CGP lil-gp 2.1;1.02
USER's MANUAL

version April 7, 1997

Cezary Z. Janikow[1]

Scott W. DeWeese

Department of Mathematics and Computer Science
University of Missouri – St. Louis
janikow@radom.umsl.edu

This document describes extensions provided to *lil-gp* facilitating dealing with constraints, as outlined in [1]. This document deals specifically with *lil-gp 1.02*, and the resulting extension is referred to as *CGP lil-gp 2.1;1.02* (the first version is for the extension, the second for the utilized *lil-gp* version). Unless explicitly needed to avoid confusion, version numbers are omitted.

1 Description

CGP lil-gp 2.1;1.02 can be invoked exactly as *lil-gp*. The only difference is the additional interactive interface reading *Function Types, Tspecs, Fspecs* and *Weights*.

CGP *lil-gp 2.1;1.02* adds to the constraints found in CGP *lil-gp 1.1;1.02.* In the earlier version the user provided constraints to indicate which functions and terminals to include and/or exclude from being an argument for certain function. In this version the ability to enter weights for each of the functions and terminals is also available. Specifying a weight will alter the probability that a particular function or terminal will be picked during generation or mutation.

A form of 'Type Checking' has also been added. The user specifies the data types allowed, all instances of a function's return type and argument types, terminal types, and the return type of the problem to be solved. This information is combined with the earlier constraints to limit even further which functions/terminals can be used as arguments to all of the functions. This Type checking occurs during all phases of tree development to ensure that the trees produced are closer to being mathematically correct.

In Section 2 (Initialization and Operator Modifications) we describe the modifications made to the operators and the parameter file. In Section 3 (Constraining lil-gp) we describe the various methods included in CGP *lil-gp* to constrain *lil-gp*. In Section 4 (CGP lil-gp Interactive Interface) we describe the interactive user interface in detail. And, Section 5 (Interface File) describes the new (*CGP lil-gp 2.1;1.02*) File Interface system. Except for that interfacing, *CGP lil-gp* runs exactly as *lil-gp* does, except that only valid trees are manipulated.

2 Initialization and Operator Modifications

Modifications to *lil-gp* have been kept to a minimum. However some changes seemed prudent to help enable it to respond more closely to the users intent.

2.1 Initialization

Please refer to the *"lil-gp 1.0 User's Manual"[3]* for a complete description of the standard initialization parameters.

2.1.1 Initialization Parameters

A couple of additional Initialization Parameter have been added. One to help control tree generation and two to provide information regarding the input files. And, like all parameters, they may be specified in a parameter file or on the command line. Please see the lil-gp Users Manual [3] for further information on using parameters and parameter files.

1. This research was supported through NASA/JSC grant NAG 9-847.

2.1.1.1 Tree Creation Parameter

- `init.depth_abs` = {true,false}, default = false.

This parameter is used to prevent the generation of initial trees shallower than that specified by the depth ramp. In the default mode, this new parameter has no effect over *lil-gp*. When `init.depth_abs` is set to `true`, `init.depth=m-n` will cause rejection of initial trees shallower than `m`. *lil-gp's* default behavior allows trees to be shallower than m. (**Technical Note: this can occur during a call to `generate_random_grow_tree`()).**

2.1.1.2 Interface Parameters

- `cgp_interface` - The file containing the constraint creation instructions.
- `cgp_input` - The file which is to be created from the constraint instructions, if the parameter `cgp_interface` is specified. This file will then act as the input to *CGP lil-gp*.

Detailed information about these parameters can be found in Section 5.2.

2.2 Mutation Operator

An additional Mutation Parameter has been added to prevent the Mutation Operator from allowing the mutated subtree from being shallower than that specified by the depth parameter.

2.2.1 Mutation Parameters

- `depth_abs` ={true,false}, default=false.

In the default mode, this new parameter has no effect over *lil-gp*. When `depth_abs` is set to `true`, `depth=m-n` will cause rejection of mutated subtrees shallower than `m` (it is a good idea to have `keep_trying` set to `true` as well). *lil-gp's* default behavior allows the Mutation operator to mutate a subtree shallower than m. **(Technical Note: this can occur during a call to `generate_random_grow_tree`()).**

2.3 Crossover Operator

Additional parameters have been added to the Crossover parameters.
The standard `internal` and `external` parameters are replaced with 2 pair of parameters which separate the functionality between the source and destination parents.

2.3.1 Crossover Parameters

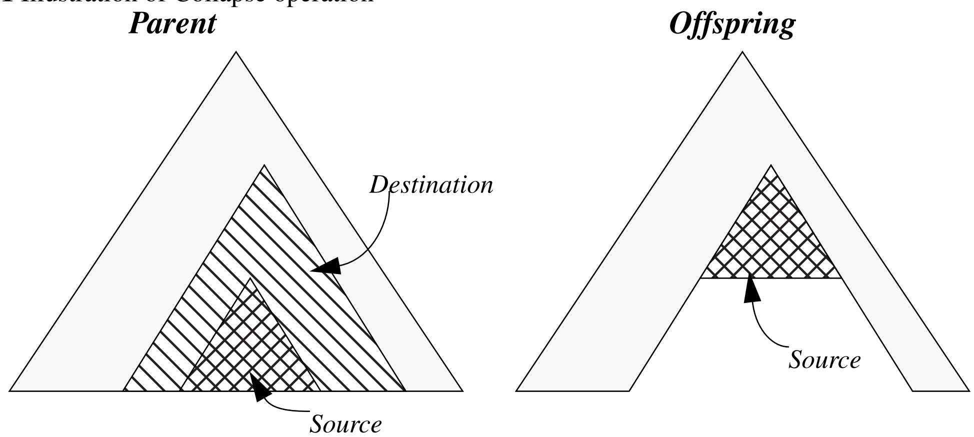
- `internal` (internal nodes in source parent), default = 0.9
- `internal_dst` (internal nodes in destination parent), default = `internal`
- `external` (external nodes in source parent), default = 0.1
- `external_dst` (external nodes in destination parent), default = `external`

These act the same way as in *lil-gp* except that there can be different frequencies for internal/external node selection in the source and destination parents.

2.4 Collapse Operator (New)

The Collapse operator is similar to the Mutation operator in that it causes a single parent to mutate into a single offspring. Collapse chooses a random subtree, from a given parent tree, to be the destination. Another subtree is chosen, from within the destination subtree, as the source (see Figure 1). All of the nodes in the destination subtree that are not also in the source subtree are removed. This results in the destination subtree being removed, and the source subtree moved into its place. See also Example 1.

Figure 1 Illustration of Collapse operation



Example 1 Here is a sample individual (tree) from an actual experiment. This can be read as LISP function calls, where `(atan2 y x)` corresponds to `atan2(y,x)` in the C language.

```
Parent:    (fkin (sub  (atan2 y x)
                       (acos (div (div x 2) x)))
                 (acos (add 2
                       (sub 1
                            (div (hypot x y) L1)))))
```

(The function `sub` performs subtraction, and the function `div` performs division. The function `fkin` simply causes its 2 subtrees to execute in order.) So, if *`sub`* on the first line is chosen as the beginning of the Destination subtree of collapse, and if `(div x 2)` is chosen as the Source subtree, then the resulting offspring would be:

```
Offspring: (fkin (div x 2)
                 (acos (add 2
                       (sub 1
                            (div (hypot x y) L1)))))
```

2.4.1 Collapse Parameters

- `select` - same as for Mutate
- `keep_trying` - same as for Mutate
- `internal` - same as for Mutate
- `external` - same as for Mutate
- `tree` -same as for Mutate
- `tree`*`n`* - same as for Mutate

3 Constraining *lil-gp*

In CGP *lil-gp*, there are three methods for constraining *lil-gp*. These are Tspec / Fspec Constraints, Weight Constraints, and Type Constraints. These constraints define and expand the method used to select function/terminals for tree growth/mutation. That method is the use of Mutation Sets.

3.1 Tspec / Fspec Constraints

Tspecs and Fspecs are sets of functions and terminals with some common characteristic. There is a detailed explanation of Tspecs and Fspecs in Section 4.1 of the Technical Manual [2].

- Tspecs are those functions/terminals which are compatible with a specific argument of a function (`T_func_`*`n`*), and those functions/terminals allowed to be the Root (`T_Root`). (syntactic constraint)

- Fspecs are those functions/terminals which are not compatible as a specific argument of a function (`F_func_`*n*), those functions which are not allowed to directly call a given function (`F_func`), and also those functions/terminals not allowed to be the Root (`F_Root`). (semantic constraint)

When specifying these constraints you must specify Tspecs for everything you wish to allow. The Tspecs and Fspecs are converted to their Normal form, consisting entirely of Fspecs. These Normal Form Fspecs are what are used to construct the Untyped Mutation Sets (see Section 3.4).

3.1.1 Tspec / Fspec Example

Listed here, in the order that they are requested in the interactive user interface are some example Fspecs and Tspecs.

Example 2 `F_add = sin cos`
This prevents sin() and cos() from calling add (e.g. sin(x+y) is not allowed).

Example 3 `F_add_0 = 0 PI log`
This prevents 0, PI, log() from being argument 0 of add (e.g. (PI + x) is not allowed, but (x + PI) may be).

Example 4 `T_add_0 = sin cos 0 PI`
This allows sin(), cos(), 0, and PI to be argument 0 of add. However, the Fspec, F_add_0, overrides this Tspec, and so only sin() and cos() are actually allowed.

Example 5 `F_add_1 = 0 PI`
This prevents 0, PI from being argument 1of add (e.g. (x + PI) is not allowed).

Example 6 `T_add_1 = sin cos 0 PI`
This allows sin(), cos(), 0, and PI to be argument 1of add. However, the Fspec, F_add_1, overrides this Tspec, and so only sin() and cos() are actually allowed.

Example 7 `F_ROOT = log 0 PI`
This prevents the function `log`, and the terminals `0` and `PI` from being the Root.

Example 8 `T_ROOT = add sin cos`
This allows the functions `add`, `sin`, and `cos` to be the Root.

3.2 Weight Constraints

Normally all functions/terminals have the same probability of being selected to be used as an argument to a function. The Weights section of the interface allows you to change this default behavior. The Weight constraints do not affect nor interact with Type constraints.

The weight of any function/terminal is in the range (0,1]. Using only the Weight constraints, you cannot absolutely prevent a function/terminal from being used. This is because any weight specified as 0 (Zero) is converted to a defined constant $MINWGHT \approx 0.00001$, defined in `kernel*/cgp_czj.c`. If you desire to absolutely prevent a function/terminal from being used, you need to specify the appropriate Fspec (see Section 3.1)

When you are given the chance to enter the weights, only those function/terminals which appear in the Untyped Mutations Set are used (see Section 3.4). So, if a function is disallowed through Fspecs, you are not asked for the weight for it.

Currently, the weights remain constant throughout the execution of the program. Work is currently underway to implement mechanisms for the automatic adjusting of the weights during program execution.

3.2.1 Weight Constraint Example

Example 9 As in Example 9 in the Technical Manual [2], assume we have 4 functions(f1, f2, f3, f4) and 4 terminals(t1, t2, t3, t4). Through Fspecs, t2 was excluded from being an argument to function f*n*. Then the weights are given as f1=1.0, f2=0.0, f3=0.1, t1=1.0, t3=1.0, t4=1.0. The total weight of all function/terminals for this argument = f1 + f2 + f3 + t1 + t3 + t4 = 4.1. The probability p(), of any function/terminal being selected as this particular argument is p((f/t)*n*) = ((f/t)*n)* / (total weight). So p(f1,t1,t3,t4) = 1.0/4.1 = 0.244, p(f2) = MINWGHT/4.1 = 2.439×10^{-6} , p(f3) = 0.1/4.1 = 0.0244, p(t2) = 0.0.

3.3 Type Constraints

Type constraints were added to CGP to help control the generation of trees representing poor mathematics. Without type constraints it would be possible, with normal arithmetical and trigonometric functions, to generate a subtree which would evaluate to: `asin(x) + sin(x)`, which under normal circumstances doesn't make much sense. With the type constraints, `add()` can be instructed not to add a non-angle with an angle, etc. **(Note: The type constraints do not in any way perform type casting of functions, terminals, or arguments. If a function is to be able to accept multiple data types, then the function must be written in such a way as to make that possible.)**

It can prevent occurrences of subtrees like that shown in Example 10, or Example 11, if that kind of restriction is what you would desire.

Example 10 `sin(mult(PI,sin(sqrt(PI))))`

Example 11 If there are variables of various units, such as S(seconds), M(mass), D(distance), T(temperature), then subtrees like this could be avoided: `add(add(S,M),add(D,T))`.

3.3.1 Type Constraint Example

To begin with, you specify every data type that your problem uses. **(Note: The data types specified here do not necessarily have anything to do with the actual data types of your functions/terminals/arguments)** Then, starting with the highest arity (argument count) functions working down to the terminals, and sorted alphabetically, every instance of a function is listed. with its arguments and return type (see Example 12). After listing every instance of every function, the return types of each of the terminals and of the root of the problem are listed.

Example 12 There may be a problem needing data types: *angle, length, force, force-length,* and *number*. And, the `multiply()` function which takes 2 arguments could have the following instances (vectors).

<arg1>	**<arg2>**	**<return>**
number	length	length
length	number	length
number	angle	angle
angle	number	angle
number	number	number
length	force	force-length
force	length	force-length

All of this means that if `multiply()` is called with a number and a length, it will return a length. If it is called with a number and an angle, it will return an angle. If it is called with two numbers it just returns a number. And, if it is called with a length and a force, it returns a force-length (e.g. 3 lbs. * 5 ft. = 15 ft-lbs.).

The type constraints also allow you to reuse a single function is several ways. This can allow you to easily create a specific "structure" for your trees as is shown in Example 13.

Example 13 You have a defined structure (full & complete binary tree of height 4, with an `and/or` for each internal node) you're trying to create from a limited set of functions. Without type constraints you would have to create different `and` & `or` functions for each level. With type constraints, however, you can specify different types (e.g. L0, L1, L2, L3, L4). So that, all terminals return type L4, and the Root takes type L0. The `and` & `or` functions then have instances such as:

Arg1	Arg2	Return
L1	L1	L0
L2	L2	L1
L3	L3	L2
L4	L4	L3

This will cause the generation of only those trees which match the above specified structure.

3.4 Mutation Sets

Mutation Sets are the constructs that CGP uses to keep track of the structure of valid trees. A mutations set consists of the functions and terminals that are allowed to represent an argument of a function.

The initial mutation set, called the Untyped Mutation Set, is constructed from the Normal Form Fspecs (see Section 3.1). Any function/terminal listed in an Fspec is removed from the corresponding portion of the Untyped Mutation Set.

The Untyped Mutation Set is combined with the Type Constraint (see Section 3.3) information to yield the final Typed Mutation Sets, which are then used for the construction of the trees. Any function/terminals which has a type that is not compatible with the argument in which it is referenced, is removed from the Mutation Set for that argument.

Upon examining the Typed Mutation Set, the true power of it may not be apparent. In Section 4.2 towards the end of the listing, the example shows the Typed Mutation Set. Even though the function `add` appears in every set for every argument of every function, it is not the same instance of the `add` function. The particular instance of the function is partially determined by which Type it is under. That is to say the `add` function under a Type 'angle' section is strictly the `add` function which takes two angles as arguments and returns an angle.

4 *CGP lil-gp* Interactive Interface

The interface to CGP *lil-gp* is an interactive system. Constraints are entered using function names. Listing order is irrelevant. Repetitions are disregarded. Functions are sorted by arty and then alphabetically by name.

4.1 Interactive Interface Description

An interactive interface is part of the modifications which CGP *lil-gp 2.1;1.02* has added to *lil-gp*. The interface can be broken up into the following sections:

1. Type Information (*optional*)
    a. User defined data entry section
    b. Display of Type Vectors
2. Tspecs & Fspecs (*optional*)
    a. User defined data entry section
    b. Display of Tspec & Fspec Constraints
    c. Display of Normal Constraints (Tspec's & Fspec's converted to Fspec-only)
    d. Display of Untyped Mutation Sets
3. Weights (*optional*)
    a. User defined data entry section
4. Display of Typed Mutation Sets

4.2 Interactive Interface Example

Note: This example is of a problem with the functions (add, asin, sin), and terminals(1, PI, x, y). All user's responses are in *italics*.

Note: Type Information Section (see Section 3.3 for more information).

```
<...>
Reading Type information...
```

Note: You are asked for the Type constraints (see Section 3.3) of the functions and their arguments.

```
Default is a single 'generic' type. Accept? [0 to change]:
   0<ENTER>
List type names: float integer angle
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :float float float<ENTER>
```

```
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :integer float float<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :float integer float<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :integer integer integer<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :angle angle angle<ENTER>
Specs for 'add' [2 arg and one ret types /<ENTER> for no more]
   :<ENTER>

Specs for 'asin' [1 arg and one ret types /<ENTER> for no more]
   :float angle<ENTER>
Specs for 'asin' [1 arg and one ret types /<ENTER> for no more]
   :integer angle<ENTER>
Specs for 'asin' [1 arg and one ret types /<ENTER> for no more]
   :<ENTER>

Specs for 'sin' [1 arg and one ret types /<ENTER> for no more]
   :angle float<ENTER>
Specs for 'sin' [1 arg and one ret types /<ENTER> for no more]
   :<ENTER>

Give ret type for terminal '1': integer<ENTER>
Give ret type for terminal 'PI': angle<ENTER>
Give ret type for terminal 'x': float<ENTER>
Give ret type for terminal 'y':float<ENTER>
Give return type for the problem: angle<ENTER>
```

Note: All valid type vectors are now displayed

```
The following types are used...
Function 'add': numArg=2, numTypeVecs=5
   vec0: 0:'float' 1:'float' -> 'float'
   vec1: 0:'integer' 1:'float' -> 'float'
   vec2: 0:'float' 1:'integer' -> 'float'
   vec3: 0:'integer' 1:'integer' -> 'integer'
   vec4: 0:'angle' 1:'angle' -> 'angle'
   Type 'float' returned from vectors: 0 1 2
   Type 'integer' returned from vectors: 3
   Type 'angle' returned from vectors: 4
Function 'asin': numArg=1, numTypeVecs=2
   vec0: 0:'float' -> 'angle'
   vec1: 0:'integer' -> 'angle'
   Type 'angle' returned from vectors: 0 1
Function 'sin': numArg=1, numTypeVecs=1
   vec0: 0:'angle' -> 'float'
   Type 'float' returned from vectors: 0
Terminal '1': -> 'integer'
Terminal 'PI': -> 'angle'
Terminal 'x': -> 'float'
Terminal 'y': -> 'float'
Root: -> 'angle'
```

Note: Tspec & Fspec section (see Section 3.1 for more information).

```
Reading F/Tspec information...


Default is empty Fspecs, full Tspecs. Accept? [0 to change]:
```

```
   0<ENTER>
3 ordinary function        (Note: this is displayed for each function)
   add asin sin
4 terminals (ordinary and ephemeral):
   1 PI x y
Separate entries by [ ,;]
Hit <ENTER> for empty set
Use function names in any order

Function 'add':
   F_add (exclusions) [up to 3 names] = <ENTER>
   F_add_0 (exclusions) [up to 7 names] = <ENTER>
   T_add_0 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>
   F_add_1 (exclusions) [up to 7 names] = <ENTER>
   T_add_1 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>
```

Note: this is asking you for the Tspecs and Fspecs (see Section 3.1) of the functions and their arguments.

- **F_add = Fspec for the add function (List all functions which cannot directly call add)**
- **F_add_0 = Fspec for Argument 0 of the add function (List all functions which cannot be argument 0 of the add function)**
- **T_add_0 = Tspec for Argument 0 of the add function (List all functions which can be argument 0 of the add function)**
- **F_add_1 = Fspec for Argument 1of the add function (List all functions which cannot be argument 1of the add function)**
- **T_add_1 = Tspec for Argument 1of the add function (List all functions which can be argument 1of the add function)**

```
<...>
Function 'asin':
   F_asin (exclusions) [up to 3 names] = <ENTER>
   F_asin_0 (exclusions) [up to 7 names] = <ENTER>
   T_asin_0 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>

<...>
Function 'sin':
   F_sin (exclusions) [up to 3 names] = <ENTER>
   F_sin_0 (exclusions) [up to 7 names] = add <ENTER>
   T_sin_0 (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>

<...>
Root:
   F^Root (exclusions) [up to 7 names] = asin<ENTER>
   T^Root (inclusions) [up to 7 names] = add asin sin 1 PI x y<ENTER>
```

Note: The Tspecs and Fspecs for the Root are slightly different.

- **F^Root = Fspec for the Root (List all functions which cannot be the root function)**
- **T^Root = Tspec for the Root (List all functions which can be the root function)**

Note: based on conditional compilation, constraints may be echoed. This section displays the constraints as you entered them, and then as converted into the Normal Constraints. The || separates functions from terminals.

```
Read the following constraints...

      CONSTRAINTS
Function 'add' [#0]:
```

```
   F_add [#Fs=0:#Ts=0] = ||
   F_add_0 [#Fs=0:#Ts=0] = ||
   F_add_1 [#Fs=0:#Ts=0] = ||
   T_add_0 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
   T_add_1 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
Function 'asin' [#1]:
   F_asin [#Fs=0:#Ts=0] = ||
   F_asin_0 [#Fs=0:#Ts=0] = ||
   T_asin_0 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
Function 'sin' [#2]:
   F_sin [#Fs=0:#Ts=0] = ||
   F_sin_0 [#Fs=1:#Ts=0] = 'add' ||
   T_sin_0 [#Fs=3:#Ts=4] = 'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'
Root:
   F_Root [#Fs=1:#Ts=0] =  'asin' ||
   T_Root [#Fs=3:#Ts=4] =  'add' 'asin' 'sin' || '1' 'PI' 'x' 'y'

The normal constraints are...

      CONSTRAINTS
Function 'add' [#0]:
   F_add_0 [#Fs=0:#Ts=0] = ||
   F_add_1 [#Fs=0:#Ts=0] = ||
Function 'asin' [#1]:
   F_asin_0 [#Fs=0:#Ts=0] = ||
Function 'sin' [#2]:
   F_sin_0 [#Fs=1:#Ts=0] = 'add' ||
Root:
   F_Root [#Fs=1:#Ts=0] =  'asin' ||
```

Note: This section displays the mutation sets as if the generic type were used. F[] = functions, T[] = terminals.

```
The following untyped mutation sets are used...

Function 'add': arity 2
   Argument 0
      Type unconstrained mutation set
         F [3 members] = 'add' 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
   Argument 1
      Type unconstrained mutation set
         F [3 members] = 'add' 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
Function 'asin': arity 1
   Argument 0
      Type unconstrained mutation set
         F [3 members] = 'add' 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
Function 'sin': arity 1
   Argument 0
      Type unconstrained mutation set
         F [2 members] = 'asin' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
Root:
      Type unconstrained mutation set
         F [2 members] = 'add' 'sin'
         T [4 members] = '1' 'PI' 'x' 'y'
```

Note: Weights Section (see Section 3.2 for more information).

Note: Entering 0 here will prompt you to enter all weights, similar to entering constraints above.

```
Setting initial weights for mutation set members...
Initial weights are all equal. Do you accept [0 to change]:
   0<ENTER>
Function 'add': 2 mutation set pairs
Argument 0:
F [3 members] = 'add' 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'

   Reading the weights for type I functions...
   Function 'add': give weight (0,1]: 0.25<ENTER>
   Function 'asin': give weight (0,1]: 0.25<ENTER>
   Function 'sin': give weight (0,1]: 0.5<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.2<ENTER>
   Terminal 'PI': give weight (0,1]: 0.2<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.4<ENTER>
Argument 1:
   F [3 members] = 'add' 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'

   Reading the weights for type I functions...
   Function 'add': give weight (0,1]: 0.25<ENTER>
   Function 'asin': give weight (0,1]: 0.25<ENTER>
   Function 'sin': give weight (0,1]: 0.5<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.2<ENTER>
   Terminal 'PI': give weight (0,1]: 0.2<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.4<ENTER>

Function 'asin': 1 mutation set pairs
Argument 0:
   F [3 members] = 'add' 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'

   Reading the weights for type I functions...
   Function 'add': give weight (0,1]: 0.25<ENTER>
   Function 'asin': give weight (0,1]: 0.25<ENTER>
   Function 'sin': give weight (0,1]: 0.5<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.2<ENTER>
   Terminal 'PI': give weight (0,1]: 0.2<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.4<ENTER>

Function 'sin': 1 mutation set pairs
Argument 0:
   F [2 members] = 'asin' 'sin'
   T [4 members] = '1' 'PI' 'x' 'y'
```

```
   Reading the weights for type I functions...
   Function 'asin': give weight (0,1]: 0.5<ENTER>
   Function 'sin': give weight (0,1]: 0.4<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 0.6<ENTER>
   Terminal 'PI': give weight (0,1]: 0.4<ENTER>
   Terminal 'x': give weight (0,1]: 0.3<ENTER>
   Terminal 'y': give weight (0,1]: 0.1<ENTER>

Root:
      F [2 members] = 'add' 'sin'
      T [4 members] = '1' 'PI' 'x' 'y'
   Reading the weights for type I functions...
   Function 'add': give weight (0,1]: 1<ENTER>
   Function 'sin': give weight (0,1]: 1<ENTER>
   Reading the weights for type II/III terminals...
   Terminal '1': give weight (0,1]: 1<ENTER>
   Terminal 'PI': give weight (0,1]: 0.2<ENTER>
   Terminal 'x': give weight (0,1]: 1<ENTER>
   Terminal 'y': give weight (0,1]: 1<ENTER>
```

Note: End of Weights Section and start of the Display of Typed Mutation Sets Section

Note: This section displays the typed mutation sets. It shows the valid types of each argument, for every function, along with the functions and terminals of that type which can be used in that argument.

```
The following typed mutation sets are used...

Function 'add': arity 2
   Argument 0
      Type 'float'
         F [2 members] = 'add' 'sin'
         T [2 members] = 'x' 'y'
      Type 'integer'
         F [1 members] = 'add'
         T [1 members] = '1'
      Type 'angle'
         F [2 members] = 'add' 'asin'
         T [1 members] = 'PI'
   Argument 1
      Type 'float'
         F [2 members] = 'add' 'sin'
         T [2 members] = 'x' 'y'
      Type 'integer'
         F [1 members] = 'add'
         T [1 members] = '1'
      Type 'angle'
         F [2 members] = 'add' 'asin'
         T [1 members] = 'PI'
Function 'asin': arity 1
   Argument 0
      Type 'float'
         F [2 members] = 'add' 'sin'
         T [2 members] = 'x' 'y'
      Type 'integer'
         F [1 members] = 'add'
         T [1 members] = '1'
Function 'sin': arity 1
```

```
    Argument 0
       Type 'angle'
          F [1 members] = 'asin'
          T [1 members] = 'PI'
Root:
       Type 'angle'
          F [1 members] = 'add'
          T [1 members] = 'PI'
Send 1 to continue, anything else to quit cgp-lil-gp: 1<ENTER>
```

5 Interface File

The Interface File, described below, uses a simple language to describe the constraints you wish to specify. It takes this information, and converts it into an input file, which CGP then reads from instead of reading from the keyboard during use of the Interactive Interface (Section 4).

5.1 Interface File Overview

There are 4 sections in the Interface File. The first three sections may appear in any order, and may even be repeated. No section has any effect on the other sections. This file must contain at least the End of File Marker. The sections are:

1. Fspec/Tspec Constraints *(optional)*
2. Weight Constraints *(optional)*
3. Type Constraints *(optional)*
4. End of File Marker *(required)*

5.2 Interface File Parameters

Two new parameters have been added to all flexibility in the use of this new interface. They are:

- `cgp_interface` - The file containing the constraint creation instructions.
- `cgp_input` - The file which is to be created from the constraint instructions, if the parameter `cgp_interface` is specified. This file will then act as the input to *CGP lil-gp*.

And, like all parameters, they may be specified in a parameter file or on the command line. Please see the lil-gp Users Manual [3] for further information on this. Depending on how you specify these parameters, several things may happen:

1. Nothing specified - Use Interactive Interface
2. Specify cgp_input only - The specified file is used for the Interactive Interface instead of the keyboard.
3. Specify cgp_interface only - A temporary file is created for the input file, and deleted when finished.
4. Specify both - The interface file is used to create the input file. The input file, may later be used as in option 2.

5.3 Interface File Definitions

- *`funclist`* = list of applicable functions
- *`termlist`* = list of applicable terminals
- *`functermlist`* = list of applicable functions & terminals
- *`arglist`* = list of applicable argument numbers (0...*arity*) (i.e. the arglist for sin() is 1 item long.)
- *`weightlist`* = list of applicable weights (user defined in interface file), $length(weightlist) \leq length(functermlist)$
- *`typelist`* = list of valid types (user defined in interface file)
- *`type`* = a single valid type from *`typelist`*
- *`argtypelist`* = list of the valid type for each argument in a particular instance (length defined by arity; i.e. the argtypelist for sin() is 1 item long.)
- *`null`* = empty list (not actually typed in, just hit the <ENTER> key)

- * = wildcard, include all elements from applicable list. Any item appearing after a wildcard is ignored.
- # = begin of comment. Comments continue until end-of-line. Valid characters are "*#" + space + alpha-numerics.

5.4 Interface File Sections

There are 3 sections corresponding to Fspecs/Tspecs, Weights, and Types. These sections may appear in any order. If a section appears, even if empty, it will override the default behavior of CGP, so care must be taken to ensure that enough information is present to allow CGP to have enough constraint information.

5.4.1 Fspec/Tspec Constraints

The Section Header and Footer must be present if this section is to be used. If this section appears, even if no specs are listed, then any items not appearing in a Tspec will be placed in the appropriate Fspec of the Normal Constraints. If an item appears in both an Fspec and the corresponding Tspec, then the Fspec will take precedents. Once a Spec is specified it cannot be overridden, with the exception of an Fspec overriding a Tspec.

5.4.1.1 Syntax

```
FTSPEC                          (Note: Section Header)
F_(funclist | *) = funclist | * | null
F_(funclist | *)[arglist | *] = functermlist | * | null
T_(funclist | *)[arglist | *] = functermlist | * | null
F_ROOT = functermlist | * | null
T_ROOT = functermlist | * | null
ENDSECTION                      (Note: Section Footer)
```

5.4.1.2 Default Behavior

If this section is omitted, then all FSpecs are left empty and all TSpecs are full. If the section header is given, then all Fspecs and Tspecs default to the empty set. If no TSpecs are entered, this will result in normal constraints being generated with full Fspecs, yielding no possible tree growth.

5.4.2 Weight Constraints

The Section Header and Footer must be present if this section is to be used. If this section appears, even if no specs are listed, then any items not appearing in a Weight specification will be given a weight of 1.0. Any weight may be overridden by specifing a new weight.

Note: if there are fewer elements in *weightlist* than in *functermlist*, the last element in *weightlist* will be used for the remaining elements in *functermlist*.

5.4.2.1 Syntax

```
WEIGHT                          (Note: Section Header)
(funclist | *)[arglist | *](functermlist | *) = weightlist
ROOT(functermlist | *)= weightlist
ENDSECTION                      (Note: Section Footer)
```

5.4.2.2 Default Behavior

Whether this section is used or not, the default behavior is to set all the weights to 1.0.

5.4.3 Type Constraints

The Section Header and Footer must be present if this section is to be used. If this section appears, TYPELIST must be the first item following it. Also, there must be an entry for every function, terminal, and the Root. Once an instance of a function has been specified, it cannot be overridden. However, terminals and the Root may be specified multiple times but the last entry for each will be the one used.

5.4.3.1 Syntax

`TYPE`	**(Note: Section Header)**
`TYPELIST =` *`typelist`*	**(Note: Define valid types)**
`(`*`funclist`*`)(`*`argtypelist`*`) =` *`type`*	
`(`*`termlist`* `|` `*) =` *`type`*	
`ROOT =` *`type`*	
`ENDSECTION`	**(Note: Section Footer)**

5.4.3.2 Default Behavior

If this section is omitted, then only a generic type is used. If the section header is given, then all types, functions, terminals, and the Root are undefined.

5.4.4 End of File Marker

`ENDFILE`	**(Note: Section Footer)**

5.5 Interface File Example

The Interactive Interface Example, Section 4.2, can be duplicated in the Interface File by the following Interface File example.

```
FTSPEC
F_(*)=                     #not required since it's empty
F_(*)[*]=                  #not required since it's empty
F_(sin)[0]=add             #prevent sin(_+_)
F_ROOT=asin                #prevent asin() from being the root
#must specify some TSpecs
T_(*)[*]=*                 #allow all TSpecs
T_ROOT=*                   #allow all functions/terminals for Root
ENDSECTION

WEIGHT
#All unspecified weights default to 1.0
#If I desire to reduce the odds of everything but that which
# I explicitly specify, I could do the following
#(*)[*](*)=0
#ROOT(*)=0
#
#Set the weights for the functions: add asin sin 1 PI x y,
#as the arguments for the add & asin functions.
(add asin)[*](*)=.25 .25 .5 .2 .2 .3 .4
#similarly for the sin function
(sin)[0](*)=.5 .4 .3 .6 .4 .3 .1
ROOT(*)=1                        #not really needed as default is 1.0
ROOT(PI)=.2
ENDSECTION

TYPE
TYPELIST = float integer angle       #list of all valid types
(add)(float float)=float             #add() instances
(add)(integer float)=float
(add)(float integer)=float
(add)(integer integer)=integer
(add)(angle angle)=angle
(asin)(float)=angle                  #asin() instances
```

```
(asin)(integer)=angle
(sin)(angle)=float                    #sin() instance
(1)=integer                           #terminal types
(PI)=angle
(x y)=float
ROOT=angle                            #Root return type
ENDSECTION
```

6 Input File Interface Redirection

If you intend to run multiple experiments, or if the data entry section is overly long, you may wish to create an input file which contains all user inputs and which can be redirected into the program at run-time. At the present time, generation of this file is a manual process, and must match **exactly** what you intend for the input. Work is underway to provide an easier-to-use user interface.

6.1 Interface Redirection File Contents

The Interactive Interface shown in Section 4.2 could be duplicated in an input file as follows. This file can be created from the example in Section 5.5 (Interface File Example). The format is very important, as this file is used instead of the user having to type all of this in. Also, since this file is simply redirected into the program, no form of comments are allowed in it. (Note: The *<ENTER>* is only shown for clarity, and the **(Note: ...)** are not part of the file.

```
0<ENTER>                         (Note: start of Types)
float integer angle<ENTER>             (Note: valid types)
float float float<ENTER>               (Note: add()Types)
integer float float<ENTER>
float integer float<ENTER
integer integer integer<ENTER>
angle angle angle<ENTER>
<ENTER>
float angle<ENTER>                     (Note: asin() Types)
integer angle<ENTER>
<ENTER>
angle float<ENTER>                     (Note: sin() Types)
<ENTER>
integer<ENTER>                         (Note: terminal Types)
angle<ENTER>
float<ENTER>
float<ENTER>
angle<ENTER>                           (Note: Root return Type)
0<ENTER>                         (Note: start of F/TSpecs)
<ENTER>                                (Note: F/T_add Constraints)
<ENTER>
add asin sin 1 PI x y<ENTER>
<ENTER>
add asin sin 1 PI x y<ENTER>
<ENTER>                                (Note: F/T_asin Constraints)
<ENTER>
add asin sin 1 PI x y<ENTER>
<ENTER>                                (Note: F/T_sin Constraints)
add<ENTER>
add asin sin 1 PI x y<ENTER>
asin<ENTER>                            (Note: F/T_Root Constraints)
add asin sin 1 PI x y<ENTER>
0<ENTER>                         (Note: start of Weights)
0.25 0.25 0.5 0.2 0.2 0.3 0.4<ENTER>         (Note: add[0]() Weights)
0.25 0.25 0.5 0.2 0.2 0.3 0.4<ENTER>         (Note: add[1]() Weights)
0.25 0.25 0.5 0.2 0.2 0.3 0.4<ENTER>         (Note: asin[0]() Weights)
0.5 0.4 0.6 0.4 0.3 0.1<ENTER>               (Note: sin[0]() Weights)
```

1 1 1 0.2 1 1<ENTER> ***(Note: Root Weights)***
1<ENTER> ***(Note: end of Weights Section. Finished, enter 1 to continue)***

Note: anything after this point will not be read by the program

6.2 Interface Redirection Example

Once the input file has been created, it is a simple matter to use it. If the file were called `experiment.input` and the executable file is called `gp` then using that file would be a matter of:

```
gp parameters < experiment.input
```

This is the standard way of redirecting a file into a program in Unix and DOS, but there may be other variations on your system. Please refer to your system documentation for further information in this area.

7 Bibliography

[1] Cezary Z. Janikow. *"A Methodology for Processing Problem Constraints in Genetic Programming"*. *Computers and Mathematics with Applications*, Vol. 32, No. 8, pp. 97-113, 1996.

[2] Cezary Z. Janikow & Scott DeWeese, "CGP lil-gp 2.1;1.02 TECHNICAL MANUAL", janikow@radom.umsl.edu

[3] Douglas Zongker & Bill Punch. *"lil-gp 1.0 User's Manual"*. `zongker@isl.cps.msu.edu`.