gparm
A Tool for General Parametrics
User Guide

Author: Z. Todd Taylor
Draft 16 – December 2020

Contents

Quick Start

- Combine an input template containing replaceable parameters with a table of parameter values:

```
gparm -t mytemplate.file -p myparameter.csv
```

- ?????????

About *gparm*

gparm is a program (written in the Perl language[1]) for generating large numbers of input files for other programs. It is designed to facilitate large-scale parametric analysis using software such as EnergyPlus, DOE-2, BLAST, or any other program that reads its input from a plain text (ASCII) file. The name *gparm* is an initialism for **g**eneral **par**a**m**etrics. Although *gparm* is useful for generating text input files for virtually any software tool, we describe it here in the context of building energy simulation.

Although many building energy simulation programs have their own macro processors and/or parametric input mechanisms, *gparm* is often better for large-scale analysis for several reasons:

- *gparm* is generic. You learn its syntax once and can use it to develop parametric inputs to many different programs.
- *gparm* is not limited to an all-combinations parametric set. Some software's parametric input facilities, for example, are limited to simple looping over one or more changeable parameters. With *gparm* you can easily set up more carefully designed analyses that focus on subsets of the full parameter space, that vary multiple parameters in concert, that define dynamic default values for some parameters based on values of other parameters, that correspond to specific instances of energy code requirements or beyond-code program provisions, etc.
- *gparm* interfaces better with other programs. The parameter file used to drive *gparm*, for example, can be directly loaded into a database management system (DBMS) or statistical analysis package (e.g., SAS, R) alongside the simulation results to facilitate widespread sharing of results, statistical analysis, custom graphing, etc.
- *gparm* has a cleaner syntax than most programs' macro languages. Compare this snippet taken from a DOE-2 input file:

```
##set1 cavitydens  #[#[0.6 * ffill[]] + #[0.016 * #[1 - ffill[]]]]
```

  with a similar construct under *gparm*:

```
{
      $cavitydens = 0.6 * $ffill + 0.016 * (1 - $ffill)
}
```

[1] *gparm* runs under any modern implementation of the Perl 5 language. Perl interpreters are available for Windows, Mac, and Linux/Unix operating systems at http://www.perl.org/.

- *gparm* has a more powerful language for customizing input files. *gparm* can handle arbitrary complexity with the facility of a full programming language. Indeed, virtually anything that can be programmed in the Perl language can be incorporated into a *gparm* template in a very readable format.[2]

Requirements

Perl must be installed and accessible on your computer. Most Linux distributions come with Perl preinstalled; Windows and Mac users will probably need to download and install a Perl 5 interpreter from http://www.perl.org/.

Your Perl installation must include two add-on modules: *Text::Template* and *Parallel::ForkManager*. These may or may not be preinstalled with your Perl interpreter; if not, follow the instructions for installing Perl modules from the Comprehensive Perl Archive Network (CPAN).

Electronic spreadsheet software such as Microsoft Excel or LibreOffice Calc can be helpful for building the parameter files used by *gparm*, but is not strictly necessary.

Running *gparm*

Because *gparm* is a Perl script, you run it like you would any other Perl script. It is a command-line tool, so it is used from an interpreter such as a Unix/Linux shell or the Windows Command Prompt tool. The syntax is slightly different between the two. In Unix/Linux, the system knows that *perl* must be used to interpret the script, so you simply name the *gparm* script along with the relevant arguments.[3] For example, if the *gparm* script is in your current directory:

```
./gparm [arguments…]
```

If the *gparm* script is not in your current directory, replace the "." with the actual path to the file. In the Windows Command Prompt tool, you must run *perl* directly, giving the full path to the *gparm* script as the first argument:

```
perl gparm [arguments…]
```

In this document, we assume a Unix/Linux system and assume that the *gparm* script is located in a directory within the user's search path, so the program is invoked without any mention of its location:

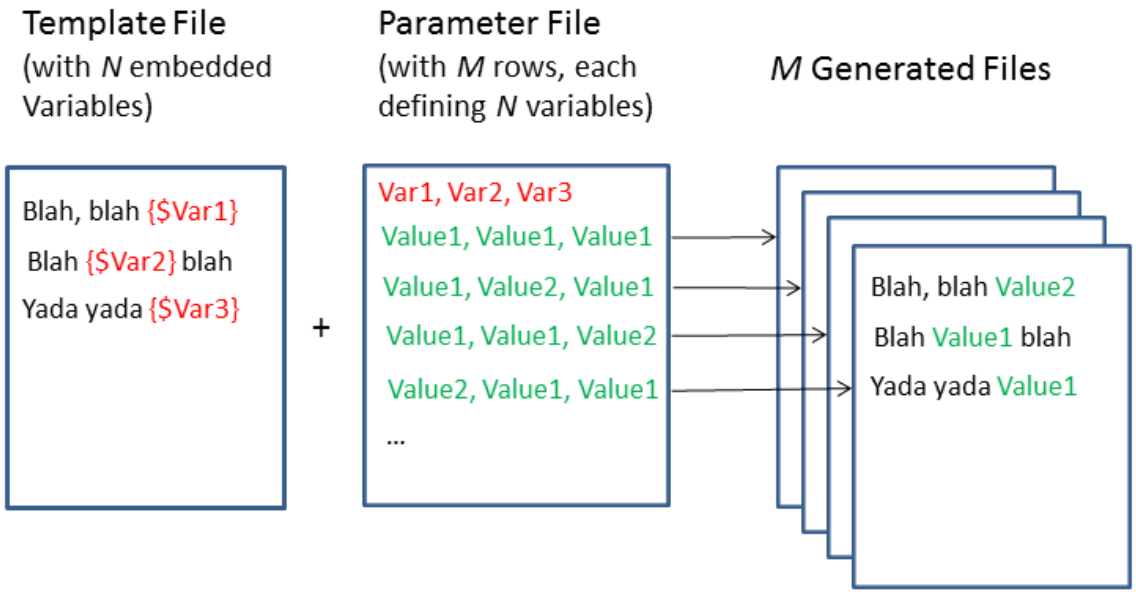
```
gparm [arguments…]
```

[2] You can of course make things unreadable too, but don't do that.
[3] This assumes you've made the *gparm* script executable: `chmod +x gparm`

Summary of how *gparm* works

gparm is somewhat like a generic system for doing a "mail merge," but instead of form letters, you're creating multiple input files for another program. You create two text files: a *template file* that contains replaceable tags (i.e., parameters), and a *parameter file* that defines all the values you want those parameters to take on. *gparm* combines those to generate a bunch of specific files representing all the combinations of the parameters given by the rows of the parameter file (see Figure 1).

FIGURE 1 – Workflow of *gparm*



For example, you might have a simple input file for a mythical program that looks like this:

```
---------- Cut Here ----------
Wall-R = 19
Ceiling-R = 38
Floor-R = 19
---------- Cut Here ----------
```

Suppose you'd like to run that program for all combinations of wall, ceiling, and floor R-values. *gparm* lets you avoid building all the input files by hand. First you'd create a template version of the input file by replacing the specific input values (e.g., "19") with replaceable tags (variable names of your own choosing, such as "Rwall"):

```
---------- Cut Here ----------
Wall-R = {$Rwall}
Ceiling-R = {$Rceiling}
Floor-R = {$Rfloor}
---------- Cut Here ----------
```

Notice that the replaceable tags are all inside curly braces and are prefixed with a dollar sign ($).[4]

Next, create a parameter file that defines the values of `$Rwall`, `$Rceiling`, and `$Rfloor` that you'd like to run. The parameter file is a simple comma-separated value (CSV) file containing a table that defines the parametric runs:

```
---------- Cut Here ----------
ID,Rwall,Rceiling,Rfloor
1, 13, 19, 19
2, 13, 19, 30
3, 13, 30, 19
4, 13, 30, 30

# Six-inch walls from here on down...

5,19,19,19
6,19,19,30
7,19,30,19
8,19,30,30  # Super efficient!
---------- Cut Here ----------
```

That example illustrates several important things about the parameter file:

1. It is simply a CSV file. You can produce it from an electronic spreadsheet if you like, or build it by hand in a text editor, or generate it from another software tool.
2. It should be a clean rectangle of data—all rows having the same number of columns, column headings being single words on a single line, etc.
3. The first row contains column headings that name the parameters. The number of headings must exactly match the number of values in each of the remaining rows.
4. The column headings should include all the parameter names in the template file,[5] without the dollar signs. The column headings should be syntactically legal variable names in the Perl language (case is significant; no spaces, dashes, or special characters; no numbers as the first character; etc.)
5. There is one special column named "ID" that doesn't necessarily appear anywhere in the template file. This column is used to name the various output[6] files that will be generated by *gparm*. The values in this column must be unique

[4] The dollar sign ($) is part of Perl's syntax—it marks the name of a scalar variable (a variable that holds a single value). Perl has other such markers (also known as "sigils"): the "@" character marks a list (array) variable; the "%" character marks a hash (associative array) variable; and the "&" character marks the name of a function. Only scalars are used for `gparm`'s parameters, though templates may make use of other Perl constructs as needed.

[5] The requirement to include all parameter names can be avoided by setting up the template file such that it defines default values for your variables. This is discussed later.

[6] Note that *gparm*'s output is a bunch of files that will be inputs for another program. We will sometimes refer to *gparm*'s output as "generated input files" or simply "generated files."

across all rows; if there are repeated values, some generated files will be overwritten. If there is no ID column, *gparm* will use the first column, regardless of its name, to determine the filenames. Many people prefer to name it "case" or "run" or something similar.[7] The values of the ID column can be simple sequential numbers as in the example above, or they can be any strings you'd like as long as there are no duplicates and they don't contain any characters that would be illegal in filenames on the operating system you're using (i.e., there should be no backslash characters in Windows, no forward slash characters in Unix/Linux, etc.)

6. Parameter values may be padded with leading and/or trailing whitespace if desired for readability; the whitespace will be ignored.
7. Blank lines (and lines with nothing but whitespace) are ignored.
8. The "#" character introduces a comment. Everything from the "#" to the end of the line is ignored.

Once a template file (say, "file.tmpl") and a parameter file (say, "parm.csv") have been created, generating the input files (one for each row in the parameter file) involves a simple call to the *gparm* program:

```
gparm -t file.tmpl -p parm.csv
```

Notice the `-t` and `-p` command-line options that allow you to specify the template and parameter files, respectively. There are several other useful options that will be discussed later. (If you run *gparm* with no options or arguments it will show a brief summary of the available options.)

The *gparm* run above will produce eight new files—one for each data row in the parameter file—named '1', '2', etc., exactly as specified in the ID column of the parameter file. Now it's up to you to run your mythical program on each of those files.

More details and some options

Template file syntax

The replaceable tags in the template file are really just little snippets of Perl code. Each snippet, which must be enclosed in curly braces `{}`,[8] will be evaluated for each row of the parameter file and its value inserted into the generated file. The snippet is usually just a variable name, which evaluates to the corresponding parameter value from the current

[7] Actually, naming this column "ID" turns out to have been a rather unfortunate choice. It seems MS-Excel assumes any text file, the first two characters of which are "I" and "D," is an SYLK file (a mostly undocumented Microsoft format for exchanging data among spreadsheets and a few other Microsoft programs). So if your first column is named "ID" and you try to open your parameter file by double-clicking on it in Windows, Excel will balk and ask you (at least a couple of times) if this is what you really want to do. Persistently answering Yes or OK will open the file, but you can save yourself the grief by naming column one "case" or "filename" or "flibbertigibbet" or anything else besides "ID".

[8] If curly braces conflict with other syntax in the input files you're generating, you can use other characters to delimit the Perl snippets. More on that later.

row of the parameter file. But it doesn't have to be just a variable name. Any legal Perl code can go inside the braces. For example, you might have:

```
---------- Cut Here ----------
Wall-R = {$Rwall}
Ceiling-R = { if ($Rwall < 19) {30} else {38} }
Floor-R = {$Rfloor}
---------- Cut Here ----------
```

There are many other tricks you can use inside the curly braces that won't be elaborated here, but Perl wizards will find the available syntax comprehensive and powerful. One important trick worth mentioning is the `$OUT` variable. If a Perl snippet assigns to a variable named `$OUT`, then the value of that variable, rather than the final value of the whole snippet, is what gets interpolated into the generated file. For example:

```
---------- Cut Here ----------
Wall-R = {$Rwall}
Ceiling-R = {
    if ($Rwall < 19) {
        $OUT = 30;
    } else {
        $OUT = 38;
    }

    if ( $roofdeck eq "insulated" ) {
        $OUT += 5;
    }
}
---------- Cut Here ----------
```

In the above example, the value of Ceiling-R could be 30, 35, 38, or 43, depending on the values of `$Rwall` and `$roofdeck`. Obviously, "OUT" should never be used as one of your parameter names.

Control over filenames

You can control how *gparm* names all of the files it generates with the `-f` option. This option allows specification of a "filename mask" that *gparm* will combine with the values from the ID column. For example:

```
gparm -t file.tmpl -p parm.file -f '{}.idf'
```
[9]

[9] In the Unix/Linux shells, the curly braces { } have to be within single or double quotes. This is not required in Windows (indeed, if used there, the generated files will have literal quote characters in their names).

The little pair of curlies will get replaced with the current row's ID value to create the filename. For example, if your ID column is just sequential integers like in the example above, you'll get files named `1.idf`, `2.idf`, `3.idf`, etc.

Where to create the generated files

For really big analyses, it might be advantageous to put all the generated files somewhere other than in the directory from which you run *gparm* (and which probably contains your template file and parameter file). The `-d` option does that:

```
gparm -t file.tmpl -p parm.file -d /my/other/dir
```

In this case *gparm* will generate filenames exactly as before, but will put them in `/my/other/dir` instead of your current directory.

Assigning default values to variables in the template file

It is often handy to include parameters in a template file that are not always (and maybe only rarely) modified by a parameter file. Your template file can assign default values to these variables so that they needn't be defined in your parameter file unless you specifically desire to modify them. Some situations where this is useful include the following:

- A parameter is left completely out of the parmfile. This is probably the most common situation. Many detailed simulation templates offer the ability to control dozens or even hundreds of parameters through the parmfile, but nobody wants to maintain a parmfile with hundreds of columns in it unless those columns are being actively varied in a particular analysis. Indeed, carrying along all those unchanging columns is an error-prone practice that should generally be avoided. By defining (in the template file) default values for many/most/all of the parameters, the parmfile for a particular analysis needs only as many columns as are being varied in the particular experiment.
- A parameter is included in the parmfile, but its value is explicitly set only in a subset of the rows. This is a common use case wherein the user wants to rely on the template's default value except in certain rows where the parameter is set to an alternate value.
- A parameter is usually set though a higher level parameter that manages a group of related parameters. For example, there may be a parameter called `code` that tells the template what energy code to assume (its values might be "`IECC_2009`", "`IECC_2012`". etc.). Based on the value of the `code` parameter, the template file can set default values for a plethora of other parameters (envelope component R-values, glazing U-factors, leakage rates, lighting power densities, etc.). In practice, this is really just a special case of one of the first two situations.

Establishing default values in your template file can be accomplished in several ways. The Perl language offers a variety of basic capabilities that can be employed to set

defaults, and *gparm* itself provides a couple of specialized functions to make the most common things easy. As with everything Perlish, there's more than one way to do it[10] and you can use any of those ways in your templates. The recommended way is number 5 below, but we'll show several others to illustrate the flexibility available:

```
---------- Cut Here ----------
{ # Snippet to set default values…

     $OUT = '';  # Suppress output from this snippet…

     # Method 1:  the most verbose way to do it…
     if (!defined($Rwall) || $Rwall eq '') {
          $Rwall = 13;
     }

     # Method 2:  another way, less verbose…
     $Rceiling = 30 unless defined($Rceiling) &&
                           $Rceiling ne '';

     # Method 3:  the shortest but dangerous way…
     # (The "||=" is usually pronounced "or equals")
     $Rfloor ||= 19;

     # Method 4:  a functional way (gparm-specific)…
     $Uwindow = val_or($Uwindow, 0.32);

     # Method 5:  the one you should usually use…
     setdef($Rslab, 5);
}

Wall-R = {$Rwall}
Ceiling-R = {$Rceiling}
Win-U = ${Uwindow}
Floor-R = {$Rfloor}
Slab-R = {$Rslab}
---------- Cut Here ----------
```

Notice several things about how we set default values:

1. We use a separate Perl snippet to assign default values.[11] You can have multiple such snippets in a template; they can appear anywhere in the template before the variables being defaulted are used.
2. We arrange for that separate snippet to evaluate to the empty string (by setting the `$OUT` variable) so the snippet introduces nothing into the generated file(s).[12]

[10] That's actually one of Perl's unofficial mottos.
[11] This is not strictly required but is a good practice for readability and maintainability.

3. We show several ways to set a default value.
    a. The first method is the most verbose, but the most flexible and the most readable to someone not versed in Perl. You can add arbitrarily many arbitrarily complex conditionals to determine how to set the default. In the example, the conditional checks whether the variable has been defined and whether it holds the empty string. This is usually what you want because when a parameter is completely missing from the parmfile, its value in your template file will be undefined, and when a parameter value is missing in the parmfile, it shows up in *gparm* as the empty string. So there must be explicit checks for both cases.
    b. The second way is less verbose and, because it fits more naturally on one line, easier to read when multiple variables are being assigned default values. But it uses a Perl idiom that may be unfamiliar to some readers.
    c. The third (shortest) way uses yet another Perl idiom, and is unsafe if zero (0) is a valid input for the variable. However, because of its brevity and readability, it is often used for parameters that are never expected to be assigned a zero value. Just be sure you never use it for parameters that might need to take on a zero value.[13]
    d. The fourth way uses a special *gparm*-provided function `val_or` that returns its first argument if it holds a defined value other than the empty string, or its second argument if not. See method 1 for more discussion on empty strings. The `val_or` function is generally unnecessary unless you need to use Perl's "symbolic references" in your template. Those are rare bits of Perl wizardry that most users will never employ.
    e. Finally, the last method is the one you should use 99% of the time. It uses a *gparm*-specific function `setdef` (or its more verbose alias `set_default`) to set a variable to the default value unless that variable already holds a defined value other than the empty string. It allows a variable to be set to any value except the empty string, including zero. As discussed above, empty strings are disallowed because leaving a value blank (missing) in the parameter file results in an empty string value when the parmfile is read in by *gparm*. We usually want those missing values to be replaced with the default value. If you need a variable to be able to take the empty string as a legal value, use something like method 1 above to deal with defaults or else use method 5 and set the default itself to the empty string.

Warning! Setting default values is conceptually simple, but can lead to unnoticed errors if you're not careful. Various parts of *gparm* templates contain different syntaxes—for example, Perl, EnergyPlus, and the vagaries of *gparm* itself are usually encountered in a single template—so you should be careful to write template text that is clear and unambiguous. Here are a few default-setting best practices:

---

[12] Actually, as written it would insert a single blank line into each generated file.

[13] And for this purpose, an empty string is the same as a zero value.

- (Almost) always use the `setdef` function (or its verbose `set_default` alias) to set default values. It works for any kind of variable except those for which the empty string is a valid input, and makes your intent clear.
- Feel free to use the `||=` operator for variables that take only non-zero and non-empty-string values. This is the exception to the previous bullet. This operator is both concise and very readable. Its only downside is that it fails for variables that can have valid zero or empty inputs. It's probably best not to use it for numeric inputs.
- Set default values only in dedicated Perl snippets that do nothing else and are marked as such with comments. Another way of saying this is don't bury default settings deep down in the syntax of the application you're building input files for (e.g., EnergyPlus). For example:[14]

  Poor practice:

```
---------- Cut Here ----------
Material,
    Lumber_2x4,               !- Name
    Rough,                    !- Roughness
    0.0890016,                !- Thickness {m}
    [setdef($cond, 0.11546)],!- Conductivity {W/m-K}
    [setdef($density, 513.)],!- Density {kg/m3}
    767.58,                   !- Specific Heat {J/kg-K}
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
---------- Cut Here ----------
```

  Better practice:

```
---------- Cut Here ----------
[ # DEFAULT VALUES FOR WOOD PROPERTIES
    $OUT = '';  # Produce no output…
    setdef($cond, 0.11546);
    setdef($density, 513.);
]
Material,
    Lumber_2x4,               !- Name
    Rough,                    !- Roughness
    0.0890016,                !- Thickness {m}
    [$cond],                  !- Conductivity {W/m-K}
    [$density],               !- Density {kg/m3}
    767.58,                   !- Specific Heat {J/kg-K}
    0.9,                      !- Thermal Absorptance
    0.7,                      !- Solar Absorptance
    0.7;                      !- Visible Absorptance
```

[14] Note that this example uses EnergyPlus syntax, in which curly braces have meaning, so the snippet delimiters must be square brackets.

```
---------- Cut Here ----------
```

Breaking up large templates

If your template file is large and complex or you need to reuse part of one template file in others, you can organize things using an include-file mechanism that is built into *gparm*. Actually, there are two such mechanisms.

The `include` function includes another file *exactly as it is* into the current template. That is, it puts the included file into each generated file without scanning for any code snippets in the included file. For example:

```
---------- Cut Here ----------
Wall-R = {$Rwall}

{ include('/path/to/it/my_common.txt') }

Floor-R = {$Rfloor}
---------- Cut Here ----------
```

The `process` function does the same thing, but processes the included file through *gparm* as it brings it in. That is, rather than inserting the included file directly into each generated file, *gparm* first scans for any code snippets in the included file, evaluates them, and inserts the filled-in results into each generated file. E.g.,

```
---------- Cut Here ----------
Wall-R = {$Rwall}

{ process('/path/to/it/my_subset.tmpl') }

Floor-R = {$Rfloor}
---------- Cut Here ----------
```

The `include` function is like processing the main template file first, then inserting the included file without doing any processing on it. The `process` function is like inserting the included file into the main template file first, then processing the whole resulting template.

Saving space

If your template files are really big (e.g., complex EnergyPlus IDF files) and really numerous, you might want to have *gparm* compress them as it generates them. The `-z` option will do that. Then, instead of generating thousands of `*.idf` files, *gparm* will generate thousands of `*.idf.gz` files, for example.[15]

[15] The compression option does not work on Windows or Mac because those operating systems do not reliably have a mechanism for doing file-by-file compression.

Multiprocessing for speed

A big, complex template file containing lots of complicated Perl snippets can take a while to fill in with each row of the parameter file. If you're generating many thousands of such files, you may want to take advantage of a multicore machine by telling *gparm* to break its work into multiple chunks that run in parallel. This is done with the `-n` option to *gparm*.

```
gparm -t file.tmpl -p parm.file -n 12
```

That command would run 12 separate *gparm* processes, each working on a subset (one-twelfth) of the parameter file. Of course, this would be of benefit only if the computer had (at least) 12 CPU cores. The rows of the original parameter file are assigned to the twelve subprocesses in a round robin style.

Persistent Storage

Occasionally for a complex template it can be advantageous to store voluminous data in external files. For example, all the complex requirements of a building code might be stored in a CSV file that gives various component requirements as functions of climate zone, building type, foundation type, etc. It's straightforward to write Perl code to read such data, but without special handling, the data will have to be re-read for every row of the parmfile, which is very slow for large parmfiles. *gparm* exports a single Perl hash table named *%DATA* that you can use to store data you'd like to persist across parmfile rows.[16]

For example, suppose you have a large table of code requirements in a CSV file `requirements.csv` that looks something like the following.

```
---------- Cut Here ----------
component,zone,r_value
roof,zone1,30
…
roof,zone8,60
wall,zone1,13
…
wall,zone8,23
…
---------- Cut Here ----------
```

You could put a snippet into your template that reads the requirements data from the CSV file if it doesn't already exist in persistent storage, and does nothing if it already exists:

```
---------- Cut Here ----------
```

[16] Ordinarily *gparm* is careful to reset all user data before processing each parmfile row to ensure nothing in your template accidentally breaks the parameter defaults defined by your template.

```
{
     $OUT = '';  # No output from this snippet

     unless ($DATA{requirements}) {
          $DATA{requirements} = << Code to read CSV >>
     } else {
          # Do nothing
     }
}
---------- Cut Here ----------
```

Thereafter you can access the requirements data with something like the following (depending on how your CSV-reading code formats its result):

```
my $rval = $DATA{requirements}{roof}{zone8}
```

Or, use it to set default values for parameters:

```
setdef($r_roof, $DATA{requirements}{roof}{zone8});
```

Special variables available in the template file

In addition to your parameter file column names, there are several other special Perl variables available for your use in the template file:

- `$IDFILE` – Holds the name of the filled-in file that will be generated by the current row of the parameter file. If you haven't told it otherwise, `$IDFILE` is identical to what's in the ID column of the parameter file. If you've told it otherwise (using the `-f` option), it'll be that. This variable is handy to include somewhere in your template file as documentation of which parm row the output file represents. It will usually go in a comment (as defined by whatever input syntax you're working with). In EnergyPlus, for example, you might put something like this near the top of your template:

  ```
  !
  ! Parameter file row: {$ID}
  ! This file: {$IDFILE}
  !
  ```

- `@FIELD_NAMES` – A Perl list (array) of all your parameter file column names, in the order they appear in the parameter file.
- `%FIELDS` – A Perl hash (associative array) that holds all the inputs defined in the current row of the parameter file. These are redundant with the individual scalar variables. For example `$FIELDS{Rwall}` is the same as `$Rwall`. The `%FIELDS` hash is handy when you want to write out an exhaustive list of each run's parameters—in a comment block at the top of each generated file, for example:

  ```
  ---------- Cut Here ----------
  {
  ```

```
        # Show all parameters as comments…

        $OUT = '';
        foreach my $p (@FIELD_NAMES) {
             $OUT .= "# $p: $FIELDS{$p}\n";
        }
}
Wall-R = {$Rwall}
.
.
.
---------- Cut Here ----------
```

In that example, the values of all parameters on the current row of the parameter file would be printed as comments in the generated file, something like:

```
# Rwall: 13
# Rceiling: 30
# Rfloor: 19
```

The "#" character should be replaced with whatever character syntactically marks a comment in the program for which you're generating input files ("$" in DOE-2, "!" in EnergyPlus, etc.)

- `%DATA` — A hash table that will retain its contents across parmfile rows. You can store any data of your choosing in the hash. It is mainly useful for storing data read from an external file so it doesn't get re-read for every parmfile row. Simply give each bit of data you want to keep a name of your choosing. Each data element can be any scalar value, including a reference that points to a complex data structure. E.g.,

```
$DATA{twelve} = 12;
$DATA{colors} = {red => '#FF0000',
                 blue => '#0000FF'};
$DATA{big} = hashref_from_simple_csv("dat.csv");
```

Functions available in the template file

All built-in Perl functions are available for use within snippets in the template file. Additionally, there are several other functions that either come from external modules pre-loaded by *gparm* or are custom written within *gparm*. These functions are:

- `include($filename)`: See above.
- `process($filename)`: See above.
- `in_set($myval, @target_set)`: Boolean test to see if a candidate value is in a target set. Use it when you want to ask if your value is equal to any of a set of values.

E.g.,

```
if (in_set('myvalue', @set_of_targets))
      { ...blah... }
```

E.g.,

```
if (in_set($Rwall, (11, 13, 15)) {
    # Deal with 4-inch walls...

} else {
    # Deal with 6-inch walls...
}
```

- `ceil($value)`: Like the C function of the same name.
- `floor($value)`: Like the C function of the same name.
- `round($value, $n_places)`: Round to a specified number of places. For example:
  - `round(123.4567, 2)   # 123.46`
  - `round(123.4567, 1)   # 123.5`
  - `round(123.4567, 0)   # 123`
  - `round(123.4567, -1)  # 120`
  - `round(123.4567, -2)  # 100`
- `signif($value, $n_digits)`: Round to a specified number of significant digits. For example:
  - `signif(123.056780, 2)  # 120`
  - `signif(123.056780, 3)  # 123`
  - `signif(123.056780, 4)  # 123.1`
  - `signif(123.056780, 5)  # 123.06`
  - `signif(123.056780, 6)  # 123.057`
  - `signif(123.056780, 7)  # 123.0568`
  - `signif(123.056780, 8)  # 123.05678`
- `is_numeric($value)`: Boolean test whether `$value` is a syntactically valid number. For example:
  - `is_numeric("123")   # True`
  - `is_numeric("R-13")  # False`
- `fmt($width, @text)`: Simple paragraph formatter. E.g., to get a nicely wrapped paragraph of text that'll fit on an 80-character screen:
  - `fmt(72, @my_paragraph_in_an_array)`
- `fmt_indented($width, $initial_indent, $subsequent_indent, @text)`: Paragraph formatter that lets the first line of the paragraph have different indenting than the rest of it. For example:
  - `fmt_indented(72, "\t", "", @txt) # indent`
  - `fmt_indented(72, "", "\t", @txt) # hanging indent`
- `min(@values)`: Minimum of a set of values.
- `max(@values)`: Maximum of a set of values.
- Trigonometric functions:

  - sin($radians)
  - cos($radians)
  - tan($radians)
  - asin($value) # returns radians
  - acos($value) # returns radians
  - atan($value) # returns radians
  - deg2rad($degrees) # returns radians
  - rad2deg($radians) # returns degrees
  - dsin($degrees)
  - dcos($degrees)
  - dtan($degrees)
  - dasin($value) # returns degrees
  - dacos($value) # returns degrees
  - datan($value) # returns degrees
  - pi # function returning a constant (3.14159...)

- deep_copy($thing): Returns a reference to a complete (recursive) copy of $thing, including all subelements therein.
- Dumper($thing1, $thing2, ...): Returns a printable dump of the objects given as arguments. It's exactly the Dumper function exported by the Data::Dumper module. Handy for debugging.
- setdef($thing, $default_value): Handy for giving parm variables default values. This function modifies the variable given as its first argument to equal $default_value unless that variable already has a defined value other than the empty string. (This function has another name (set_default) that you can use if you want your templates to be more readable to newcomers.) For example, to give parm variable $n_stories a default value of 1 if there was no value given in the parmfile:
  - setdef($n_stories, 1);
- val_or($thing, $default_value): An alternative to setdef for giving parm variables default values. Unlike setdef, val_or does not modify its argument. The val_or function simply returns its first argument if that is a defined value other than the empty string; otherwise it returns the second argument. The only reason to use the more verbose val_or instead of setdef is if you need to use a Perl "symbolic reference" in your template. (This is a rare thing and should only be used if you know why you're using it.) Here are two examples that do the same thing, one using a normal Perl variable and the other using a symbolic reference:
  - $n_stories = val_or($n_stories, 1);

  - $what = 'n_stories';
  - $$what = val_or($$what, 1);
- hashref_from_simple_csv($csvfile): A utility function to read simple CSV data into your template file Perl interpreter. See discussion in the section above.

- `globals()`: Returns a reference to a hash holding all the *scalar* variables (and their values) known to gparm at the time the function is called. Its purpose is to aid in debugging complex templates. Typically, you would call `globals` at the top of a template and store the result, then call it again at the bottom of the template so that a report on what was changed by the template can be generated. That report is generally produced by the `globals_report` function (see next).
- `globals_report($hashref1, $hashref2, $prefix)`: Generates a textual report showing two things. First it lists all scalar variables that were changed from their original parmfile values during fill-in of the template, showing both the original and new values. Second, it lists all scalar variables that were created anew during fill-in of the template. The latter are usually either parm variables taking default values (i.e., not listed in the parmfile) or temporary variables used in the template that should be changed to lexical (`"my"`) variables.[17] The `$prefix` parameter allows specifying a comment character or string to be prepended to every line of the report (defaults to `"! "`, to make readable EnergyPlus comments). Typical usage:

```
---------- Cut Here ----------
{
     # At top of template…

     $OUT = '';
     $vars_top = globals();
}
.
.
.
{
     # At bottom of template…

     $vars_bottom = globals();
     $OUT = globals_report(
          $vars_top, $vars_bottom, "! "
     );
}
---------- Cut Here ----------
```

Alternative snippet delimiters

If you're creating input files for a program that uses curly braces `{}` as part of its input syntax, you'll need to replace *gparm*'s default delimiters with something else. For example, EnergyPlus input (IDF) files use curly braces to mark the labels for units (e.g.,

[17] There will also be various internal Perl variables unrelated to the template; these may occasionally be illuminating if something unusual is happening (error messages, etc.).

`{gallons}`), so you'll have to change *gparm*'s delimiters from `{}` to, say, `[]`. You do that with the `-o` (open_string) and `-c` (close_string) options. E.g.,

```
gparm ...  -o '['  -c ']'
```

The open and close strings can consist of more than one character if needed to avoid conflicting with your program's syntax:

```
gparm ...  -o '(--'  -c '--)'
```

In that case your template snippets would look like:

```
---------- Cut Here ----------
Wall-R = (-- $Rwall --)
Floor-R = (-- $Rfloor --)
---------- Cut Here ----------
```

The delimiters don't have to include any of the "balanced" grouping characters such as parentheses and braces. So if you really need it, you can make the open_string and close_string virtually any character sequences you like:

```
gparm ...  -o 'Here we go:'  -c ':that is all.'
```

Hence (ugh!):

```
---------- Cut Here ----------
Wall-R = Here we go: $Rwall :that is all.
Floor-R = Here we go: $Rfloor :that is all.
---------- Cut Here ----------
```

Error processing

Because *gparm* snippets are Perl code, it's entirely possible during template development that you'll have bugs in your template. For example, consider the following snippet:

```
---------- Cut Here ----------
Wall-R = {$Rwall}
Ceiling-R = { if ($Rwall < 19) {30x} else {38} }

##### Syntax error here ----------^

Floor-R = {$Rfloor}
---------- Cut Here ----------
```

Your typo ("30x" instead of "30") will cause Perl to emit a syntax error message when *gparm* tries to fill in your template. When that or any other kind of error happens, *gparm* will insert some (hopefully) helpful messages into the generated file it was working on at

the time and will try to tell you where to look to find the problem. In the example above when you run *gparm*:

```
gparm -t file.tmpl -p parm.csv -f 'test.{}'
```

you'll get a message something like:

```
Filling in main template failed on row 10 of parm file
Take a look in ./test.1 for more information
Died at /tools/bin/gparm line 414, <PARM> line 10.
```

If you follow the instructions and look in the generated file *test.1*, you'll see:

```
---------- Cut Here ----------
Wall-R = 13
Ceiling-R = ------------------------------------…
ERROR 1:
    syntax error at file.tmpl line 6, at EOF


The offending code fragment starts at about line 6:
 if ($Rwall < 19) {30x} else {38}

END ERROR 1
-------------------------------------------------…

Floor-R = 19
---------- Cut Here ----------
```

That's hopefully enough direction to help you track down the problem and fix it. Just remember that because you're mixing *gparm* syntax, Perl syntax, and your target program's (e.g., E+'s) syntax, typos can sometimes be hard to figure out. So don't expect *gparm*'s helpful hints to always be completely illuminating.