RPL2 User Guide

Project title: RPL2

Document title: RPL2 User Guide

Document identifier: EPCC-BG-PAP-RPL2-UG 0.2

Authorship: Patrick D. Surry

Document history:

Personnel	Date	Summary	Version
PDS	30 November 1993	Initial draft	0.1
PDS	24 January 1994	Draft for alpha-distribution	0.2

Approval List: *ISG Manager, Technical Reviewer, British Gas Project Manager*
Nicholas J Radcliffe, Felicity George, Jim Mallinson

Distribution List: *British Gas Project Team*
Ian Boyd, Mark Green

Contents

List of Figures

List of Tables

Known Problems with this Document

- *section 4* This section is incomplete.
- *section 3* A generic “rpl2” interface which invokes rpl2p or rpl2s as appropriate should be defined.
- *appendix A* A selection of the example plans from the distribution should be included.
- *appendix A* More useful examples based on “real” work outlined in the literature would be an asset (e.g. meta GAs). Better coverage of language features in the examples would be useful as well.

1 Overview

1.1 Genetic Algorithms

A great deal of interest in genetic algorithms has been generated in recent years, based primarily on the ground-breaking work done by John Holland [8] at the University of Michigan. The genetic algorithm is a novel approach to the general problem of adaptive search. Genetic algorithms are often applied to function optimisation, when the goal is to maximise (or minimise) the value of a given objective function that is parameterised by some set of independent variables. Typically it is extremely difficult to determine exact extremal values of the function, so usually "good" (near-optimal) sets of parameters are sought. There are many techniques for this, based on calculus (gradient methods, solution of non-linear systems), enumerative search (exhaustive search, random search), linear and quadratic programming and stochastic search (hill-climbing, simulated annealing). The genetic algorithm falls into the last of these categories. The most appealing aspect of genetic techniques is their *robustness* with respect to the range of problems they can solve. In contrast, calculus methods require "nice" functions (continuous, differentiable, etc.) and enumerative techniques are of limited usefulness for large search spaces. Genetic methods are in general *weak* search techniques, in that they use little or no problem specific information. Hybrid solutions that couple the general genetic search framework with domain specific local optimisation can often be formulated which achieve performance superior to either method used in isolation.

In a genetic algorithm, a *population* consisting of some number of *genomes* is evolved over a number of *generations*. Each genome (or *chromosome*) is an encoded representation of some point taken from the search space. In the case of function optimisation, the search space is the parameter space of the objective function. The encoding is commonly a string of symbols such as binary digits, but there is no reason for this to be so in general. Biological terms are then borrowed in this new context: *chromosomes* are strings of *genes* (symbols), each of which can take one a set of values called *alleles*. In some cases, a distinction is made between the *gene* and its position in the string or *locus*.

Each genome is assigned a *fitness* based on the value of the objective function at the point encoded by the gene. The actual value of the function is termed the *raw fitness*, which will be either maximised or minimised. It is often convenient to introduce one or more types of *scaled fitness* (based on ranking, or normalised in some way) which can be used to distinguish more meaningfully between the genomes than the raw fitness. The scaled fitness is always maximised.

The evolution of the population mirrors the biological process of sexual reproduction. Genomes are *selected* for reproduction on the basis of their fitness. This introduces a *selective pressure* as fitter solutions are afforded more opportunities to reproduce. *Children* are formed from two or more *parents* by various *recombination* operators, such as *simple crossover*. Children are also modified by random *mutations* which cause occasional small changes in the genome. The children may then replace some or all of the parents, when they are *inserted* into the population, giving rise to the next generation. This process is repeated until some *termination* condition is satisfied (perhaps a fixed number of generations, or a specified value of the objective function). The sequence and exact nature of the steps (evaluation, selection, recombination, mutation, insertion, and termination) used to solve a particular problem is termed a *reproductive plan*.

There are many variations upon this theme, including multitudes of operators, probabilistic choice of which operators to apply and so forth. Many ideas have been taken from biological systems which may or may not make a particular genetic algorithm more effective, including the ideas of *elitism*, *dominance*, *inversion*, *speciation* and so forth. For a much more detailed look at these and other ideas, see for instance [5, 4, 17].

1.2 Problem Specific Representations

The process of encoding the objective function's parameter space into a string of symbols induces a number of *equivalence classes* in the space. Each equivalence class is made up of all possible *genomes* in which some subset of the *genes* take on particular values (and the other genes have arbitrary values). The equivalence classes can be thought of as meta-genomes in which each gene can either be one of the normal symbols in the representation (e.g. binary digits) or a new wildcard symbol indicating that the given gene is arbitrary. These meta-genomes are often called *schemata*, or *similarity templates*.

It can be shown that the standard genetic algorithm discussed above explores these equivalence classes based on the *observed* average fitness of all genomes in the class represented in the population. Members of the population will tend (probabilistically) to favour above average schemata instantiated in the current population. For the search to have any opportunity to exceed the performance of a random search, a reasonable proportion of the equivalence classes must contain genomes of correlated fitness. This essentially means that each gene should correspond to some meaningful characteristic of the parameter space. Furthermore, the recombination operators should combine classes in meaningful ways, as discussed in [16].

Most discussions of genetic algorithms are based on genomes composed of fixed length strings of binary digits. These are both easy to understand and to theorise about. However, it is clear that such a representation is not useful for all problems. Consider, for example, a function of a single integer which has a period of 3. If the integer parameter is encoded as a simple binary integer, then there is clearly no possible characterisation of the infinite family of optimal genomes in terms of equivalence classes, implying that a genetic algorithm using this representation would be unsuccessful. On the other hand, if a ternary string (of base 3 digits) is used, then there would be an extremely strong correlation between schema membership and fitness.

In real world applications of genetic algorithms, it has become increasingly clear that different representations are required for different problems in order to achieve satisfactory performance [4]. As it is impossible to characterise all possible representations *a priori*, any general genetic algorithm language must allow users to define their own.

1.3 Parallelism and Structured Populations

Genetic algorithms are inherently parallel in the sense that — depending on the exact reproductive plan used — each chromosome update is largely independent of the others. There are a number of options for implementing them on parallel computers. Several techniques for this have been proposed and investigated, as discussed below.

The use of structured populations (*fine grained* or *diffusion* genetic algorithms, *island* or *stepping stone* genetic algorithms — see *figure 1*, on page 3), provides a convenient mechanism for exploiting parallelism without the bottlenecks of simpler schemes. Typically population structure is tied closely to the architecture of a particular target machine, but there is no reason, in general, why this need be so.

In the *fine-grained* or *diffusion model*, a geometry of some sort is imposed on the genomes, so that they all "live" at a given co-ordinate in some arbitrary space. Typically only a single genome can be at any one co-ordinate. The idea of a *distance* between the genomes is introduced, and used as a parameter to govern their interaction. This leads to the concept of a genome's neighbourhood or *deme* which contains all of the other genome that it can interact with. This gives a spatial diversity to the population, and allows niche populations to grow, each specialising in some area of the space. Good characteristics can slowly diffuse back and forth across the boundaries of the regions, helping to prevent the problem of premature convergence. See, for example the work in [6, 10, 13, 14, 19, 3].

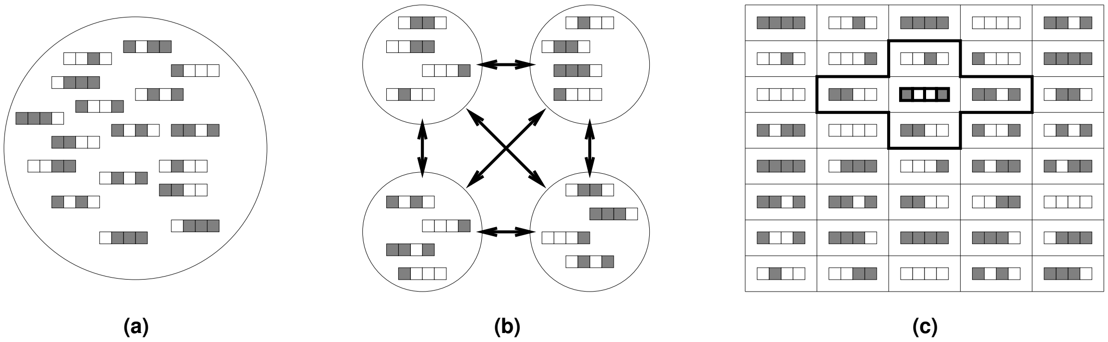

Figure 1: The most common types of population structure are illustrated: (a) an unstructured, or *panmictic* collection of genomes in which arbitrary matings are possible, (b) an island model, in which a number of panmictic populations evolve separately, interacting occasionally via migration of genomes, and (c) a fine-grained structure, where each genome has some physical location and interacts only with its neighbours (its *deme*, as shown).

In the *island model*, a much coarser spatial structure is imposed on the genomes — several panmictic populations are allowed to develop in parallel, and occasionally exchange genomes (perhaps the best in the local population) in migration steps that might occur after some number of generations. This is very loosely synchronous structure, and can work well on distributed systems with high compute to communications ratios. One of the earliest experiments with this technique is found in [15]. The related *stepping stone* model is similar except that migration only occurs to neighbouring islands.

Other types of genetic algorithms have been used which employ more than one representation at a time. These sometimes involve *meta-GAs* (for example when one genetic algorithm is used to evolve optimal parameters concurrently for another one which is solving a given problem). Work of this kind which also includes a structured population is explored in [7].

1.4 The Reproductive Plan Language

The Reproductive Plan Language (RPL) was developed in order to simplify the job of writing and experimenting with genetic algorithm codes. A prototype implementation described in [18] was developed during 1991 by a summer student, Claudio Russo and an MSc student, Graham Jones. Jones used RPL to explore large TSP problems in [9]. The language was implemented as an interpreter, for the convenience and speed it offered in program development. Typical real world genetic algorithm codes spend most of their time in the evaluation function, which would be a compiled C routine called as an instruction from RPL, thus maintaining reasonable efficiency.

The software supported arbitrary (user-defined) genome representations, and also allowed the user to add functionality to the language by means of C functions, for both representation dependent and independent operations.

The language was based on three collections of genomes called the population, cache and stack. The population represented the genomes making up the current generation, the cache was a mirror of the population that could be used as temporary space for forming the next generation, and the stack was a LIFO stack of genomes which was used by language functions to push and pop genome arguments from. This was useful in some ways but sometimes led to obscure code and difficulty in writing new operators. RPL also implemented a form of "island model" parallelism.

In a collaborative effort, EPCC and British Gas worked to standardise and extend the language RPL language to produce a new version, RPL2. In particular, its support for general structured populations across a variety of hardware platforms, including both serial and parallel machines is enhanced. The project has been properly engineered and documented, utilising the lessons learned from both the users and programmers who worked with RPL. Some additional and enhanced language features (checkpointing, statistics gathering) are also be provided. Furthermore, the flexibility and modularity of the environment, including the ability to make local customised versions of the interpreter, have been improved.

Essentially RPL2 is an interpretive language coupled with C library functions. The interpreter reads a "plan" (program) supplied by the user and converts it into an internal representation. Interactive commands (for running, setting variables, and so on) can then be used, making it easy to investigate the effects of different parameters and to do batch runs.

1.5 User versus Programmer

Those using RPL2 to solve genetic algorithm problems can be divided into two classes.

The first group ("Users") make use of existing representations and operators to build arbitrarily complex RPL2 plans. Much of the real work in efficiently solving a problem using a genetic algorithm is taken up with determining good operator combinations, population structures, parameter settings, etc. all of which can be done within the RPL2 interpreter. This guide introduces and explains RPL2 from this point of view.

The second group ("Programmers") either have problem-specific requirements for new genome representations, or want to design new or modified genetic operators. This requires C programming, but the procedure has been standardised and streamlined as much as possible, to make it feasible for non-experts. The Programmer's Guide [22] gives more details in this area.

It is likely that most will start as "Users" and will be more and more likely to become "Programmers" as they gain experience with the system.

1.6 Conventions Used in this Document

In this document, variables and function arguments are named using "Hungarian Notation". All variables begin with a lower case letter or letters indicating what the type of the variable is. Integer variables (`int`) are prefixed with an `i` or `n`, according to whether they are indices (counters) or numbers respectively (e.g. `for iPop = 1, nPop`). Floating point variables (`real`) start with `r` (`rCrossRate := 0.78`), `bool`s with `b` (`bMaxIsBest := TRUE`), and `string`s with `s` (`sOutFile := "data.out"`). The RPL2 data types `genome` and `gstack` will be prefixed with `g` and `gs` respectively (e.g. `Push(gSomeGenome, gsTheStack)`). Pointers to types have a `p` prefixed, as in a pointer to an integer, `piCount`.

Optional parts of language constructs and command line options are surrounded by square brackets ("`[ ]`"). One of several choices is indicated by a list of items in square brackets separated by vertical bars, for instance `ReduceBool[And|Or|Xor]` represents three separate functions.

Code and pseudo-code is indicated by `typewriter` font, and comments by *italics*.

Function declarations are presented as shown below. The `rettype` indicates the type of value that is returned by the function. RPL functions which return no value are referred to as procedures, and are indicated by a return type of `proc`. The list of arguments consists of the required `type` of each argument, a descriptive sample argument name (`tVar` here), and a brief comment indicating the usage of the argument. This declaration is then followed by a description of the action of the function, including the handling of special cases.

Function `FunctionName`

```
rettype  FunctionName (
      type  tVar          Comment
)
```

2 Installation

2.1 Installing RPL2

RPL2 is distributed as a compressed tar file, `rpl2.tar.Z`. A suitable directory to contain the distribution should be selected, and its named stored in the shell variable `$(RPLHOME)`.

Move the tar file into `$(RPLHOME)`, and extract the distribution using:

```
zcat rpl2.tar.Z | tar xvf -
```

The resulting directory structure is organised as shown in *figure 2*. Typing `make` in the root of the tree simply invokes sub-makes in various directories to build applicable libraries, culminating in `local/Makefile` which builds the `rpl2s` and `rpl2p` executables (serial and parallel versions of RPL2). The parallel executables require the CHIMP and PUL libraries to be installed on your system, refer to *section 2.2*

The main framework code, as well as each set of representation-independent operators and the representations themselves, exist as libraries in `lib/$(ARCH)`. The framework is linked together with a collection of the libraries as specified in `local/Makefile`.

Central executable versions of `rpl2s` and `rpl2p` exists in `$RPLHOME/bin`, as does the RPL2 preprocessor, `rpp` (used to prepare user C code for use within the RPL2 environment).

The include directory, `$(RPLHOME)/include`, contains header files which present the interface to RPL2 (data structures and external functions), and preprocessor macros for the `rpp` preprocessor. The files are called `rpl2.h` and `rpp.h` respectively. Typically both will be included in any user written C code. Other design dependent include files shared between the framework and pre-processor may be located in this directory.

The `doc` directory contains the most recent copies of the project documentation, including the functional specification, the design document and the programmer and user guides. Each directory will additionally contain a `README` file (not shown in *figure 2*) which explains the contents of the directory.

Object code produced from any `.c` files will reside in a subdirectory `obj/$(ARCH)` below the location of the C code, where `$(ARCH)` expands to the name of the local machine architecture.

2.2 Installing the CHIMP and PUL Communications Libraries

The CHIMP communications interface is available by anonymous ftp from: `anonymous@ftp.epcc.ed.ac.uk` in directory `pub/chimp/release`. A `README` file describes the procedure for retrieving and installing the software. This code is required to run the parallel version of RPL2.

The PUL libraries are available on request by contacting `rpl2-support@epcc.ed.ac.uk`. These are required for building the parallel executable for RPL2 (e.g. when extending or adding representations and operators).

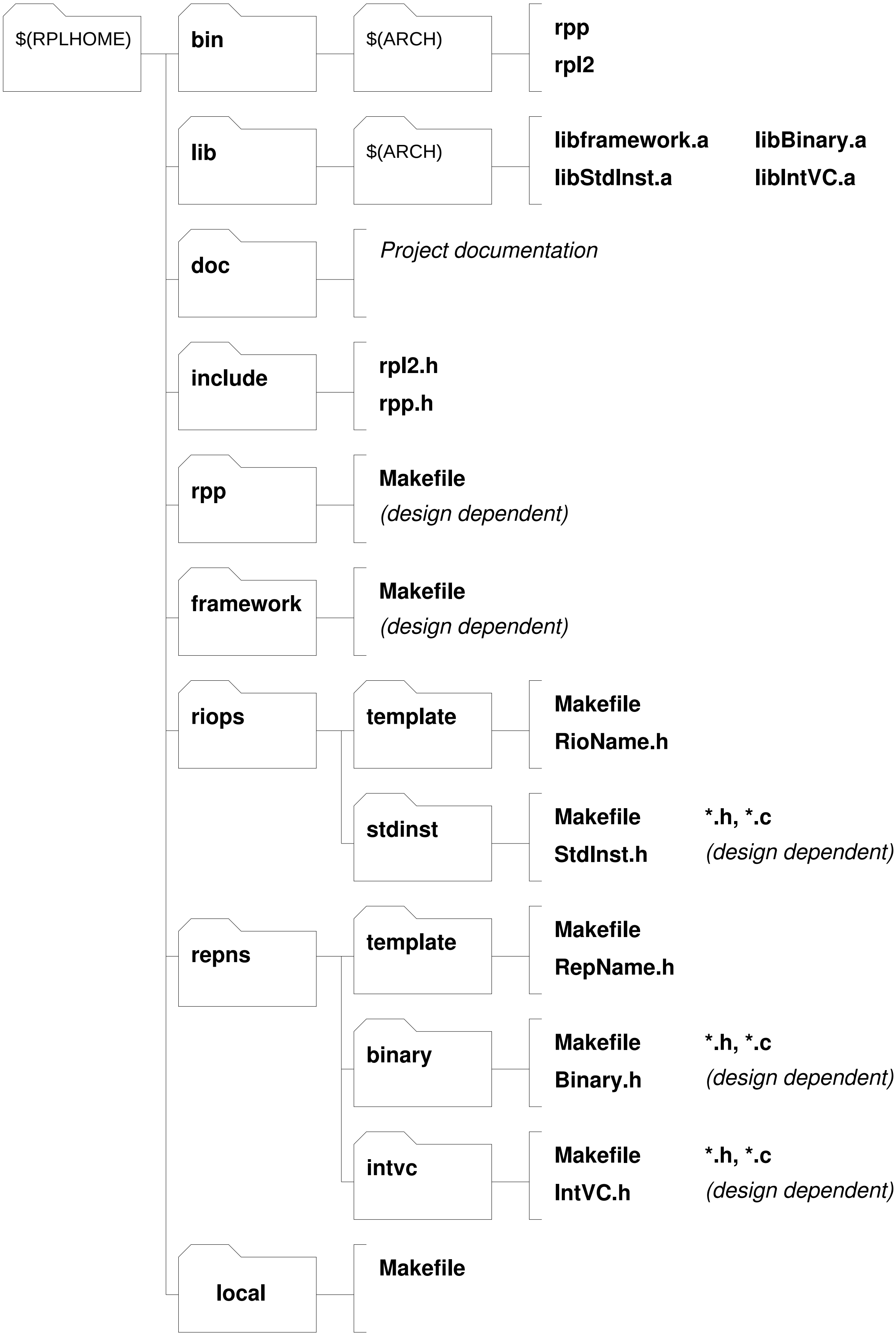

Figure 2: RPL2 directory structure

3 Using RPL2

3.1 Serial and Parallel Versions

RPL2 is provided as an executable command interpreter, which reads user input from `stdin` and writes output to `stdout`. The interpreter parses and executes interactive commands until encountering the `plan` keyword (see *section 5.5*), at which point it "compiles" a syntax tree for the specified plan, until the `endplan` token is reached. This plan may be executed (using the `go` command) to generate various output files. Values of variables can be changed and the code may then be re-run. The framework provides for batch use by means of file redirection.

The framework is modular and extensible, as discussed in the *RPL2 Programmer's Guide*, [22]. Addition of functions requires re-linking of the run-time system, and the addition of entirely new representations requires re-compiling (in order to provide the framework with an entry-point to the library).

Two distinct versions of RPL2 exist: a serial executable called `rpl2s` and a parallel executable called `rpl2p`. The serial executable can be executed directly from the command line, while the parallel executable is invoked via CHIMP, using a script such as those found in `$RPLHOME/examples/*.chp`.

3.2 Command Line Arguments

To use RPL2, simply type `rpl2` at the command line. A file from which to take input can be specified. If none is specified, or when EOF is reached, standard input is used. Various flags can be specified on the command line to control the operation of the command. These are documented below:

```
rpl2 [-o outfile] [-e errfile] [-l logfile] [-b] [-q] [ [-r|-s] chkptfile | -h |
-v | rplfile ]
```

`-o outfile` Copy output (`stdout`) to `outfile`.

`-e errfile` Cause errors (`stderr`) to be logged to `errfile`.

`-l logfile` Log input (`stdin`, but not `rplfiles` or `execed` files to `logfile`.

`-b` Batch mode. Suppress output to `stdout` and `stderr`.

`-q` If an `rplfile` is specified, quit the interpreter after exhausting the file (do not continue to read from `stdin`). Equivalent to a `quit` command appended to the `rplfile`.

`-r chkptfile` Resume execution from a checkpoint file called `chkptfile`. Equivalent to `-s chkptfile` and then `resume` in the interpreter.

`-s chkptfile` Load the state from a checkpoint file called `chkptfile`, but do not execute it.

`-h` Print out a help message describing the usage of RPL2, then exit.

`-v` Print out the version number of the RPL2 kernel and the libraries linked with it, then exit.

`rplfile` Read input from this file until it is empty, then read from `stdin`.

4 Language Tutorial

See the example plans in `$RPLHOME/*.rpl`.

5 Language Reference

5.1 Lexical Conventions

An RPL2 program consists of free format text. This text is broken down by a lexical analyser into tokens, which can be separated by whitespace (blanks, tabs, and newlines). Ambiguous sequences of the characters in the file (which could be interpreted as tokens in more than one way) are resolved by accepting the longest possible sequence of characters that constitutes a valid token. Strictly speaking, this means that not all tokens need to be separated by whitespace, but it is good practice, and highly recommended. Multiple whitespace is treated in the same way as single whitespace.

Comments are introduced by the "%" character and are terminated by a newline (the only distinction between types of whitespace). They are completely ignored by the lexical analyser. The "%" character does not introduce a comment if it is inside double quotes (as part of a string literal).

After removal of comments, the input file is broken down into tokens, which fall into one of the following classes:

- identifiers
- keywords (reserved words)
- constants
- string literals
- operators
- separators

An identifier is the name of a function or variable. It must start with a letter which may be followed by any number of letters, numbers or underscores. (i.e. it matches the regular expression `[a-zA-Z][a-zA-Z0-9_]*`). Identifiers are case sensitive (so that `g1` is distinct from `G1`).

The following list of words is reserved and may not be used as identifiers:

- `go quit help exec state resume`
- `not and or xor`
- `plan endplan`
- `use structure deme`
- `if else endif`
- `while endwhile`
- `for endfor to step`
- `structfor endstructfor`
- `forall endforall in`
- `int real bool string genome gstack`

A constant is a numeric or boolean value which may be used in function calls, assignments, loop constructs and so on. A string literal is a string constant surrounded by double quotes. See *section 5.2* for more detailed description of these tokens.

There are several tokens in RPL2 which are classed as operators. "`:=`" is the assignment operator, and "`&`" is used to indicate a variable passed by reference. The logical operators `not`, `and`, `or`, `xor`, ">", ">=", "<", "<=", "<>" and "=" are also in this category.

Various punctuation marks are used as separators in RPL2, including parentheses "( )" for grouping function arguments, brackets "[ ]" used with structured populations, commas "," for separating function arguments, and semi-colons ";" for terminating statements.

All statements in RPL2 are terminated by a semi-colon except for the multi-part grouping constructs `if else endif`, `while endwhile`, `for endfor`, `structfor endstructfor`, `forall endforall`, and `plan endplan`.

A distinction is made between interactive and non-interactive commands. Interactive commands (see *section 5.4*) are executed immediately upon being processed, while non-interactive commands (see *section 5.5*) are simply recognised and added to a parse tree for later execution. All text entered inside a `plan endplan` construct is treated as non-interactive commands. Other text is treated as interactive. During the processing of interactive commands, the newline character is equivalent to the semi-colon in that it serves to terminate statements.

5.2 Data Types

RPL2 supports six data types, namely: `int, real, bool, string, genome, gstack`. The first four of these are similar to C data types. They can be assigned (using the `:=` construct) and passed to and returned from functions. Examples of allowable values for the types are given in *table 1*. Strings may contain double quotes by preceding them with a backslash character as indicated in the table. Note that the definition of the `real` type is a superset of `int`. Expressions (such as function arguments or looping parameters) involving mixtures of `reals` and `ints` (either constants or variables) are legal and will be resolved through promotion to `real`, and possible truncation to `int` for assignment. All of these basic types are passed by value, so cannot be changed by functions.

Type	Regular Expression	Examples
`int`	[–+]?[0-9]+	1, 2, –6, 0
`real`	[–+]?(([0-9]+(\.[0-9]*)?) | (\.[0-9]+)) ([eE][–+]?[0-9]+)?	1.3, .3, –.3, –2, –3.7e–6
`bool`	TRUE | FALSE	TRUE, FALSE
`string`		"Anything at all inside \"s!"

Table 1: Definitions and examples of base data types

The additional data types `genome` and `gstack` are specific to RPL2. A `genome` is a data structure which contains an instance of the genetic representation used to parameterise the search space. A generic part of the data structure holds information which applies to all `genomes` such as fitness values. Part of the `genome` will be representation-dependant information (such as the string of bits used in a binary representation). A `genome` variable can only be assigned as a result of function calls, or by copying from existing `genomes`. (Note that I/O to and from `genome` variables takes place via function calls, and is not an explicit part of the language.)

The `gstack` data structure is essentially a variable length array of `genomes`. For convenience, it can be treated as a LIFO stack, but direct random access is also supported (see the operators described in *section 6.1*).

Before use, all variables must be declared in the declaration section of a reproductive plan (see *section 5.5*). This is accomplished by stating a type, an optional structure quantifier and a list of variable names, separated by commas. The structure quantifier is a comma-separated list of dimensional quantifiers ("*" or "-") surrounded by brackets ("[ ]"). The number of entries in the list must be equal to the number of dimensions of the population structure that has been declared. This concept is explained in *section 5.3*. The example below illustrates the declaration of several structured arrays[1].

```
genome  [*,*]   gPopulation, gMate;
gstack  [*,*]   gsStack;
int     [*,-]   iCount;
```

Variables are not required to be assigned values inside the `plan`. The act of declaring each variable sets up an appropriate uninitialised storage space, which may be assigned by means of interactive commands outside the `plan` before it is run. However, by run time all variables must be defined before they can be used. Attempting to make use of an uninitialised variable will cause a run time error.

5.3 Structured Population Support

The "fine-grained" structured population model (see *figure 3*) is supported in RPL2 through the `fine` axes in the population structure. Neighbourhoods (demes) are defined using the `deme` keyword. The function `DemeCollect` allows the programmer to treat the genomes in each deme as a `gstack`, avoiding the need for explicit "parallel" constructs in the language.

The "island" and "stepping stone" models are (see *figure 3*) supported by `island` axes in the population structure. Migration is supported by the functions `MigrateRandom` and `MigrateLocal`.

A population structure is defined in the pre-instructions of a plan. It takes the form of the keyword `structure` followed by a list of axis descriptors in brackets "[ ]", separated by commas. An optional `deme` statement may follow, before the terminating semi-colon. Each axis descriptor consists of three parts: a dimension for the axis (a positive integer), an axis boundary condition (":" or "@", corresponding to unconnected ends (static) and connected ends (cyclic) respectively), and a type for the axis (`fine` or `island`). This population structure applies for the whole plan, and is referred to in variable declarations and loops over part (or all) of the population structure (`structloop`).

If the population structure includes any `fine` axes, then a distance metric should be specified (which will define the "deme" of each genome). This is accomplished by adding the keyword `deme` and the name of a distance function, with its argument list, after the list of axis descriptors. Some example declarations follow. The first describes a cylindrical structure of 500 islands, with 50 along the length of the cylinder and 10 around its circumference. The second defines the surface of a torus with 10000 genomes on it, with the neighbourhood of each site composed of all sites within 3 grid points of it, measured using Euclidean distance. Finally, the third defines a planar array of nine islands, each of which is composed of a toroidal surface with 100 genomes on it, with distance on the torii measured using the Taxicab metric.

```
structure [50:island, 10@island];
structure [100@fine, 100@fine] deme Euclidean(3);
structure [3:island, 3:island, 10@fine, 10@fine] deme Taxicab(2);
```

5.4 Interactive Commands

Interactive commands (entered outside a `plan`) are executed immediately upon processing. Several operations can be performed both inside and outside a `plan`, while others may only be used in one mode.

[1] Note that a structured `gstack` may contain different numbers of elements at different locations in the population structure, so that the concept of the depth of a structured `gstack` is not well-defined.

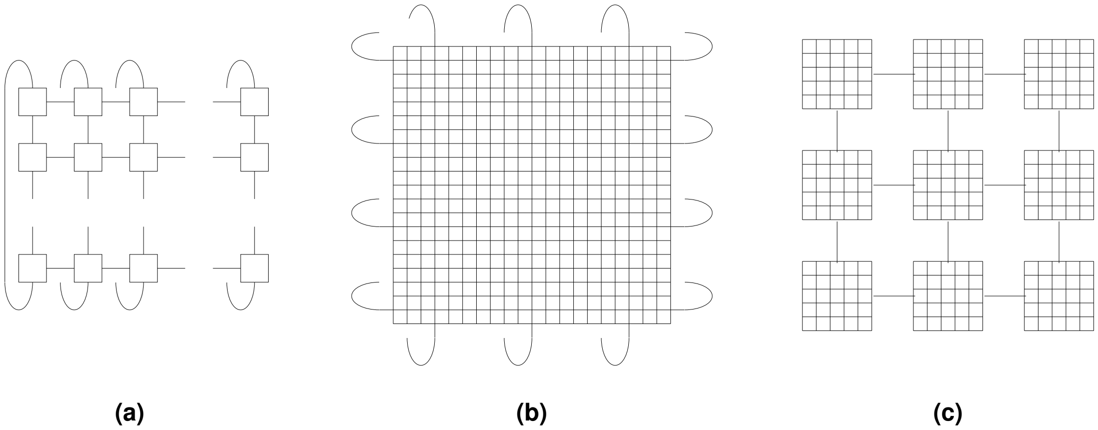

Figure 3: Several population structures are illustrated, including (a) a two-dimensional array of islands (with cyclic boundary conditions in the vertical axis), (b) a two-dimensional fine-grained structure (with toroidal boundaries), and (c) a four dimensional structure made up of an array of islands, each of which is itself a fine-grained array (with non-cyclic boundaries).

5.4.1 `go`

The command `go(PlanName)` executes the named plan. The argument is optional — `go` will execute the currently defined plan. (RPL2 supports only a single plan at present).

5.4.2 `quit`

The `quit` instruction causes the RPL2 run-time system to terminate.

5.4.3 `help`

The `help([libname|funcname|libname::funcname])` is used to show a usage string and user supplied description of a function. If `libname` is not specified, all functions with the given name are listed. If `funcname` is omitted, all functions in the specified library are listed. If both arguments are omitted, all available functions are listed.

5.4.4 `exec`

The `exec("filename")` instruction causes input to be read from the specified file as if it had been typed. Upon reaching the end of the file, input is again expected from the source of the `exec` statement.

5.4.5 `state`

The command `state("chkptfile")` loads the system state defined in the specified checkpoint file. This command is identical to the `-s` command line flag described in *section 3.2*.

5.4.6 `resume`

The `resume` command resumes execution of the current plan in its current state. If the `state` command has been used, this state will be after a `Checkpoint` call in the `plan`. Otherwise, `resume` is equivalent to `go`. The `state` and `resume` instructions are separated to allow the user to examine and change plan variables prior to continuing.

5.5 Non-interactive Commands

5.5.1 `plan endplan`

```
plan (name)
        pre-instructions;
        variable-declarations;
        instructions;
endplan
```

A number of non-interactive commands can be used to build up a "genetic algorithm" by means of the `plan/endplan` construct. The plan may optionally be given a name. Execution is deferred until the `go` command is issued (see *section 5.4*).

The body of the plan consists of pre-instructions, which include the `use` declaration for libraries, and the `structure/deme` definition of the population structure. Next come variable declarations (see *section 5.2*), and finally the body of the plan is made up of the executable instructions and control flow constructs that are carried out when the plan is executed.

5.5.2 `use`

5.5.3 `structure deme`

5.5.4 `for step by endfor`

The syntax of the `for` loop is illustrated below. The loop variable `loopvar` is assigned the successive values `start`, `start+stepval`, `start+stepval+stepval`, etc. until its value is greater (less, if `stepval` is negative) than `stop`. If the `step` argument is omitted, its value is assumed to be 1. All of the instructions in the loop are executed for each value of `loopvar` within the loop range. Instructions in the loop are not necessarily executed even once (if `stop - start` and `stepval` are of different signs). The value of `loopvar` after exiting the loop will be outside the loop range (e.g. after the loop `for i := 1 to 3`, `i` will be equal to 4). The values `start`, `stop`, and `stepval` can be constants or variables, and may safely be changed during the loop (the comparison takes place *before* the instructions, and the increment takes place *after* them). A zero value of `stepval` will cause an infinite loop. Normal rules for promotion/truncation apply (see *section 5.2*) — note that the loop variable will be incremented and rounded at each step, so that a loop over an integer variable with a step size of 0.3 would loop forever.

```
for loopvar := start to stop [ step stepval ]
        instructions;
endfor
```

5.5.5 `if else endif`

An `if` construct is formed in the following way. The `else` and corresponding statements are optional. The `bool` inside parentheses may be either a constant or variable.

```
if (bool)
        true-statements;
else
        false-statements;
endif
```

5.5.6 `while endwhile`

A `while` construct is illustrated below. The boolean variable is checked and if it is `TRUE` then the statements are executed. This procedure is repeated until the boolean is `FALSE` at which point execution proceeds to the statement after the `endwhile`.

```
while (bool)
        true-statements;
endwhile
```

5.5.7 `structfor endstructfor`

A `structfor` loop includes a list of axis quantifiers which corresponds to the declaration of the population structure. This list is made up of the elements "`*`" and "`-`" which indicate whether or not (respectively) each corresponding, unquantified dimension of the population structure is to be looped over (quantified) inside this `structfor`. Thus the first `structfor` loop in a plan will contain a quantifier list with a number of elements equal to the number of dimensions of the population structure. The next `structfor` loop's list will have a number of elements equal to the number of "`-`" (unquantified) axes from the first loop, and so forth.

The `structfor` construct thus permits the programmer to write SIMD-like parallel code that acts on all elements of a structured population concurrently. Every statement of an RPL2 plan should be considered to have some inherent parallelism, quantified by any enclosing `structfor` loops. All variables used in a statement must be declared with a quantifier that matches the currently nested `structfor` loops, with two exceptions. It is legal to substitute a scalar (non-quantified), base, by-value type (see *section 5.2*) in place of a quantified argument of the same type. This essentially allows the user to pass the same scalar constants to all parallel instances of a given function. Also, some of the explicitly parallel structured population operators described in *section 6.1* take arguments of strictly higher dimension than the current `structfor` loop nesting.

An example of such nested loops is given below. The three successive calls to `RandomGenome` create 1, 9 and 900 `genomes` respectively. Note the use of the scalar argument `rParameter` inside the loops.

```
structure [3:island, 3:island, 10@fine, 10@fine] deme Taxicab(2);

genome gSingle;
genome [*,*,-,-] gTwoD;
genome [*,*,*,*] gFourD;
real   rParameter;

rParameter := 0.3;

gSingle := RandomGenome(rParameter);

structfor [*,*,-,-]
        gTwoD := RandomGenome(rParameter);
        structfor [*,*]
                gFourD := RandomGenome(rParameter);
        endstructfor
endstructfor
```

5.5.8 `forall endforall`

Additional parallelism is provided by means of the `forall/endforall` construct. This is primarily to support parallel execution of *panmictic* (unstructured) by means of a task farm, but may also be used with structured populations. A common use of the construct would be to evaluate all of the genomes in a given stack. The construct takes the following form:

```
forall g in gs
       instructions;
endforall
```

Here, `g` is the "loop-variable" of `genome` type, which will take on the value of each `genome` is the specified `gstack` (`gs` in this case). Iterations of the loop may be farmed (executed in arbitrary order), although the individual statements in each iteration will be executed sequentially. For this reason, the instructions inside the loop should not refer to any variable by reference, nor any `gstack` or `genome` variable save the specified loop variable. Assignments should also be avoided.

5.6 Functions

Most of the statements in an RPL2 program are made up of calls to externally-defined functions which are provided as libraries, and may be written by the programmer — see the *RPL2 Programmer's Guide*, [22].

5.6.1 Using a Library

In order to call functions in a given library, the name of the library must be listed in the program's `use` declaration. The declaration consists of a list of library names with parameters (if necessary), separated by commas. The example below would permit the use of functions in the `StdInst` and `Binary` libraries. Note the required parameter for the second library.

```
use StdInst, Binary(64);
```

5.6.2 Calling Conventions

Functions are called by giving their name and arguments. The arguments are surrounded by parentheses. The statement is terminated with a semi-colon. The function call may be preceded by a "disambiguator" as discussed below. Arguments are separated by commas, and may be constants, string literals or variables. Some functions may require arguments to be passed by reference. In this case the argument must be a variable of the appropriate type, and must be preceded by an ampersand "&".

Functions may or may not return a value. Functions which do return one of the six data types must be assigned (":=") to a variable of the appropriate type. Those which do not return a value (referred to as procedures) are simply stated in the program with their argument list, and terminated by a semi-colon. Some example calls are:

```
gChild := SelectRandom(gsParents);
Push(gChild, gsParents);
PlusEquals(&iA, iB);   % do the C op iA += iB
```

5.6.3 Overloading and Disambiguators

RPL2 supports a form of function "overloading" whereby different libraries (and/or representations) may have functions with the same name[2]. This allows for more readable programs (if the feature is not abused) since, for instance, functions which perform essentially the same task

[2] True function overloading essentially allows functions to branch (perform different operations) based on the *types* of their arguments and required return values.

on different genome representations can have the same name. If an RPL2 program uses only a single representation or a set of libraries in which there are no name clashes, then the function is simple called using its name, as shown below:

```
use Binary(64);
genome g1;

g1 := RandomGenome();
```

If the program uses multiple representations which share some function names, then the base name of the function should be disambiguated by prefixing the name of the representation followed by "::". RPL2 will generate an error if a function call is ambiguous (i.e. it appears in two or more used libraries and is not disambiguated).

```
use Binary(64), Set(10);
genome g1;

g1 := Binary::RandomGenome(); % qualified as binary, default parms
```

Programs which use multiple instances of a parameterised representation should disambiguate function calls with the name of the representation and one of the parameter lists declared in the use line, as shown. Disambiguation using a parameter list causes RPL2 to notify the library of the specified value of the parameters — it does not actually distinguish between two versions of the same function. If a parameter list is not specified, the current values of the parameters are preserved (the library is not notified of any change).

```
use Binary(64), Binary(128);
genome g1;

g1 := Binary(64)::RandomGenome(); % qualified as binary with parms
```

5.7 Expressions

RPL2 supports standard nested algebraic expressions, where data of varying types are combined by operators into results. The operators which are supported are described below. Functions (see *section 5.6*) which return a result may also be used in expressions.

5.7.1 Logical Operators

The following operators combine boolean expressions to produce a boolean result.

not The result is true if the right hand expression is false. This is a unary operator.

and The result is true only if both the left and right hand expressions are true.

or The result is true if either of the left and right hand expressions are true.

xor The result is true only if exactly one of the left and right hand expressions are true.

5.7.2 Mathematical Operators

The following mathematical operators combine two numeric expressions into a resulting value. The type of the result will be `int` if and only if both the left and right hand expressions are of type `int`. Otherwise the result will be `real`.

+ The result is the arithmetic sum of the left and right hand expressions.

- (binary) The result is the arithmetic difference of the left and right hand expressions.

- (unary) The result is the negative of the right hand expression. (There is no left hand expression.)

`*` The result is the produce of the left and right hand expressions.

`/` The result is the quotient of the left and right hand expressions. Integer division is carried out in floating point and the fractional part discarded.

5.7.3 Comparison Operators

The following comparison operators yeild a boolean result from two numerical expressions.

`>=` True if the left hand expression is numerically greater than or equal to the right hand expression.

`>` True if the left hand expression is numerically greater than the right hand expression.

`<=` True if the left hand expression is numerically less than or equal to the right hand expression.

`<` True if the left hand expression is numerically less than the right hand expression.

`<>` True if the left hand expression is numerically not equal to the right hand expression.

`=` True if the left hand expression is numerically equal to the right hand expression.

5.7.4 Miscellaneous Operators

`:=` Assigns the quantity on the right hand side to a variable of compatible type on the left hand side. The left hand side must be a variable (it cannot be a compound expression). An assignment construct is *not* an expression (so that the C-style `a := b := c;` is not legal). In the case of genomes, this only copies a *reference* to the underlying genome, not the genome itself.

() Parentheses are used for grouping in the ordinary way. The value of a bracketed expression is simply the value of the expression inside the brackets.

6 Supplied Libraries

6.1 The Standard Instruction Library

Selection Operators

The following instructions are provided to perform common selection operations. All choose one or more `genomes` from a given `gstack`. The operators may be grouped into three classes: first are operators that have no requirements on the `gstack`, second are those that act based on the raw fitness of the input `genomes` (so they must be evaluated), and third are those that act based on scaled fitness (so the stack must have been using a scaling operator). None of the instructions actually modify the input stack; they merely return one or more references to `genomes` contained therein.

Function `SelectRandom`

`genome` SelectRandom (

`gstack`	`gs`	*Set of genomes*

)

Returns a uniformly chosen random member of `gs` if it is non-empty; otherwise the `NULL genome` is returned.

Function `SelectNth`

`genome` SelectNth (

`gstack`	`gs,`	*Set of genomes*
`int`	`nIndex`	*Index of the genome to pick*

)

Returns the genome which is in position `nIndex` in `gs`, where the bottom member of the stack (last to be popped) has index 1. If `nIndex` is out of range (non-positive or greater than the stack depth), the `NULL genome` is returned.

Function `SelectRawBest`

`genome` SelectRawBest (

`gstack`	`gs,`	*Evaluated set of genomes*
`bool`	`bMaxIsBest`	*True if maximal fitness is best*

)

Returns the `genome` in `gs` that has the best raw fitness (maximal or minimal according to `bMaxIsBest`). If `gs` is empty or contains unevaluated genomes, the `NULL genome` is returned.

Function `SelectRawTournament`

`genome` SelectRawTournament (

`gstack`	`gs,`	*Evaluated set of genomes*
`bool`	`bMaxIsBest,`	*True if maximal raw fitness is best*
`int`	`nTournSize,`	*Number of genomes in the tournament*
`real`	`rProbOfBest,`	*Probability of selecting best genome*
`bool`	`bWithReplacement`	*True if tournament can have duplicates*

)

Selects a `genome` from the given `gstack` using a k-tournament procedure based on raw fitness. A tournament pool of `nTournSize` members are chosen uniformly at random from `gs` (with or without replacement according to the value of `bWithReplacement`). The `genome` with the *best* raw fitness (maximal or minimal according to `bMaxIsBest`) in the tournament is then returned with probability `rProbOfBest`. In the case that the best genome is not returned, the second best is returned with probability `rProbOfBest`, and so on. If only the single (worst) genome remains, it is returned unconditionally. If `nTournSize` is less than one or greater than the depth of the stack, or if the stack contains unevaluated genomes, the `NULL genome` is returned.

Function `SelectScaledBest`

`genome` SelectScaledBest (

`gstack`	`gs`	*Evaluated, scaled set of genomes*

)

Returns the best `genome` in `gs` based on scaled fitness. If `gs` contains members without a valid scaled fitness, the `NULL genome` is returned.

Function `SelectScaledRoulette`

`genome` SelectScaledRoulette (

`gstack`	`gs`	*Evaluated, scaled set of genomes*

)

Make a roulette selection based on the scaled fitness values of the `genomes` in `gs`. Each `genome` is assigned a sector of a conceptual roulette wheel whose size is proportional to its scaled fitness. A random number is used to determine a position on the wheel, and the selected `genome` is returned. If `gs` contains members without a valid scaled fitness, the `NULL genome` is returned.

Function `SelectScaledSUS`

`proc` SelectScaledSUS (

`gstack`	`gs,`	*Evaluated, scaled set of genomes*
`int`	`nChildren,`	*Number of children to select*
`gstack`	`gsOut`	*Stack to add children to*

)

Use Stochastic Universal Sampling to choose `nChildren` from `gs` which will be pushed onto the top of `gsOut`. Nothing is returned by the procedure. The selection technique is similar to roulette selection except that `nChildren` `genomes` are picked at once, spaced equally around the conceptual wheel. This ensures a low-variance, representative sample of the initial set of `genomes` in `gs`. If `nChildren` is out of range (less than one or greater than the depth of the stack), or if `gs` contains members without a valid scaled fitness, no change is made to `gsOut`.

Replacement Operators

The instructions below provide common replacement operations. All replace a single member of a given `gstack`. The `genome` to replace is chosen in a manner analogous to the selection operators, and may require that the `gstack` be evaluated or scaled. The replacement of a genome removes the reference to it from the stack (although it may not actually be destroyed if it is referenced somewhere else). The `genome` used as the replacement is not altered.

Function `ReplaceRandom`

`proc` ReplaceRandom (

`gstack`	`gs,`	*Non-empty set of genomes*
`genome`	`gReplace,`	*Genome to use as replacement*
`bool`	`bElitist`	*True if the best genome should not be replaced*

)

Replaces a uniformly chosen random member of `gs` with `gReplace`. If `gs` is empty, no action is performed. If `bElitist` is `TRUE`, the member of the population with the best raw fitness will never be replaced. The value of this flag is ignored if the population is unevaluated.

Function `ReplaceNth`

`proc` ReplaceNth (

`gstack`	`gs,`	*Set of genomes*
`genome`	`gReplace,`	*Genome to use as replacement*
`int`	`nIndex`	*Index of the genome to replace*

)

Replaces the genome which is in position `nIndex` in `gs` with `gReplace`. The bottom member of the stack (last to be popped) has index 1. If `nIndex` is out of range (non-positive or greater than the stack depth), no action is performed.

Function `ReplaceRawWorst`

`proc` ReplaceRawWorst (

`gstack`	`gs,`	*Evaluated, non-empty set of genomes*
`genome`	`gReplace,`	*Genome to use as replacement*
`bool`	`bMaxIsBest`	*True if maximal fitness is best*

)

Replaces the member of `gs` that has the worst raw fitness (maximal or minimal according to `bMaxIsBest`) with `gReplace`. If `gs` is empty or contains unevaluated genomes, no action is performed.

Function `ReplaceRawTournament`

`proc` ReplaceRawTournament (

`gstack`	`gs,`	*Evaluated set of genomes*
`genome`	`gReplace,`	*Genome to use as replacement*
`bool`	`bMaxIsBest,`	*True if maximal raw fitness is best*
`int`	`nTournSize,`	*Number of genomes in the tournament*
`real`	`rProbOfWorst,`	*Probability of replacing worst genome*
`bool`	`bWithReplacement,`	*True if tournament can have duplicates*
`bool`	`bElitist`	*True if best genome should not be replaced*

)

Replaces a `genome` from the given `gstack` using a k-tournament procedure based on raw fitness. A tournament pool of `nTournSize` members are chosen uniformly at random from `gs` (with or without replacement according to the value of `bWithReplacement`). The `genome` with the *worst* raw fitness (maximal or minimal according to `bMaxIsBest`) in the tournament is replaced with `gReplace`, with probability `rProbOfBest`. In the case that the worst genome is not replaced, the second best is replaced with probability `rProbOfBest`, and so on. If only the single (best) genome remains, it is replaced unconditionally. If `bElitist` is `TRUE`, the member of the population with the best raw fitness will never be chosen as part of the tournament pool, and hence will never be replaced. If `nTournSize` is less than one or greater than the depth of the stack, or if the stack contains unevaluated genomes, no action is performed.

Function `ReplaceScaledWorst`

`proc` ReplaceScaledWorst (

`gstack`	`gs,`	*Evaluated, scaled set of genomes*
`genome`	`gReplace`	*Genome to use as replacement*

)

Replaces the worst `genome` in `gs` based on scaled fitness. If `gs` is empty or contains members without a valid scaled fitness, no action is performed.

Stack Manipulation Operators

Function `Pop`

`genome` Pop (

`gstack`	`gs`	*Set of genomes*

)

Returns the top member of `gs`, which is removed from the `gstack`. If `gs` is empty, the `NULL` `genome` is returned.

Function `Push`

`proc` Push (

`genome`	`g,`	*Any genome*
`gstack`	`gs`	*Set of genomes*

)

Adds the `genome` `g` to the top of `gs`.

Function `Swap`

```
proc  Swap (
      gstack  gsA,          Set of genomes
      gstack  gsB           Set of genomes
)
```

Swaps the contents of `gsA` and `gsB`. Essentially, the handles of the two stacks are exchanged, so that what was `gsA` is now identical to what is now `gsB`, and vice versa. This allows a population/cache model to be implemented straightforwardly.

Function `Empty`

```
proc  Empty (
      gstack  gs            Set of genomes
)
```

Remove all `genomes` from `gs`.

Function `IsEmpty`

```
bool  IsEmpty (
      gstack  gs            Set of genomes
)
```

Returns `TRUE` if `gs` contains no elements, `FALSE` otherwise.

Function `RankRaw`

```
proc  RankRaw (
      gstack  gs,           Evaluated set of genomes
      bool    bMaxIsBest    True if maximal fitness is best
)
```

Sort the given `gstack` in order of raw fitness where best is either maximal or minimal depending on the value of `bMaxIsBest`. The best `genome` will end up at the top of `gs` (so would be first to be popped). If any member of `gstack` is unevaluated, no action is performed.

Function `RankScaled`

```
proc  RankScaled (
      gstack  gs            Evaluated, scaled set of genomes
)
```

Sort the given `gstack` in order of scaled fitness. The `genome` with highest scaled fitness will end up at the top of `gs` (so would be first to be popped). If any element of `gstack` does not have a valid scaled fitness, no action will be performed.

Function `ScaleRaw`

```
proc ScaleRaw (
      gstack  gs                 Evaluated set of genomes
)
```

This scaling function calculates a scaled fitness value for all members of the given stack `gs`. The scaled fitness for each genome is simply set to the raw fitness value. If any member of `gs` is not evaluated, no action is performed.

Function `ScaleRanked`

```
proc ScaleRanked (
      gstack  gs,                Evaluated set of genomes
      bool    bMaxIsBest,        True if maximal fitness is best
      real    rMinScaled,        Minimum scaled fitness value
      real    rMaxScaled         Maximum scaled fitness value
)
```

This scaling function calculates a scaled fitness value for all members of the given stack `gs`. The scaled fitness of each genome is set to a value between `rMinScaled` and `rMaxScaled` based simply on the rank of the genome relative to the other members of the stack. For example, a genome that has fourth best (highest if `bMaxIsBest` is `TRUE`) raw fitness in a stack of eleven genomes would be assigned a scaled fitness of `0.3 * rMinScaled + 0.7 * rMaxScaled`. `gs` need not be ranked, and its order will not be changed by this procedure. If any elements are not evaluated, or if `rMinScaled` > `rMaxScaled`, then no action will be performed.

Function `ScaleLinear`

```
proc ScaleLinear (
      gstack  gs,                Evaluated set of genomes
      bool    bMaxIsBest,        True if maximal fitness is best
      real    rScaleFactor,      Positive multiple of average fitness for best
      bool    bCutoffNotPivot    True to cutoff negative fitnesses at zero
)
```

This scaling function calculates a scaled fitness value for all members of the given stack `gs`. The linear scaling technique calculates a scaled fitness of the form $f_{scaled} = af_{raw} + b$, with the following properties: $\bar{f}_{scaled} = \bar{f}_{raw}$, and $\max f_{scaled} = C\bar{f}_{scaled}$, where C is equal to the `rScaleFactor` argument. If this scaling would generate negative values for the scaled fitness, two possible actions can be taken. If `bCutoffNotPivot` is `TRUE`, then negative values are arbitrarily set to zero. If it is `FALSE`, then a new, smaller value of C will be calculated to ensure all scaled fitnesses are non-negative. The `bMaxIsBest` flag determines whether high scaled fitness values correspond to high or low raw fitness values. If any member of `gs` is not evaluated, or if `rScaleFactor` is negative, no action is performed.

Function `ScaleSigma`

```
proc ScaleSigma (
```

`gstack`	`gs,`	*Evaluated set of genomes*
`bool`	`bMaxIsBest,`	*True if maximal fitness is best*
`real`	`rNumSigma,`	*Number of standard deviations to subtract*
`real`	`rScaleFactor`	*Scale factor for linear scaling phase*

)

This scaling function calculates a scaled fitness value for all members of the given stack `gs`. In order to avoid the problem of negative scaled fitnesses which can cause problems when using linear scaling, this technique normalises the raw fitnesses using standard deviation information prior to scaling. A "normalised" fitness is first calculated as $f_{norm} = f_{raw} - (\bar{f}_{raw} - C\sigma)$ where C is equal to `rNumSigma` and σ is the population deviation calculated for the genomes in `gs`. These normalised values are then scaled using the linear scaling technique described previously, with proportionality constant `rScaleFactor`. Negative values are arbitrarily set to zero (as when `bCutoffNotPivot` is `TRUE`). The `bMaxIsBest` flag determines whether high scaled fitness values correspond to high or low raw fitness values. If any member of `gs` is not evaluated, or if `rScaleFactor` or `rNumSigma` are negative, no action is performed.

Structured Population Operators

Function `DemeCollect`

`proc` DemeCollect (

`gstack`	`gsDeme,`	*Stack to hold genomes in deme*
`genome`	`gPop`	*A structured population of genomes*

)

This function adds all genomes in the deme (as determined by the `deme` specified in the `uses` declaration) of `gPop` to `gsDeme`, which is first emptied if it is not already empty. `NULL genomes` in the deme are ignored (not placed on the stack).

Function `MigrateRandom`

`proc` MigrateRandom (

`gstack`	`gsImmigrants,`	*Stack to add immigrants to*
`genome`	`gEmigrant,`	*The genome that may emigrate*
`real`	`rProbMigrate`	*The probability with which migration occurs*

)

If the `gEmigrant` is not the `NULL genome` then it migrates, with probability `fProbMigrate` from its current `island` position to a random site (the `fine` axes co-ordinates are unchanged). Otherwise, no emigration occurs from the site. Any `genomes` arriving at the site are added to the top of `gsImmigrants`.

Function `MigrateLocal`

`proc` MigrateLocal (

`gstack`	`gsImmigrants,`	*Stack to add immigrants to*
`genome`	`gEmigrant,`	*The genome that may emigrate*
`real`	`rProbMigrate`	*The probability with which migration occurs*

)

If the `gEmigrant` is not the `NULL genome` then it migrates, with probability `fProbMigrate` from its current `island` position to a neighbouring site, with a randomly chosen axis co-ordinate different by one (the `fine` axes co-ordinates are unchanged). Otherwise, no emigration occurs from the site. Any `genomes` arriving at the site are added to the top of `gsImmigrants`.

Function `ReduceRandom`

`genome` ReduceRandom (

`genome`	`g`	*A higher dimensional genome to be reduced*

)

In order to allow mapping of data between structured arrays of different sizes, a number of reduction and projection functions have been supplied in the `StdInst` library. These instructions are explicitly parallel, and take some arguments which are of higher dimension than the current nesting of `structfor` loops. The return values (if any) correspond, like all other functions, to the current loop nesting.

The `ReduceRandom` reduction function is similar to the `SelectRandom` function, except that it selects from a `genome` array of higher dimension than the returned array instead of selecting from a `gstack` which conforms to the return argument. The returned value will be appropriate to the current nesting of `structfors`, whereas the argument `g` may be of higher dimension as discussed above.

Function `ReduceRawBest`

`genome` ReduceRawBest (

`genome`	`g,`	*A higher dimensional genome to be reduced*
`bool`	`bMaxIsBest`	*True if maximal fitness is best*

)

This reduction function is similar to the `SelectRawBest` function, except that it selects from higher dimensional `genome` arrays instead of simple `gstacks`. The returned value will be appropriate to the current nesting of `structfors`, whereas the argument `g` may be of higher dimension as discussed previously. The structured genome `g` must be evaluated or no action will be performed.

Function `ProjectRandom`

`proc` ProjectRandom (

`genome`	`gTarget,`	*The higher dimensional target genome*
`genome`	`gProject`	*The genome to be projected*

)

This function is similar to the `ReplaceRandom`, except that `gProject` is used to replace a corresponding member of the higher dimensional target `gTarget` instead of a `gstack`. The dimension of the `gProject` argument must agree with the current nesting of `structfors`, which the target is higher dimensional.

Function `ProjectRawWorst`

`proc` ProjectRawWorst (

`genome`	`gTarget,`	*The higher dimensional target genome*
`genome`	`gProject,`	*The genome to be projected*
`bool`	`bMaxIsBest`	*True if maximal fitness is best*

)

This function is similar to the `ReplaceRawWorst`, except that `gProject` is used to replace a corresponding member of the higher dimensional target `gTarget` instead of a `gstack`. The dimension of the `gProject` argument must agree with the current nesting of `structfors` while the target should be of higher dimension. Both genomes must be evaluated, or no action will be taken.

Function `Reduce[Int|Real][Min|Max|Sum|Product]`

`int` Reduce[Int|Real][Min|Max|Sum|Product] (

`type`	`t`	*The higher dimensional array to be reduced*

)

This function reduces an array of integers or reals of higher dimension that the current nesting of `structfors` to a return array that matches the loop nesting level. Various operations are supported via different function names, as listed below. For the `ReduceIntOp` forms, the argument `t` must be of type `int`, while for the `ReduceRealOp` forms, the argument should be `real`.

`Min` Reduce to the *minimum* of corresponding elements.

`Max` Reduce to the *maximum* of corresponding elements.

`Sum` Reduce to the *sum* of corresponding elements.

`Product` Reduce to the *product* of corresponding elements.

Function `ReduceBool[And|Or|Xor]`

`bool` ReduceBool[And|Or|Xor] (

`bool`	`b`	*The higher dimensional array to be reduced*

)

This function reduces a higher dimensional array of booleans to an array that corresponds to the current nesting of `structfors` using a specified logical operator. Various operations are supported through the different function names as listed below:

`And` Reduce to the logical *and* of corresponding elements.

`Or` Reduce to the logical *or* of corresponding elements.

`Xor` Reduce to the logical *xor* of corresponding elements.

Function `Collect`

`proc` Collect (

`genome`	`gSource,`	*Higher dimensional genome to collect*
`gstack`	`gsTarget`	*Stack to collect genome to*

)

This function collects (copies) all members of a higher dimensional `genome` source, `gSource` onto a target `gstack`. The target `gsTarget` must have a structure corresponding to the current nesting of `structfors`, and the source should be higher dimensional. The order in which elements will be collected is identical to how the elements of a C array corresponding to the `gSource` array would be laid out in memory. Thus the lowest element on the stack has co-ordinate `[1,1,...,1]` and the last dimension (reading left to right) of the structure is varied fastest.

Function `Spread`

`proc` Spread (

`gstack`	`gsSource,`	*Stack containing source genomes*
`genome`	`gTarget`	*Higher dimensional genome to fill*

)

This function is the converse of `Collect`, and spreads (copies) successive members of the source stack `gsSource` onto the higher dimensional target genome `gTarget`. The source must correspond in dimension to the current nesting of `structfors`. If there are insufficient `genomes` on `gsSource` at any location in the structure, the `NULL genome` is substituted for those values. The order in which elements are spread is exactly the same as the `Collect` operation, so that collecting and then spreading to the same array leaves it unchanged.

Standard Demes

Function `Euclidean`

`deme` Euclidean (

`real`	`rRadius`	*Radius in grid points of the deme*

)

Used only in the declaration of a structured population, as the `deme` declaration. Declares the deme of each grid point to be all grid points within `rRadius` (inclusive) points of the co-ordinate under consideration (in the fine grained axes), measured in Euclidean terms.

Function `Taxicab`

`deme` Taxicab (

`int`	`nRadius`	*Radius in grid points of the deme*

)

Used only in the declaration of a structured population, as the `deme` declaration. Declares the deme of each grid point to be all grid points within `nRadius` (inclusive) points of the co-ordinate under consideration (in the fine grained axes), measured by the “taxicab metric” (the sum of the vertical and horizontal distances).

Function `InfNorm`

```
deme InfNorm (
      int  nRadius              Radius in grid points of the deme
)
```

Used only in the declaration of a structured population, as the `deme` declaration. Declares the deme of each grid point to be all grid points within `nRadius` (inclusive) points of the co-ordinate under consideration (in the fine grained axes), measured by the "infinity-norm" (the maximum co-ordinate difference — this defines the deme to be a hypercube of side length twice `nRadius`).

Miscellaneous Operators

Function `Randomize`

```
proc Randomize (
      int  nSeed                Seed for the random number generator
)
```

This function initialises Marsaglia's "Universal" random number generator, as described in [12, 11] . A non-zero value of `nSeed` will initialise the generator in a deterministic way so that results are reproducible given the same seed value (running the same program on identical hardware). A seed value of zero will do a pseudo-random initialisation of the generator (probably based on the system clock).

Function `Checkpoint`

```
proc Checkpoint (
      int     iLoopIndex,       The current value of the loop index
      int     nFrequency,       The number of iterations between saves
      string  sOutfile          The name of the file to write checkpoint data to
)
```

This function saves the current state of the RPL2 environment if `iLoopIndex` is some integer multiple of `nFrequency`. Typically this statement is used inside time consuming loops in order to provide a means for recovering partial runs (after a hardware fault), or doing a run in multiple parts. If `nFrequency` is 0 or negative, a checkpoint is performed regardless of the value of `iLoopIndex`.

The routine ensures the system is in a consistent state and then ensures that all processes save sufficient information to allow the run to be resumed. Significant performance degradation can be expected if this routine is called frequently.

The checkpoint information will be saved to the file named in `sOutfile`. If the special character `'#'` is present in the string `sOutfile`, the generated filename will be `sOutfile` with `'#'` replaced by the value of `iLoopIndex`. This allows multiple checkpoint files to be maintained for a single run. The contents of the string `sOutFile` will not be changed. The specified file will be overwritten (if it exists) as a result of this call. However, this will be achieved via a temporary file (named `#sOutfile#`), ensuring that a valid checkpoint file exists at all times.

Function `Write[Genome|Gstack|Bool|Int|Real]`

`proc` Write[Genome | Gstack | Bool | Int | Real] (

`type`	`tArg,`	*The variable to write to a file*
`string`	`sOutFile`	*The file to write the data to*

)

These functions save the contents of the specified variable to the file `sOutFile`. Each function must take the appropriate type of argument (e.g. `WriteBool` requires a `bool` first argument). The output file is overwritten if it exists, via a temporary file so that a valid version will exist at all times. These files can be read using the `Read[Type]` functions. The special string `"stdout"` is recognised, and will redirect output to `stdout`.

Function `Read[Genome|Gstack|Bool|Int|Real]`

`proc` Read[Genome | Gstack | Bool | Int | Real] (

`type`	`[&]tArg,`	*The variable to read from a file*
`string`	`sInFile`	*The file to read the data from*

)

These functions read data from a file written using the `Write[Type]` functions. The `ReadReal`, `ReadInt` and `ReadBool` functions require the first argument to be passed by reference (with `&` prepended to the argument name). If `sInFile` cannot be opened, or contains data which doesn't match the type/size of `tArg`, no action is performed. Otherwise, the variable `tArg` is set to the values specified in the file.

Function `StatsPrint`

`proc` StatsPrint (

`int`	`iLoopIndex,`	*The current value of the loop index*
`int`	`nFrequency,`	*The number of iterations between saves*
`gstack`	`gsPop,`	*An evaluated population to collect statistics about*
`string`	`sOutFile`	*The name of the file to write the data to*

)

This function writes statistics concerning the generic parts of `genomes` if `iLoopIndex` is an integer multiple of `nFrequency`. Setting `nFrequency` to 1 or 0 ensures that the data will be printed (the function need not be called with a "real" loop index inside a `for` loop). All data is written to the specified file. The current value of the loop index is printed, along with the maximum, minimum, mean, median and variance of the raw fitness values. If `gs` is not evaluated, no action will be taken.

The specified file `sOutFile` will be appended to if it exists already. The name special string `"stdout"` will be recognised and will cause output to be directed to `stdout` instead of to a file.

6.2 The Binary Representation

The binary representation is provided with the basic RPL2 framework in order to allow the user to immediately write plans, and as an example for those users wishing to create their own representations. Although the binary representation is commonly used in theoretical discussions of genetic algorithms, it is not suitable for many practical problems. It should by no means be considered a panacea or a substitute for writing a more suitable problem-specific representation.

The representation is parameterised with a single integer, indicating the number of bits in the genome.

To use the representation in a plan, include the name and required number of bits in the `use` line as described in *section 5.6*, e.g. `use Binary(64)`.

Function `RandomGenome`

```
genome  RandomGenome (

)
```

Creates and returns a random binary bit string, with the currently specified bit length.

Function `EvalDeJongF2`

```
proc  EvalDeJongF2 (
      genome  g                    Genome to be evaluated
)
```

Evaluate the real fitness of the given `genome` using De Jong's F2 function (as described in [5]). This evaluation treats the genome `g` as two binary fractions, with both represented as the same number of bits if the genome length is even, and the first having the extra bit if odd. (Thus the eight bit string 10100111 would encode $1/2 + 1/8$, $1/4 + 1/8 + 1/16$.) The fractions are used to form two values x_1 and x_2 in the range $-2.048 \leq x_1, x_2 \leq 2.048$ by linear interpolation, and the fitness of the genome is evaluated using the F2 function:

$$f_2(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$$

Ideally the function should be minimised.

Function `Mutate`

```
proc  Mutate (
      genome  g,                   Genome to be mutated
      real    rRate                Mutation rate
)
```

This function mutates the bits making up the `genome g`. The probability that each bit will mutate (flip its state) is specified by `rRate` which must be between 0 and 1 inclusive.

Function `CrossNpt`

```
genome  CrossNpt (
      genome  gA,                  First parent genome for cross
      genome  gB,                  Second parent genome for cross
      int     nCrossPoints,        Number of crossover points to choose
      real    rCrossProb           Probability that crossover will occur
```

)

Performs a linear N-point crossover of the provided parent genomes and returns a child, with probability `fCrossProb`. If crossover does not occur, then one of the parents is selected at random and cloned to produce the child. The crossover points are selected uniformly along the length of the parent genomes, and the child is made up of sections taken alternately from each parent. The value of `rCrossProb` must be in the interval $[0, 1]$.

Function `PrintGenome`

`proc` PrintGenome (

`genome`	`g,`	*The genome to print*
`string`	`sOutFile`	*The file to print to*

)

Prints the given `genome` showing its fitness and raw fitness (if evaluated), along with its number of bits to the specified file. If `sOutFile` exists, data will be appended. The special string `"stdout"` is recognised, and will redirect output to `stdout`.

6.3 The Variable Cardinality Integer Representation (IntVC)

The variable cardinality integer representation is provided with the basic RPL2 framework in order to facilitate the implementation of the pipe-network optimisation problem posed by British Gas.

The representation consists of a string of integers, and is parameterised with a string which names a data file describing the genome. The data file should contain a list of integers, one per line. The first value indicates the number of genes N in the genome. Following this, the cardinality C_i of each gene should be specified. Thus there should be $N + 1$ lines in the file. An instance of the genome is simply a sequence of N integers $a_1 a_2 ... a_N$ with $a_i \in \{0, 1, ..., C_i - 1\}$.

To use the representation in a plan, include its name and the data file parameter in the `use` line as described in *section 5.6*, e.g. `use IntVC("data.dat")`.

Function `RandomGenome`

`genome` RandomGenome (

)

Creates and returns a random string of integers, corresponding to the currently specified data set (as explained above).

Function `MutateRandom`

`proc` MutateRandom (

`genome`	`g,`	*Genome to be mutated*
`real`	`rRate`	*Mutation rate*

)

This function mutates the integers making up the `genome g`. The probability that each integer will be mutated is specified by `rRate` which must be between 0 and 1 inclusive. Each mutated gene will be replaced by a random integer in the proper range (e.g. $a_i \in \{0, 1, ..., C_i - 1\}$ as above).

Function `MutateCreep`

```
proc  MutateCreep (
      genome  g,          Genome to be mutated
      real    rRate,      Creep rate for genes
      bool    bCyclic     Allow creep to wrap around
)
```

This function mutates the genes making up `g`. Each gene is mutated with probability $(C_i - 1)\cdot$ `rRate`. (Thus if all $C_i = 2$ this function is equivalent to `Binary::Mutate`.) Each gene that mutates has its value changed by plus or minus one. If the `bCyclic` flag is `TRUE`, then genes with value 0 can creep to $C_i - 1$ and vice versa, otherwise this is prohibited.

Function `CrossNpt`

```
genome  CrossNpt (
      genome  gA,             First parent genome for cross
      genome  gB,             Second parent genome for cross
      int     nCrossPoints,   Number of crossover points to choose
      real    rCrossProb      Probability that crossover will occur
)
```

Performs a linear N-point crossover of the provided parent genomes and returns a child, with probability `fCrossProb`. If crossover does not occur, then one of the parents is selected at random and cloned to produce the child. The crossover points are selected uniformly along the length of the parent genomes, and the child is made up of sections taken alternately from each parent. The value of `rCrossProb` must be in the interval $[0, 1]$.

Function `CrossUniform`

```
genome  CrossUniform (
      genome  gA,           First parent genome for cross
      genome  gB,           Second parent genome for cross
      real    rBias,        Bias toward first parent
      real    rCrossProb    Probability that crossover will occur
)
```

This function performs a uniform crossover between two parent genomes to create a new child, with probability `fCrossProb`. If crossover does not occur, then one of the parents is selected (`gA` with probability `rBias`, otherwise `gB`) and cloned to produce the child. Uniform crossover produces a child by taking each gene from the first parent with probability `rBias` and from the second parent with probability $1-$ `rBias`. The values of `rBias` and `rCrossProb` must be in the interval $[0, 1]$.

Function `PrintGenome`

`proc` PrintGenome (

`genome`	`g,`	*The genome to print*
`string`	`sOutFile`	*The file to print the information to*

)

Prints the given `genome` showing its fitness and scaled fitness (if evaluated), along with its number of bits to the specified file. If `sOutFile` exists, data will be appended. The special string `"stdout"` is recognised, and will redirect output to `stdout`.

7 Further Information

Further information about RPL2 and related areas are available as follows:

- RPL2 Functional Specification, [21]
- RPL2 Design Document, [20]
- RPL2 Programmer's Guide, [22]
- PUL-RD, [2]
- PUL-TF, [23]
- PUL-EM, [24]
- CHIMP V2, [1]
- Manual pages for RPL2 and RPP are included in the distribution.

The authors of RPL2 can be reached at the following electronic mail addresses:

rpl2-support@epcc.ed.ac.uk Usage queries, design questions and suggestions for improvements should be reported here, and may be answered on a time-permits basis after the alpha and beta test phases.

rpl2-faults@epcc.ed.ac.uk Bug reports received at this address will be acknowledged, and may be corrected in future releases.

rpl2-contributions@epcc.ed.ac.uk Contributions of new libraries of operators and representations may be sent to this address, and will be considered for inclusion with future general releases of RPL2.

A Example RPL2 Programs

See the example plans in `$RPLHOME/*.rpl`.

References

[1] R. A. A. Bruce and A. G. Smith. CHIMP version 2.0 interface. Technical Report EPCC-KTP-CHIMP-V2-IFACE, Edinburgh Parallel Computing Centre, 1993.

[2] S. Chapple. *PUL-RD Prototype User Guide*. Technical Report EPCC-KTP-PUL-RD-PROT-UG 1.5, Edinburgh Parallel Computing Centre, 1993.

[3] Y. Davidor. A naturally occuring niche & species phenomenon: The model and first results. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 257–263, San Mateo, 1991. Morgan Kaufmann Publishers.

[4] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Publishing Co., Inc., 1989.

[6] M. Gorges-Schleuter. Asparagos an asynchronous parallel genetic optimization strategy. In *ICGA*, pages 422–427, 1989.

[7] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimisation procedure. *Physica D*, 42:228–234, 1990.

[8] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[9] G. P. Jones. Parallel genetic algorithms for large travelling salesrep problems. Master's thesis, University of Edinburgh, September 1992.

[10] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433, San Mateo, 1989. Morgan Kaufmann Publishers.

[11] G. Marsaglia. A current view of random number generators. In *Computer Science and Statistics: 16th Symposium on the Interface*, 1984.

[12] G. Marsaglia, A. Zaman, and W. W. Tsang. Toward a universal random number generator. *Statistics and Probability Letters*, 9(1):35–39, 1990.

[13] H. Muhlenbein. *Foundations of Genetic Algorithms & Classifier Systems*, chapter Evolution in Time and Space - The Parallel Genetic Algorithm, pages 316–337. Morgan Kaufmann Publishers, San Mateo, 1991.

[14] H. Muhlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 271–278, San Mateo, 1991. Morgan Kaufmann Publishers.

[15] M. Norman. A genetic approach to topology optimisation for multiprocessor architectures. Technical report, University of Edinburgh, 1988.

[16] N. J. Radcliffe. Forma analysis and random respectful recombination. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms.*, San Mateo, 1991. Morgan Kaufmann Publishers.

[17] G. J. E. Rawlins, editor. *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, San Mateo, 1991.

[18] C. V. Russo. A general framework for implementing genetic algorithms. Technical Report EPCC–SS91–17, Edinburgh Parallel Computing Centre, University of Edinburgh, 1991.

[19] P. Spiessens and B. Manderick. A massively parallel genetic algorithm: Implementation and first analysis. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of*

the Fourth International Conference on Genetic Algorithms, pages 279–286, San Mateo, 1991. Morgan Kaufmann Publishers.

[20] P. D. Surry. RPL2 Design Document. Technical Report EPCC-BG-PAP-RPL2-DD, Edinburgh Parallel Computing Centre, November 1993.

[21] P. D. Surry. RPL2 Functional Specification. Technical Report EPCC-BG-PAP-RPL2-FS, Edinburgh Parallel Computing Centre, July 1993.

[22] P. D. Surry. RPL2 Programmer's Guide. Technical Report EPCC-BG-PAP-RPL2-PG, Edinburgh Parallel Computing Centre, November 1993.

[23] S. Trewin. *PUL-TF Prototype User Guide*. Technical Report EPCC-KTP-PUL-TF-PROT-UG 1.7, Edinburgh Parallel Computing Centre, 1993.

[24] S. M. Trewin and S. R. Chapple. *PUL-EM Prototype User Guide*. Technical Report EPCC-KTP-PUL-EM-PROT-UG, Edinburgh Parallel Computing Centre, 1993.