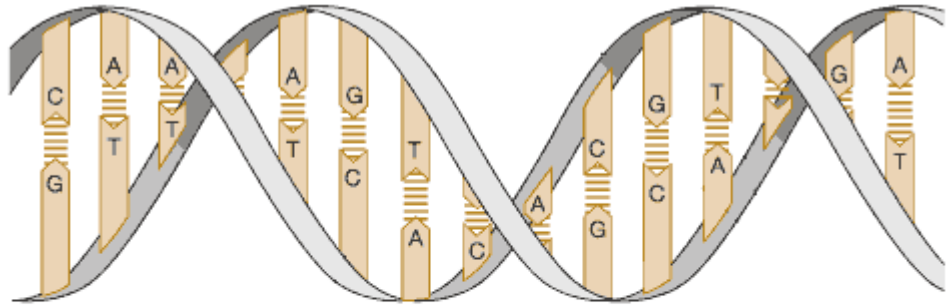
· UNIVERSIDAD CARLOS III DE MADRID ·
FINAL PROJECT
COMPUTER SCIENCE

USER MANUAL



NEPNPROGENGPO(

Student: Alberto Vegas Estrada
Supervisors: César Estébanez Tascón
Ricardo Aler Mur

Version: 1.0
Leganés, June 2008

INDEX:

PAGE:

1. INTRODUCTION

This manual will help you to solve every doubt that you have about how to use ProGen. If you still don't know what is ProGen or what is it useful for please dedicate a couple of minutes to read the point "frequently asked questions" at the end of this manual. As a matter of general definition we can say that ProGen is a tool that allows you, in a very easy and quick way, to define experiments of Genetic Programming, to execute them and to obtain results. ProGen (the name is a combination of the Spanish words Programación y Genética) has been designed with several clear goals: Ease, efficiency, scalability, portability, robustness and transparency.

ProGen is written in Java. It has been designed using every advantage offered by an object oriented programming language, including dynamic polymorphism. This allows ProGen to grow and adopt new code and new classes without any problem. Also, ProGen can work with any kind of data since it is strongly typed.

ProGen's design is thought to satisfy a wide set of users. Those who have not a big knowledge about Genetic Programming or programming in general can create their projects in an easy way. They can use the tool fearless because ProGen expects the minimum from them. Just a java file where the fitness function is written is enough. Once the file is compiled, users can execute infinity of different experiments and they will not even need to compile anymore.

Users with a bigger experience in GP (Genetic Programming) could need more complex functionality. ProGen offers it and it is highly flexible at the setting time. Almost any experiment desired can be designed in a few minutes with the intuitive experiment file.

Expert users could need very specific features, functions, operators that can adapt themselves perfectly to a particular problem. ProGen is ready to incorporate everything that it does not contain yet. In this user manual it is explained how to achieve that in very few steps.

We have cared very much about the design and the source code. We have tried to make it readable and understandable for users to feel comfortable exploring it and, if they consider it necessary, modify it fearless.

2. HOW TO USE THIS MANUAL

We don't want you to lose your time, and you don't need to read this manual as if it was a novel. In the index you can find a very accurate idea of what you are going to find across the document, and where each part is located. Anyway you can have a look to the following table and find out how many pages you can skip.

Excuse me, I'm just arrived… what is an individual?	Go to the definitions page. After that we advise you to read the frequently asked questions section and following you can read the quick start guide to start using ProGen.
I would like to launch an experiment. Nothing really difficult. The set of functions, operators and selectors provided with ProGen will be more than enough.	Go to the quick start guide. You will learn how to create and launch your experiment in a very few minutes.
I already know other Genetic Programming tools and what I want is to check if ProGen is as easy to use as they say.	Do you already know how to execute ProGen? If so, let's for example give ProGen a new function. Follow the instructions in section "inclusion of new functions" and check it yourself (see you here in 5 minutes). If you don't know how to execute ProGen you should start reading de quick start guide.
I have already made some projects in ProGen and now I have invented a new genetic operator. I want to analyze if this operator works better or worse than the conventional	You can read the section "inclusion of new operators" Once you have included it you can use it in all your projects as any other operator simply by selecting it in the experiment

operators.	properties file.
I am working on a complex investigation project and I need to take advantage of every Genetic Programming feature (multiple trees, ADFs, ARGs, etc)	In your case you can find interesting information in each part of this manual. You can use the index to go directly to the specific section you want to check in each moment.
Can I incorporate a new module to ProGen? I have a very good idea, but I don't want to implement a Genetic Programming engine from scratch. If I could use ProGen I would save a lot of time.	You can. ProGen is very modular and it is written completely using Java. It has been designed to grow and actually we are also working in new and very interesting modules. What you need to know is the interface of the classes to engage your module. You can find a detailed description of each class and each method in ProGen's documentation. If you need more help don't hesitate contacting us.

3. DEFINITIONS

With the objective of making readers understand better what is written in this manual, especially for those that are not very familiar with the concepts of Genetic Programming, we think it is useful collecting some definitions.

- ADF: (Automatic Defined Function). If the individual's main trees (see below) can be considered the representation of the main program, an ADF is the representation of a subroutine of the program. ADFs evolve parallel and independently from the rest of the trees. ADFs are invoked from main trees as any other function (see below).

- ARG: Argument. When in one tree we have a node that points to an ADF (a subroutine invocation), is possible that this node has more nodes hanging from it (its branches or children). These branches are called arguments (from ARG0 to the number of branches hanging from the node minus one). ADF trees can contain nodes that point to those branches to return the execution flow to the tree that invoked the ADF. The next figure illustrates more clearly the described structure.

Note: ProGen trees are executed in **preorder.**

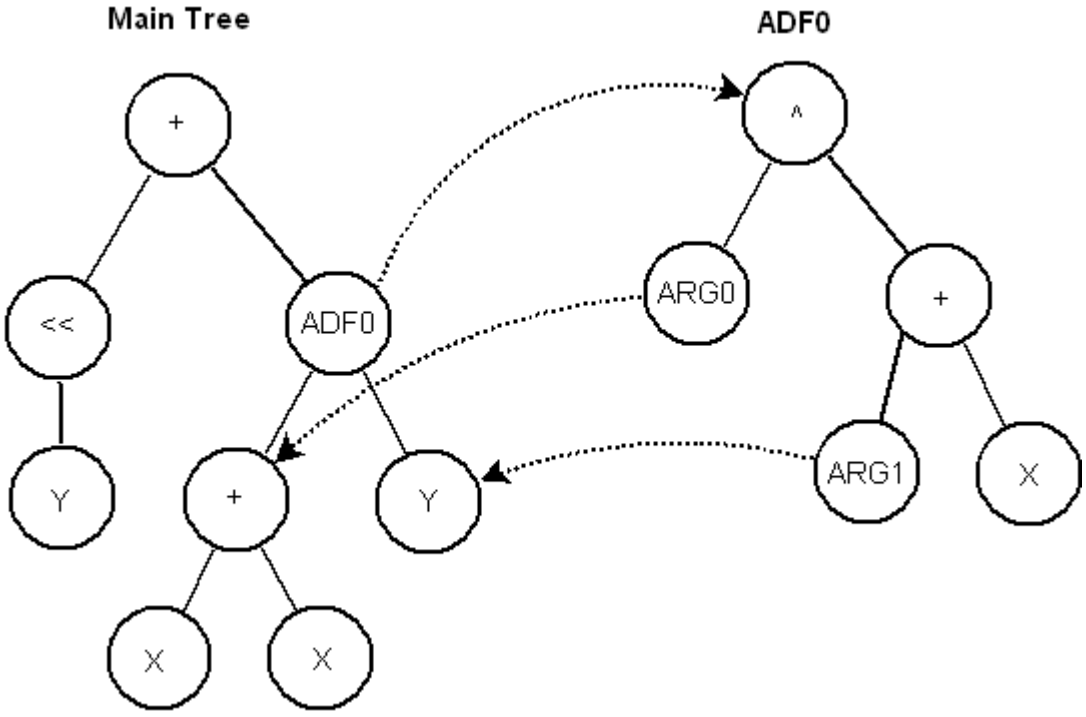


Figure 1: Execution flow with ADF's and ARG's

- Experiment: We call experiment to an execution of a project under a determined configuration. For example, the regression problem can be executed under infinite different configurations. Each time we execute the regression problem we are launching a different experiment (even when it's the same configuration).

- Fitness: Numeric value that measures how appropriate is the individual to solve certain problem that means, how good the individual is. ProGen, following a non written standard, minimizes fitness to measure individuals. This means that lower values are considered better than higher values. When an individual is evaluated using the fitness function, and the value returned is zero, this individual is considered perfect.

- Function: At the end, a program is a set of operations and data. Each different function can accomplish a different operation over the data or over the environment like for example add, subtract, multiply, jump, eat… whatever.

Each node in the tree contains a pointer to a function. Leaf nodes contain pointers to functions that do not need to operate over data, or data used by the nodes that they are hanging from. Some examples of functions included in ProGen are PlusFc or LwThanFc.

- Individual: The base of Genetic Programming is emulation of the biological life. In biology individuals are leaving beings. A ProGen individual is the equivalent. It is a data structure that codifies a program. More in detail, this structure is a set of trees divided in two groups (main trees and ADF's). In the simplest case an individual is a single tree. The evaluation of the tree returns a result that is a piece of data that can be any type. The evaluation process will look for an individual whose tree or trees codify a program able to solve a certain problem as for example the symbolic regression.

- Operator: An operator is an object able to turn one or more individuals into one or more different individuals. This new individuals belong to a new generation of individuals, that hopefully will be more able to find the solution for the given problem, or at least a better approach than the individuals belonging to the previous generation. Said with another words, application of operators over individuals usually returns individuals with better fitness. Examples of genetic operators included in ProGen are: PointMutation or Crossover.

- Selector: A selector is an object able to select individuals among a population, according to certain criteria (best fitness value, winners in a tournament, etc). Examples of selectors included in ProGen are: Roulette or Tournament.

- Terminal: With terminal we are defining the data, or the functions that don't receive arguments. Since ProGen can manage typed evaluation, a terminal can be an integer, a boolean, a map, a tree… anything. ARG nodes are also terminals as they have not branches.

4. QUICK START GUIDE

This will help you to start using ProGen. Here are briefly explained the minimum steps to create a project and to set up and run an experiment. We will create step by step a full example. Reading this chapter at least once is advisable for all users. Users interested in knowing in detail all the functionality and not only those modules or parts that are more basic should read the next chapter "Detailed user manual" where they will find deeper explanations and they will learn to take advantage of all the functionality of ProGen.

4.1. SYSTEM REQUIREMENTS

To use ProGen you will need:

- 64M of RAM
- 600Mhz processor
- 20M of empty space in your hard drive.
- Java 1.2 or higher

It is recommended to have:

- 256M or more RAM.
- 1.5Ghz or faster processor.
- 50M of empty space in your hard drive.
- Java 1.5

ProGen has been tested on UNIX / Linux, Mac OS X and Windows environments, but it will run on any platform where you have installed Java 1.5.

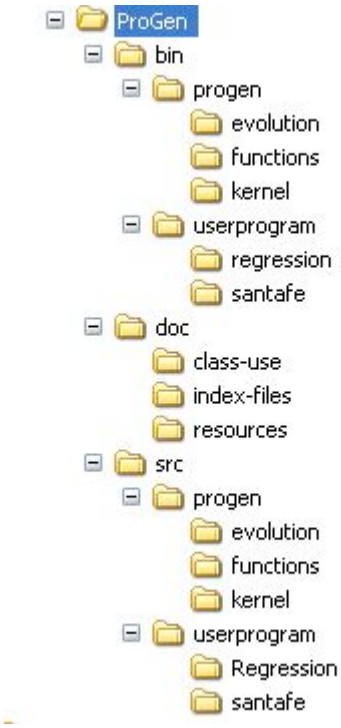
Genetic Programming problems tend to be of high computational complexity, and hence the results will be obtained more quickly the higher the speed of the computer. Additionally, ProGen can be configured to generate large amounts of data. The amount of disk space required depends entirely on the configuration of the experiment (number of individuals, generations, nodes,

etc.). Logically the minimum required disk space is therefore strongly linked to the experiment we want to run.

ProGen is written using Java 1.5 but respecting backward compatibility to 1.2.

4.2. INSTALLATION OF PROGEN

To install ProGen simply unzip the file with the source code in the desired directory. The directory hierarchy you get will look like:



At the root directory (ProGen) you will find the following:

- The **master_file.cfg** file (main properties file). ProGen runs the experiment referenced by this file.

- The **src** directory (where the .java files are located as in the hierarchy shown in the image)

- The bin directory (where the files .class files are located following the same hierarchy as for the .java files)

- The doc directory (where is the javadoc documentation of the project).

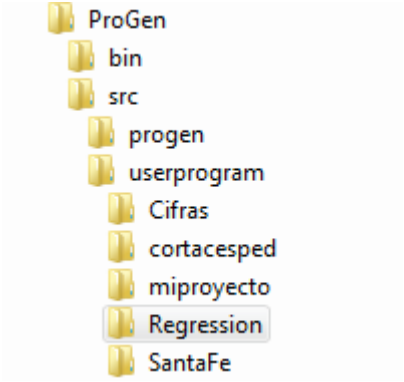
4.3. CREATION OF A NEW PROJECT

The different projects are stored in subdirectories in the directory ProGen/src/userprogram. Let's create a new project that we will call "Regression". In this project we want ProGen to find a program capable of solving the symbolic regression problem, or to put it another way, to approximate an equation like this: Y = X ^ 3 + X ^ 2 + X.

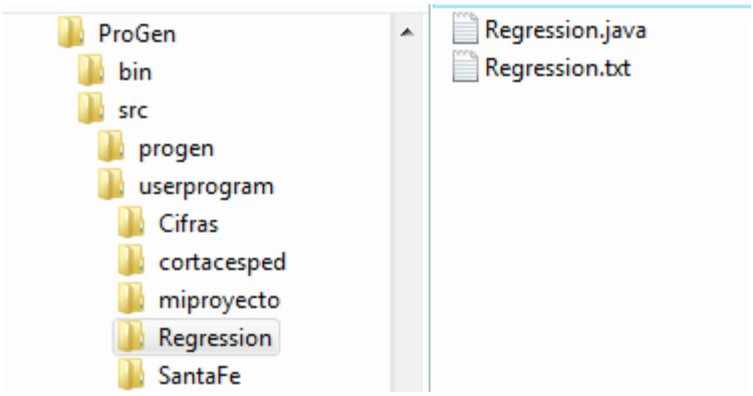
Let's see what are the steps to follow:

1. Create a new directory in the directory userprogram with the name you want to give to the project. In our project we call Regression to this directory.

The new directory hierarchy is as follows:



2. Now we need to have in the newly created directory at least two files. One will contain our program code (at least the fitness function). We will call this file Regression.java. The other file will contain the experiment configuration. We will call this one Regression.txt. The easiest way to start is to copy these files from another project and rename them. Then edit them to suit our particular problem. Once this is done the situation at the moment is that shown in the following figure:



3. Now we need to edit the files Regression.java (we call it the project file) and Regression.txt (we call it the experiment properties file). Let's see how:

4.3.1. THE PROJECT FILE

The project file is the file where is the java code of our project. It must necessarily contain the implementation of the method fitness and then, it can grow to be as complex as the user needs. It may include other classes, other files, databases and an unlimited etc.

In our case we are implementing a simple problem and we just need the fitness function. We will do the following:

4. We change references in the file that we are using as a template for our own project (change the name of the class by " class Regression" and in the package declaration in the first line we put the name of our package. In this case we have to type "package userprogram.Regression").

5. We delete the code of the fitness method and write our own fitness function. Remember that the best fitness is 0.0. The methods initialize and uninitialize are not mandatory. What they contain will be executed only once before and after the process of evolution. It can be used to initialize data, print a welcome message as in our case or you can simply delete them. See

how the code of the fitness function of the class Regression.java would be (you can find the file among the examples of ProGen).

```
public double fitness(Individual ind){
      //System.out.println("evaluando individuo");
      double fitness = 1000;
      double error = 0.0;
      double y = 0.0;
      double result = 0.0;

      for (double i = 0; i < 1000; i++){
            setVariable("X", i);
            y= Math.pow(i,4)+ Math.pow(i,3) + Math.pow(i,2) + i;
            result = ((Double)ind.evaluate(this)).doubleValue();
            error += Math.pow( y-result, 2 );
      }
      fitness = Math.sqrt(error / 1000);

      return fitness;
      }
```

We just take 1000 test cases. We assign values to X from 0 to 999 and calculate the corresponding value of Y. We evaluate the individual and compare the result with the previously calculated. The error is accumulated. The fitness of the individual is the mean quadratic error obtained (the square root of the total accumulated error divided by the number of test cases).

Important is to underline how simple we asing value to variables. With the sentence setVariable ("X", i); we assigned to the variable X (which corresponds to a terminal as defined in our experiment configuration file) the value of i. This method is inherited from class UserProgram.java and can be invoked directly from anywhere in your program. The method works with objects, so that the same method can be used to for example load a map or restore a "snapshot" of any object.

If we want ProGen to run some code before or after the execution of the program (some message, some initialization, etc.) use the methods initialize and uninitialize respectively. If the problem implemented was SantaFe, in which an ant explores a map, the initialize method is perfect to create the map with the values that the user wants. If what you want is that the map is initialized before the evaluation of each individual, and not only once prior to the

program execution and the evolution process, the best thing would be to use the method setVariable mentioned above, within the fitness function.

For the regression example is not necessary to initialize or uninitialize anything so that our program is already created, and we are ready to configure and run experiments. Next we will configure an experiment.

4.3.2. THE EXPERIMENT PROPERTIES FILE

6. Open the file that you renamed as Regression.txt and set up the experiment to meet your preferences. We believe it is sufficiently intuitive, so in this Quick Start guide we just mention the most important properties for our example. If you have any doubt, go to the point "5.2 Data input" of the documentation of ProGen (in this user manual). There you will find a detailed explanation of each property.

In any problem about Genetic Programming, you must ask yourself the following questions. One advantage that ProGen gives to you is that specifying the responses is really easy. So, coming back to our particular example about symbolic regression.

a. Do I need to define ADFs?

The ADFs allow us to evolve independently functions that can be complex or can be handled easier and better separately. For more information about the ADFs read Section 5.3: Using ADF's. To find a program able to give an approach to the equation Y = X ^ 3 + X ^ 2 + X, in a first instance it seems like we don't need to use them.

b. What functions and terminals do I need?

At a first glance one might say that it will be necessary to use the addition and the power functions. If we do not have the power function we can implement it in a moment (see section 5.4: Inclusion of new functions), or we can use the multiplication and leave it to ProGen to do the job. Regarding

terminals we need one to be the variable X. Taking a look at the library included in ProGen, we see that we can use the following:

DoublePlusFc: Sum of numbers of double type.

DoubleMultFc: Multiplication of numbers of double type.

D1: Variable double.

c. Do I need several main trees?

ProGen offers the possibility to use several main trees, however the answer to this question is usually no.

d. What type of data should the trees return?

In this case the individual will only have one tree. We want ProGen to find a program that is able to calculate the result of a particular equation. So trees must return a number, in this case of type double.

e. Which genetic operators do I need? What selectors are going to make ProGen converge towards the solution faster? Which probabilities should I set? What size for the population? Which initialization method is better?...

This is one of those parts of Genetic Programming you must experiment with. The answer to these questions is different for each problem, which is why we offer a tool really fast and flexible to configure. ProGen makes easier the configuration process, allowing you to change, remove or include new selectors, operators, parameters, etc. in the experiment and run it again without even re-compilation. Modifying your experiment is matter of very few seconds. In addition, thanks to the experimenter you can also define in a few seconds batteries of experiments so that it is ProGen who alters the values of the properties and launches an experiment after another. For more information go to point 5.8: The experimenter.

7. Well, you already have a first idea of what you need to solve the symbolic regression. Let's describe your wishes to ProGen by writing them in the properties file.

We had decided that we will use the following functions and terminals: DoublePlusFc, DoubleMultFc and D1. Since we are not going to use ADFs and we want to use only a main tree per individual (which returns a double value), we only need a function set. In the properties file, these especifications are corresponds to the following:

```
prp_num_function_sets: 1
prp_function_set_0: DoublePlusFc, DoubleMultFc, D1
prp_return_type_fs_0: double

prp_number_of_trees: 1
prp_tree0_function_set_number: 0
```

We are saying to ProGen that we use a single main tree, the tree uses the function set 0 that contains the list of functions and terminals desired and finally we are indicating that the trees created with this function set return the type double.

8. We now define the evolution parameters. As already commented, these are totally dependent of the problem to be solved and generally they must be adjusted using the experience, intuition and test and error. To start we ca say that we want to use the crossover with a 40% of probability, the mutation with growth (GrowMutation) with a 60% of probability, and also that we want the 16.34% of the population move to the next generation by elitism. We want Crossover to use RandomSelector to select individuals and GrowMutation to use tournament of size 4. The translation of thise in the properties file looks like the following:

```
prp_elitism: 16.34%

prp_operator_number: 2
prp_op1_name: Crossover(internal=0.9)
prp_op1_probability: 0.4
prp_op1_selection: RandomSelector()

prp_op2_name: GrowMutation(internal=1, levels=3, mode=grow)
prp_op2_probability: 0.6
prp_op2_selection: Tournament(size=4)
```

What if we would like Crossover to use as its selector RandomSelector 30% of the times and roulette (Roulette) the remaining 70%?

We simply consider them different operators and then we adjust the probabilities of occurrence:

```
prp_elitism: 16.34%

prp_operator_number: 3
prp_op1_name: Crossover(internal=0.9)
prp_op1_probability: 0.12
prp_op1_selection: RandomSelector()

prp_op2_name: Crossover(internal=0.9)
prp_op2_probability: 0.28
prp_op2_selection: Roulette()

prp_op3_name: GrowMutation(internal=1, levels=3, mode=grow)
prp_op3_probability: 0.6
prp_op3_selection: Tournament(size=4)
```

Note: While operators are here numbered starting from 1, it is expected that in future versions their numeration will start from 0.

Note: The values passed as parameters to the selectors and operators (internal, size, etc.) are parameters with which the user can specify the behavior of these operators and selectors. For more information about the parameters accepted by each operator or selector read section 6: Library available. If you want to know how to include new parameters for redefining the behavior of operators or selectors, or improve them with new possibilities that you like read paragraph 5.7: Pass-parameters: easy and unlimited.

9. What else remains undone? Now we simply have to adjust the value of other properties as for example the number of generations, the population size, the maximum depth of individuals ... We believe it is an intuitive task, however all the properties of this file are explained in section 5.2: Data input.

10. Now we are ready to run our problem and find a program that can calculate regression. If the result is not satisfactory at the first attempt, just look at the output of ProGen, try to guess how you can you improve the evolution (more generations perhaps? Another configuration for the

operators?). Edit the file Regression.txt to set new values, save the file and launch the new experiment. You do not need to compile anything.

4.3.3. THE MAIN PROPERTIES FILE

11. Go to the ProGen´s root directory. Here is the main properties file (by default it is called master_file.cfg). At the moment just make sure that the option "prp_experiment_file" is referencing your experiment properties file (src/userprogram/Regression/Regression.txt). To have more information about the main properties file go to point "5.2 The Data input" inside this user manual). Your experiment is now ready to be launched.

4.4. EXECUTION OF AN EXPERIMENT

To run an experiment simply execute the following command:

java ProGen <file>

where <file> is the name of the main properties file (if you have not created your own, this file will be named master_file.cfg). Remember that this file must reference the experiment file. Since the name of this file is passed as a parameter, you can have as many main properties files as you wish in the main directory.

Note: Neither the filename or its extension are important as far as it contains the property "prp_experiment_file." You can launch ProGen executing for example "java ProGen my_file.pro" or "java ProGen properties.txt", and so on. As long as the file referenced by the parameter exists on the proper path and contains the property "prp_experiment_file" pointing to the experiment properties file, the program will be launched successfully.

Note: To compile simply execute the command javadoc followed by the name of the file or files you want to compile.

5. DETAILED USER MANUAL

In this section we are going to explain in more detail the characteristics of ProGen and how can we get the maximum advantage of it.

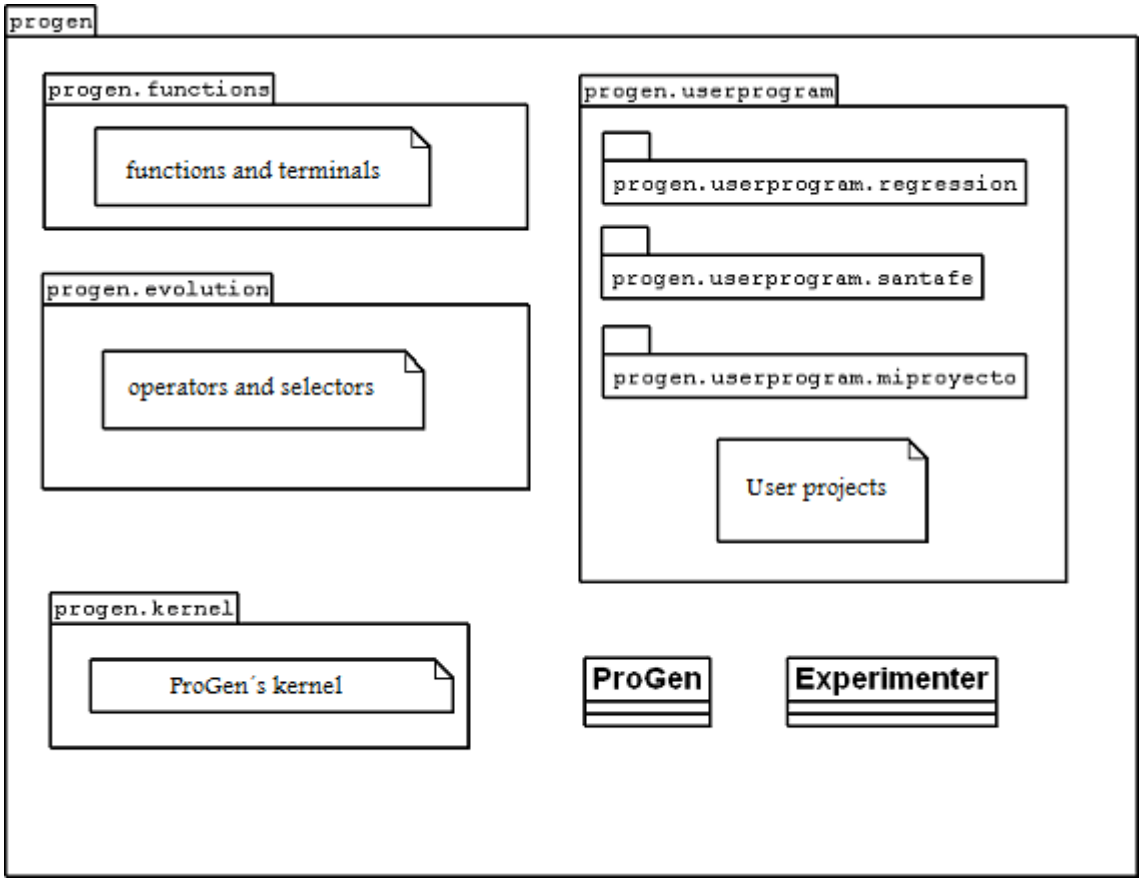
5.1. DESCRIPTION OF CLASSES

Doubtless is that the best way to get to know ProGen in dept is by exploring its classes and the interface that each of them offers. Knowing the source code in detail we will be able to modify it as we wish. However, ProGen is a program that contains several thousands of lines of code and understanding every method could be a heavy task. This does not mean that you can not use every feature or even modify ProGen. ProGen was created to be easily modified. From point 5.2 you will find information about how to do that. Even the operators, selectors and functions that we offer can be modified quick and easily and deep knowledge of ProGen classes is not required. Just follow the instructions that you will find from section 5.3 (Inclusion of new functions). The source code is divided in packages in such a way that we can guess by intuition what we are going to find inside each of them.

Nevertheless, for all users, expert or not, who are interested on exploring the code and knowing how does ProGen's kernel work, how is it implemented, what exactly happens when a particular method is invoked, etc. we offer inside ProGen´s documentation a detailed description of every package, every class, every method and every attribute.

Note: The detailed description of each component can be already found in this document (out of the user manual) in the section under the title "**Detailed Design**". To avoid unnecessary repetition of the information, from this manual (considered the fact that it is released together with the rest of the documentation) we will just point to the corresponding sections.

5.1.1. PROGEN'S PACKAGES



5.1.1.1. Package progen

This is the package that encloses the main class. It includes the following classes: ProGen.java and Experimenter.java. The detailed information about this package can be found in the section "Detailed design" (inside this document, out of this manual).

5.1.1.2. Package progen.functions

Package that stores the library of functions and terminals provided with ProGen. It has been specially designed to be improved with new functions and terminals in a fast and easy way. It also includes the class ADF, considered a

special kind of function and the class ARG, considered a special kind of terminal. The detailed information about this package can be found in the section “Detailed design” (inside this document, out of this manual).

5.1.1.3. Package progen.evolution

This is the package that stores all those classes that have a direct implication in the evolution process. It includes evolutive operators and selectors. The detailed information about this package can be found in the section “Detailed design” (inside this document, out of this manual).

5.1.1.4. Package progen.kernel

This package builds the kernel of ProGen. Basically includes those classes that are necessary to create, evaluate and control the population. The detailed information about this package can be found in the section “Detailed design” (inside this document, out of this manual).

5.1.1.5. Package progen.userprogram

In this package are included all the user programs. A user program is a problem that we want ProGen to solve, as for example the symbolic regression, SantaFe trail or any other problem that we intend to solve using Genetic Programming. The detailed information about this package can be found in the section “Detailed design” (inside this document, out of this manual).

5.2. DATA INPUT

The data imput handled by ProGen v1.0 is read from text files, called properties files. The first of them, passed as an argument in the ProGen execution is called the main properties file. In this file there is a reference to the second file, or experiment file. Following we explain its content and how must both files be configurated.

The main properties file:

ProGen recibe como argumento el fichero principal de propiedades, que tiene el siguiente aspecto:

```
#Master file configuration:

prp_experiment_file: Regression\Regression.txt

#EXPERIMENTER
prp_experimenter: off
prp_population_size: 2,10:2
prp_initialization_mode: grow, full
#prp_max_nodes: 100,300:50
#prp_max_depth: 6,10:1
prp_generations: 10,20:5
```

Mainly, this file contains a reference to the experiment properties file and properties used by the experimenter (For details about the experiment please read the section 5.8 of this document). Although this data may be included inside the experiment properties file in order to have a single input file, we expect in the future that together with the experimenter properties, this file will contain properties to manage other functionality as parallelism or multipopulations. Because of this, we decided to use both of the files, one with the properties for a specific experiment and the other, the main one, for higher level options.

The experimenter will be explained later, so about this file we will only say for the moment that the property “prp_experiment_file” must have a value, and this value must contain the path to the experiment properties file.

Note: In UNIX/Linux or Mac OS X environments, the character delimiting directories in the paths is ‘/’ instead of ‘\’, so the path to the file in the example above would look like:

```
prp_experiment_file: Regression/Regression.txt
```

The experiment properties file:

The experiment properties file is the place where we will set the whole configuration for a specific experiment. It is highly flexible and potent. Any error in the settings in this file will be detected by ProGen that will return an explanation of the particular error. Further in this document you can find a detailed description of every possible error and how to correct it. An experiment properties file has the following look.

```
#*********************************************
#        Experiment properties file           *
#*********************************************


#----PROJECT----

#The project must be written in a java file with the same name that
you specify in this property.
#it will be also used to name output files.
prp_project_name: Regression

#Available: English, Español
prp_language: español

#Output: detailed or summarized
output_mode: detailed


#----POPULATION----

prp_load_population: population.xml
prp_population_size: 100
#Full, grow, half and half
prp_initialization_mode: half and half
#Maximun number of attempts to generate a valid tree (both for the
initial population and during the evolution)
prp_max_attempts: 50



#----INDIVIDUALS----

#valid trees during the evolution
prp_max_nodes: 50
prp_max_depth: 10

#Valid depth range for the initial population trees
prp_depth_interval: 2,4


#----EVOLUTION----

prp_generations: 10
#Evolution will stop when this value (or any other inside the allowed
error interval) is reached
prp_stop_fitness: 0.5
#Upper and lower bounds from the stop_fitness value. Reaching a value
in this interval will also stop the evolution
```

```
prp_error_interval: 0.0, 100.0

#exact number or percentage value
prp_elitism: 10%

prp_operator_number: 4
prp_op1_name: Crossover(internal=0.5)
prp_op1_probability: 0.2
prp_op1_selection: RandomSelector(amongTheBest=100%)

prp_op2_name: GrowMutation(internal=1, levels=3, mode=grow)
prp_op2_probability: 0.2
prp_op2_selection: Tournament(size=1)

prp_op3_name: PointMutation(internal=1)
prp_op3_probability: 0.5
prp_op3_selection: Roulette()

prp_op4_name: Reproduction(internal=1.0)
prp_op4_probability: 0.1
prp_op4_selection: RandomSelector(amongTheBest=100%)


#----FUNCTIONS and ADF´S----

#if you want to use ADF´S name them like ADF0, ADF1 and so. To refer
its branches use ARG0, ARG1...

prp_num_function_sets: 3
prp_function_set_1: DoublePlusFc, DoubleMinusFc, D1, ADF0
prp_return_type_fs_1: double

prp_function_set_2: DoublePlusFc, DoubleMultFc, D1
prp_return_type_fs_2: double

prp_function_set_3: DoublePlusFc, DoubleMinusFc, D1, ADF0
prp_return_type_fs_3: double


prp_number_of_trees: 1
prp_tree0_function_set_number: 1
prp_tree1_function_set_number: 3

prp_adf_number: 1
prp_ADF0_function_set_number: 2
prp_ADF0_interface: double
```

These properties define the configuration with which an experiment is going to be run. Following you can find an explanation for each of them.

A. Block Project

- **prp_project_name**: Project name. Important: This name must be the same than the name of the java file that contains the project´s fitness

function. If in this property you write the value “MyProgram”, ProGen in order to evaluate the population will look for the fitness function in MyProgram.class. If it doesn’t find the file, the execution will be aborted returning the corresponding error.

- **prp_language:** This property allows you to decide in which language you want ProGen to inform you about the possible errors. In this first version of ProGen the languages available are Spanish and English. If you leave this property blank, or if the property is not found in the properties file, or if you write something different from one of the available languages, by default the error output will be in English.

- **prp_output_mode:** Especifies the level of detail that ProGen will use to log what has happened during the evolution process. Possible values for this variable are:

Mode	Information provided in the output
Normal	- Grammars used, initial population, best individual, fitness and generation in which it was found.
Detailed	- Logs everytime that an operator or selector is aplied, the individuals involved, the population in every generation, the best individual for each generation and its fitness, and stadistical data about the experiment.

B. Block Population

In this block you can specify the data related with the population.

- **prp_population_size:** Wished size for the population.

- **prp_initialization_mode:** Initialitation mode for the initial population. Possible values for this property are:

Mode	Information provided in the output
Grow	Trees grow randomly.
Full	Trees grow full. For each tree to be generated it´s chosen a depth value randomly between the minimum and maximum specified depth. Then the tree is generated full. (All the terminal nodes are in the last level)
Half and half	Aproximately half of the trees are generated using the grow method, and half using the full method.

- **Prp_max_attempts:** Maximun number of times that ProGen will retry generating a tree. Every time that it is generated a tree that does not meet the conditions specified in the properties file, ProGen will retry as long as the number of retries does not get higher than the value of this property.

C. Block Individuals

In this block we specify how do we want the individuals for the experiment to be.

- **prp_max_nodes:** Maximun number of nodes that the trees of an individual can have. A tree whose number of nodes is higher than this value will be rejected. This value is taken into account when generating the initial population and also during the evolution.

- **prp_max_depth:** This property defines the maximun depth that the individuals can reach. Unlike the last property, this one is taken into

account only during the evolution process. To control the máximun depth of the trees when generating the initial population we use the following property.

- **prp_depth_interval:** It indicates the minimum and maximum depth which may have the individual trees from the initial population. The value of this property must be two numbers separated by a comma. Example: "5, 8." Where the first number indicates the minimum depth (5) and the second number the maximum depth (8).

D. Block Evolution:

- **prp_generations:** Maximum number of generations that we want the evolution to last.

- **prp_stop_fitness:** Represents a condition for stopping. If the fitness function defined by the user returns this value, the evolution will stop.

- **prp_error_interval:** Stop condition. The user can define an error range for the stop_fitness (previous property). For example: If we define prp_stop_fitness to have the value 17 and prp_error_interval to have the value: 4, 8 it means that the evolution will be halted if the user´s fitness function returns 17, but also if it returns any value between 17-4 and 17 +8 . When the evolution is stopped for having been found a fitness stop or fitness within the range of error, it is reported by ProGen using the standard output.

- **prp_elitism:** Number of individuals who will pass intact to the next generation because they are the best. This value can be expressed in absolute terms or as a percentage of the population size.

  Examples:

  prp_elitism: 30 # The 30 best individuals will pass to the next generation.

prp_elitism: 15% # Starting with the best individual, the 15% of the population will pass to the next generation.

- **prp_operator_number:** Number of genetic operators that the user wants to use in the experiment. For each of them it should appear in the properties file the properties prp_opN_name, prp_opN_probability and prp_opN_selection, where N ranges from 1 to the specified value in this property. In other words, if in this property we declare that we will use 4 operators, the three properties cited must appear with N = 1, N = 2, N = 3 and N = 4.

  Note: We only use this numeration method (from 1 to N) for the genetic operators for implementation reasons. In the rest of parameters with numerations with have stuck to the convention widely accepted by programmers of counting from 0. In future releases of ProGen also genetic operators will be counted from 0.

  Prp_op1_name: Name of the operator. This name coincides with the Java file name where it is implemented (without extension). The parameters are passed in brackets as in the following example:

  Example: prp_op1_name: Crossover (internal = 0.6)

- **prp_op1_probability:** probability of this operator to be used to generate new individuals. This probability is taken into account, whenever ProGen requires an operator to evolve an individual. The value logically must be between 0 and 1 and if we use more than one genetic operator it must also guarantee that the sum of the probabilities of all of them is equal to 1.

  prp_op1_selection: Name of the selector used by this operator. The name of the selector must match the Java filename where it is

implemented (no extension). Any operator can use any selector. The parameters are passed in brackets as in the following example:

Example: prp_op1_selection: Tournament (size = 4, amongTheBest = 40%)

Pass-Parameter: Operators and selectors can receive parameters. In some cases these parameters are mandatory (see section dedicated to the explanation of operators and selectors to know exactly what parameters can or must receive each operator / selector). To pass several parameters to a selector or to an operator we simply separate them by commas as in the example above.

Note: You can pass any information to an operator or selector using pass-parameter. This system is especially useful to improve, modify or customize operators and selectors offered by ProGen and it is also very useful to create new operators and selectors easily and without limitations. For a deeper explanation of this method please read the section 5.7: Pass-Parameter: Easy and unlimited in the page 42.

E. Block Funciones y ADF´s:

In this block we specify the functions that we use in the trees and in the ADF's of ProGen individuals, as well as how many main trees to have in each individual, how many ADF's, return types and so on.

- **prp_num_function_sets:** Number of function sets that we are going to use. A function set is a set of functions and terminals. Allowing the existence of several sets we allow the possibility of having different trees evolving with different functions, which is particularly useful when we want to evolve with ADFs. If this property has the value N then in the same properties file they must also appear the properties "prp_function_set_i" and "prp_return_type_fs_i" where i is a number

ranging **from 0** to N-1. In other words, if we declare the use of 3 function sets, the two properties cited must appear for i = 0, i = 1 and i = 2.

prp_function_set_1: Here user should write a list of functions and terminals separated by commas. The trees that we will indicate later to be created using this function set, will have in their nodes functions and terminals from this list. The list of functions and terminals offered by ProGen can be found in this document (see index), but it has been designed to grow in a really simple way. In this document you will find detailed information on how to include your own functions to the list. The names of the functions that should appear on this property are the names of the Java files that implement them (without extension).

If ADFs are declared (see how to do this in the following properties) they can be included on the list as any other function assigning them numbers **from 0**. In this case it is not necessary the existence of a Java file for every ADF we want to use, since the file ADF.java is generic and ProGen is responsible for creating the objects needed.

Example: prp_function_set_1: DoublePlusFc, DoubleMinusFc, D1, D2, ADF0

- **prp_return_type_fs_1:** Return type of the trees that will be generated using this function set. ProGen is a tool that allows typed Genetic Programming. All functions and terminals have both, return type and types for the arguments they receive (type of possible child nodes). All trees are created using grammars so that the trees are always semantically correct. To evaluate a tree it is necessary to know what type is the final value that this tree returns. Such information is provided to ProGen through this property.

- **prp_number_of_trees:** ProGen can handle several main trees. This property specify how many main trees we want to use. Typically this value is always one but we wanted to allow the existence of n especially for researchers. For every main tree we want to use we must include a property "prp_treeN_function_set_number" where N start counting **from 0**.

- **prp_tree0_function_set_number:** Specifies the function set with which we want to generate trees of this index.

  # Example: Main trees number 0 of each individual will be generated with the function set number 3
  Prp_tree0_function_set_number: 3

- **prp_adf_number:** Declaration of the number of ADFs you want to use. The ADFs are numbered from 0. For every ADF we want to use we have to include in this properties file the properties "prp_ADF0_function_set" and "prp_ADF0_interface".

- **prp_ADF0_function_set:** Specifies the function set with which we want to generate trees of this index.

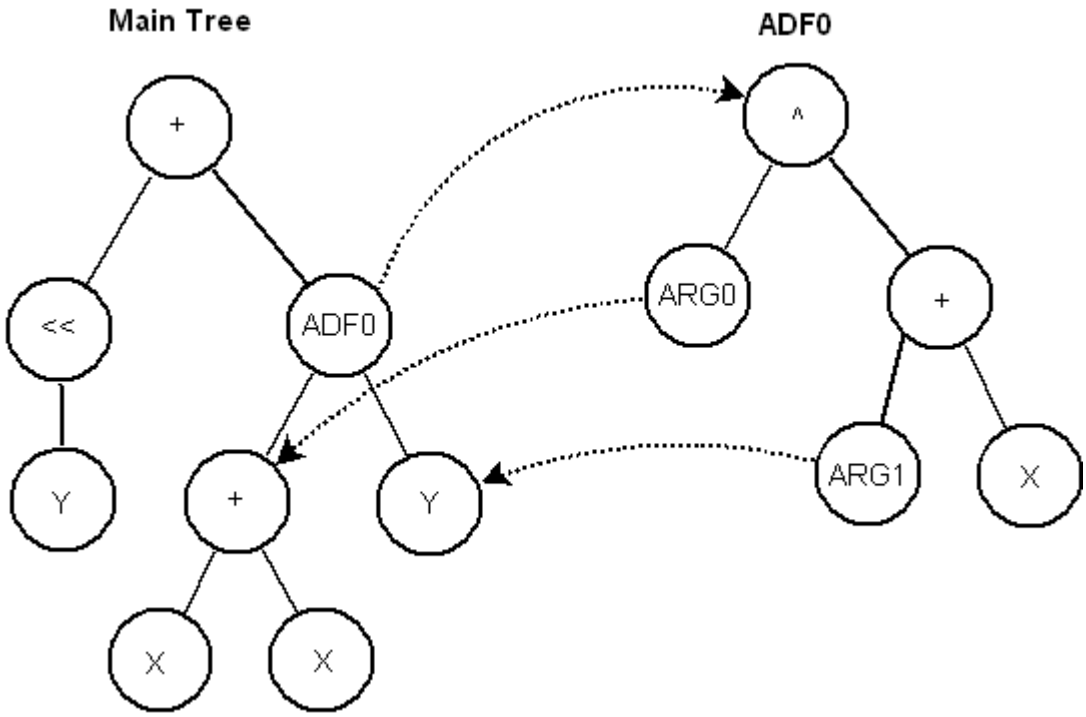
  # Example: The ADF0 of each individual will be generated with the function set number 3
  Prp_ADF0_function_set: 3

- **prp_ADF0_interface:** Specifies the ADF interface using the following format: returnType$$typeArg1$$typeArg2$$TypeArgN…
  Specifying the interface for the ADF we allow that grammars can integrate ADF nodes into the trees as any other function..

  Nota: When declaring the interface of an ADF we declare types for its children, it is being defined implicitly the arity of the ADF. If the ADF has arity N, the function set that we will use to generate these trees must

include in its list of functions and terminals functions ARG (ARG0, ARG1… until ARG (N-1)). This allows that in the ADF trees we can have nodes that allow jumping back to the tree that invoked the ADF. Running ARG0 the execution jumps to the child number 0 of the node that called the ADF.

The figure below helps understanding better this structure of "jumps" or calls.



Some important considerations about the properties files:

- The properties in both properties files can appear in any order.

- If any mandatory property does not appear in the file, ProGen will warn you of the error when trying to run the experiment.

- You can add any comments using the "#" character. Everything from this symbol to the end of the line will be ignored.

- The values of the properties are written without quotation marks. The value "10" with the quotation marks will not be recognized as a number but as a string.

- You can add new properties if you find it useful. The java class responsible for reading (Parser.java) will be explained below. The use of the string "prp_" at the beginning of each property is a convention of style. If you choose to include any property to the file, you are free to adopt it or not.

Examples:

Found	Consequence
prp_property1: value	ProGen will read this property and store its value.
#prp_property1: value	This line is commented; it will be completely ignored by ProGen.
Comments unsigned at the beginning (without the #)	ProGen will read this line and will treat it as a property. It is not altering the execution but it is advised that the comments begin with the "#" for ProGen not to store unnecessary data.
prp_property1:	This property will be read and its value will be stored as null. If ProGen needs that this property has a value other than null, user will be informed with an error.

Note: The inclusion of properties by the user is entirely unnecessary. Whether you wish to extend the functionality of ProGen with your own functions, operators, selectors, etc as if you want to change any of the existing ones, the easiest and most powerful way of passing on information is through "pass-parameter" method.

- Do not be afraid to edit the file properties to configure the experiment entirely to meet your wishes. We have already seen how easy it is, and also be aware that any mistakes you could incur in will be informed by ProGen. All mistakes have an error code. In this document you will also find a detailed explanation of any potential errors that can occur and how to solve them.

- Properties files allow the user to modify the configuration of the experiment to run many different tests and experiments quickly without recompiling anything. There is also the possibility of automating the launch of multiple experiments using the experimenter (see section 5.8: The experimenter).

5.3. USING ADF'S

ADFs, as it has been already mentioned before, are trees that evolve independently. They are created from a different function set whose functions and terminals are chosen by the user and which encode a sub-function of the main program.

If we want to use ADFs in our experiment we must follow the next steps:

1. The use and composition of the ADFs are specified in the properties file, so the first thing we need to do is open it for editing.

2. Let's say we want to use an ADF (it must be named ADF0). The first thing is to think about what functions we should include in this ADF
   For example:
   The ADF0 uses functions: DoublePlusFc, DoubleMultFc and D1 and its return type will be double. In other words, evaluating the ADF0 will return a double.

3. Do we have any function set of these characteristics? Imagine that we have a function set (the main tree), but it uses a different set of functions

and terminals. This means that we have to create a new function set to be used by the ADF0. As we already have a function set (which is the number 0) we create function set number 1. We do the following:

a. We add one to the value of the property "prp_num_function_sets." Before this value was 1 (we only had a function set). Changing the value with a 2 we tell ProGen that we are going to use two sets function (that may or may not be different).

b. We include the properties:

```
prp_function_set_1: DoublePlusFc, DoubleMultFc, D1
prp_return_type_fs_1: double
```

With this we tell ProGen that function set number 1 contains the functions DoublePlusFc, DoubleMultFc and D1, and that the return type for this function set should be double.

4. We already have the function set. Now we have to tell ProGen we are going to use an ADF.

a. In property "prp_adf_number" write the value 1. (One ADF).

b. We include property "prp_ADF0_function_set_number: 1." With this we are telling ProGen that we want that the ADF0 will be built using the function set number 1. (The one we just created).

c. We include property "prp_ADF0_interface: double$$double" This tells ProGen that ADF0 is a function that returns a double and receives a double. In other words, it is not a terminal function and therefore it will not be placed in leaf nodes of the tree that will be generated by this function set (function set 1).

5. Note: The ADF0 receives an argument (in its interface we have declared so). When you run a node ADF0 the execution flow jumps to the tree ADF0. Then we must have the possibility of returning to the original tree for the ADF0´s child node to be also run. It is mandatory to establish a way to evaluate this child and this is done through ARG functions. Since the ADF0 have only one child, we call him ARG0 (if it had two they would

be ARG0 and ARG1, etc.). The function set we use to build the ADF0 (function set 1 in this case) must include the function ARG0. It would look like this:

"`prp_function_set_2: DoublePlusFc, DoubleMultFc, D1, ARG0`"

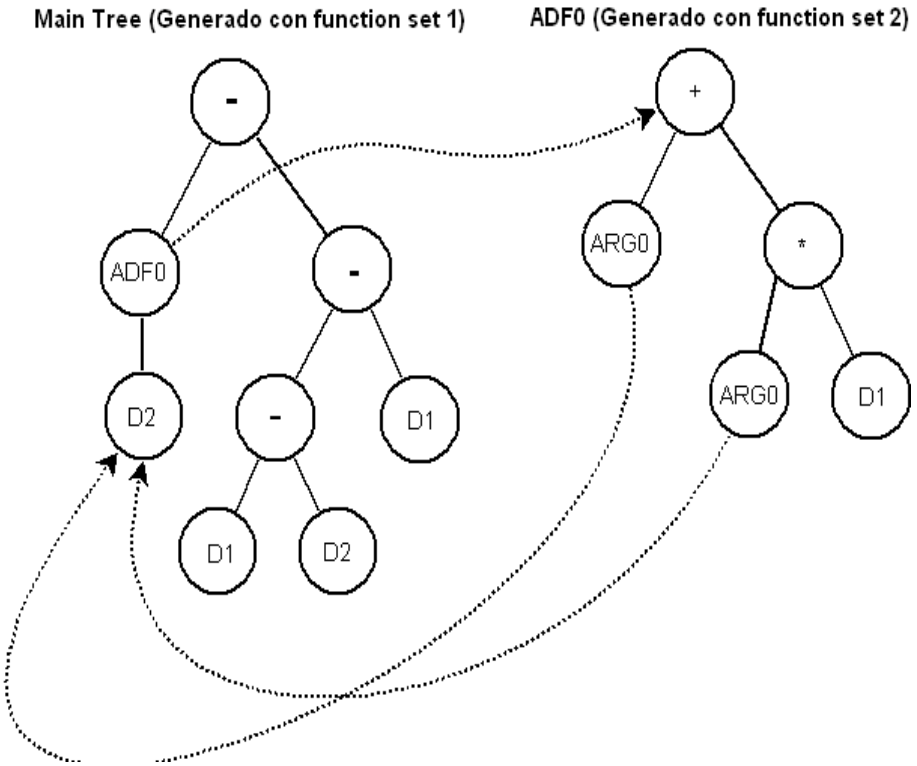
Note: The functions ADF (ADF0, ADF1 ...) and the terminals ARG (ARG0, ARG1 ...) do not require the existence of a Java file with its same name, as it is the case of other functions and terminals. These special functions are created automatically using generic classes ADF.java and ARG.java

6. Are we done? Yes, and no. The ADF0 is declared, it uses a function set and there are no mistakes (we included ARG0 in the function set and the return types of both the ADF (declared in prp_ADF0_interface) and the function set that it uses (declared in Prp_return_type_fs_1) are not contradictory (double in both cases)). Now we only have to use it. Our main tree is not using ADF0 so the ADF0 will be never invoked. In our example, the main tree is using function set 0. We can edit its property as follows:

```
prp_function_set_1: DoubleMinusFc, ADF0, D2, D1
prp_return_type_fs_1: double
```

Note: If you use several ADFs in such a way that they form loops ProGen will inform you of the mistake.

Let's look at a possible individual generated under this configuration:



It is relevant to say that the value of the variables is the same at a given moment for all of the individual trees. For example, if in the figure above the values of D1 and D2 were 10.0 and 2.0 respectively, when evaluating this individual it would be obtained the value: 22.0 - (-2) = **24.0**

Lets remember that trees are executed in preorden, so the first thing is to evaluate the left branch of the main tree, jumping to ADF0 (which returns 22). From this value it is subtracted the result of the execution of the right branch, that is -2.

This structure can get very complicated if we use several ADFs and especially if they have arguments (what results in ARG functions). If we have many ADFs with many ARGs it is very normal that the ADF and ARG nodes get nested causing executions to be very long. We advise not to abuse of the use of ADFs. Use ADFs if you have evidence that they will be useful and note that for obvious reasons (they multiply the number of nodes to process) the time needed to run an experiment will be much higher than if you do not use them. If it is beneficial or not to use ADFs is a controversial issue which has conducted several studies. We can say that the benefit depends on the specific problem and that ADFs are an important part of the Genetic Programming. We have put

special care to optimize performance in the evaluation of the trees so that you have the best tools to experiment as you wish.

5.4. INCLUSION OF NEW FUNCTIONS

This section we will explain how to use your own functions and how to increase ProGen´s own function library.

Creating a function is really easy. The time required depends on the operation that the function has to perform, but for a simple operation we assure you that two minutes is plenty of time. Follow these steps:

1. Go to progen.functions package and open the file of a function to use it as a template. For example PlusFc.java (sum of two integers). The code is shown below:

```
package progen.functions;

import userprogram.UserProgram;
import progen.kernel.*;

/**
 * This class implements the function "Plus".
 */
public class PlusFc extends Function{

	/**
	 * Constructor. Passes to the upper class Function its
	 * arity, signature and symbol
	 */

	public PlusFc(){
		super(2, "int$$int$$int", "+");
	}

	/**
	 * This function adds two numbers and returns the result
	 * @param children the nodes hanging from the node
	 * that contains this function, arguments for the operation.
	 * @param uProgram Reference to the user´s project file.
	 * @param stack Used only for ADFs and ARGs. You can ignore it.
	 * @return The result of the operation
	 */
	public Object execute(PGNode[] children, UserProgram uProgram, PGStack
stack){
		Integer child_1 = (Integer)children[0].evaluate(uProgram, stack);
		Integer child_2 = (Integer)children[1].evaluate(uProgram, stack);

		return  (Integer) child_1.intValue() + child_2.intValue();
	}
}
```

2. We have already opened it, we are now going to edit it. We have marked with bold font things that we need to change (ignoring the comments). Lets go line by line.

a. The name of the class. We put ours. The suffix Fc is a convention used in ProGen, absolutely not mandatory.

b. The name of the constructor method to match the class name as in any Java class...

c. The super sentence: We will specify the arity, the interface and the symbol of our function.

d. The content of the method execute: If our function has arity 2, we know that we can access to children [0] and children [1]. Executing the method **evaluate** we get returned the value of the child. It is returned as an Object so we must make casting to the corresponding type. (We know the type since we have declared it in the interface). Every class is valid as data type. We do with the values of the children whichever operation we want to perform (in this case add the Integer value) and then we return the result.

3. We already have the new function ready to be used. If we want to include a new terminal it will be easier if we take a terminal object as a template (like D1.java). The process is the same, even shorter and simpler.

Note: If we want the function to be visible to all the projects we implement, then we save it in the package progen.functions with all the others and it will be ready to be used for any project that you do. If you prefer that the function is visible only from the current project, it should be saved in your project directory within the directory progen.userprogram. <nameofmyproject> In that case you must change the header of the file so instead of declaring membership to progen.functions package it does to the package progen.userprogram. <nameofmyproject> (without symbols <>).

Note: In the case that the name of your own defined function is the same as other function inside ProGen´s library, ProGen will use yours (it look first in the user's home directory and then in the directory "progen\functions"

5.5. INCLUSION OF NEW OPERATORS

What happens if I need an operator that is not included in ProGen? For example, a new operator that I invented and I want to prove. Can I include it in ProGen and use it as any other? The answer is yes. Simply follow these steps:

1. Go to the directory progen\evolution. You will find there operators which ProGen does include. Use one as a template, for example open Crossover.java.
2. Choose a name for your new operator and save the file with that name plus the extension .java.
3. Use the same name for the class declaration and the constructor method.
4. Edit in the constructor the “super” sentence:
    a. Change the first argument for the name of your operator.
    b. As the second argument put the number of individuals that your operator needs.
    c. As the third argument put the number of individuals that your operator returns.
    d. It is not necessary to touch the three remaining arguments (selector, probability and params). Their values are taken from the properties file.
5. Write the program sentences for the **apply** method. When your operator is selected it will execute the code of this method. Typically all operators begin with the sentence:

```
Individual[] individuals =
_selector.select(_individualsNeeded, file);
```

This sentence asks the selector used by the operator (which we specified in the properties file) to select as many individuals as the operator needs and leave them in the matrix of individuals' individuals'. Typically also all operators end with the sentence

```
return individuals;
```

That returns the individuals (which we operated with during the execution of the method) for ProGen to check their validity, that means to check whether they are in compliance with the restrictions imposed by the user on how a valid individual should be (in terms of maximum number of nodes, maximum depth, and so on.). Resulting individuals considered valid will be part of the new population.

Note: Individuals returned by selectors are copies; that means that operating on them does not alter the original population.

Note: To implement the method “apply” in your operators you will find very useful to observe how ProGen operators access to individuals, their trees, their nodes, and so on. Anyway in the next point we introduce to you some methods that can be used and can be very useful.

Note: Once you finish writing your operator, compile it and leave it in the directory progen\evolution with the other operators so it will be visible to all projects. Once this is done you can select it in your experiment file like any other operator. ProGen dynamically creates the objects from the information specified in the experiment file.

5.5.1. SOME USEFUL METHODS

Here are some methods that can be very useful when it comes to implementing your own genetic operators.

Method	Class it belongs to	Description

Select	Selector	Returns an array with as many individuals as you ask for parameter
GetRandomTree	Individual	Returns a tree Individual randomly chosen
GetParam	Operator	Returns the value of a parameter passed from the properties file. (See section 5.7 The pass-parameters method)
GetRandomNode	PGTree	Returns a node of the tree chosen randomly (among those whose type is the type specified by the parameter)
getRandomCompatibleTree	Individual	Returns randomly an individual tree that has been generated by the same grammar than the tree received by parameter.
getRandomCompatibleNode	PGTree	Returns (randomly chosen) a node of the tree that is compatible (interchangeable) with the node recibed by parameter.
Swap	PGSubTree	Exchanges the subtree that executes the method with the subtree received by parameter. Updates also de number of nodes, depths and roots (if necessary) of the resultant trees.

The methods in this table are presented in a very general way. To get accurate information on what they do, how they do it, parameters they receive,

and so on see the section "detailed design" (within this document, but out of this user manual).

5.6. INCLUSION OF NEW SELECTORS

As with the operators, you can create and include in ProGen new selectors built to meet your specific requirements. To do so just follow these steps:

1. Go to the directory progen\evolution. There you will find the selectors which includes ProGen. Use one as a template, for example open Tournament.java.
2. Choose a name for your new selector and save the file with that name plus the extension .java (progen / evolution).
3. Use the same name for declaring the class and the constructor.
4. Modify the constructor method, specifically the sentence "super":
    a. Change the first argument for the name of your selector.
    b. The remaining two arguments (pop and params) don't need to be touched. Their values are obtained automatically.
5. Edit the program code of the method "select". When your selector is invoked it executes the code inside that method. Keep in mind the following:
    a. The reference to the population is the variable _population defined in Selector.java.
    b. You can resize the visible population by calling to the method amongTheBest, defined in the class Selector.java (it returns new dimension), and then you can use the parameter amongTheBest any time you want (see section 6.3 Selectors).
    c. ProGen v1.0 uses generational evolution. Thus individuals who are selected for the execution of an operator must not return to the original population but to a new one. This means that we always have to return copies of the individuals selected calling to the method **copy**. Take a look at any of the selectors included in ProGen for more clarity.

5.6.1. SOME USEFUL METHODS

Method	Class it belongs to	Description
AmongTheBest	Selector	Resizes the visible population so only individuals among the best “n” can be selected. It returns the new size (virtual size) of the population. See section 6.3 Selectors.
GetRandomTree	Individual	Returns a tree of the individual, randomly chosen.
PrintToMatlab	PGTree	Prints the tree to a file that can be open with Matlab, in order to see the tree shape.
getSize	Population	Returns the number of individuals that the population has.
getIndividual	Population	It returns an individual of the population.
copy	Individual	Returns a copy of the individual.
getAdjustedFitness	Individual	Returns the fitness of the individual.

The methods in this table are presented in a very general manner. To get accurate information on what they do, how they do it, the parameters they receive and so on, see the point "detailed design" (within this document, but out of this user manual).

5.7. PASS-PARAMETER: EASY AND LIMITELESS.

What would happen if you want to expand the possibilities of a selector or an operator? Imagine that what you want is that the RandomSelector (for example) does not return an individual at random, but you want to specify a minimum amount of fitness and return and individual (at random) whose fitness is higher than this value. The selector should look into the population (which is always ordered by their fitness value), and choose randomly an individual between the best individual and the individual border. Whether you create a new selector as if you decide to edit RandomSelector, you need to specify the value of that parameter (the minimum fitness). Creating a new selector in this case does not make much sense because if you specifie for the new selector minimum fitness = 0.0 the behaviour will be identical to the RandomSelector included in ProGen. In other words, you do not need both. The simplest way is to use parameters to extend the functionality of RandomSelector. Let's see step by step how it's done:

1. Choose a name for the parameter. For example “minimum fitness” (Note that you can include spaces if desired). The parameter can be used as in the example:
   “prp_op1_selection: RandomSelector(**minimun fitness = 0.5**)”

2. When you run ProGen, RandomSelector knows already that the parameter "minimum fitness" has the value 0.5. But we are not taking this into account yet. We must change the method of select RandomSelector to use the new parameter, for example:

```
...
// We get the parameter value with getParam and convert it to double.
double fitmin = Double.parseDouble(getParam(“minimun fitness”));
do{
        // What RandomSelector was already doing
        IndividualSelected = selectIndividualRandomly ();
while (individualSelected has fitness <fitmin)
```

```
        return IndividualSelected;
```

As it can be seen, flexibility is maximum, and you only need to modify the file selector whose functionality you want to extend, modify, etc. or create another selector considering as many parameters as desired.

Note: In the experiment properties file you can specify the parameters of the selectors and operators in any order.

Note: If in the experiment properties file you declare the use of a parameter that does not exist (that the selector does not take into account) absolutely nothing happens.

Note: If within your selector's select method you invoke getParam asking for a parameter that has not been passed through the experiment properties file, the method getParam will return the string "null".

5.8. EL EXPERIMENTER

What we are going to describe below is another feature that you will only find in ProGen. This is a simple and elegant way to define and execute a battery of experiments one after another without further intervention from the user. The experimenter allows you to:

1. Define the value of one or several properties in the form of ranges with the following shape : prp_property: "*initial value, final value: offset*
Example: prp_population_size: 100, 500: 200

Including this property in the experimenter will turn your experiment into 3 different experiments: One for a population size of 100, one for a population size of 300 and another for a population size of 500. The remaining properties will be taken from the experiment properties file.

2. Get a separate output for each experiment launched, so that it can be easily analyzed and then processed to obtain statistics, graphics, or any type of data.

Note: When several properties are included in the experimenter, all experiments resulting from all the possible combinations with all the values of these properties will be launched.

For the experimenter to be put into operation, the property "**prp_experimenter**" in the main properties file must be set to have the value **ON** (no matter if you use capital or small letters).

Then include the properties you want to play with (whose value you want to change). As it was already mentioned, if you include more than one property in the experimenter, it will generate all the possible experiments arising from the combination of the possible values for each property. Let's illustrate this with an example: If you include the following lines in the main properties file:

```
#EXPERIMENTER
prp_experimenter: on
prp_population_size: 100,500:100
#prp_initialization_mode: grow, full
#prp_max_nodes: 100,300:50
#prp_max_depth: 6,10:1
prp_generations: 100,150:50
```

The experimenter will be launched (since it is configured with the value "on") and ProGen will proceed to launch **10** experiments. Why 10? Note that the only properties that are not commented are:

```
prp_population_size: 100,500:100
```

(5 possible values: 100, 200, 300, 400 and 500) and

```
prp_generations: 100,150:50
```

(2 possible values: 100 and 150).

From these values they can be obtained 10 different combinations that will define 10 different experiments. (100 individuals and 100 generations, 100 individuals and 150 generations, 200 individuals and 100 generations ... etc.).

Note: The only properties that can be used in the experimenter are those getting numerical values and the property "prp_initialization_mode".

5.9. THE OUTPUT

For the output we have mainly developed three java classes.

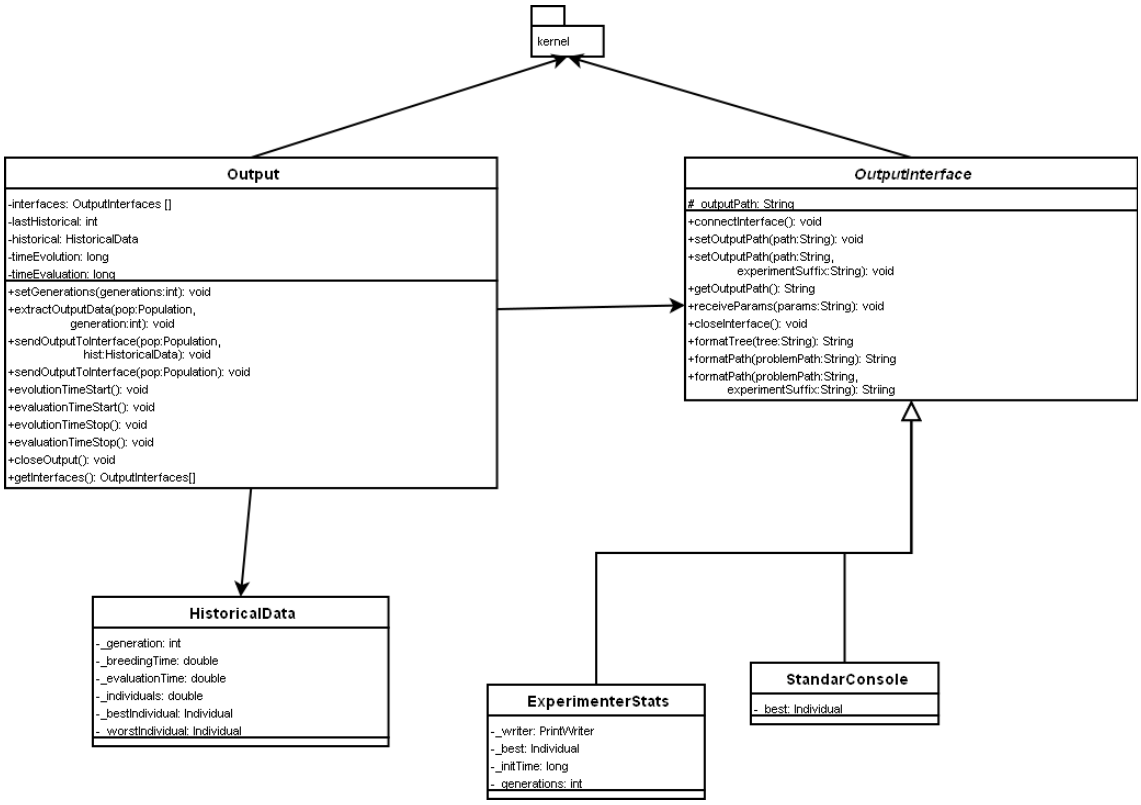
One of them is the HistoricalData class. The purpose of this class is to store all important information about the execution of a generation in order to make it accessible at the end of the execution. We have considered that it would be interesting to use this information to construct graphical diagrams and others. For each generation this class stores: the number of the generation, the breeding time, the evaluation time, the number of individuals, the best, worst and average individual, etc.

The main class of the output package is the class Output. In each generation this class extracts all information from the population at a generation and set it at a HistoricalData object wich is stored into an array. Once all data is processed, Output sends it to the OutputInterfaces.

The flexible element of the output package is the abstract class OutputInterface. This class defines the expected functionality of all compatible output interfaces at the time it implements some of the methods which will be common to all of them.

Progen offers a variety of classes which extends this class, and offers the most usual output requirements such as ExperimentsStats, StandardConsole, StandardFile, etc. At any moment the user will be able to define his own OutputInterfaces in an easy way.

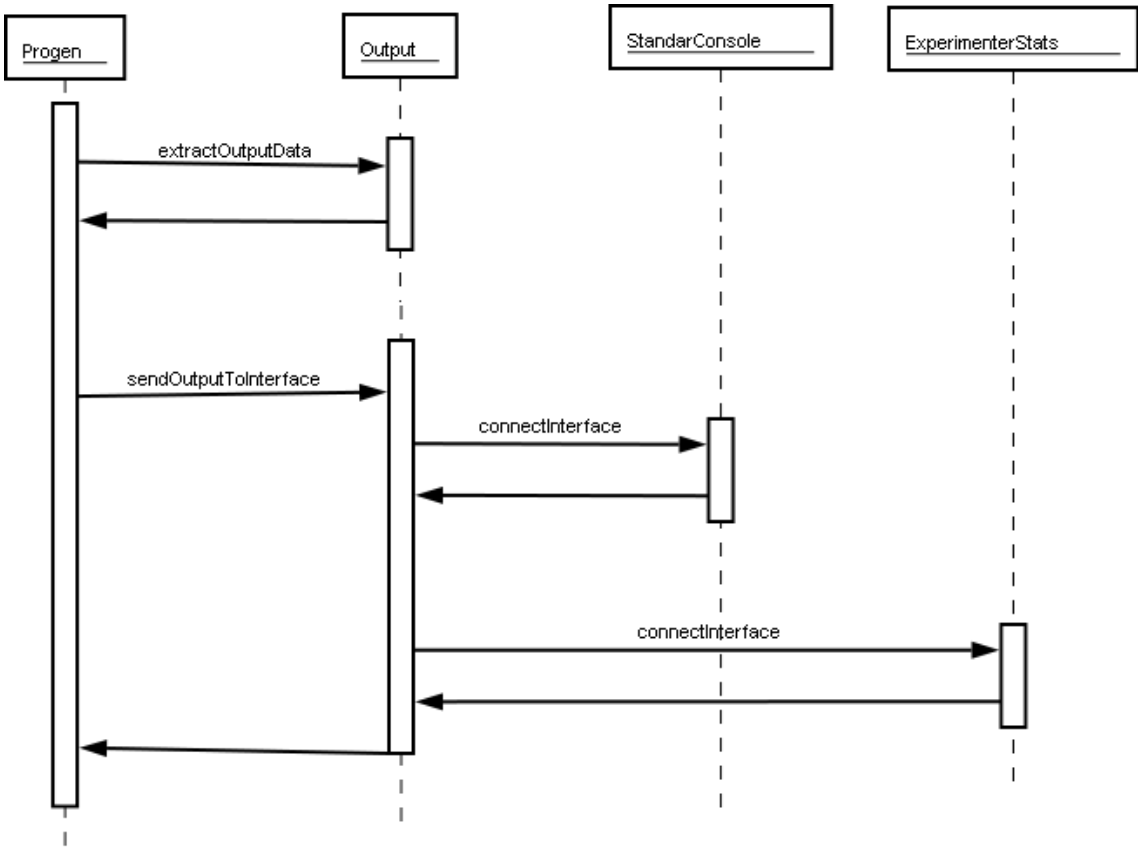
Here we have a simplified class diagram. Into this diagram, doesn't appear all attributes and all methods of each class. In the same way all classes which extends OutputInterface aren't shown.



Any user can make his own OutputInterface class by programming a class which extends the abstract class OutputInterface. This class will have to implement the abstract methods defined by OutputInterface such as the following:

- receiveParams which allows ProGen to give arguments, if necessary, to the class.
- closeInterface which finalizes the OutputInterface, if necessary, as it is for file streams.
- connectInterface which manage the information in the required way, such as calling System.out.println, etc.

The functionality of this package is defined in the following sequence diagram:



This allows that each OutputInterface manages the information in their own way, being called by the Output class.

5.9.1. CREATING A NEW OUTPUTINTERFACE

It is easy and usefull to the user to be able to create their own output system. This can be done through the following steps:

1. Create a class which extends OutputInterface into the output package.
2. Implement the abstract method according to the user requirements.
3. Include the class into the o_interfaces property of the master_file.cfg file.

The output of ProGen allows you to encapsulate output systems into an output element. For example, you can program a class which contains other three OutputInterfaces. By this way, into the new outputinterface method connectInterface you will call the connectInterface method of the other OutputInterfaces encapsulated.

5.9. THE OUTPUT

For the output we have mainly developed three java classes.

One of them is the HistoricalData class. The purpose of this class is to store all important information about the execution of a generation in order to make it accessible at the end of the execution. We have considered that it would be interesting to use this information to construct graphical diagrams and others. For each generation this class stores: the number of the generation, the breeding time, the evaluation time, the number of individuals, the best, worst and average individual, etc.

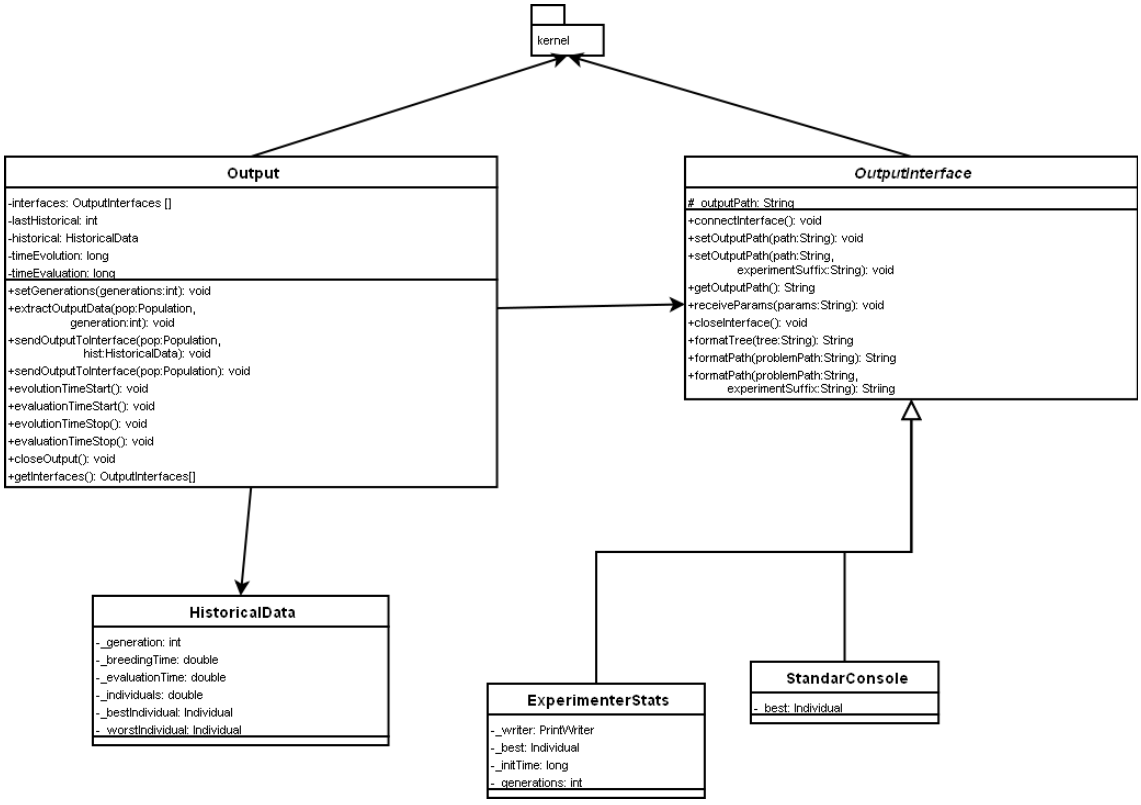
The main class of the output package is the class Output. In each generation this class extracts all information from the population at a generation and set it at a HistoricalData object which is stored into an array. Once all data is processed, Output sends it to the OutputInterfaces.

The flexible element of the output package is the abstract class OutputInterface. This class defines the expected functionality of all compatible output interfaces at the time it implements some of the methods which will be common to all of them.

Progen offers a variety of classes which extend this class, and it offers the most usual output requirements such as ExperimentsStats,

StandardConsole, StandardFile, etc. At any moment the user will be able to define his own OutputInterfaces in an easy way.

Here we have a simplified class diagram. Into this diagram, it doesn't appear all attributes and all methods of each class. In the same way all classes which extend OutputInterface aren't shown.

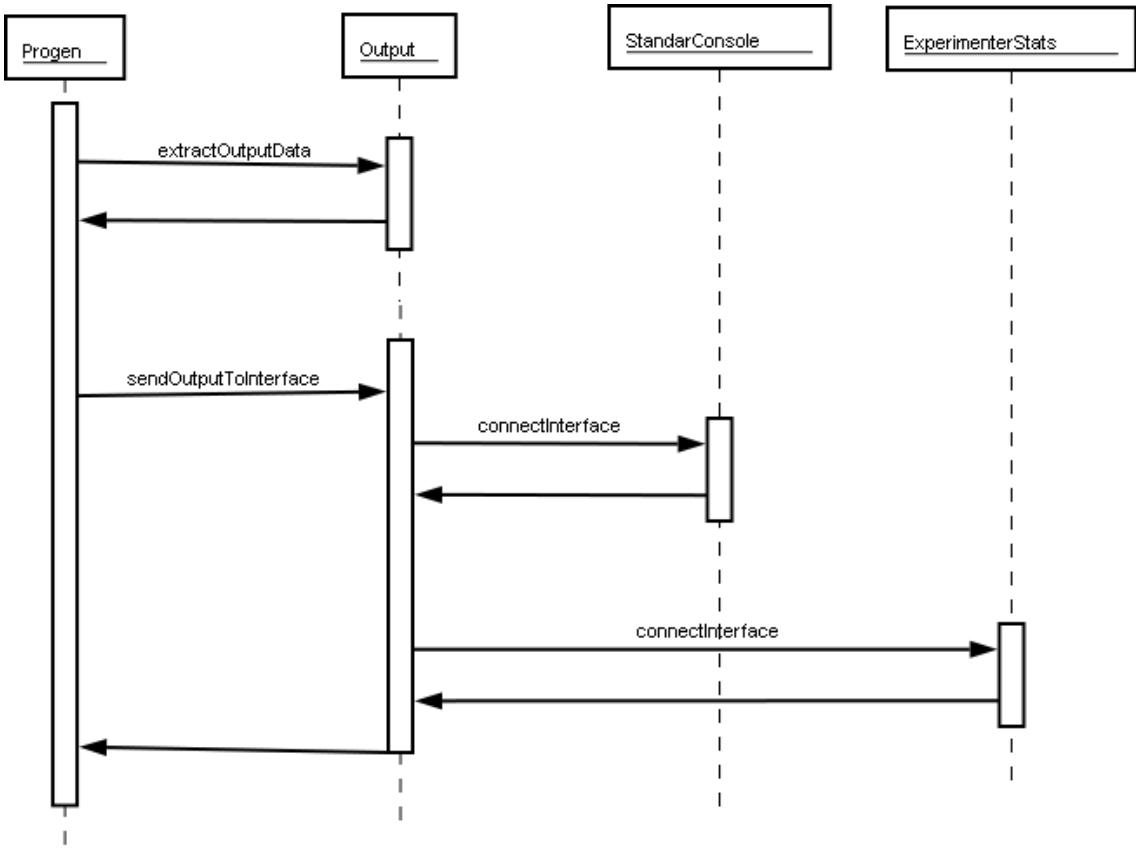


Any user can make his own OutputInterface class by programming a class which extends the abstract class OutputInterface. This class will have to implement the abstract methods defined by OutputInterface such as the following:

- receiveParams which allows ProGen to give arguments, if necessary, to the class.

- closeInterface which finalizes the OutputInterface, if necessary, as it is for file streams.
- connectInterface which manage the information in the required way, such as calling System.out.println, etc.

The functionality of this package is defined in the following sequence diagram:



This allows that each OutputInterface manages the information in their own way, being called by the Output class.

5.9.1. CREATING A NEW OUTPUTINTERFACE

It is easy and useful for the users to be able to create their own output system. This can be done through the following steps:

4. Create a class which extends OutputInterface into the output package.
5. Implement the abstract method according to the user requirements.
6. Include the class into the o_interfaces property of the master_file.cfg file.

The output of ProGen allows you to encapsulate output systems into an output element. For example, you can program a class which contains other three OutputInterfaces. By this way, into the new outputinterface method connectInterface you will call the connectInterface method of the other OutputInterfaces encapsulated.

5.9.2. THE OUTPUT OF PROGEN

Despite the OutputInterfaces indicated into the master_file.cfg, ProGen generates some initial information and shows it into the console.

```
****************************************
*                ProGen                *
****************************************

Construyendo gramatica: Valor de retorno:Integer
R0: A0 , A1 , Aterminal2
R1: Aterminal3
A0: ( userprogram.Gphash.BitAndFc R0 R0 ) , (
userprogram.Gphash.BitOrFc R0 R0 ) , ( userprogram.Gphash.BitXorFc R0
R0 ) , ( userprogram.Gphash.BitMultFc R0 R0 ) , (
userprogram.Gphash.BitSumFc R0 R0 )
A1: ( userprogram.Gphash.BitNotFc R0 ) , (
userprogram.Gphash.BitVrotdFc R0 )
Aterminal2: userprogram.Gphash.A0 , userprogram.Gphash.Hval
Aterminal3: userprogram.Gphash.Bit32ERC
Axiom: A0 , A1 , Aterminal2 , Aterminal3

ProGen---> Generating population RPB0
I:0 A:1 Depth:4 Nodes:10: ( userprogram.Gphash.BitXorFc (
userprogram.Gphash.BitNotFc ( userprogram.Gphash.BitNotFc (
userprogram.Gphash.BitVrotdFc userprogram.Gphash.A0 ) ) ) (
userprogram.Gphash.BitNotFc ( userprogram.Gphash.BitNotFc (
userprogram.Gphash.BitMultFc userprogram.Gphash.Hval
userprogram.Gphash.A0 ) ) ) )
I:1 A:1 Depth:4 Nodes:9: ( userprogram.Gphash.BitSumFc (
userprogram.Gphash.BitVrotdFc userprogram.Gphash.A0 ) (
userprogram.Gphash.BitVrotdFc ( userprogram.Gphash.BitOrFc (
```

```
userprogram.Gphash.BitOrFc userprogram.Gphash.Hval
userprogram.Gphash.Hval ) userprogram.Gphash.Hval ) ) )
...
I:198 A:1 Depth:3 Nodes:12: ( userprogram.Gphash.BitXorFc (
userprogram.Gphash.BitOrFc ( userprogram.Gphash.BitVrotdFc
userprogram.Gphash.A0 ) ( userprogram.Gphash.BitNotFc
userprogram.Gphash.A0 ) ) ( userprogram.Gphash.BitOrFc (
userprogram.Gphash.BitXorFc userprogram.Gphash.A0
userprogram.Gphash.A0 ) ( userprogram.Gphash.BitVrotdFc
userprogram.Gphash.A0 ) ) )
I:199 A:3 Depth:6 Nodes:18: ( userprogram.Gphash.BitNotFc (
userprogram.Gphash.BitMultFc ( userprogram.Gphash.BitVrotdFc (
userprogram.Gphash.BitNotFc ( userprogram.Gphash.BitVrotdFc (
userprogram.Gphash.BitXorFc userprogram.Gphash.Hval
userprogram.Gphash.A0 ) ) ) ) ( userprogram.Gphash.BitNotFc (
userprogram.Gphash.BitMultFc ( userprogram.Gphash.BitOrFc (
userprogram.Gphash.BitNotFc userprogram.Gphash.A0 ) (
userprogram.Gphash.BitNotFc userprogram.Gphash.Hval ) ) (
userprogram.Gphash.BitNotFc ( userprogram.Gphash.BitVrotdFc
userprogram.Gphash.Hval ) ) ) ) ) )
ProGen---> Initial population generated
```

First of all, we can see the welcome of ProGen. Then it will be shown information about the grammar used to generate the population, and finally a description of all the individuals of the initial population.

Once this initial information has been shown, ProGen starts managing the information of the generations. The information corresponding to the StandardConsole has the following view:

```
--------------------
|-=   Generation 0 =-|
=============================================================================
   Individual  |        Raw Fitness  | Adjusted Fit.  | Nodes tree 0 | Depth tree 0 |
-----------------------------------------------------------------------------
  Best of Gen. |        48,4510 |       0,0202 |        21 |         6 |
Generation Mean |      8026,4785 |       0,0022 |    10,6917 |     3,9500 |
  Worst of Gen. |     44236,4651 |       0,0000 |         7 |         3 |
-----------------------------------------------------------------------------

-------------------------------------------------------
   Time (ms.)  |   Population Mean  | Total Population Time  |
-------------------------------------------------------
  Breeding Time |          0,0000 |           0,0000 |
Evaluation Time |          1,9867 |        1192,0000 |
-------------------------------------------------------

New Best Individual:

        Raw fitness: 48.45102520710212
        Adjusted fitness: 0.020222027669031636

          Tree  Nodes  Depth
        --------------------
        Tree 0     21     6
        --------------------

Tree 0:
        (>>>        (^          (>>>        (^          (~          (>>>        hval
                                                                    )
                                                        )
                                                        (|          (~          hval
                                                                    )
                                                                    (&          a0
                                                                                a0
                                                                    )
                                                        )
                                            )
                                (~          (>>>        (+          (>>>        a0
                                                                    )
                                                                    (*          a0
                                                                                hval
                                                                    )
```

```
                                                    )
                                                )
                                            )
                                        )
                                    )
```

```
----------------------
|=    Generation 1 =-|
```

Individual	Raw Fitness	Adjusted Fit.	Nodes tree 0	Depth tree 0
Best of Gen.	48,3787	0,0203	21	5
Generation Mean	2032,2948	0,0087	13,7600	4,4050
Worst of Gen.	44236,4651	0,0000	13	4

Time (ms.)	Population Mean	Total Population Time
Breeding Time	0,0667	40,0000
Evaluation Time	2,2200	1332,0000

New Best Individual:

Raw fitness: 48.37872516488049
Adjusted fitness: 0.020251636644342277

Tree	Nodes	Depth
Tree 0	21	5

Tree 0:

```
(>>>        (*          (~          (*          (>>>        hval
                                                )
                                                (+          a0
                                                            a0
                                                )
                                    )
                        )
                        (^          (*          (&          a0
                                                            hval
                                                )
                                                (>>>        hval
                                                )
                                    )
                                    (+          (>>>        a0
                                                )
                                                (>>>        hval
                                                )
                                    )
                        )
                    )
```

```
----------------------
|=  Generation 932 =-|
```

Individual	Raw Fitness	Adjusted Fit.	Nodes tree 0	Depth tree 0
Best of Gen.	3,7087	0,2124	25	19
Generation Mean	217,2546	0,1382	23,9383	18,1550
Worst of Gen.	44236,4651	0,0000	25	19

Time (ms.)	Population Mean	Total Population Time
Breeding Time	0,0333	20,0000
Evaluation Time	3,2200	1932,0000

New Best Individual:

Raw fitness: 3.7086689608497205
Adjusted fitness: 0.21237424170492997

Tree	Nodes	Depth
Tree 0	25	19

Tree 0:

```
(*          (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (*          (>>>
(>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (^          hval
                                                                                    a0
                                                                        )
                                                            )
                                                )
                                    )
                        )
                    )
                )
            )
                                                                                                )
                                                                                                (^
hval
```

```
        a0
                                                                                    )
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                a0
                            )
=============================================================================

----------------------
|= Generation 1000 =-|
=============================================================================
    Individual  |         Raw Fitness  |  Adjusted Fit.  | Nodes tree 0 | Depth tree 0 |
-----------------------------------------------------------------------------
  Best of Gen.  |          3,2569  |        0,2349  |         25  |         19  |
Generation Mean |        112,0751  |        0,1416  |    24,0750  |    18,2467  |
  Worst of Gen. |      12342,4580  |        0,0001  |          1  |          0  |
-----------------------------------------------------------------------------

-------------------------------------------------------------------
    Time (ms.)  |    Population Mean  | Total Population Time  |
-------------------------------------------------------------------
  Breeding Time |             0,0667  |             40,0000  |
Evaluation Time |             3,2883  |           1973,0000  |
                ===================================================
```

As we can see, the information about the different generations is divided into three different sections.

The first of them shows the number of the generation whose information is shown into the table.

The second one shows information about the individuals which belong to that population. In this table it is shown: the raw fitness, the adjusted fitness, the number of nodes and the depth of the individual. All this information is shown about the best, worst and average individual of the population.

The third table shows the time in milliseconds used to breed and evaluate each individual and the full generation.

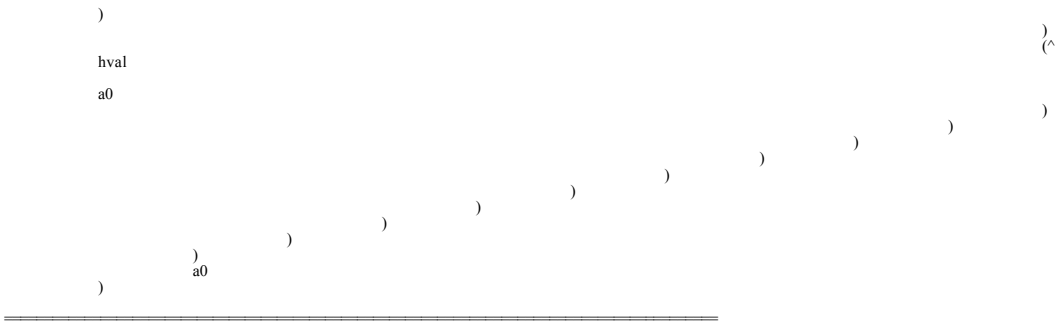
Whenever a new best individual is reached the following information is shown.

```
New Best Individual:

        Raw fitness: 3.7086689608497205
        Adjusted fitness: 0.21237424170492997

          Tree  Nodes  Depth
        ----------------------
        Tree 0     25     19
        ----------------------

Tree 0:
        (*          (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (*          (>>>
        (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (>>>        (^          hval
                                                                                        a0
                                                                                    )
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                a0
                            )
```



The first part talks about the information of the best individual, including the raw fitness, the adjusted fitness, the number of nodes and the depth. Then there are shown the trees and adfs that define the individual. Note that the trees and adfs are shown in preorden notation.

5.10. ERRORS: DESCRIPTION AND SOLUTIONS

Experiments on ProGen are configured from a properties file. This properties file specifies for example which functions will include each of the function sets you want to use, the genetic operators, the number of trees, and so on. For the experiment to run, it is necessary that ProGen checks that the data entered is valid, that there are no inconsistencies, that not necessary information is missed, and so on. When ProGen encounter any error in the properties file it informs the user by the standard output printing an error, which consists of a numeric code and a textual description. The error message is usually completed with specific information about the exact location where it was found.

Below there is a list of the different errors that ProGen can detect and you can get, as well as its solution.

Code	**0**

Message	"Error 0: Function set wrongly built. You are using ADF´s or funtions whose arguments have an unreachable type"
Solution	For the grammar to be able to generate trees, it is neccesary that for every function or ADF, the types that they receive (i.e. the return type of their children) are returned by other functions of the function set, or that in the function set there are terminals of that type. For example, if we use a function that needs a double, and we dont use any function or Terminal returning double, it will be imposible that the grammar can generate trees including that function or ADF. For the function or ADF that raised the error we must make sure that there are other functions or terminals returning the types that that function or ADF receives as parameters. (Or remove that function or ADF from the function set if you don't want to use it).

Code	**1**
Message	"Error 1: Setting a value of incompatible type to the following variable"
Solution	The user, in his program, has tried to asign to the mentioned variable a value of an incorrect type. When the user invokes setVariable he must get sure that he is passing an object of the same type of the variable. For more information read the section 7.4: the method setVariable()

Code	**2**

Message	"Error 2: The function set you try to use to generate the tree or ADF is invalid (it must be a number between 0 and the specified value in \"prp_num_function_sets\""
Solution	Get sure that all the trees and ADFs that you want to use are generated using function sets previously declared.

Code	**3**
Message	"Error 3: It could not be found the file"
Solution	You are trying to use a file that has not been found. Get sure that the file exists in the correct path.

Code	**4**
Message	"Error 4: A function set that is used by a main tree (not ADF) can´t contain ARG´s"
Solution	Main trees are not invoked from other trees, so it makes no sense that they include ARG nodes (used to jump back to the n-child of the node that invoked the tree where the ARG is). Make sure that there are no ARG terminals in the function sets used by main trees.

Code	**5**
Message	"Error 5: Invalid name for an ADF funcion (they must be named ADFi being i a natural positive number)"
Solution	Make sure that in your function sets there are no ADFs that does not match the correct format.

Code	6
Message	“Error 6: ADF not declared,”
Solution	Make sure that in your function sets there are no ADFs whose index is bigger or equals than the number of ADFs declared in the property “prp_adf_number”.

Code	7
Message	"Error 7: The return type of an ADF must be the same as the return type of the function set that it uses"
Solution	If an ADF returns certain data type and uses certain function set, the type returned by the trees generated by that function set must match with the type returned by the ADF. The oposite would be a contradiction.

Code	8
Message	"Error 8: All ADF´s using the same function set must have the same interface"

Solution	If several different ADFs use the same function set, these ADFs must have the same interface (return and wait for the same types and have the same arity). Failure to do so leads to ambiguity in the evaluation of ARGs. Example: Suppose we have the following situation: - An individual with a tree and two main ADFs (ADF0 and ADF1) - The main tree uses the function set number 0 where they are included ADF0 and ADF1 - ADF0 has arity 3 and ADF1 has arity 2 . - The function set that uses ADF0 (for example function set number 1) must necessarily contain ARG0, ARG1 and ARG2, since ADF0 has arity three and it should be possible to evaluate any of its branches . - From this situation, we can conclude that ADF1 can not use the function set number 1, as it may contain functions ARG2 when ADF1 has only arity two. The evaluation of a node ARG2 in ADF1 would be an error. With this we explain the need for all ADF using the same function set to have the same arity... But, do they also need to have exactly the same interface? The answer is yes. When creating the ADF trees, for ARG nodes can be placed in the tree, they need to have a type, but ... What is the type of the ARG? At the moment of creating the ADF tree it is necessary that the ARGs have a type. But if the child 0 of ADF0 is an integer, and the son 0 of ADF1 is not, and ADF0 and ADF1 use the same function set (they have been generated by the same grammar and therefore are interchangeable) could happen that after a crossover, evaluating a ARG node returns a different type that the type that the parent node of that ARG is expecting.

Code	9
Message	"Error 9: Invalid name for an ARG function (they must be named ARGi being i a natural positive number)"
Solution	Make sure that no function set includes ARGs that does not match the correct format.

Code	10
Message	"Error 10: Acording to the interface of the ADF´s that uses the following function set, it doesn´t exist such ARG"
Solution	A function set contains an ARGN, but the ADFs that use that function set don't have that N child. Remove that ARG from the function seto r modif. The interface of the ADFs that use that function set for the inconsistency to dissapear.

Code	11
Message	"Error 11: Acording to the interface of the ADF´s that uses the following function set, it must be present the following ARG"
Solution	If an ADF receives arguments, then the function set that it uses must include an ARG function for each of them. Include the ARG indicated or modify the interface of the ADFs using that function set, removing from it the specified child.

Code	12
Message	"Error 12: The way the ADF´s are being used can generate loops"

Solution	One or more of your ADF are using function sets that try to invoke to each other directly or by means of one or more other function sets. Review your function sets to avoid this situation.

Code	**13**
Message	"Error 13: Since no ADF is using the following function set the type of the ARG´s is impossible to be determinated so they must be removed from the function set"
Solution	In a function set there are functions ARG and no ADF is using that function set. That means that the parser is unable to determine the type of the ARG. Remove it from the function set.

Code	**14**
Message	"Error 14: Make sure that the following property has a natural positive number as a value"
Solution	The message is quite self-explanatory

Code	**15**
Message	"Error 15: You must write a value for the following property"
Solution	The message is quite self-explanatory

Code	**16**
Message	"Error 16: It couldn’t be found the following compulsory property"
Solution	Include (or uncomment) the specified property.

Code	17
Message	"Error 17: The following property must have one of the following values"
Solution	The message is quite self-explanatory

Code	18
Message	"Error 18: It was impossible to instantiate the file"
Solution	Make sure that the file exists in the correct path and that its corresponding .java file does not implement interfaces not imported.

Code	19
Message	"Error 19: Illegal access when instantiating the file"
Solution	This error can not be produced by a wrong configuration of the properties files. If you get this error, review the indicated file (that will for sure be one of those created by the user) and check if it has access restrictions.

Code	20
Message	"Error 20: Make sure that all function sets include at least one terminal function of the same type as the function set return type"
Solution	This error shouln not be launched, since it is not true that a tree cant be generated if it returns certain type and the function set doesn't include at least one terminal of that type.

Code	21
Message	"Error 21: The following property was defined by the user as compulsory, but it was not found in the properties file"
Solution	This error corresponds to a disabled feature. It can not be produce in this version of ProGen.

Code	22
Message	"Error 22: The parameters of the following operator or selector doesnt fit the correct format"
Solution	Operators and selectors can receive parameters. The format is name = value. Parameters must be separated by commas and the whole set of parameters must be passed in brackets. Example: operator (name1=value1, name2=value2,…)

Code	23
Message	"Error 23: Coulndt be invoked the constructor for the class"
Solution	This error is launched when you invoke a constructor of a clase (using reflexion) and there is any error with the parameters. This error should not take place since you do not need to edit the code responsible for this features (although of course you can do it if you want).

Code	24
Message	"Error 24: Incorrect value: The following property or param must contain either a positive numeric value or a percentage value (positive number from 0 to 100, followed by the sign \"%\")"

Solution	The message is quite self-explanatory.

Code	**25**
Message	"Error 25: Probability properties must contain a value between 0 and 1"
Solution	The message is quite self-explanatory.

Code	**26**
Message	"Error 26: The adition of the probabilities of all operators must be equals 1"
Solution	The message is quite self-explanatory.

Code	**27**
Message	"Error 27: It couldn't be created the file"
Solution	The file we want to create could not be created. Make sure that the path where you want to create the file exists.

Code	**28**
Message	"Error 28: The file couldn't be written"
Solution	Make sure the file exists and it’s not write-protected.

Code	**29**
Message	"Error 29: The file couldn't be closed"

Solution	Make sure the file exists and it has not been previously closed.

Code	**30**
Message	"Error 30: You must specify a size param for the Tournament operator (a positive number between 1 and the population size)"
Solution	If you use the selector “Tournament” it is mandatory that you specify the parameter “size”.

Code	**31**
Message	"Error 31: The max_attempts value was reached while trying to get valid individuals by applying the operator"
Solution	Raise the value of this property, or relax the conditions needed for a tree to be valid (increase the number of nodes, the maximun depth of the tree…).

Code	**32**
Message	"Error 32: Values that represent intervals must have the following format: Number1,Number2:Number3 All the numbers must be positive and follow the relation: Number1 <= Number2"
Solution	Write the range using the format specified in the message.

Code	**33**

Message	"Error 33: The population size must be greater than zero. If it´s already set to a value greater than zero in your experiment file, check the experimenter in your main properties file"
Solution	Make sure you specify a value for the population size bigger than zero. Note that it is possible that you have the experimenter on and you are specifying there the zero value.

Code	**34**
Message	"Error 34: The variable you are trying to access to, doesn't exist"
Solution	Make sure that a terminal with the name specified in the message exists (and that you are using it in any function set).

Code	**35**
Message	"Error 35: In the property prp_depth_interval the format is: 'n1, n2' where: n1, n2 are integer numbers and (1 < n1 <= n2)"
Solution	Write the value of the property in the correct format.

Code	**36**
Message	"Error 36: Your function sets must contain at least one function returning the type specified as the return type for that function set. Otherwise in the best case all your trees will have only one node"
Solution	If a tree must return certain data type, the function set used to generate that tree must contain at least one function that returns that same type. Make sure this condition is met.

Code	37
Message	"Error 37: Trying to generate randomly a valid individual, the max_attempts value was reached"
Solution	Raise the value of this property, or relax the conditions needed for a tree to be valid (increase the number of nodes, increase the range prp_depth_interval…).

Code	38
Message	"Error 38: You declared an ADF that does not appear in the function set of any principal tree (RPB). ADF is inaccessible and thus useless. If you want to use trees that cannot be accessed by RPBs, then define a RPB instead of an ADF (PRoGen supports multiple RPBs in he same individual). This error was produced in ADF"
Solution	Message is self-explanatory.

6. LIBRARY AVAILABLE

In this point we are going to introduce the library of functions, genetic operators and selectors included in this first version of ProGen. Note that the list of funcionts as well as the lists of operators and selectors can be enhanced with your own functions, operators or selectors respectivement.

6.1. FUNCTIONS AND TERMINALS

The functions available in ProGen 1.0 are:

Name	Return type	Simbol	Arity	Type of its children	Description
AndFc	boolean	^	2	boolean, boolean	Logic and
NotFc	int	~	1	boolean	Logic nor
OrFc	boolean	v	2	boolean	Logic or
ExclOrFc	boolean	·	2	boolean	Exclusive or
DoubleMinusFc	double	-	2	double, double	Subtraction of two doubles
DoubleMultFc	double	*	2	double, double	Multiplication of two doubles
DoublePlusFc	double	+	2	double, double	Addition of two doubles
DoubleDivFc	double	/	2	double, double	Protected division of two doubles
GrtEqThanFc	boolean	>=	2	int, int	Greater or equals than
LwEqThanFc	boolean	<=	2	int, int	Lower or equals than
LwThanFc	boolean	<	2	int, int	Lower than

GrtThanFc	boolean	>	2	int, int	Greater than
Equals	boolean	==	2	Object, Object	Equals
MinusFc	int	-	2	int, int	Subtraction of two integers
MultFc	int	*			Multiplication of two integers
PlusFc	int	+	2	int, int	Addition of two integers
DivFc	int	/	2	int, int	Protected division of two integers

The terminals available are:

Name	Return type	Simbol	Description
B1	boolean	B1	Variable boolean
B2	boolean	B2	Variable boolean
D1	double	D1	Variable double
D2	double	D2	Variable double
I1	int	I1	Variable integer
I2	int	I2	Variable integer

All these functions can be selected to be part of any function set in any properties file of any experiment and there is no need to import any file for it.

Note: If we want to override any of these functions or terminals (to use the same name for another function or terminal with a different behaviour) we can do it including the function or terminal in the project directory. ProGen will search first in the project directory and only if the function or terminal is not found there it will search in its own library.

Note: Symbols can be repeated without any problem; although we recommend to repeat them only when doing so doesn't affect the clarity. (For example using the symbol + for adding integers and doubles).

6.2. OPERATORS

The genetic operators available in ProGen 1.0 are the following:

Name	**Crossover**	
Description	Chooses two nodes in two individuals and crosses (exchanges) the subtrees hanging from them.	
Supported parameters	**Name**	Internal
	Values	Between 0.0 and 1.0
	Description	Probability of selecting for the crossover an internal node of the tree.

Name	**PointMutation**	
Description	Chooses a node of an individual and changes the function that it contains with another equivalent (same interface) of the same function set that generated the tree.	
Supported parameters	**Name**	internal
	Values	Between 0.0 and 1.0
	Description	Probability of selecting for the crossover an internal node of the tree.

Name	**GrowMutation**

Description	Chooses a node of an individual, cuts the subtree hanging from it and creates a new branch in the place of the old one.	
Supported parameters	**Name**	internal
	Values	Between 0.0 and 1.0
	Description	Probability of selecting for the crossover an internal node of the tree.
	Name	levels
	Values	Integer number greater than zero.
	Description	Maximun number of levels that we want the new branch to grow under the selected node.
	Name	mode
	Values	grow or full
	Description	Growing mode for the new branch.

Name	**Reproduction**	
Description	Returns the individuals untouched.	
Supported parameters	**Name**	---
	Values	---
	Description	---

6.3. SELECTORS

The selectors available in ProGen 1.0 are the following:

Name	**RandomSelector**
Description	Choose an individual randomly from the population.

Supported parameters	**Name**	amongTheBest
	Values	A percentage (0% to 100%), or an absolute value between 0 and the population size.
	Description	Resizes the population from which the selector will select individuals. If the value received is a percentage, the visible population will be that percentage of the best individuals in the population (example 50% best individuals). If the value received is an absolute value the population visible is that number of individuals counting from the best (example the 94 best individuals).

Name	**Roulette**	
Description	Chooses an individual randomly according to its fitness proportional. That is, individuals who have fitness values closer to 1 are proportionately more likely to be selected than individuals with fitness values close to 0.	
Supported parameters	**Name**	amongTheBest
	Values	A percentage (0% to 100%), or an absolute value between 0 and the population size

	Description	Resizes the population from which the selector will select individuals. If the value received is a percentage, the visible population will be that percentage of the best individuals in the population (example 50% best individuals). If the value received is an absolute value the population visible is that number of individuals counting from the best (example the 94 best individuals).

Name	**Tournament**	
Description	Chooses randomly several individuals (as many as indicated in the parameter size) and returns the one that has the best fitness (the highest).	
Supported parameters	**Name**	amongTheBest
	Values	A percentage (0% to 100%), or an absolute value between 0 and the population size
	Description	Resizes the population from which the selector will select individuals. If the value received is a percentage, the visible population will be that percentage of the best individuals in the population (example 50% best individuals). If the value received is an absolute value the population visible is that number of individuals counting from the best (example the 94 best individuals).
	Name	size

	Values	Integer number.
	Description	Size of the tournament.

7. IMPLEMENTATION DETAILS

This section is dedicated to clarify some details about ProGen´s implementation. We will dedicate a few lines to explain how ProGen internally works in some interesting points:

7.1. THE GRAMMARS

The class Grammar.java is responsible first and foremost for building grammars capable of generating individuals that are valid according to the functions included in the function sets (the types of their children and their return types). For every function set there will be a grammar in charge of generating all the trees that use this function set. For example, the following function set:

```
prp_function_set_1: DoublePlusFc, DoubleMinusFc, DoubleMultFc, D1, D2,
AndFc, LwThanFc, GrtEqThanFc, B, NotFc, PlusFc, MinusFc, I1
```

Where the interfaces of each function are:

```
DoublePlusFc:     double$$double$$double
DoubleMinusFc:    double$$double$$double
DoubleMultFc:     double$$double$$double
D1:               double
D2:               double
AndFc:            boolean$$boolean$$boolean
LwThanFc:         boolean$$int$$int
GrtEqThanFc:      boolean$$int$$int
B:                boolean
NotFc:            boolean$$boolean
PlusFc:           int$$int$$int
MinusFc:          int$$int$$int
I1 :              int
```

The corresponding grammar is:

```
R0: A0 , Aterminal1
```

```
R1: A2 , A3 , Aterminal4 , A5
R2: A6 , Aterminal7
A0: ( DoublePlusFc R0 R0 ) , ( DoubleMinusFc R0 R0 ) , ( DoubleMultFc
   R0 R0 )
Aterminal1: D1 , D2
A2: ( AndFc R1 R1 )
A3: ( LwThanFc R2 R2 ) , ( GrtEqThanFc R2 R2 )
Aterminal4: B
A5: ( NotFc R1 )
A6: ( PlusFc R2 R2 ) , ( MinusFc R2 R2 )
Aterminal7: I1
Axiom: A0 , Aterminal1 , A2 , A3 , Aterminal4 , A5 , A6 , Aterminal7
```

Grammars in ProGen have four different kinds of rules:

- **R Rules**: There is one for each return type. In this case we have three different types: double, boolean and int, which correspond to R0, R1 and R2 respectively.
- **A Rules**: There is one for every different interface, in other words, are grouped under the same rule those functions that have equivalent interfaces.
- **Rules Aterminal**: Equivalent to the rules A but they put together only terminals.
- **Axiom**: Rule from which the grammar starts generating words (in this case words are lisp strings that encode programs).

Note: Although the grammar allows that from the axiom we can jump to any A rule or Aterminal, in the practice we prohibited the jumps from the axiom to any A rule whose return type is not the type that the tree generated by the grammar has to return (specified in the "prp_return_type_fs1" in this example) and also the jumps to Aterminal rules, so single node trees, or trees returning types different than the specified in the mentioned property are not going to be generated.

The use of grammars allows the use of types in the functions and prevents the generation of individuals with incompatibilities among the types of their nodes. This ensures that the initial population will be generated much faster, since rejecting individuals is not necessary. It is important to underline that the generation of individuals remains completely random, as we put barriers only to prevent the generation of invalid individuals.

7.2. GENERACIONAL EVOLUTION

ProGen, in the version 1.0 works in generational mode, this means that when an operator of the population takes one or more individuals to modify, resulting individuals are left in a new population. When this new population is full of individuals, it becomes the only population thus completing a generation.

For later versions, it will be offered the possibility that in addition to operating in generational mode, ProGen will be able to run experiments in "Steady state." In this mode the new individuals are placed in the old population (it is necessary to use replacement policies as “less selected individual”, “worse fitness”, etc.) so that the same individual can suffer several transformations before moving on to the next generation.

7.3. REJECTION OF INDIVIDUALS

When an operator returns one or more individuals, ProGen checks their validity depending on the configuration supplied by the user in the experiment properties file. Those individuals that are valid will be inserted into the new population and invalid individuals will be rejected. This means that an operator who returned two individuals could have 0% success (none of the individuals returned is valid), 50% or 100% of success. Once an operator has been selected, it will continue being applied until it accumulates 100% of successful individuals returned, that is, if the crossover is selected ProGen will insist on its use until it returns two individuals valid (in one or as many attempts as needed, provided that they do not exceed the value of the property "prp_max_attempts"). To make it clearer: If crossover is selected it has to return two valid individuals. If in a first attempt it returns only one valid individual, it will be applied again, this means: It will select another two individuals from the population and apply the crossover. If this time it returns **one or two** valid individuals (remember we already got one in the previous attempt), a different operator can be chosen to go on generating the offspring. If in this second

attempt, crossover returns zero valid individuals, a new attempt will be performed.

Keeping every valid individual we optimise performance.

7.4. THE METHOD SETVARIABLE()

The method setVariable is the way to specify the value of terminals. This method is located in the class UserProgram.java, inside the package userprogram. Since user programs must inherit from the class UserProgram, any user program can invoke directly the method setVariable.

Sintaxis: void setVariable (String variable, Object valor)

What the method does is to assign a value to a variable. Variables are the terminal functions defined in the function sets. That way, if I have declared in a function set the terminal M1 that whose type is Map, in any where inside my program I can:

- Create a map: Map myMapa = new Map();
- Asign it to M1: setVariable (M1, myMap);

In this moment the Map object inside the terminal M1 will become myMap.

As you can see, variables are referenced by their symbol (the symbol of the terminal). This method is specially useful when you want to set initial conditions before evaluating the individuals with the fitness function. Coming back to the regression example where we want to find a program able to solve the equation Y = X^3 + X^2 + X, we can use setVariable to fix the value of X to be 2.17 with only writing anywhere inside the code of our program: setVariable("X", 2.17). We only have to ensure that there is a terminal named X whose value is double type. Otherwise ProGen will inform about the error.

Internally, setVariable does the following:

- Searchs (by the symbol) the variable inside the variable vector_variables.
- Asigns it the new value.
- Executes the variable (terminal function) for it to return its value. With this we control that the object assigned has the proper type. Otherwise java will launch a controlled exception and the execution will end returning the error 1.

8. FREQUENTLY ASKED QUESTIONS

1. What is ProGen?

ProGen is a genetic programming engine designed to be used in a simple way for all kinds of users. It is powerful, simple, error-preventive and elegant.

2. What can I do with ProGen?

With ProGen you can solve any problem that is affordable using Genetic Programming. It is very flexible so you can quickly set up experiments that meet your needings.

3. Aren't there other programms that do the same already?

There are other Genetic Programming tools widely used, like lil-gp, BeagleGP, ECJ, etc. They are all very good in some points, but weak in some others. Some are very rigid and impose unnecessary constraints; Others have not been updated for years and have many errors and bugs. Most of them are hermetic and inaccessible, so the simple task of adding a genetic operator of your own becomes a real odyssey. And almost none of them have a documentation that is appropriated for every level, that clearly explains from simple tasks like implementing a first problem of Genetic Programming, to other more complicated like including selectors or genetic operators, change the parameters that they receive, and so on. ProGen has been created by a group of researchers at the Carlos III university of Madrid, who already had a big experience using this kind of tools. The challenge we faced from the very beginning was to inherit the good features of those tools we knew, completing them with all those we missed while eliminating all those that we found uncomfortable or inadequate. As a result we created a modular tool, very easy to use at the most basic level but designed explicitly

for the user to be able to include deep changes and that we distribute together with a documentation that seeks to facilitate to the maximun the interaction with all level users.

In addition, this tool belongs to the ProGen´s free software project, that already counts with five official developers and the support of the group EVANNAI of the department of computer science at Carlos III university of Madrid, ensuring the continuity in the short and medium term of the project, and its proper maintenance.

ProGen's main characteristics are:

- Support for ADFs, ERCs, multiple main trees, etc.
- An experimenter to program batteries of experiments.
- Strongly-typed Genetic Programming
- Library of standard functions and terminals
- Several genetic operators and selectors
- Several methods of generation for the initial populations

Examples, templates, tutorials and detailed and friendly documentation

4. What and when should I compile?

You just have to compile when you change the source code. The entire configuration of ProGen is done outside of the code, so that when your project is written (the fitness function is enough) compile it and you are ready to launch as many experiments as you want without the need for new compilations.

5. What does it mean that ProGen´s evaluation is typed?

This means you can use any type of data. Any java class can be a variable, and therefore your functions can receive and return any type of data as well. ProGen trees are generated using grammars created dinamically depending on the types used by the functions you use, and that ensure that all trees are evaluable because they are well formed.

6. Who can get more out of ProGen? Beginners of experts?

Both alike. The simplicity is an advantage for all of them, and experts have all the power and especially all the flexibility they need.