Along with the [[fitness function|Evaluation#fitness-functions]], grammars are one of the most problem-specific components of the PonyGE2 algorithm. The performance of PonyGE2 can be greatly affected by the quality of the grammar used. When tackling a problem with GE, a suitable BNF (Backus-Naur Form) grammar must initially be defined. The BNF can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared towards the problem at hand.

In GE, a BNF grammar is used to describe the output language to be produced by the system. BNF is a notation for expressing the grammar of a language in the form of production rules.

Each production rule (or "rewrite rule") is composed of a left-hand side (a single non-terminal), followed by the "goes-to" symbol `::=`, followed by a list of production choices separated by the "or" symbol `|`. Production choices can be composed of any combination of terminals or non-terminals. Non-terminals are enclosed by angle brackets `<>`. For example, consider the following production rule:

```
<a> ::= <b>c | d
```

In this rule, the non-terminal `<a>` maps to either the choice `<b>c` (a combination of a new non-terminal `<b>` and a terminal `c`), or a single terminal `d`.

A full grammar is built up of any combinations of such rules.

***NOTE** that all non-terminals must be fully defined in the grammar, i.e. every non-terminal that appears in the grammar must have a full production rule with defined production choices.*

Recursion

One of the most powerful aspects of GE is that the representation can be variable in length. Notably, rules can be recursive (i.e. a non-terminal production rule can contain itself as a production choice), which can allow GE to generate solutions of arbitrary size, e.g.:

```
<a> ::= <a> + b | b
```

The code produced by a grammar will consist of elements of the terminal set `T`. The grammar is used in a developmental approach whereby the evolutionary process evolves the production rules to be applied at each stage of a mapping process, starting from the start symbol, until a complete program is formed. A complete program is one that is comprised solely from elements of `T`.

In PonyGE2, the BNF grammar is comprised entirely of the set of production rules, with the definition of terminals and non-terminals implicit in these rules. The first non-terminal symbol is by default the start symbol. As the BNF grammar is a plug-in component of the system, it means that GE can produce code in any language thereby giving the system a unique flexibility. However, in PonyGE, the most common scenario is for the solutions to be executable Python code.

Writing Grammars

Grammars can have any of the following features:

- Production separators in multiple lines. A single production rule can span multiple lines. For example, the following code is valid:

  ```
  <a> ::= x | y | z
  ```

  The following code is also equally valid:

  ```
  <a> ::= x |
          y |
          z
  ```

- Entire lines of commented text
  - Note that multi-line comments need each line to be commented out.
- Comments at the end of any line,
- Single quotation within double quotation and vice versa, and
- Any characters can be used in quotation, even separators (`"|"`) or angle brackets (`"<"` / `">"`).

These options can make the code more readable as well as maintainable. Importantly, this method of writing BNF grammars is not as error prone as previous techniques used with GE. Example grammars are provided in the `grammars` folder.

Grammars are parsed using regular expressions. Examples on parsing some of grammars can be found here:

- Whole grammar
- Rule
- Production

Special Grammars: expressions vs code

Grammars are inextricably linked to [[fitness functions|Evaluation#fitness-functions]] in GE. The following sections will refer to relevant fitness evaluation documentation.

Expressions

Many users of GP systems will be familiar with regression and symbolic expressions. These expressions are essentially single-line equations that can be evaluated in order to capture their output. The key term here is "evaluate". With many GE applications, phenotypic solutions can be evaluated using Python's built-in `eval()` statement. This statement will take a string input and will attempt to evaluate the output of the string by parsing it into a Python expression, evaluate the expression, and return the output. For example, let us define a local variable `x = 3`. Now, consider the following expression `s`, which is saved as a string:

```
x = 3
s = "5 + x"
```

Note that `s` is a string. If `"5 + x"` were not a string, `s` would merely be stored as the sum of the two numbers, since x is a local variable stored as an `int`.

```
print(x, type(x))

>>> 3 <class 'int'>

print(s, type(s))

>>> 5 + x <class 'str'>
```

Now, we can evaluate this expression in Python in order to capture the output. Using the built-in `eval()` function, we can evaluate the string expression `s` in order to find the output of the proposed expression:

```
output = eval(s)
print(output, type(output))

>>> 8 <class 'int'>
```

Note that the expression evaluated correctly, as `x` was defined as a local variable. When attempting to evaluate the string `s`, Python encountered the variable call `x` within `s`, and since `x` is defined as a local variable, `s` is capable of being evaluated. Knowing this, it is possible to write a grammar which contains references to local and global variables that are either defined or imported into your fitness function. When evaluating your phenotype string, Python will then use these variables if they appear in the string to be evaluated.

The `eval()` statement is used by PonyGE2 to evaluate symbolic expressions in a number of fitness functions, such as those in supervised learning. See the supervised learning fitness functions and grammars for more examples of this functionality.

NOTE *that locally or globally defined variables or functions can be referenced and accessed directly by expressions at the time of evaluation.*

Code

Arguably the most powerful feature of GE is its ability to generate arbitrary phenotype strings in an arbitrary language. In particular, this allows GE to generate *programs* (or pieces of code) in an arbitrary language.

Programs differ from most traditional grammar outputs in a number of ways:

1. Phenotype outputs in the form of code typically span multiple lines,
2. Many languages (e.g. Python) use indentation as a necessary part of the code structure. Others use indentation simply for readability.
3. Whereas expressions are `eval`-able, code must be *executed* in order to exert some phenotypic effects (e.g. performing actions, calling external functions, etc.) or to provide some output returns. (Note that in this documentation we discuss the execution of ***Python*** code specifically. Other languages may require specialised execution environments.)

While the execution/evaluation of the code phenotype (i.e. point 3 above) is handled in the [[fitness function|Evaluation#fitness-functions]], provision has been made in PonyGE2 for these first two points through the use of specialised flags. Saving a grammar file with the extension `".pybnf"` rather than the traditional `".bnf"` will automatically activate additional functionality for code-generating grammars through the use of PonyGE2's Python filter (the Python filter can be found at `utilities.representation.python_filter`). This Python filter is automatically called during the mapping process and converts a single-line phenotype string into a multi-line, correctly indented code string. This code string can then be executed using Python's built-in `exec()` statement (rather than the `eval()` statement discussed in [[expressions|Grammars#expressions]] above).

The most important aspect of Python code grammars is the ability to both generate new lines and to reliably indent and dedent code blocks. Both of these are done through the use of curly brackets and colons.

Indentation

To create a new line and open an indented code block, open a new indentation block with an opening curly bracket followed by a colon: `{:`. To close the previous code block and then de-dent back to the previous indentation level, simply close the previous indentation block with a colon followed by a closing curly bracket: `:}`. An example phenotype code string (generated by a grammar with a `.pybnf` extension) is shown below:

```
s = "def print_arg(arg):{:print(arg):}print_arg("Hello")"
```

Since the grammar file used the `.pybnf` extension rather than the traditional `.bnf` extension, PonyGE2's Python filter is automatically used to parse the phenotype *before* evaluation. The parsed phenotype string then becomes:

```
s = "def print_arg(arg):
         print(arg)
     print_arg("Hello")"
```

This string is a complete program, and can be executed using Python's built-in `exec()` module:

```
exec(s)

>>> Hello
```

***NOTE** that just like normal code, code phenotype strings may need to contain calls to any functions that are defined therein.*

New Lines

Since opening a new indentation block with `{:` creates a new line, one simply needs to create a complete empty indentation block `{::}` in order to create a new line. This can be seen in the following example:

```
s = "arg = 'Hello World!'{::}def print_arg(arg):{:print(arg:}print_arg(arg)"
```

When automatically parsed through PonyGE2's Python filter, this phenotype string becomes:

```
s = "arg = 'Hello World!'
     def print_arg(arg):
         print(arg)
     print_arg(arg)"
```

This string is a complete program, which can then be executed:

```
exec(s)

>>> Hello World!
```

Capturing Output, and Local & Global Variables

While capturing the output of `eval`-able expressions is a simple matter of assigning a new variable to the `eval()` call (as seen [[above|Grammars#expressions]]), capturing the output (if any) of a code string execution is not so simple. Python's `exec()` function can take a dictionary as an optional extra input variable. Unlike with evaluable expression strings, executable code strings cannot directly access local and global variables or functions beyond the core built-in Python library. However, if a dictionary is specified in the `exec()` call, then the code string is executed with full read and write access to the specified dictionary. This has two effects:

1. Any keys defined in the given dictionary will be accessible as local variables to the code string during execution, and
2. Any outputs from executing the code string (including defined function objects and generated variables) will be saved as new key:value pairs in the given dictionary.

For example, let us define a dictionary `d`, with a key `"x"`:

```
d = {"x": 5}
```

Let us now define a code string (for convenience, let's say this string has already been parsed by the Python filter):

```
s = "def square(var):
         return var*var
     ans = square(x)"
```

Note that this code string `s` references a local variable `x`, which is contained in the dictionary `d`. Now, if we execute the code string `s` on its own (i.e. with no dictionary), we get no output or returns:

```
out = exec(s)
print(out)

>>>
```

However, if we execute the code string `s` along with the dictionary `d`, then `s` has access to any local variables defined in `d` at the time of execution (i.e. `x`). Furthermore, any output or variables defined during the execution of `s` will be captured and stored in `d`:

```
exec(s, d)
for key in sorted(d.keys()):
    print(key, type(d[key]), d[key])

>>> ans <class 'int> 25
>>> square <class 'function'> <function square at 0x10ad80048>
>>> x <class 'int'> 5
```

Here, we see that the dictionary `d` has been updated with both the newly defined variable `ans`, and the actual `square()` function object itself. This data can then be accessed and used directly by the main body of code in the rest of the fitness function.

NOTE *that the dictionary `d` above also contains a dictionary __`builtins`__ of built-in core Python modules. We have omitted this from the above example for clarity's sake.*

NOTE *that while it is possible to pass in either the `locals()` or `globals()` dictionaries (or both) during execution of a code string, this can potentially have adverse effects such as the over-writing of existing data. It is safer to retain full control over the execution environment of phenotype code strings by explicitly defining the desired usable variables in a specific execution dictionary.*

Calling predefined functions in grammars

It is possible to call predefined functions in grammars. The requirement is that the function must be available, eg in the `local` namespace, when the solution string is `eval`-ed or `exec`-ed. Built-in functions such as `len` are already available. Other well-known GP functions such as `plog` are defined in `src/fitness/supervised_learning/supervised_learning.py` and are imported in the base fitness function class. Other functions such as `np.sin`, or `cv2` functions, or user-written functions, can be imported by the user in a custom fitness function module.

Variable ranges in grammars

A useful special case is available when writing grammars: ranges of numbers can be specified dynamically in the grammar. This allows for more compact and readable grammars to be written. For example, if a non-terminal `<n>` were to specify all numbers from 0 to 9, the traditional way would be to write:

```
 <n> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A cleaner method has been implemented in PonyGE2, allowing for such ranges to be directly specified using `GE_RANGE:N`, where `N` is an integer. Wherever the PonyGE2 grammar parser finds an instance of `GE_RANGE`, it substitutes it with a list of the appropriate terminal choices of the given range. The example `<n>` above would therefore be the result of using `GE_RANGE:9` in a grammar.

This functionality can be quite powerful and useful, as it allows for grammars to create dynamic ranges of numbers. Furthermore, fitness-function specific optional arguments can be accessed by the grammar in order to generate such dynamic lists. One such example is the use of an `n_vars` attribute within the fitness function. If the grammar parser encounters `GE_RANGE:dataset_n_vars`, and if the fitness function has an `n_vars` attribute (note that `n_vars` must be an integer), the grammar parser will automatically substitute a range of productions equal to the range of the n_vars attribute found in the fitness function. An example use case for this can be seen in the `supervised_learning/supervised_learning.bnf` grammar file and the `supervised_learning.supervised_learning` fitness function, where the `n_vars` attribute is set by the number of columns in the dataset file. Using grammar productions like the following, we can avoid hard-coding the number of independent variables in the grammar:

```
<var> ::= x[<varidx>]
<varidx> ::= GE_RANGE:dataset_n_vars
```

This is further explained in the [[evaluation|Evaluation#dynamic-ranging-of-variables-within-the-grammar]] section.

Grammar Files

All grammars are stored in the grammars folder. Grammars can be set with the argument:

```
--grammar_file [FILE_NAME]
```

or by setting the parameter `GRAMMAR_FILE` to `[FILE_NAME]` in either a parameters file or in the params dictionary, where `[FILE_NAME]` is a the full file name of the desired grammar file including the file extension.

NOTE *that the full file extension (e.g. ".bnf")* ***must*** *be specified, but the full file path (e.g.* `grammars/example_grammar.bnf`*)* ***does not*** *need to be specified.*

A note on unit productions.

Traditionally GE would not consume a codon for unit productions. This was a design decision taken by O'Neill et al [[[O'Neill and Ryan, 2003|References]]]. However, in PonyGE2 unit productions consume codons, the logic being that it helps to do linear tree-style operations. Furthermore, the checks needed for unit productions during the running of the algorithm can add up to millions of checks that aren't needed if we just consume codons for unit productions.

The original design decision on unit productions was also taken before the introduction of evolvable grammars whereby the arity of a unit production could change over time. In this case consuming codons will help to limit the ripple effect from that change in arity. This also replicates non coding regions of genome as seen in nature.

In summary, the merits for not consuming a codon for unit productions are not clearly defined in the literature. The benefits in consuming codons are a reduction in computation and improved speed with linear tree style operations. Other benefits are an increase in non-coding regions in the chromosome (more in line with nature) that through evolution of the grammar may then express useful information.