From: ISMB-95 Proceedings. Copyright © 1995, AAAI (www.aaai.org). All rights reserved.

Classifying Nucleic Acid Sub-Sequences as Introns or Exons Using Genetic Programming

Simon Handley
Computer Science Department
Stanford University
Stanford, CA 94305
(415) 723-4096 shandley@cs.stanford.edu

Abstract

An evolutionary computation technique, genetic programming, created programs that classify messenger RNA sequences into one of two classes: (1) the sequence is expressed as (part of) a protein (an *exon*), or (2) not expressed as protein (an *intron*).

1. Introduction

Cells convert DNA sequences into proteins in two stages: first, the DNA is *transcribed* into *messenger RNA* (mRNA). Second, the messenger RNA is *translated* into amino acid residues. In eukaryotes, the messenger RNA transcribed from the DNA does not all necessarily end up being expressed as protein. After an mRNA sequence is transcribed from a DNA sequence, and before it is translated into amino acid residues, contiguous subsequences of the mRNA sequence are *spliced* out. The subsequences that are removed are called *introns*; the intervening subsequences that get expressed as protein are called *exons*. Singer and Berg (Singer & Berg 1991) discuss eukaryotic translation in detail.

In this paper an evolutionary computation technique, genetic programming, is shown to produce programs that can distinguish between exons and introns.

2. Genetic Programming

Genetic programming (Koza 1992; Koza 1994b; Koza & Rice 1992) is an inductive learning technique that is particularly suited to problems in which some underlying regularity or structure must be discovered.

Genetic programming can be thought of as a black box whose output is a program that is (hopefully) fit according to the given fitness measure. See figures 1 and 2.

Genetic programming starts with a population of randomly generated computer programs, a fitness measure, and a termination criterion and uses artificial selection and sexual reproduction to produce increasingly fit populations of computer programs.

Each program is a composition of functions and terminals. For example, "`1`" is a program that consists of a single terminal; "`(+ 1 2)`" is another program, it is a composition of two terminals (`1` and `2`) and a function (+); "`(* 1 (+ 2 3))`" is another program.

The programs in each generation of a run of genetic programming are created either by copying them from the previous generation (with probability proportional to their fitness) or by crossing over two parental programs (also chosen with probability proportional to their fitness) and placing the two resulting child programs in the new generation.

The crossover operation works as follows. Let the two parental programs be

```
(+ (* 1 2) (* 3 4)), and
(+ (+ 5 (+ 6 7)) 8).
```

The crossover operator randomly chooses an internal point in each program. Two such choices are highlighted here:

```
(+ (* 1 2) (* 3 4)), and
```

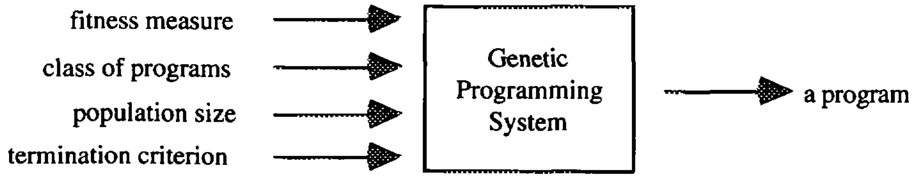

Figure 1. Genetic programming as a black box.

`(+ (+ 5 (+ 6 7)) 8).`

The crossover operator then swaps these two fragments, `(* 3 4)` and `(+ 6 7)`, to produce the child programs:

`(+ (* 1 2) (+ 6 7))`, and

`(+ (+ 5 (* 3 4)) 8).`

Genetic programming has been applied to the following sequence analysis problems: predicting whether or not a residue is in a helix (Handley 1993; Handley & Klingler 1993); recognizing the cores of helices (Handley 1994a); predicting the degree to which a residue in a protein is exposed to solvent (Handley 1994b); predicting whether or not a DNA sequence contains an *E. coli* promoter (Handley 1995b); predicting whether or not a nucleic acid sequence contains a splice site (Handley 1995a); predicting whether or not a sequence contains an omega loop (Koza 1994b); and predicting whether or not a sequence contains a transmembrane domain (Koza 1994a; Koza 1994b).

3. Applying Genetic Programming: Choosing Values for Parameters

The black box in figure 1 has four inputs: the fitness measure, the class of programs, the population size and the termination criterion.

3.1. The fitness measure

The programs being evolved are supposed to assign sequences into one of two classes: intron or exon. A reliable way of measuring the performance of two-state classifiers is the correlation coefficient, *C*. The choice of parents to participate in the genetic operations—crossing-over and fitness-proportionate reproduction—that create each generation (after generation 0) was based on this fitness measure.

3.2. The class of programs

There are statistical differences between introns and exons; for example different codons appear with different frequencies. Programs that differentiate between introns and exons must be able to compute non-trivial statistics based on sequences.

I chose to evolve programs in a modified version of the language FP, as described in John Backus's 1977 Turing Award Lecture (Backus 1987). The language I implemented differs in five main ways from that described in the above reference: (1) I added a new function ("<", described below), (2) I simplified the language to do computations on arrays and numbers, rather than on arrays, numbers and symbols, (3) Boolean values are represented by -1 (false) and +1 (true); (4) numbers as functions are interpreted as constants (as if they were written $\overline{42}$ in FP) not as element selectors (1st, 2nd, etc.); and (5) I changed the error-handling (also described below).

The intron/exon problem is different from other molecular biology problems in that genetic programming has been applied to in that the programs to be evolved must compute statistics based on sequences. Other problems, such as the helix-cores problem (Handley 1994a), were solved by evolving programs that looked at individual residues/base pairs and then an external wrapper was applied to each evolved program that ran the program on all residues/base pairs of a sequence. Such a summing-up wrapper is not useful in the

```
function genetic-programming( fitness-measure, class-of-programs, pop-size, termination-criterion )
  gen ← 0
  for i ← 0 ... pop-size - 1 do
    population[0][i] ← randomly-generate-a-program( class-of-programs )
    compute & store fitness-measure( population[0][i] )
  end
  while not termination-criterion( population[gen] ) do
    create population[gen+1] by crossing-over and copying probably-highly-fit programs from
            population[gen]
    for i ← 0 ... pop-size - 1 do compute & store fitness-measure( population[gen+1][i] )
    gen ← gen + 1
  end
  return fittest individual in population[gen]
end
```

Figure 2. Genetic programming pseudocode.

intron/exon problem because the evolved programs must compute more complex statistics, such as di-codon frequencies.

FP was chosen not because it is known to be amenable to evolutionary change (it's not known to be) but because it is a well-defined starting point for experimenting with evolvable array-based languages.

It isn't necessary to understand the details of all the functions below. What is important is that all the functions operate on and return numbers and lists of (lists of...) numbers.

There are two classes of functions: *ordinary functions*, that take as input a sequence or a number and return a sequence or a number, and *functional forms*, that take as input one or more functions and return a function.

Appendix 1 lists the ordinary functions. These functions indicate an error by returning their input unchanged. Angle brackets ("<" and ">") are used to delimit sequences; for example, `<1, 2>` is a two-element sequence.

Appendix 2 lists the functional forms. Parentheses ("(" and ")") are used to group functions together.

Here are some example programs. The notation $p{:}x \rightarrow y$ means that the program p when given the input x returns the value y.

- `2nd:<1, 2, 3>` → `2`. (Returns the 2nd element of the input sequence.)
- `4th:<1, 2, 3>` → `<1, 2, 3>`. (4th element doesn't exist so it returns its input.)
- `4th:42` → `42`. (Input isn't a sequence so it returns its input.)
- `(ins +):<1, 2, 3>` → `6`. (Returns `+:<1,+:<2,+:<3>>>` = `+:<1,+:<2,3>>` = `+:<1,5>` = `6`.)
- `(ata 1):<1, 2, 3>` → `<1, 1, 1>`. (Returns `<1:1, 1:2, 1:3>`, "1" being a function that ignores its input.)
- `([]2 id (ata 1)):<1, 2, 3>` → `<<1, 2, 3>, <1, 1, 1>>`. (Returns `<id:<1, 2, 3>, (ata 1):<1, 2, 3>>`.)
- `(o trans ([]2 id (ata 1))):<1, 2, 3>` → `<<1, 1>, <2, 1>, <3, 1>>`. (Returns `trans:<<1, 2, 3>, <1, 1, 1>>`.)
- `(o (ata +) (o trans ([]2 id (ata 1)))):<1, 2, 3>` → `<2, 3, 4>`. (Adds 1 to each element of the input sequence.)

Random programs in the above language can require an enormous amount of computer resources. For example, the program `(while 1 ([]2 id id))` requires an exponentially increasing amount of memory. The havoc caused by such programs was controlled in two ways: (1) limiting to 1500 the total number of while-loop iterations per fitness case; (2) limiting the number of list elements (*conses*) that can be in use (5000). Both ordinary functions and functional forms just return their input when they need a new cons but the limit has been reached.

3.3. The population size

The genetic algorithm is designed to search spaces that have many dimensions and that are moderately epistatic. The population is used to implicitly store multiple partial solutions to sub-spaces of the search space. For this reason, the size of the population is directly proportional to the difficulty (the degree of epistasis and the number of dimensions) of the problem that can be solved. Because programs written in the above language can take a long time to run, populations of 12,800 individuals were the largest that would fit on the available hardware.

3.4. The termination criterion

Normally genetic programming runs are stopped when some criterion is true of the most recent population, for example if the population contains a solution to the problem then the run would usually stop. Because the evaluation of individuals in this problem is so slow, I ran each evolution for a fixed time period.

4. The Data

I extracted all human sequences from Genbank 85 (Benson et al. 1994) that contained the phrase "complete cds". Then all introns and all exons were extracted whose starting and ending points were well-defined (i.e., whose location was "x..y", not "<x..y", "x..>y", etc.). I then extracted the largest possible multiple of 3 codons from each intron and exon, making sure that the reading frame was correct (exons don't necessarily start at the start of a codon). This resulted in 1055 introns and 1014 exons. To keep the computation tractable, introns and exons containing more than 500 bases were filtered out. Finally, 150 introns/exons were randomly chosen (without replacement) for the training set. A testing set (with which to check for overfitting) was not used because the lack of computer time precluded overfitting; similarly, no evaluation set (with which to independently evaluate the resultant programs) was used.

A fitness case in this problem is a complete intron or a complete exon. Each evolved program takes as input a sequence of numbers representing the entire sequence of bases that make up the intron/exon and returns a value indicating whether or not the program classifies that sequence as an intron or an exon. (The DNA sequences were presented to the programs as sequences of floating point numbers, where base N = 0.0, A = 1.0, C = 2.0, G = 3.0, T = 4.0, R = 5.0.) The result of the execution of an individual program was converted into a Boolean classification as follows: if the result was a number then it was converted to a Boolean as per the "and" function; otherwise the flattened result was summed and the sum was converted to a Boolean value.

5. The Results

One run was done. Computing the fitness of a single individual took ~20 sec on an SGI Challenge. Table 1 shows the fitnesses of the best three individuals. Appendix 3 shows the actual individuals.

6. Previous Work

Lapedes et al. (Lapedes et al. 1990) used 5th order perceptrons[1] to predict whether or not a sequence of DNA was expressed as a protein. The bases were grouped together in the input representation. Two nets were created: one for *H. sapiens* and one for *E. coli*. The training set for the *H. sapiens* net contained liver cDNAs (the positive examples) and the negative examples were chosen from introns. The training set for the *E. coli* net was ~50 sequences from GenBank; the testing set was the remainder of *E. coli*. The sizes of the training and testing sets weren't given. They achieved accuracies (expressed as the average of Q_I on the positive and negative examples) of 99.5% and 98.4% on *E. coli* and *H. sapiens*. It is difficult to compare the Q_I values obtained by Lapedes et al. with the results in this paper because (1) the individuals in table 1 and appendix 3 were selected for good performance as measured by C, not Q_I. Comparing Q_I values is not meaningful. (2) It is not possible to compute Q_3 or C from Q_I because the number of false positives is not known.

[1] An nth order perceptron has the inputs to each neuron grouped in all possible groups of n and weighted individually; this enables the net to learn nth-order correlations without a hidden layer.

Uberbacher and Mural (Roberts 1991; Uberbacher & Mural 1991; Xu et al. 1994) wrote a program, called GRAIL, that classified DNA bases as being in introns or in exons. GRAIL contains 7 hand-crafted feature-detectors that are combined by a neural net. This neural net's input is a 99-base window of DNA and it's output is the likelihood of the central base being in an exon. Their training set was 18 human genes and their testing set was 19 human genes. Of the 1113 residues in exons in the testing set, 1029 (92%) were correctly identified and 84 (8%) were incorrectly identified.

Uberbacher and Mural successfully created a program that classifies bases as exon/intron with very high accuracy: approximately 90% of exons are correctly predicted. GRAIL is not directly comparable to the work in this paper, however, because of the hand-crafted feature detectors. (GRAIL also differs in that it has to parse the DNA sequence, the programs described in this paper were given sequences that were either entirely an exon or entirely an intron.) The point of this paper is to try evolving both the feature detectors and the logic that combines them. This problem is different than that solved by GRAIL.

7. Conclusions

This paper has shown that an evolutionary computation technique, genetic programming, can create programs that differentiate between introns and exons. Due to the computational difficulties inherent in array-based languages it was not possible to evolve programs that were accurate on many training examples. The above experiments are promising, however, in that they indicate that with more computer time it should be possible to evolve a program that accurately differentiates between introns and exons.

8. Acknowledgments

Thanks to: John Koza for his comments; David Andre for his programming advice; and Monica Lam for the use of the machine on which the software was developed.

	generation	C
best individual	59	0.54
second-best	27	0.53
third-best	23	0.52

Table 1. Performance of the top three individuals. The "generation" column is the generation in which the individual first appeared.

9. References

Backus, J. 1987. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. In *ACM Turing Award Lectures: The First Twenty Years*, 63–130. New York, NY: ACM Press.

Benson, D.; Boguski, M.; Lipman, D. J.; and Ostell, J. GenBank. 1994. *Nucleic Acids Research* 22 (17): 3441–4.

Handley, S. G. 1993. Automated learning of a detector for α-helices in protein sequences via genetic programming. In Proceedings of the Fifth International Conference on Genetic Algorithms, 271–8. Urbana-Champaign, IL.: Morgan Kaufmann.

Handley, S. G. 1994a. Automated learning of a detector for the cores of α-helices in protein sequences via genetic programming. In First IEEE Conference on Evolutionary Computation, 474–9. Walt Disney World Dolphin Hotel, Orlando, FL.: IEEE.

Handley, S. G. 1994b. The prediction of the degree of exposure to solvent of amino acid residues via genetic programming. In Second International Conference on Intelligent Systems for Molecular Biology, 156–60. Stanford University, Stanford, CA.: AAAI Press.

Handley, S. G. 1995a. Predicting Whether Or Not a 60-Base DNA Sequence Contains a Centrally-Located Splice Site Using Genetic Programming. Forthcoming.

Handley, S. G. 1995b. Predicting Whether Or Not a Nucleic Acid Sequence is an *E. coli* Promoter Region Using Genetic Programming. In First International IEEE Symposium on Intelligence in Neural and Biological Systems, Herndon, VA.: IEEE Press.

Handley, S. G. and Klingler, T. 1993. Automated learning of a detector for α-helices in protein sequences via genetic programming. In *Artificial Life at Stanford 1993*, ed. Koza, J. 144–52. Stanford, CA: Stanford Bookstore.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Koza, J. R. 1994a. Evolution of a comuter program for classifying protein segments as transmembrane domains using genetic programming. In Second International Conference on Intelligent Systems for Molecular Biology, 244–52. Stanford University, Stanford, CA.: AAAI Press.

Koza, J. R. 1994b. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

Koza, J. R. and Rice, J. P. 1992. Genetic Programming: The Movie. The MIT Press. Cambridge, MA.

Lapedes, A.; Barnes, C.; Burks, C.; Farber, R.; and Sirotkin, K. 1990. Application of neural networks and other machine learning algorithms to DNA sequence analysis. In *Computers and DNA, SFI studies in the sciences of complexity*, ed. Bell, G. and Marr, T. 157–82. VII. Addison-Wesley.

Roberts, L. GRAIL seeks out genes buried in DNA sequence. 1991. *Science* 254 (5033): 805.

Singer, M. and Berg, P. 1991. *Genes & genomes: A changing perspective.* Mill Valley, CA: University Science Books.

Uberbacher, E. C. and Mural, R. J. Locating protein-coding regions in human DNA sequences by a multiple sensor-neural network approach. 1991. *Proceedings of the National Academy of Science (USA)* 88 :11261–5.

Xu, Y.; Einstein, J. R.; Mural, R. J.; Shah, M.; and Uberbacher, E. C. 1994. An improved system for exon recognition and gene modeling in human DNA sequences. In Second International Conference on Intelligent Systems for Molecular Biology, 376–84. Stanford University, Stanford, CA.: AAAI Press.

Appendix 1. The ordinary functions

Descriptive name	Short name & Definition
First[2]	$\mathrm{1st}{:}x \equiv \begin{cases} x_1 & \text{if } x = < x_1,\ldots,x_n >, n \geq 1 \\ x & \text{otherwise} \end{cases}$
Tail	$\mathrm{tl}{:}x \equiv \begin{cases} < x_2,\ldots,x_n > & \text{if } x = < x_1,\ldots,x_n >, n \geq 2 \\ x & \text{otherwise} \end{cases}$
Reverse tail	$\mathrm{tlr}{:}x \equiv \begin{cases} < x_1,\ldots,x_{n-1} > & \text{if } x = < x_1,\ldots,x_n >, n \geq 2 \\ x & \text{otherwise} \end{cases}$
Identity	$\mathrm{id}{:}x \equiv x$
Atom?	$\mathrm{atm}{:}x \equiv \begin{cases} -1 & \text{if } x = < x_1,\ldots,x_n > \\ +1 & \text{otherwise} \end{cases}$
Equality test	$\mathrm{eq}{:}x \equiv \begin{cases} \begin{cases} +1 & x_1 = x_2 \\ -1 & x_1 \neq x_2 \end{cases} & \text{if } x = < x_1,x_2 >,\ x_1,x_2 \text{ are numbers,} \\ x & \text{otherwise} \end{cases}$
Logical and[3]	$\mathrm{and}{:}x \equiv \begin{cases} \begin{cases} +1 & x_1 > 0, x_2 > 0 \\ -1 & \text{otherwise} \end{cases} & \text{if } x = < x_1,x_2 >,\ x_1,x_2 \text{ are numbers,} \\ x & \text{otherwise} \end{cases}$
Null sequence?	$\mathrm{null}{:}x \equiv \begin{cases} +1 & \text{if } x = < x_1,\ldots,x_n >, n = 0 \\ -1 & \text{otherwise} \end{cases}$
Reverse	$\mathrm{rev}{:}x \equiv \begin{cases} < x_n,\ldots,x_1 > & \text{if } x = < x_1,\ldots,x_n > \\ x & \text{otherwise} \end{cases}$
Distribute left	$\mathrm{distl}{:}x \equiv \begin{cases} << y,z_1 >,\ldots,< y,z_n >> & \text{if } x = < y, < z_1,\ldots,z_n >> \\ x & \text{otherwise} \end{cases}$
Distribute right	$\mathrm{distr}{:}x \equiv \begin{cases} << z_1,y >,\ldots,< z_n,y >> & \text{if } x = << z_1,\ldots,z_n >, y > \\ x & \text{otherwise} \end{cases}$
Length	$\mathrm{len}{:}x \equiv \begin{cases} n & \text{if } x = < x_1,\ldots,x_n > \\ x & \text{otherwise} \end{cases}$
Addition[4]	$+{:}x \equiv \begin{cases} x_1 + x_2 & \text{if } x = < x_1,x_2 >,\ x_1,x_2 \text{ are numbers} \\ x & \text{otherwise} \end{cases}$
Division	$\%{:}x \equiv \begin{cases} x_1 / x_2 & \text{if } x = < x_1,x_2 >,\ x_1,x_2 \text{ are numbers, } x_2 \neq 0 \\ x & \text{otherwise} \end{cases}$
Transpose	$\mathrm{trans}{:}x \equiv \begin{cases} << x_{11},\ldots,x_{n1} >,\ldots, \; < x_{1m},\ldots,x_{nm} >> & \text{if } x = << x_{11},\ldots,x_{1m} >,\ldots,< x_{n1},\ldots,x_{nm} >>^{5} \\ x & \text{otherwise} \end{cases}$
Append left	$\mathrm{apndl}{:}x \equiv \begin{cases} < y,z_1,\ldots,z_n > & \text{if } x = < y, < z_1,\ldots,z_n >> \\ x & \text{otherwise} \end{cases}$

[2]Second (2nd), third (3rd), ..., sixth (6th) are defined similarly. Reverse-first (r1st), ... reverse-sixth (r6th) are also similar: reverse-first returns the last element, reverse-second the second-to-last etc.
[3]The functions or, < and not are defined similarly.
[4]The functions * and - are defined similarly.
[5]Also, all of x's subsequences must be the same length.

Append right	$\text{apndr}{:}x \equiv \begin{cases} <z_1,\ldots,z_n,y> & \text{if } x = << z_1,\ldots,z_n >,y> \\ x & \text{otherwise} \end{cases}$
Rotate left	$\text{rotl}{:}x \equiv \begin{cases} <x_2,\ldots,x_n,x_1> & \text{if } x = < x_1,\ldots,x_n > \\ x & \text{otherwise} \end{cases}$
Rotate right	$\text{rotr}{:}x \equiv \begin{cases} <x_n,x_1,\ldots,x_{n-1}> & \text{if } x = < x_1,\ldots,x_n > \\ x & \text{otherwise} \end{cases}$
numbers $\in \Re$	numbers evaluate to themselves (i.e., they're constant functions)

Appendix 2. The functional forms

Descriptive name	Short name & Definition
Composition	$(\text{o f g}){:}x \equiv f{:}(g{:}x)$
Construction	$([]2\ \text{f g}){:}x \equiv < f{:}x, g{:}x >$
Conditional	$(\text{if p t f}){:}x \equiv \begin{cases} \begin{cases} t{:}x & p{:}x \geq 0 \\ f{:}x & p{:}x < 0 \end{cases} & p{:}x \text{ is a number} \\ x & \text{otherwise} \end{cases}$
Insert	$(\text{ins f}){:}x \equiv \begin{cases} f{:}< x_1, (\text{ins f}){:}< x_2,\ldots,x_n >> & \text{if } x = < x_1,\ldots,x_n >, n \geq 2 \\ x_1 & \text{if } x = < x_1 > \\ x & \text{otherwise} \end{cases}$
Apply to all	$(\text{ata f}){:}x \equiv \begin{cases} < f{:}x_1,\ldots,f{:}x_n > & \text{if } x = < x_1,\ldots,x_n > \\ x & \text{otherwise} \end{cases}$
While	$(\text{while p f}){:}x \equiv \begin{cases} \begin{cases} (\text{while p f}){:}(f{:}x) & p{:}x \geq 0 \\ x & p{:}x < 0 \end{cases} & p{:}x \text{ is a number} \\ x & \text{otherwise} \end{cases}$

Appendix 3. The best three individuals.

The best individual:

```
(o (if (if len (o (if ([]2 (if (if 2nd not r5th) not r5th) (if tl r6th -.9439))
(if (ata (if (if (if ([]2 (if (ins or) (if (ins or) (ata (if tl r6th -3.9439))
(while ([]2 (if ([]2 * *) (o (o or r4th) (if not 6th not)) ([]2 ([]2 len r2nd)
(ins 5th))) (ata (ins (ins +)))) (ata (o (ata tl) (ins distl))))) (if (o 1st
(distr)) (o distl trans) (ata (rotl)))) (if tl r6th -3.9439)) (if (ata tl) (if
and atm ([]2 distl trans)) (ata -)) (while (ins apndr) (o (ins (ata (ata len)))
([]2 < ([]2 rotr (while ([]2 (rotr) (if (if (while (ata rotl) -) (ins (o 5th
(while (if or and ([]2 (<) r1st)) trans))) ([]2 ([]2 (and) *) +)) (ata r4th) ([]2
(ata (o (ata tl) (ins (distl)))) ([]2 2nd (while (o distl ([]2 (if rotl id distr)
(ins r1st))) len))))) trans)))))) or ([]2 ([]2 and *) +)) (ata r4th) (ins
distl))) (if (if (if (while (ata rotl) (o or 5th)) (ins (o 5th (while tlr
trans))) ([]2 ([]2 and (ata trans)) +)) (ata r4th) ([]2 (ata (o (ata tl) (ins
distl))) ([]2 rotr tlr))) atm r5th) (ata -)) (while (ins apndr) (ins rotl)))
trans) rotr) (ins rotl) ([]2 (if rotl id distr) (ins r1st))) ([]2 (ata r5th) ([]2
(if rotr rotr or) (if (if (if ([]2 (if (if (if 2nd (if (ins (o ([]2 * ([]2 ([]2
and *) ([]2 (if ([]2 (if rotl id distr) (ins r1st)) % (if tl r6th -3.9439)) (ins
(ins rev))))) (while tlr trans))) (o (if ([]2 (if 2nd not r5th) (if tl (if (if
([]2 ([]2 (if 2nd ([]2 (if rotl id distr) (ins r1st)) (o (5th) (while tlr (if
(ata r5th) (if and (atm) r5th) (ata -))))) r6th) ([]2 (ins (if eq % (while (ata
(o distl ([]2 (if - r6th %) (ins r1st)))) (while rotr and)))) (while ([]2 ([]2
and *) rotr) (if ([]2 (o distl (%)) *) 9.5347 4th)))) * len) (atm) r5th)
-3.9439)) (if (ata r5th) (if and atm r5th) (ata -)) (while (ins apndr) len))
trans) rotr) atm) (ins not) (while null apndl)) (if (ins (or)) (ata (if tl r6th
-3.9439)) (while ([]2 (if ([]2 * *) (o (o or r4th) (if not 6th (not))) ([]2 ([]2
len r2nd) (ins 5th))) (ata (ins (ins +)))) (ata (o (ata tl) (ins distl))))) (if
(o 1st distr) (o distl trans) (ata rotl))) (if tl r6th -3.9439)) (if (ata tl) (if
and atm ([]2 distl trans)) ([]2 distl trans)) (while (ins apndr) (o (ata 1st)
([]2 < ([]2 rotr tlr))))) or ([]2 ([]2 and *) +)) ([]2 5th rotr) (ins distl)))))
```

The second-best individual:

```
(o (if (if len (o (if ([]2 (if (if 2nd not r5th) not r5th) (if tl r6th -3.9439))
([]2 < ([]2 rotr tlr)) (while (ins apndr) (ins rotl))) trans) rotr) (ins rotl)
([]2 (if rotl id distr) (ins (if 2nd ([]2 (if rotl id distr) (ins r1st)) (o 5th
(while tlr (if ([]2 (if 2nd ([]2 (if rotl id distr) (ins r1st)) (o 5th (if ([]2
(if 2nd (if (if or 4th rotl) (ins %) len) r5th) (if (ata tl) r6th (ata (if or *
([]2 (if (if 2nd not r5th) not r5th) (if tl r6th -3.9439)))))) (if (ata r5th) (if
and atm r5th) (ata -)) (while (ins tlr) len)))) (if tl r6th -3.9439 ))(if andatm
(r5th))(ata -)))))))) ([]2 (ata r5th) ([]2 (if rotr rotr or) (if (if (if ([]2 (if
(if (if 2nd (if (ins (o ([]2 (ata (o ([]2 (if rotl id trans) (ins r1st)) ([]2
r4th len))) ([]2 ([]2 and *) +)) (while tlr trans))) (o (if len (if distl (if and
atm r5th) (ata -)) (while (ins apndr) len)) trans) rotr) (atm)) (ins not) (while
null apndl)) (if (ins or) (ata (if (o (if (if (if (while r2nd 3rd) (while eq
distl) (o eq (apndr))) + rev) ([]2 - or) ([]2 apndl rotl)) (ata (o ([]2 (if rotl
id trans) (ins r1st)) ([]2 r4th len)))) r6th -3.9439)) (while ([]2 (if ([]2 * *)
(o (o or r4th) (if not 6th not)) ([]2 ([]2 len r2nd) (ins 5th))) (ata (ins (ins
+)))) (if eq % (while (ata (o distl ([]2 (if rotl id distr) (ins r1st))))
r5th)))) (if (o 1st distr) (o distl trans) (ata rotl))) (if tl r6th -3.9439)) (if
(ata tl) (if and ([]2 distl trans) ([]2 distl trans)) (ata -)) +) (while (ins
apndr) len) not) ([]2 5th rotr) (ins distl)))))
```

The third-best individual:

```
(o (if (if len (o (if ([]2 (if (if 2nd not r5th) not r5th) (if tl r6th -3.9439))
(if (ata (ata (o (ata tl) (ins distl)))) (if ([]2 ([]2 and *) +) (atm) r5th) (ata
-)) (while (ins (apndr)) (ins rotl))) trans) rotr) (ins rotl) (if eq r3rd distr))
([]2 (ata r5th) ([]2 (if rotr rotr or) (if (if (if ([]2 (if (if (if 2nd (if (ins
(o ([]2 * ([]2 (ins +) +)) (while tlr trans))) (o (if ([]2 (if and atm ([]2 distl
trans)) (if tl (if (o or 5th) atm r5th) -3.9439)) (if distl 2nd (ata -)) (while
(ins apndr) len)) trans) rotr) atm) (ins not) (while null apndl)) (if (ins or)
(ata (if (ata len) r6th -3.9439)) (while ([]2 (if ([]2 * *) (o (o or r4th) (if
not 6th (not))) ([]2 ([]2 len r2nd) (ins 5th))) (ata (ins (ins +)))) (ata (o (ata
tl) (ins distl))))) (if (o 1st distr) (ata r4th) (ata rotl))) (if tl r6th
-3.9439)) (if (ata tl) (if and atm ([]2 distl trans)) (ata -)) +) or ([]2 ([]2
and 5th) +)) ([]2 len ([]2 (if distl distr apndl) +)) (ins distl)))))
```