_1, s_2, s_3 \in L(G)$ are interpreted in sequence, however, after substring s_2 has been interpreted and before starting to interpret s_3 the turtle is reset to its prior position and orientation after interpretation of s_1. This allows the formation of tree-like structures and branches as the visualization of iterated turtle movement for the example above shows (figure 1).

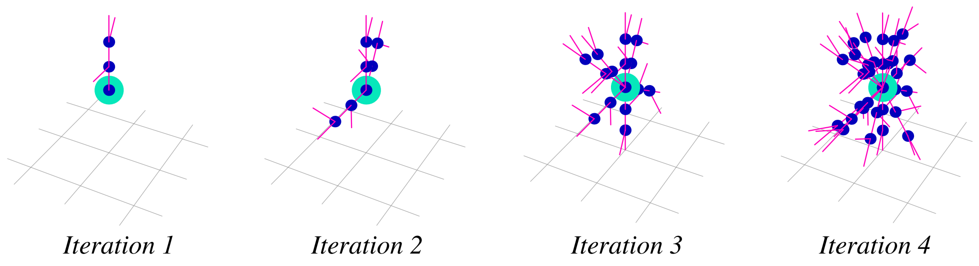


Fig. 1. Artificial structure generated with DOL-system described in (1). The turtle is oriented upward and its origin is situated at the big spot in the center.

2 L-Systems and Genetic Programming

2.1 Synthesis of L-systems

The inference problem for L-systems involves finding a proper axiom ω and rewrite rules P for a given structure or growth process, i.e. a sequence of structures. For the development of an L-system for a particular (biological) species one usually has to perform the following steps [8]:

1. analyzation of the biological object,
2. informal rules definition,
3. definition of L-system axiom and rules,
4. computer simulation and interpretation of generated strings,
5. translation into a graphical output,
6. comparison of the artificial object with the behavior of the real object,
7. correction of the L-system and repetition of the steps above (if necessary).

This shows that L-system synthesis is an overall difficult and sometimes tedious process. But what methods do we have at hand for (automatic) generation of L-systems? As P. Prusinkiewicz [9] points out, random modification of production rules generally gives little insight into the relationship between L-systems and the figures they generate. Algorithms reported in the literature up to now are still too limited to be of practical value for complex structure formation [9, p. 39, 62]. Obviously an evolutionary approach is sensible for points 3, 6 and 7. So what we need to support L-system inference on an evolutionary basis is:

- functions to *generate* (possibly codings of) L-systems that are subject to certain constraints (alphabet, iterations, context-sensitivity, parameters etc.),
- *evaluation* functions that return a fitness measure for each interpreted L-system,
- *modification* and *selection* functions which enable interactive L-system editing as well as automatic control through evolutionary techniques.

In the following sections we discuss preliminary ideas about the use of evolutionary techniques for breeding populations of L-systems that describe growth processes which are interpreted in a problemspecific domain and evaluated by a fitness measure with respect to a target growth process.

2.2 Extended GP and GLP

Here we briefly describe what kind of evolutionary algorithm system we use for L-system development and coding. Similar to the genetic programming (GP) paradigm introduced by J. Koza [5] who uses LISP-S-expressions our structures undergoing adaptation are hierarchical, typed expressions (terms).[2]

One of the main differences to the common GP paradigm is the use of higher-order building blocks ('patterns') for expression generation and modification. The coarse structures of problemdependent genotype expressions are generated by combining 'macro-patterns' taken from a predefined pattern pool $Pool = \{p_1, \ldots, p_M\}$ (see the example patterns around the centered circle in fig. 2). The combinable subexpressions rely on a set of function symbols $F = \{f_1, f_2, \ldots, f_N\}$ for each of which an arity range $A = \{a_1, \ldots, a_N\}$ with $a_i = (minarg(f_i), maxarg(f_i))$ has to be specified.

Each expression from the pattern pool serves as a (possibly partial) descriptions of "organism" genotypes for a problem dependent environment. Only the expression patterns are used for expression generation, i.e. parametrized, possibly constrained, high-level data structures serve as building blocks. Each of these patterns p_i is associated with a set of attributes as e.g. a number of predicates constraining the set of subexpressions that can be 'plugged in'. Another attribute is the pattern rank $r(p_i)$ which serves as a kind of fitness measure among patterns that compete for being selected as subexpressions during the expression generation process.[3] This concerns patterns with the same root symbol - as is the case with the recursive and non-recursive version of the *stack*-pattern - as well as with different function symbols within alternatives (fig. 2).

2. For an alternative grammar-based approach see [1]. An excellent overview of current extensions and applications of GP can be found in [5]

3. Similar ranks control pattern selection of the genetic operators.

Specialized meta-operators for rank adjustment take care about which patterns enhance the pool and for which patterns focus is increased or decreased through ranks adjustment.

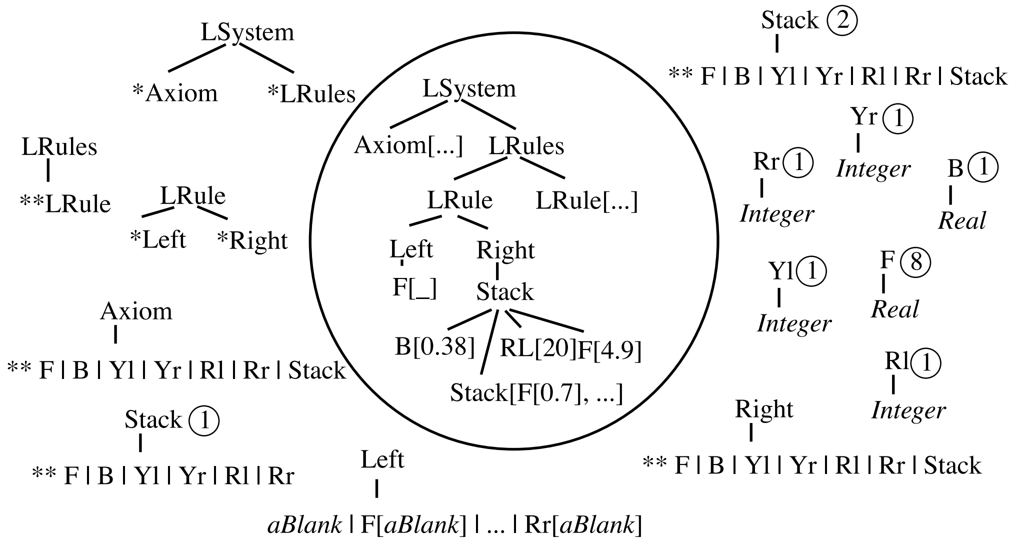


Fig. 2. : Pool of expression patterns. The coarse structure of the L-system description within the centered circle is built by using the depicted expression patterns. *X and **X stand for any single expression or (non-empty) sequence of expressions with head X, respectively. | denotes alternatives. Pattern ranks are depicted within small circles.

2.3 Expression generation

Evolution starts with random generation of an initial population of expressions. Each expression is constructed from a start pattern in a recursive manner by combining expressions from the expression pattern pool, always respecting the pattern constraints as discussed in the previous section.

Generation of an L-system expression might result in the following generation steps[4]:

```
LSystem[_Axiom,_LRules]
LSystem[Axiom[Stack[_F,_RR,_F]],_LRules]
...
LSystem[Axiom[Stack[F[0.8],RR[70],F[1.5]]],_LRules]
LSystem[Axiom[Stack[F[0.8],RR[70],F[1.5]]],
        LRules[_LRule,_LRule,_LRule]]
LSystem[Axiom[Stack[F[0.8],RR[70],F[1.5]]],
        LRules[LRule[_Left,_Right],_LRule,_LRule]]
...
```

4. Here '_' and '_X' represent formal parameters referring to any expression and expressions with head *X*, respectively.

```
LSystem[Axiom[Stack[F[0.8],RR[70],_F[1.5]]],
   LRules[LRule[Left[F[aBlank]]],Right[Stack[RL[110],F[2.],Stack[...],P[50]]],
            LRule[Left[PB[aBlank]]],Right[Stack[B[2.8],Stack[...],RR[70]]], _LRule]]
...
```

These expressions are decoded into a parametrized bracketed L-system of the following form

ω: F(0.8) RR(70) F(1.5)
P: F(_) $\rightarrow$ RL(110) F(2.) [...] P(50)
PB(_) $\rightarrow$ B(2.8) [...] RR(70)

which is then interpreted by a 3D-turtle as demonstrated above.

2.4 Evaluation and reproduction of expressions

The population of L-system genotypes consists of symbolic expressions (data structures) the head symbols of which denote (abstract) data types for which decoding, interpretation and evaluation functions are easily definable by pattern matching mechanisms. This enables simultaneous use of different kinds of L-system genotypes, e.g. merging context-free and context-sensitive L-systems within the same population by introducing a new context-dependent *CLSystem* data type with according interpretation functions. Fitnesses are derived from the L-system interpretation functions so that each L-system genotype receives an associated fitness value.

In order to build the next generation of expressions a genetic operator op_i is chosen from an ‘operator pool’ $OpPool = \{op_1, \ldots, op_K\}$ depending on its operator rank $r(op_i)$. Each operator op_i performs a mapping $o(p_i) : G^{a_1(i)} \to G^{a_2(i)}$ from an $a_1(i)$-to a $a_2(i)$-dimensional genotype vector where G is the set of genotype expressions. The individual genotypes are selected according to their fitness values (fitness proportionate, rank-based or other selection schemes). The resulting, possibly modified expressions are entered into the next generation. The selection of genetic operators terminates when the new population is filled up to its maximum size.

2.5 Variations on expressions

Size and shape of the expressions change dynamically during the evolution process through genetic operators. Table 1 gives an overview of operators we currently use. We introduce an alternate selection scheme for subexpressions as arguments for the genetic operators: (possibly constrained) patterns provide templates used for extracting subexpressions for modification or recombination. This enables operators to be applied only within predefined expression contexts where context may vary in the course of the evolution process. For the definition of new patterns and contexts meta-operators (*te*, *ec*) are necessary. Similar to the pattern pool for expression generation there is a pattern pool $Pool_{op_i}$ for each genetic operator op_i; each pattern is associated with a rank number which controls selection among competing patterns. We explain these ideas in detail for the mutation and crossover operators which rely on the templates defined in table 2.

Pattern-Operator		**Short explanation ...**
Mutation	*mu*	Replace subterms of an expression meeting template constraints by newly generated, equivalent subexpressions.
Crossover	*co*	Exchange subexpressions meeting template constraints between two expressions.
Shrink	*sh*	Delete a subexpression.
Duplication	*du*	Duplicate a subexpression.
Permutation	*pe*	Permute expression arguments randomly, by left or right shift, or by reversion.
Template Extr.	*te*	Extract a template from an expression. 'Successful' templates are inserted into the pattern pools.
Encapsulation	*ec*	Replace a subexpression by a single reference symbol.

Tab. 1 : GP operators collection. For operand selection all the operators rely on operator templates.

GP operator	**Rank**	**Templates for selection of operator agruments**
Mutation $T_1(mu)$	3	Axiom[i: *Stack /; Q[i]] *Restrict mutation to expressions with head* Axiom *that have a* Stack *expression complying with a predicate* Q *as their argument.*
$T_2(mu)$	1	*LRules Restrict mutation to expressions with head LRules.
$T_3(mu)$	2	LRule[*Left,Right[**]] Restrict mutation to expressions with head Left appearing within an LRule expression and with a Right term as right context.
Crossover $T_1(co)$	1	*LRule[Left[*], Right[**, Stack[**], **]] Restrict crossover to LRule expressions that contain at least one Stack expression among the Right term arguments.

Tab. 2 : Pattern pool (templates) for GP operators

2.5.1 Pattern Mutation

To perform *pattern mutation* on an individual expression (figure 3a) a mutation template $T_1(mu)$ is selected from the pattern pool $Pool_{mu}$ according to the pattern ranks. Suppose the first template has been selected with predicate Q demanding at least three arguments for the *Stack* term. The subexpression with head

Axiom is then replaced by a newly generated *Axiom* term with a *Stack* argument expression resulting in a modified individual genotype (figure 3b).

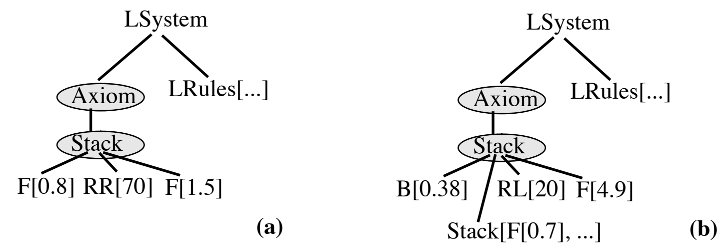


Fig. 3. : Pattern mutation on an L-system genotype

2.5.2 Pattern Crossover

Pattern crossover is used as a recombination operator which enables exchange of structures of the same type between two individuals. Given the two expressions in figure (4, top) a crossover template $T_1(co)$ is chosen from pattern pool $Pool_{co}$ with according ranking scheme. Subexpressions with head *LRule* meeting the restrictions of template $T_1(co)$ are selected randomly within each expression and exchanged between the two individuals resulting in two modified expressions (figure 4, bottom).

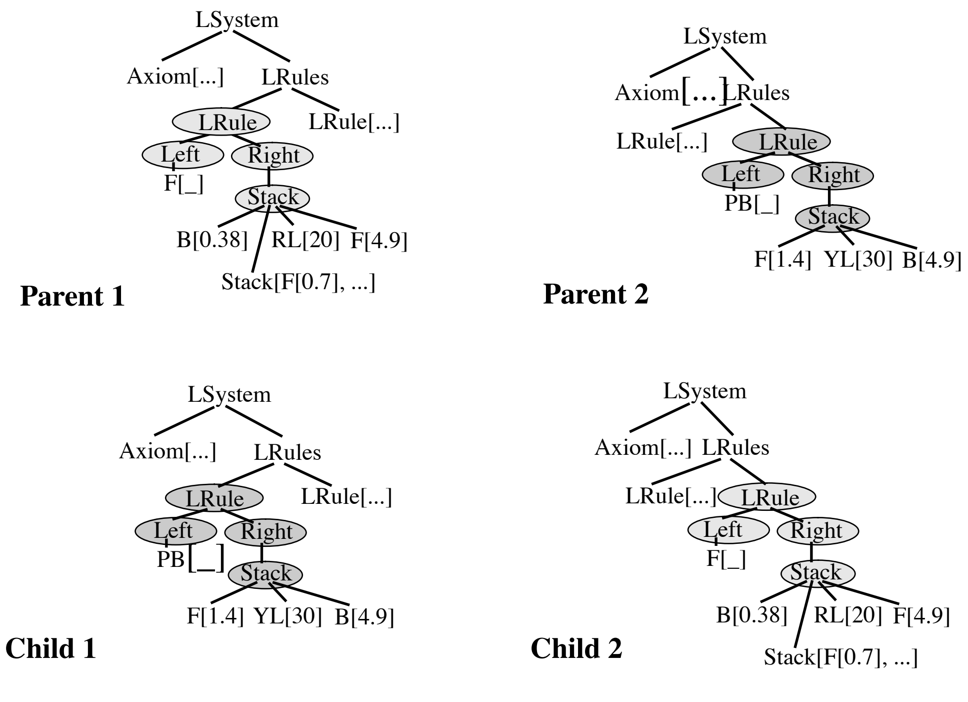


Fig. 4. : Expression recombination: pattern crossover

3 Virtual Genetic L-System Laboratory

In order to test and support problemspecific generation and evolution of expressions within the extended genetic programming paradigm we are designing a virtual GLP laboratory as one part of our genetic programming environment *MathEvolvica*. The following examples should give a brief impression of the surprisingly easy formation of complex structures even with very small populations (between 10 and 20 individuals per generation) and over a short period of generations. The simple problem to be solved was to generate L-systems that form a complex structure (with a number of $0 < b \le 100$ branches) and with the majority of tree end points (leaves) situated outside the inner cube but within the outer cube boundaries with regard to the horizontal x_1- and x_2-directions (figure 5). The number of L-system iterations was fixed to 3. The axiom and L-rule expressions had to be evolved. The fitness value $f(g_i)$ for each individual L-system genotype g_i, $1 \le i \le N$, to be maximized was defined as

$$f(g_i) = b(g_i)^{1 + penalty(g_i)} \quad , \quad penalty(g_i) = \sum_{k = 1, 2} penalty(x_k, g_i)$$

where $penalty(x_k, g_i)$ is the portion of x_k leaf coordinates lying within the specified boundaries with each leave having coordinates of the form (x_1, x_2, x_3). The following figures show a collection of interpreted L-systems all derived from a single genotype by applying crossover and mutation over 20 generations. The phenotypes develop to densely packed structures with broad branching.

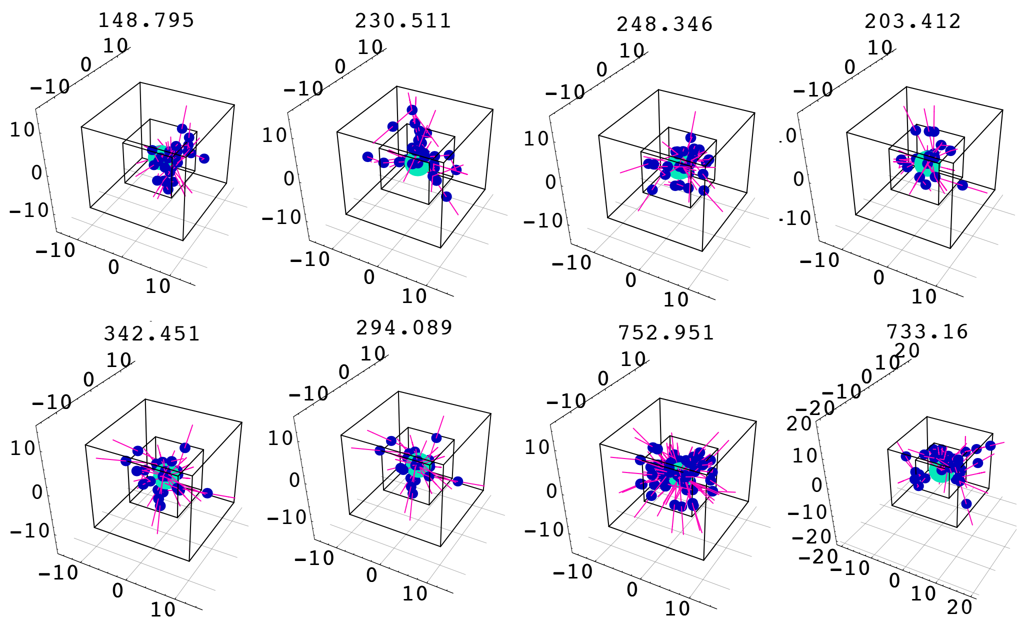


Fig. 5. : Collection of L-system turtle interpretations derived as mutants from the genotype of the first L-system individual (upper left corner). Depicted numbers refer to phenotype fitness.

4 Conclusion and Further Research

We demonstrated how parallel rewrite systems can be designed by evolution processes. Of course, this is only a very limited and rudimentary description of how genetic programming techniques support the design of hierarchical (program or data) structures. Extensions of the GLP/GP laboratory[5] are currently developed and implemented with respect to the following areas:

- using fitness functions that measure similarities among growth processes in order to infer L-systems for (sequences of) target structures,
- including growth functions into fitness evaluation,
- extending the set of interpretation functions,
- extension to context-sensitive, stochastic and table L-systems.

Another important area of research are genetic operators that support hierarchical, modularized expression evolution. A variant of the described GLP system will be used for the design of artificial neural networks.[6]

References

1. Antonisse, H.J., *A Grammer-Based Genetic Algorithm*, in: G. Rawlins (ed.), Foundations of Genetic Algorithms, San Mateo, 1991.
2. Jacob, C., Rehder, J., *Evolution of neural net architectures by a hierarchical grammar-based genetic system*, ICNNGA'93, International Conference on Neural Networks and Genetic Algorithms, Innsbruck, Austria, 1993.
3. Jacob, C., *Typed expressions evolution of artificial nervous systems*, to appear in: ICANN'94, International Conference on Artificial Neural Networks, Sorrento, Italy, 1994.
4. Kinnear, K.E., *Advances in Genetic Programming*, MIT Press, London, 1994.
5. Koza, J.R., *Genetic Programming, On the Programming of Computers by Means of Natural Selection*, MIT Press, London, 1993.
6. Lindenmayer, A., Rozenberg, G. (eds.), *Automata, Languages, Development*, North-Holland, 1975.
7. Lindenmayer, A., *Mathematical models for cellular interaction in development*, Parts I and II, Journal of Theoretical Biology 18, 1968, pp. 280-315.
8. Peitgen, H.-O., Jürgens, H., Saupe, D., *Chaos and Fractals*, New Frontiers of Science, Springer-Verlag, 1993, p. 363.
9. Prusinkiewicz, P., Lindenmayer, A., *The Algorithmic Beauty of Plants*, Springer-Verlag, 1990, pp. 11ff.

5. The GP laboratory is implemented in C and *Mathematica*. Currently the user interface is realized through *Mathematica* notebooks and will be extended by a NeXTSTEP based graphical user interface (*Mathematica* is a trademark of Wolfram Research, Inc. NeXTSTEP is a trademark of NeXT Computers, Inc.)
6. Alternative approachs for neural net design with the expression evolution system are described in [2] and [3].