Automatic composition of music by means of Grammatical Evolution

Alfonso Ortega de la Puente
Universidad Autónoma de Madrid
e-mail: alfonso@ii.uam.es

Rafael Sánchez Alfonso
Universidad Autónoma de Madrid & IBM
e-mail: rafael_sanchez@es.ibm.com

Manuel Alfonseca Moreno
Universidad Autónoma de Madrid
e-mail: Manuel.Alfonseca@ii.uam.es

Main topics: computer music, automatic composition, grammatical evolution, formal languages and grammars, genetic algorithms, evolutionary programming.

Acknowledgement

This paper has been sponsored by the Spanish Interdepartmental Commission of Science and Technology (CICYT), project numbers TEL1999-0181 and TIC2001-0685-C02-1.

Abstract

This work describes how grammatical evolution may be applied to the domain of automatic composition. Our goal is to test this technique as an alternate tool for automatic composition. The AP440 auxiliary processor will be used to play music, thus we shall use a grammar that generates AP440 melodies. Grammar evolution will use fitness functions defined from several well-known single melodies to automatically generate AP440 compositions that are expected to sound like those composed by human musicians.

1. Introduction

The automatic generation of musical composition is a multi disciplinary area of interest and research. Some of the current approaches try to simulate how the musicians play [1] or improvise [2] while other do not deal with the time spent in the process. The greater part of them applies models of Theoretical Computer Science (cellular automata [3], parallel derivation grammars [1] and evolutionary programming [4-7]) to the generation of complex compositions. A musical meaning is given to the models and the music is automatically found (composed) by means of genetic programming. Our group is interested in the simulation of complex systems by means of formal models, their equivalence, and their design not only by hand, but also by means of automatic processes such as genetic programming. Grammatical evolution is a variation of genetic programming that uses formal grammars to represent the populations, so the formal aspect of evolutionary computation is reinforced.

A brief introduction to formal grammars

Formal grammars are one of the main topics in Theoretical Computer Science and were introduced in the fifties by the North American linguist Avram Noam Chomsky and are used to formalize both natural and programming languages. The reader could find detailed descriptions and definitions of formal grammars in his publications [8] or in any book on Theoretical Computer Science [9]

Formal grammars formalize the syntactic rules used to build or analyze phrases of a given language. The following example shows a formal grammar for a subset of an arbitrary programming language:

the sentences that assign an arithmetic expression to a numeric variable. Let us suppose that the allowed operations are `+`, `-`, `×` and `÷`, and the valid operands are numbers and variables which are named by vowels. The first word of an assignment must be the name of the variable, the second must be the symbol `←` and the rest of the statement is the expression whose value is assigned. The following sentences are correct assignments:

```
a←2×a          b←a×(365+b)
```

A formal grammar could describe the structure of the assignment in this way:

```
Rule 1: Assignment is Variable ← ArithmeticExpression
```

Where the words `Assignment`, `Variable` and `ArithmeticExpression` will be further defined and will not be written like this in the final assignment. The symbol `←` must literally appear.

Variables' names could be described by the grammar by means of the rule

```
Rule 2: Variable is a or e or i or o or u
```

Arithmetic expressions follows the rule

```
Rule 3: ArithmeticExpression is Name or Variable or (ArithmeticExpression)
          or ArithmeticExpression Operation ArithmeticExpression
```

Where numbers are written in the usual way and operations are defined as follows:

```
Rule 4: Operation is + or - or × or ÷
```

The structure of the first example could be described by means of the successive application of these rules. Figure 1 shows the process.

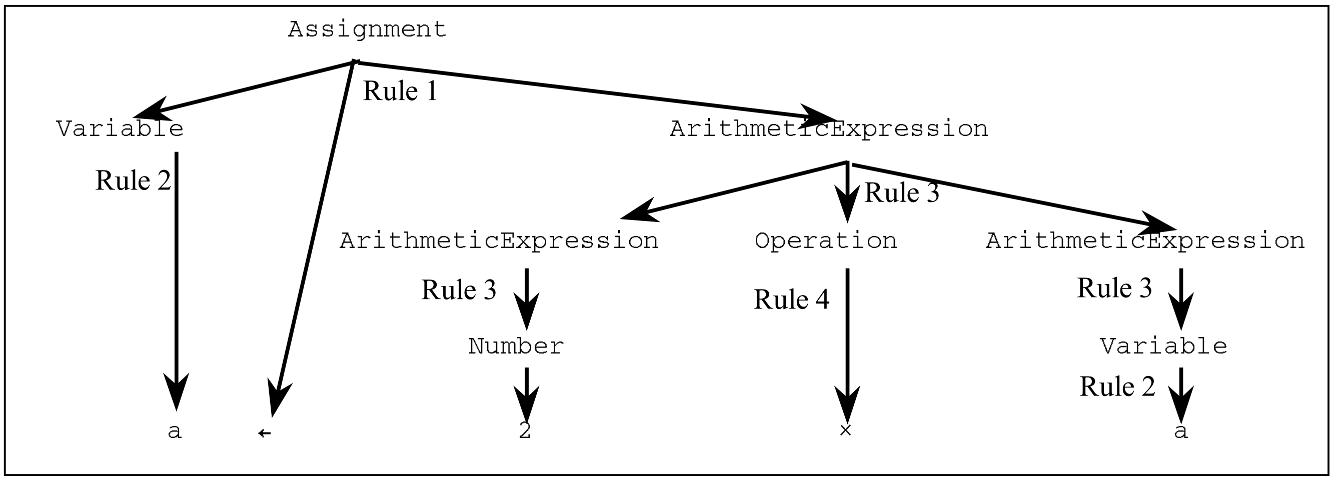

Figure 1: A grammar is used to analyze the assignment `a←2×a`

There are several kinds of grammars. This example shows a Chomsky grammar that does not depend on the context (a context free grammar). All the grammars used in this paper are context free.

A brief introduction to musical parameters

In the following paragraphs, some essential musical theory concepts will be reviewed. Melody, rhythm and harmony are considered the three fundamental elements in music. This work is focused on melody.

Melody is a series of musical silences and sounds (or notes) with different lengths and stress arranged in succession in a particular rhythmic pattern to form a recognizable unit.

Notes' names belong to {A, B, C, D, E, F, G}. These letters represent musical pitches and correspond to the white keys on the piano. The black keys on the piano are used to play sharp or flat notes. From left to right, the key that follows any white one is its sharp key and the previous key is its flat key. The name of these keys adds respectively the symbols # and ***b*** to the white key's name (for example A# or B***b***). The interested reader could find an amusing simulation of a virtual keyboard at [20].

The distance from a note to its flat or sharp notes is called "a half step" and is the smallest unit of pitch used in the piano, where there is a half step between every pair of adjacent keys, no matter their color. Two consecutive half steps are called a whole step.

The maximum length of notes and rests is considered the unit. There are seven different lengths (from 1 to 1/64), each of which has double duration than the next: complete, half, quarter, quaver, semi quaver, quarter quaver and half quarter quaver. The complete specification of notes and silences includes their lengths.

An interval is the number of half steps between two notes.

A brief introduction to genetic algorithms

The following paragraphs are extracted from the excellent work on heuristics by Z. Michalewicz and D. Fogel [19] The interested reader could find a more detailed introduction to this topic in any general text on evolutionary computation, genetic programming or heuristics.

Genetic algorithms provide a computational model inspired by the evolutionary process of natural competition and selection. When trying to solve a problem, genetic algorithms starts with a population of initial candidate solutions usually generated at random and represented as strings of symbols. A proper evaluation or fitness function is defined to measure how good is a solution. Those solutions which are better, as determined by the fitness function, become the parents for the next generation of offspring that replace the discarded individuals. Gene transmission (crossover), mutation and other biological processes can be implemented in genetic algorithms as string operations: two new individuals may be obtained from their two parents by taking the first half of one parent and the second half of the other; an individual may be mutated by randomly changing some symbol.

With each generation, the individuals compete for inclusion in the next iteration. After a series of generations, it is expected that the quality of the tested solution converges toward a nearly optimum solution to the problem.

A brief Introduction to grammatical evolution

Grammatical evolution [10-17] is a grammar based, linear genome system, which has been applied in the area of automatic programming to automatically generate programs or expressions in a given language to solve a particular problem. Programming languages can be represented usually by context free Chomsky languages. In grammatical evolution, the Backus Naur Form (BNF) specification of a language is used to describe the output produced by the system (a compilable code fragment). Different BNF grammars can be used to automatically produce code in any language.

In grammatical evolution, the genotype is a string of binary numbers (each of which is named *codon*) generated at random, treated as integer values. The phenotype is a running computer program generated by a genotype-phenotype mapping process. The mapping benefits from genetic code degeneracy; i.e. different integers in the genotype generate the same phenotype.

The genotype-phenotype mapping in grammatical evolution is deterministic; i.e. each individual is always mapped to the same phenotype. Two mechanisms can be used to minimize the numbers of invalid individuals in each generation: punishing them with poor fitness values, or using a steady state replacement method [15, 16]. The last method seems to improve greatly the performance of the algorithm.

Figure 2 shows a scheme of the process as a biological metaphor. Grammar evolution adds a grammatical translation level (genotype to phenotype mapping) to classic genetic algorithms: genetic operators act on genotypes while fitness function evaluates phenotypes.

This technique has been successfully applied to the automatic programming of problems in different domains: symbolic regressions, finding trigonometric identities, the Santa Fe ant trail, and caching algorithms. Our group has previously applied grammatical evolution in the domain of fractal curves.

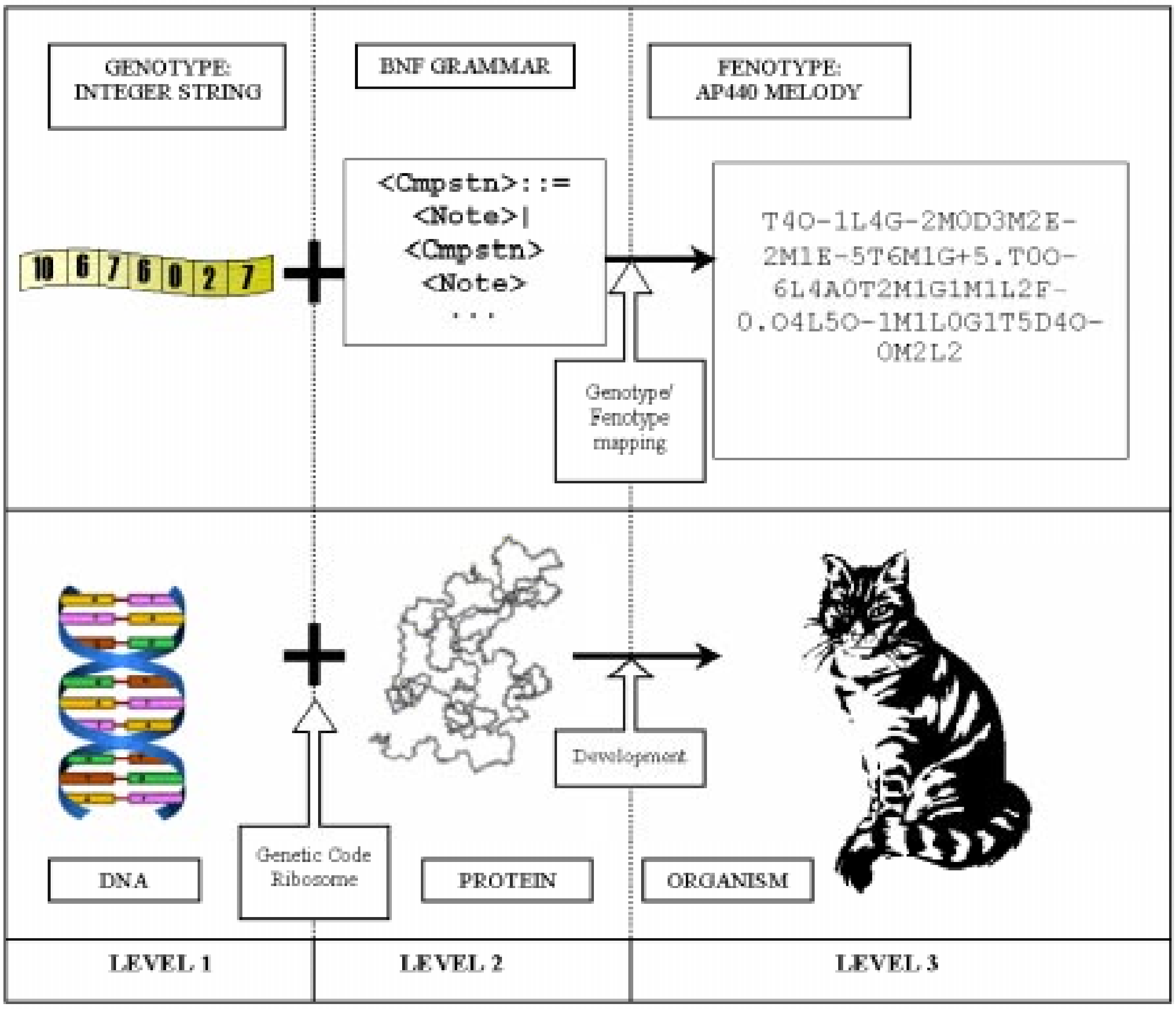
GENOTYPE: INTEGER STRING
BNF GRAMMAR
FENOTYPE: AP440 MELODY
10 6 7 6 0 2 7
<Cmpstn>::= <Note>| <Cmpstn> <Note> ...
T4O-1L4G-2MOD3M2E-2M1E-5T6M1G+5.TOO-6L4AOT2M1G1M1L2F-0.O4L5O-1M1LOG1T5D4O-OM2L2
Genotype/ Fenotype mapping
Development
DNA
Genetic Code Ribosome
PROTEIN
ORGANISM
LEVEL 1
LEVEL 2
LEVEL 3

Figure 2: Grammar evolution layout

2. Music and APL2

APL2 provides the auxiliary processor AP440 to easily create music from the PC speaker. The reader can find a description of AP440 in any IBM PC APL2 manual [18].

The auxiliary processor interprets as music every character string assigned to the shared variable with the following syntax:

```
SHR440←∇{[tempo][octave][mode][length][note][pause]}→∇
```

Where

tempo: Tn(n=0 to 6; default 4)	*Length: Ln(n=0 to 6; default 0)*
Octave: On[{+ -}](n=0 to 6; default 3)	*Note: tone[{# + -}][n][.]* *(tone=A to G; n=0 to 6, default 0)*
Mode: Mn(n=0 to 2; default 0)	*Pause: P[n][.](n=0 to 6; default 0)*

3. Description of the experiment

The goal of this paper is to automatically generate melodies that sound like those composed by human authors. First, several well-known compositions will be selected and translated into AP440 melodies. Then, a set of n parameters will be extracted from the melodies and studied; each melody will become a vector whose components will be the selected parameters. Our purpose is to identify a combination of parameters and values that will distinguish random and hand-made melodies. We expect that a melody whose parameters are close to the values typical for melodies composed by humans will sound like these melodies. Finally, grammar evolution will be used to automatically generate melodies with the characteristics previously described. This step involves the specification of a fitness function that will be based on the distance between the target parameters and the candidate melodies' vector.

A grammar for AP440's music

In order to solve this problem by means of grammatical evolution we have defined a formal grammar to express compositions.

The following Chomsky grammar generates AP440 music. The grammar is expressed in BNF (Bakus-Naur Form) and uses the following conventions:

- Angular brackets embrace non-terminal symbols, expressions that do not belong to the language of AP440 compositions, but can be replaced by other symbols to generate correct AP440 melodies (for example `<Composition>`).
- Names not embraced by angular brackets represent terminal symbols, expressions that can belong to an AP440 composition, and cannot be replaced by any symbol. They will appear literally in the final composition.
- The derivations of the grammar are represented in the following way:

  `<Non terminal>::=Expression`

  - Where `Expression` is a string of terminal and non terminal symbols
- The symbol | replaces the word "or" used in the first example of a formal grammar.
- The right hand side of several rules is followed by parenthesized real numbers. These rules are probabilistic and their numbers show the probability for each option to be chosen.
- The symbol ε stands for the empty word (a word without symbols) and it is used to erase non-terminal symbols.

```
<Composition ::=<Note (1/5)|
                  <Composition <Note (4/5)

<Note ::=<T <O <M <L <N

<T ::=□ | T<n_7

<n_7 ::=0|1|2|3|4|5|6

<O ::= □ | O<□_O <n_7

<□_O ::= □ | + | -

<M ::= □ | M<n_3

<n_3 ::=0|1|2

<L ::= □ | L<n_7

<N ::= □ (1/7)|<_Note (5/7)|
      <Pause (1/7)

<_Note ::=<Sound <□_N <n_7 <Dot

<Sound ::=A|B|C|D|E|F|G

<□_N ::= □ |<Sharp |<Flat

<Sharp ::=#|+

<Flat ::=-

<Dot ::= □ |.

<Pause ::=P<n_7 <Dot
```

Description of the GE algorithm implementation

Initial population

The initial population has 64 vectors of 32 random codons (integer numbers from the interval [0,255]). A length equal to thirty-two seems to be suitable for the genotypes.

Genetic operators

The classic genetic operators crossover and mutation are extended with an enlarging operator (that splices the genotype and another string of codons) and a cutting operator (that removes some of the codons).

The number of discarded individuals from a generation to another is 16, that is, 25% of the whole population. They are replaced by the offspring of the 16 better genotypes randomly coupled. Crossover is made at a single random position. If both parents are the same, the probability of mutation is 0.95 otherwise it is 0.25. When an individual mutates, each of its codons is randomly changed with a probability of 0.25.

The probability for an offspring to be enlarged with one of its parents is 0.05 and the probability to be reduced is 0.05 too.

Fitness function

As it can be deduced from the AP440 syntax, there are 83 valid different notes with 7 possible lengths. In this work, both notes and lengths are identified by means of natural numbers taken respectively from the intervals [0, 82] and [0, 6]. Among other possible parameters we intend to consider those extracted from three random variables: the pitch and the differences between consecutive notes and lengths that belong to the sets [-82, 82] and [-6, 6]. Melodies are quantified as vectors with some measures computed from these three variables.

APL2 implementation

GEMUSIC is the main function and executes the following steps:

[1] Create the AP440 grammar

[2] Generate the initial population of genotypes.

[3] Evaluate the fitness function on every new phenotype.

[4] Sort the population of genotypes by their merit.

[5] Replace the worse individuals by the offspring of the better.

[6] Go to step [3] until finding a good enough solution.

The APL2 functions MKGRMMR, GENER, TEST and SEX called by GEMUSIC are used to respectively solve the steps [1], [2], [3] and [5].

Listing 1 shows GEMUSIC's code.

```
[0]    Z←GEMUSIC G;I;P;V;⎕IO;Grammar_;ProductionRules_;Terminals_;MAXFIT
[1]     ⎕IO←1
[2]     Grammar_←MKGRMMR                    ⍝ AP440 grammar is created
[3]     ProductionRules_←GRM2RULE Grammar_  ⍝ and its components are extracted
[4]     Terminals_←GRM2TRM Grammar_
[5]     NotaANum_←MKNT2NUM
[7]     MAXFIT←1000                         ⍝ Fitness of the target melody
[8]     ⎕IO←G←0
[9]    P←GENER¨64⍴32                        ⍝ Initial population is generated
[10]    V←TEST¨P                            ⍝ Current population is evaluated
[11]   L:V←V[I←⍒V]                          ⍝ Current population is sorted
[12]    P←P[I]
[13]    'GENERATION'(G←G+1)'MAX ='(↑V)(TRANS↑P)  ⍝ A message is prompted
```

```
[14]    →(MAXFIT≤↑V)/F                         ⍝ If the better individual is in the
[15]                                           ⍝ ball the process finishes
[16]    P[48+⍳16]←SEX P[16?16]                 ⍝ Worse individuals are replaced
[17]    V[48+⍳16]←TEST¨P[48+⍳16]               ⍝ and their merit is updated
[18]    →L                                     ⍝ Next iteration
[19]   F:Z←↑P                                  ⍝ Best genotype is returned
```

Listing 1: Current function GEMUSIC's code

4. Further research

The goal of our future experiments will be to automatically compose melodies with the style and flavor of well-known composers. Thus, the work described in this paper will be extended in the following way:

- Select several well-known composers and a set of distinctive melodies from their compositions and express them with AP440 syntax.
- Identify a combination of the compositions' parameters that will distinguish and cluster the melodies of each composer.
- Automatically generate a melody with the style of a given composer by means of grammar evolution.
- Each composer's set of melodies will be represented by the vector of the arithmetic average of the coordinates of the set. The fitness function is based on the Euclidean distance from this average vector to each individual candidate.

Figure 3 graphically shows the goal of the process: several composers' populations are represented as points in a two-dimensional landscape. A distinctive point is emphasized in each population. A ball surrounds the target composer's population. The dotted arrow shows the expected path followed by the best melody from a random beginning place in the plane until one of its descendants reaches the ball.

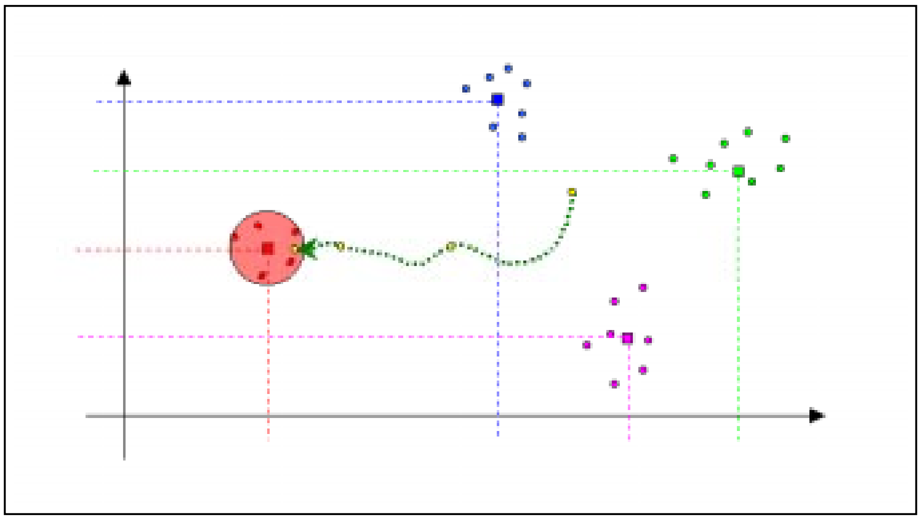

Figure 3: Grammar evoution and music layout

5. References

1. J. McCormack, 1996 *Grammar-based music composition.* In Complex International Vol 3
2. J. Biles, 1994 *GenJam: A Genetic Algorithm for Generating Jazz Solos.* In Proceedings of the 1994 International Computer Music Conference, ICMA, San Francisco, 1994
3. E. Bilotta, P. Pantano, V. Talarico, 2000 *Synthetic Harmonies: an approach to musical semiosis by means of cellular automata.* In Artificial Life VII: Proceedings of the Seventh International Conference, edited by M. Beday et al. MIT Press.

4. D. LIDOV, J. GABURA, 1973 *A melody writing algorithm using a formal language model*, in Computer Studies in the Humanities 4(3-4), pp. 138-148, 1973

5. P. LAINE, M. KUUSKANKARE, 1994 *Genetic Algorithms in Musical Style oriented Generation* in: Proceedings of the First IEEE Conference on Evolutionary Computation, Orlando, Florida, 1994.

6. D. HOROWITZ, 1994 *Generating Rhythms with Genetic Algorithms,* in: Proceedings of the ICMC 1994, International Computer Music Association, Århus, 1994

7. B. JACOB, 1994 *Composing with Genetic Algorithms* In Proceedings of the 1995 International Computer Music Conference, ICMA, San Francisco.

8. N. CHOMSKY, 1956 *Three models for the description of a language* IRE Transaction on Information Theory. 2:3 113-124.

9. J.E. HOPCROFT, R. MOTWANI, J. D. ULLMAN, 2001 *Introduction to Automata Theory, Languages, and Computation*. Pearson Education.

10. M. O'NEILL AND C. RYAN: *Grammatical Evolution*. IEEE Transaction on Evolutionary Computation. August 2001. Vol. 5, number 4 ITEVF5 (ISSN 1089-779X)

11. M. O'NEILL AND C. RYAN: *Evolving Multi-line Compilable C Programs*. In Proceedings of the Second European Workshop on Genetic 1999. Berlin, Germany: Springer-Verlag, 1999, vol. 1598, LNCS, pp. 83-92

12. M. O'NEILL AND C. RYAN: *Under the Hood of Grammatical Evolution*. In GECCO '99: Proceedings of the Genetic and Evolutionary Computation Conference 1999, W. Banzhaf et al. Eds. San Mateo, CA: Morgan Kaufmann, 1999, vol. 2, pp. 1143-1148

13. M. O'NEILL AND C. RYAN: *Genetic code degeneracy: implications for grammatical evolution and beyond*. In ECAL'99: Proceedings of the Fifth European Conference on Artificial Life, Lausanne, Switzerland, Sept. 1999, pp. 149-153

14. RYAN, J.J. COLLINS, AND M. O'NEILL*: Grammatical Evolution: Evolving Programs for an Arbitrary Language*. In EuroGP'98: Proceedings of the First European Workshop on Genetic Programming. Berlin, Germany: Springer-Verlag, 1998, vol.. 1391, LNCS, pp. 83-95

15. RYAN, O'NEILL M. *Grammatical Evolution: A Steady State Approach*. In Late Breaking Papers, Genetic Programming 1998, pages 180-185.

16. RYAN, M. O'NEILL: *Grammatical Evolution: A Steady State Approach*. In Genetic Programming 1998: Proceedings of the 3rd Annual Conference, J. R. Koza et al. Ed.s Cambridge, MMA, 1998, pp. 180-185

17. C. RYAN, M. O'NEILL, AND J. J. COLLINS: *Grammatical Evolution: Solving Trigonometric Identities*. In Mendel'98: Proceedings of the 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks, and Rough Sets. Brno, Czech Republic: Tech. Univ. Brno, 1998, pp. 111-119

18. IBM: *APL2 for the IBM PC: Reference summary Version 1.02*. IBM Scientific Centres Madrid, Spain and Winchester, England. SC33-0601-01. Nov. 8, 1991.

19. Z. MICHALEWICZ, D. FOGEL: *How to solve it: modern heuristics*. Springer-Verlag Berlin Heidelberg, 2000

20. M. MONCUR: *THE WWW VIRTUAL KEYBOARD,* http://www.xmission.com/~mgm/misc/keyboard.html