# Neobem

Mitchell T. Paulus

Version 0.9.0, January 9, 2025

**Abstract**

Neobem is a preprocessor language for compiling to the input files for building energy simulation programs, such as the idf file syntax expected by EnergyPlus.

## Introduction

While creating your building energy simulation input files, have you ever wanted to:

- Use variables?
- Break out sections into custom templates?
- Use arithmetic?
- Build objects directly from Excel or JSON data?
- Loop over lists?
- Easily incorporate work from others?
- Do all this without setting up a full Python, R, or Ruby working environment?

If so, Neobem is what you have always wanted.

At its core, Neobem is a command line application that follows the Unix principle of doing one thing and doing it well. That thing is compiling an expressive programming syntax into building energy simulation input.

It is a Unix filter program, taking input via standard input or a file and writing the results to standard output. It is designed to play one role in larger processing pipelines.

I hope you find it as useful as I do.

## Getting Started

### Installation

#### Quick Instructions

1. Download program file from GitHub.
2. Add directory location containing program file to `PATH` environment variable or symlink the executable `nbem` to a location in `PATH`.
3. Execute `nbem` in shell or command interpreter.

See below for additional details on these steps.

#### Download Program

The latest release of Neobem is on GitHub, at https://github.com/mitchpaulus/neobem/releases. There you will see zips containing a compiled executable for various operating systems and CPU architectures.

1. linux-arm64.zip
2. linux-arm.zip
3. linux-musl-x64.zip
4. linux-x64.zip
5. osx-x64.zip
6. win-arm64.zip
7. win-arm.zip
8. win-x64.zip
9. win-x86.zip

Download the zip file that matches your operating system and architecture.[1] For most people, this will be `win-x64`, `linux-x64`, or `osx-x64`.

The zip file will contain a single self contained executable. Extract that file from the zip file to a location that you will want the program to live. It doesn't really matter where you put it, but recommended places would be:

- `C:\Program Files\neobem\neobem.exe` on Windows
- `/usr/local/bin/nbem` or `~/.local/bin/nbem` on Linux

Neobem is a console or command line application. It is meant to be run from a shell environment, that could be anything like[2]:

- `cmd.exe` or PowerShell on Windows
- `bash`, `zsh`, or `fish` running in any terminal emulator, such as:
  - Windows Terminal
  - Terminal.app
  - iTerm2
  - Gnome Terminal
  - Alacritty
  - Konsole
  - Terminator

### Add Program Location to PATH Variable

Once the program files are installed in your preferred location, you will want to add the folder to the `PATH` environment variable (if the location you put it in isn't already there). Another option is to symlink the executable to a location that is already in the `PATH` variable. Here's a link for creating symlinks in Windows, and another link for creating symlinks in Linux.

**Windows** On Windows, you can get to the dialog box to change the `PATH` variable by doing a search for 'Edit System Environment Variables'. There are several biog posts on the Internet that can guide you through this with screenshots, here are a curated few:

- https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/
- https://www.howtogeek.com/118594/how-to-edit-your-system-path-for-easy-command-line-access/

Here's a post on Superuser answering the question, "What are PATH and other environment variables, and how can I set or use them?"

**Linux/OSX** Setting the `PATH` variable is most often done in the initialization of the particular shell that you are using. The default shell on many systems is `bash`. To add a location to the `PATH` every

[1]If you are running the Windows Subsystem for Linux within Windows, I would recommend the Linux version, and installing like a Linux program

[2]If none of this makes sense, take a look at this link or other web searches for 'terminal vs. shell'

time bash is invoked, you follow the steps here. You add the location to the existing `PATH` variable in the `.bash_profile` or `.bashrc` initialization file, making sure it is exported.

If you are using a different shell, you already likely know how to add locations to the PATH, but for example the syntax for fish (the interactive shell I personally use), the syntax looks like:

```
set -gxp PATH "/path/to/directory"
```

## Creating Neobem Input Files

The input files are simple text files - you can use any editor of choice to create them. Here's a list of popular text editors that you might want to try. If you've never heard of a "text editor", I'd begin with Visual Studio Code, Sublime Text, or Atom.

**Cross-Platform:**

1. Visual Studio Code
2. Sublime Text
3. Atom
4. Vim
5. Neovim
6. Kate
7. Emacs

**Windows:**

1. Notepad++
2. Notepad - yes, that Notepad built into Windows

**Linux:**

1. gEdit
2. Nano

## Execute the Program

On Windows, the program is called `nbem.exe`. On Linux and OSX, it is just `nbem` with no extension.

From the shell, you can test that things are working by running the command with the help argument like:

```
mp@mp-computer:~$ nbem -h
```

on Windows:

```
C:\Users\mpaulus> nbem.exe -h
```

If things are working correctly, you should see help text like:

```
mp@mp-computer:~$ nbem -h
USAGE: nbem [options..] [input file]
Compile Neobem file to EnergyPlus or DOE-2 input files.

With no [input file], input is read from file named 'in.nbem' in the
current directory. If the input file is '-', input is read from standard
input rather than from a file.
```

```
OPTIONS:

    --doe2             Parse input file in DOE-2 Building Description Language format
-h, --help             Show this help and exit
-f, --fmt              Format file instead of compiling
-o, --output <filename> Output file name. Output is printed to standard output by default.
    --tokens           Print lexed tokens for debugging
    --tree             Print parse tree in Lisp format for debugging
-v, --version          Print version number and exit
```

In general, you will call the execute the program `nbem`, passing in your Neobem input file as an argument.

To compile a Neobem file to an idf file, execute a command like

```
mp@mp-computer:~$ nbem in.nbem
```

where `in.nbem` is the relative path to the file you want to compile. In the example above, this would be the `in.nbem` file in my home directory ('~'). By default, the compiled output is printed to standard output, which you will see on the screen. To put the output into a file, either specify the file path as a option, or redirect the output in the shell.

**Using Option:**

```
mp@mp-computer:~$ nbem -o output.idf in.nbem
```

**Using Redirection:**

```
mp@mp-computer:~$ nbem in.nbem > output.idf
```

Please see this screencast link that shows an example workflow from start to finish. This particular workflow example used `bash` as the shell and the files were edited with Neovim.

## Tutorials

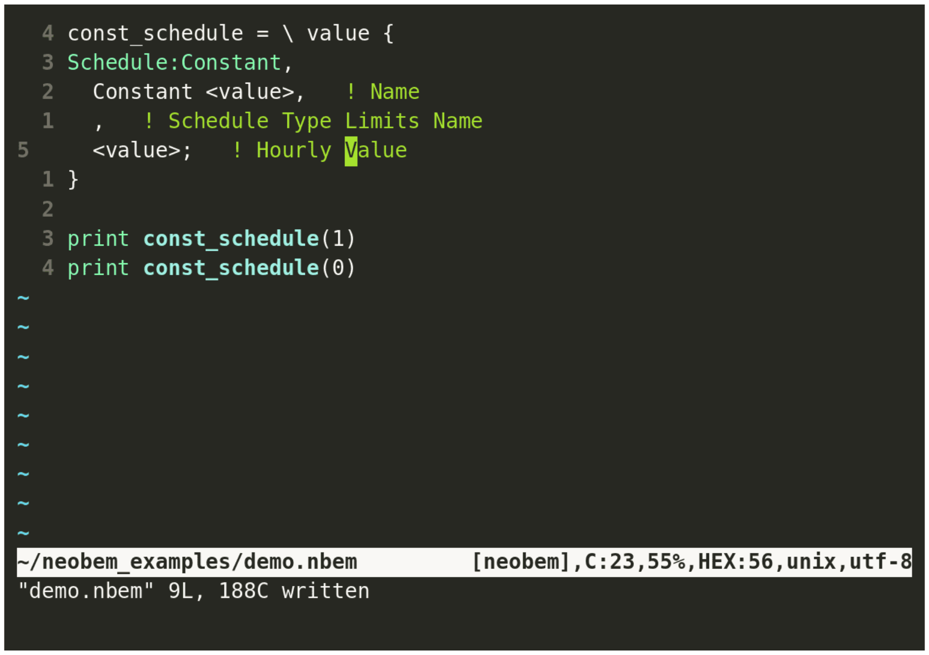
Instead of starting with the boring details about installation and reference material, let's jump right into the good stuff. This will focus on the basics, giving you a general understanding of what this tool is all about, without worrying too much about the details.[3]

### Tutorial 1: Variables

You are building your EnergyPlus input file using conventional tools, and you end up with objects like so:

```
Zone,
  Atrium,        ! Name
  0,             ! Direction of Relative North {deg}
  0,             ! X Origin {m}
  0,             ! Y Origin {m}
  0,             ! Z Origin {m}
```

[3]Inspiration from the TikZ-PGF manual.



```
const_schedule = \ value {
Schedule:Constant,
  Constant <value>,    ! Name
  ,    ! Schedule Type Limits Name
  <value>;    ! Hourly Value
}

print const_schedule(1)
print const_schedule(0)
~
~
~
~
~
~
~
~
~
~/neobem_examples/demo.nbem          [neobem],C:23,55%,HEX:56,unix,utf-8
"demo.nbem" 9L, 188C written
```

Figure 1: Sample screenshot from demo at: asciinema.org.

```
  1,              ! Type
  1,              ! Multiplier
  autocalculate, ! Ceiling Height {m}
  autocalculate, ! Volume {m3}
  autocalculate, ! Floor Area {m2}
  ,               ! Zone Inside Convection Algorithm
  ,               ! Zone Outside Convection Algorithm
  Yes;            ! Part of Total Floor Area


Schedule:Constant,
  Atrium Schedule, ! Name
  Atrium Limits,    ! Schedule Type Limits Name
  1;                ! Hourly Value
```

Do you see all those `Atrium`s? If you're a programmer, you know those can be abstracted out to a variable. If you ever want to change the name, you can easily do it in one place. How do you do it? Like so:

```
atrium_name = 'Atrium'

Zone,
  <atrium_name>, ! Name
  0,              ! Direction of Relative North {deg}
  0,              ! X Origin {m}
  0,              ! Y Origin {m}
  0,              ! Z Origin {m}
  1,              ! Type
  1,             ! Multiplier
  autocalculate, ! Ceiling Height {m}
  autocalculate, ! Volume {m3}
  autocalculate, ! Floor Area {m2}
  ,              ! Zone Inside Convection Algorithm
  ,              ! Zone Outside Convection Algorithm
  Yes;           ! Part of Total Floor Area


Schedule:Constant,
  <atrium_name> Schedule, ! Name
  <atrium_name> Limits,   ! Schedule Type Limits Name
  1;                      ! Hourly Value
```

A few pieces of syntax to see here:

1. Variable declaration

   `var_name = <expression>`

   One thing to note about the variable name. It *must* begin with a lowercase alphabetical character. This is to help differentiate between variable declarations and regular Object declarations.

2. Expression placeholders

   `< <expression> >`

   In our Object declaration, we can insert expressions anywhere by putting them in angle brackets ‘< >’. If you really need to have a angle bracket, you can escape them by doubling them up, like ‘≪’.

## Tutorial 2: Functions

Say you’ve continued to build your input file, and you end up with the six following objects:

```
Schedule:Constant,
  Zone Space Temperature Setpoint Schedule, ! Name
  Zone Space Temperature Limits,            ! Schedule Type Limits Name
  22;                                       ! Hourly Value

Schedule:Constant,
  Supply Air Temp Setpoint Schedule, ! Name
  Supply Air Temp Limits,            ! Schedule Type Limits Name
  13;                                ! Hourly Value

Schedule:Constant,
  CHW Supply Temp Setpoint Schedule, ! Name
  CHW Supply Temp Limits,            ! Schedule Type Limits Name
  7;                                 ! Hourly Value

ScheduleTypeLimits,
  Zone Space Temperature Limits, ! Name
  20,                            ! Lower Limit Value
  24,                            ! Upper Limit Value
  Continuous,                  ! Numeric Type [Continuous, Discrete]
  Temperature;                 ! Unit Type

ScheduleTypeLimits,
  Supply Air Temp Limits, ! Name
  10,                     ! Lower Limit Value
  16,                     ! Upper Limit Value
  Continuous,             ! Numeric Type [Continuous, Discrete]
  Temperature;            ! Unit Type

ScheduleTypeLimits,
  CHW Supply Temp Limits, ! Name
  5,                      ! Lower Limit Value
  10,                     ! Upper Limit Value
  Continuous,             ! Numeric Type [Continuous, Discrete]
  Temperature;            ! Unit Type
```

Can you see the repetition here? Each schedule has an associated schedule limits, in which the names match. We can make this situation better by introducing a *function*.

```
const_temp_schedule = \ name value lower upper {
Schedule:Constant,
  <name> Schedule, ! Name
  <name> Limits,   ! Schedule Type Limits Name
  <value>;         ! Hourly Value

ScheduleTypeLimits,
  <name> Limits, ! Name
  <lower>,       ! Lower Limit Value
  <upper>,       ! Upper Limit Value
  Continuous,    ! Numeric Type [Continuous, Discrete]
  Temperature;   ! Unit Type
}

print const_temp_schedule('Zone Space Temperature', 22, 20, 24)
print const_temp_schedule('Supply Air Temp'       , 13, 10, 16)
print const_temp_schedule('CHW Supply Temp'       , 7 , 5 , 10)
```

This will produce the exact same objects as before. But we now can trust that our names will be correct, we went from 35 lines to 17, and adding a new constant schedule like this is just one more line.

Let's discuss what we've seen here - Here's the official grammar for defining a function (named `lambda_def` in the grammar):

```
lambda_def : ('\\' | 'λ' ) IDENTIFIER* '{' (expression | function_statement*) '}' ;
```

In words, a lambda definition is a backslash character (or $\lambda$ symbol), followed by 0 or more identifiers, with a single expression *or* one or more function statements in curly braces.

So technically, in our example with the schedule, this is a *variable declaration* where the expression on the right hand side being assigned is a *function expression*. Pretty much everything in Neobem is an *expression*. And in Neobem, functions are first class. They can be assigned to variables and be returned by other functions.

This means you can build higher order functions like:

```
my_add = \ x { \y { x + y }}

add_two = my_add(2)

print add_two(5)
```

The same thing could even be written on a single line like:

```
print (\x { \y { x + y }})(2)(5)
```

The result of this is `7`. See, functions can return other functions, and they have the concept of "closures". That is, functions can see all the defined variables in the above scopes. For example, in the inner defined function `\y { x + y }`, it knows of the `x` from the surrounding function.

The second example showed how you don't even need to necessarily name your function if you want. You can define anonymous functions wherever you like.

This example showed the one of function forms, where inside the curly braces is a single expression. However, inside a function, you can also provide one or more *function statements*. This is actually what you saw in the example with the schedules.

A function statement can be:

1. An object declaration
2. A variable declaration
3. An input file comment
4. A return statement

Neobem can be considered a mostly *functional* language. In a truly functional programming language, there *are no side-effects*. However, for practical reasons, the Neobem language has one important built in side-effect. That is the object declaration.

When the compiler encounters an object declaration from inside or outside of a function, it will print it out to the final result, making any required replacements (The same is true for input file comments). *This is the only way objects are printed out.*

This is how the concept of what some might think of as a "template" and a mathematical function are the same thing in Neobem. These are both functions to Neobem. One has a side effect of printing to the result, and the other just computes a value.

```
version_template = \ ver {
Version,
    <ver>;
}

math_function = \ x { (x + 2)^3 }
```

Beside object declarations and input file comments, the other two possible statements are the variable declaration and the return statement.

A variable declaration is the same declaration introduced earlier. However, if a variable is declared within a function, it is not visible from outside that function. It is normally used to break up a complicated expression that make be easier to understand in more steps.

The return statement sets the resulting expression that is returned by the function. An example of variable declaration and the return statement would be:

```
my_hard_computation = \ input {
    numerator   = sin(input * 2) + ceiling(input)
    denominator = ln(input)
    return numerator / denominator
}
```

## Tutorial 3: Utilizing Tabular Data, Dictionaries, and Lists

The goal with a domain specific language like this is to reduce the amount of syntax required to express our intent. At the extreme, the minimum we must provide is the *data*. So a novel feature of the Neobem language is the integration of data entry directly into the language.

Lets say I have tabular data about my zones and want to create the required objects for them. An example is the easiest way to show how this would be done in Neobem.

```
zones =

----------------------------------------
'name'        | 'x_origin'   | 'y_origin'
--------------|--------------|-----------
'Bedroom'     | 0            | 0
'Living Room' | 10           | 20
'Kitchen'     | 5            | 12
----------------------------------------

zone_template = λ zone {
Zone,
  <zone.'name'>,      ! Name
  0,                  ! Direction of Relative North {deg}
  <zone.'x_origin'>,  ! X Origin {m}
  <zone.'y_origin'>,  ! Y Origin {m}
  0,                  ! Z Origin {m}
  1,                  ! Type
  1,                  ! Multiplier
  autocalculate,      ! Ceiling Height {m}
  autocalculate,      ! Volume {m3}
  autocalculate,      ! Floor Area {m2}
  ,                   ! Zone Inside Convection Algorithm
  ,                   ! Zone Outside Convection Algorithm
  Yes;                ! Part of Total Floor Area
}

print map(zones, zone_template)
```

Woah - there's a lot going on here. Let's unpack this. There are 3 lines, the first two are variable declarations, followed by a print statement.

The first variable declaration

```
zones =

----------------------------------------
```

```
'name'        | 'x_origin'  | 'y_origin'
--------------|------------ |-----------
'Bedroom'     | 0           | 0
'Living Room' | 10          | 20
'Kitchen'     | 5           | 12

----------------------------------------
```

shows an *inline data table expression*. It looks like a pretty table, but it is actually an expression that evaluates to a *list* of *dictionaries*. The borders can be made with hyphens `'-'` and pipe characters `'|'`

A list expression is what you would expect. It's a list of expressions. This is like an "array" in other programming languages. You can define your own lists anywhere an expression is expected (because it is an expression) using square brackets `'['` and `']'` with items separated by commas.

Example:

```
my_list = [ 1, 2, 'a string!', sin(3.1415926), \x { x + 1 } ]
```

Notice that you can put whatever expression *types* you want into a single list. The example has 2 numeric expressions, a string expression, a numeric expression that is the result of function application, and a function expression.

A *dictionary* is a way to group expressions into a single identifier. This is the concept of an "associative array" implemented as a "hash table" with a string key, the same concept as in many other programming languages. Unfortunately for me, I cannot name this an "object" since the concept of an object is already taken by the "objects" in the resulting building energy simulation files we are trying to create.

You define a dictionary like this:

```
my_dict = {
    'prop 1' : 42,
    'prop 2' : 'Some text',
    'prop 3' : {
        'nested_dict_prop' : 'I am nested'
    },
    'prop 4' : [ 1, 2, 3]
}
```

You can access the contents of the dictionary using the member access operator, a period **.**, followed by the string key.

So for our dictionary example:

```
my_dictionary.'prop2' == 'Some text'
my_dictionary.'prop_3'.'nested_dictionary' == 'I am nested'
```

Note the key used in the member access can be *any* string. For example, you can use pretty much any Unicode characters you'd like. Using music notes is totally valid:

```
dictionary = {
    '♬': 9.4
}
```

```
Version,
    < dictionary.'♬' >;
```

And the key doesn't have to be a *string literal*, it just has to *evaluate* to a string. So this is no problem either:

```
assigned = 'my key'
dictionary = { 'my key': 10 }

Version,
  < dictionary.assigned >;
```

In fact, in the original definition, they key doesn't even have to be a string literal, it can be any arbitrary expression that evaluates to a string. If you were feeling wild, you could do something like:

```
dictionary = {
    if 'this is' == 'totally wild' then 'wild' else 'totally cool': 9.4
}
Version,<dictionary . ( 'totally ' + 'cool' )>;
```

That does evaluate to `Version,9.4;`, give it a try!

Now that we know what a list and dictionary are, what is the result of that inline data table? It's a *list of dictionaries*.

That means these are exactly the same.

```
zones =

-----------------------------------------
'name'        | 'x_origin' | 'y_origin'
--------------|------------|------------
'Bedroom'     | 0          | 0
'Living Room' | 10         | 20
'Kitchen'     | 5          | 12

-----------------------------------------

zones = [
    {
        'name': 'Bedroom',
        'x_origin': 0,
        'y_origin': 0
    },
    {
        'name': 'Living Room',
        'x_origin': 10,
        'y_origin': 20
    },
    {
        'name': 'Kitchen',
        'x_origin': 5,
        'y_origin': 12
    }
]
```

Why would you use one version over another? I would generally prefer the inline data table for situations when you have data *that doesn't include nested dictionaries.* If nested dictionaries are required, then you'd have to use the normal dictionary syntax.

That covers the first variable declaration of our example file.

The second variable declaration

```
zone_template = λ zone {
Zone,
  <zone.'name'>,      ! Name
  0,                  ! Direction of Relative North {deg}
  <zone.'x_origin'>, ! X Origin {m}
  <zone.'y_origin'>, ! Y Origin {m}
  0,                  ! Z Origin {m}
  1,                  ! Type
  1,                  ! Multiplier
  autocalculate,      ! Ceiling Height {m}
  autocalculate,      ! Volume {m3}
  autocalculate,      ! Floor Area {m2}
  ,                   ! Zone Inside Convection Algorithm
  ,                   ! Zone Outside Convection Algorithm
  Yes;                ! Part of Total Floor Area
}
```

is a normal function with an object declaration. The only unique thing about it is that the input parameter is expected to be a dictionary that has 3 keys: `name`, `x_origin`, and `y_origin`.

Did you also notice that lambda (λ) character in the function definition? It's allowed instead of the boring \ character if you'd like. Why the lambda character? If you don't know why, search for lambda calculus and dive into that rabbit hole.

The final statement is

```
print map(zones, zone_template)
```

This is the evaluation of the built in `map` function, with the function `zone_template` and list of dictionaries `zones` as inputs.

The `map` function is a extremely important function in functional programming languages. It allows you to "map" or evaluate a function over each element of a list. The resulting expression is a new list, with each list item being transformed by the function.

However, in Neobem remember, functions can have one important side-effect: printing out objects.

So the output to our compiled idf file from the `map` function is:

```
Zone,
  Bedroom,                  ! Name
  0,                        ! Direction of Relative North {deg}
  0,                        ! X Origin {m}
  0,                        ! Y Origin {m}
  0,                        ! Z Origin {m}
  1,                        ! Type
  1,                        ! Multiplier
  autocalculate,            ! Ceiling Height {m}
  autocalculate,              ! Volume {m3}
  autocalculate,              ! Floor Area {m2}
  ,                           ! Zone Inside Convection Algorithm
  ,                           ! Zone Outside Convection Algorithm
  Yes;                        ! Part of Total Floor Area

Zone,
  Living Room,                ! Name
  0,                          ! Direction of Relative North {deg}
  10,                         ! X Origin {m}
  20,                         ! Y Origin {m}
  0,                          ! Z Origin {m}
  1,                          ! Type
  1,                          ! Multiplier
  autocalculate,              ! Ceiling Height {m}
  autocalculate,              ! Volume {m3}
  autocalculate,              ! Floor Area {m2}
  ,                           ! Zone Inside Convection Algorithm
  ,                           ! Zone Outside Convection Algorithm
  Yes;                        ! Part of Total Floor Area

Zone,
  Kitchen,                    ! Name
  0,                          ! Direction of Relative North {deg}
  5,                          ! X Origin {m}
  12,                         ! Y Origin {m}
  0,                          ! Z Origin {m}
  1,                          ! Type
  1,                          ! Multiplier
  autocalculate,              ! Ceiling Height {m}
  autocalculate,              ! Volume {m3}
  autocalculate,              ! Floor Area {m2}
  ,                           ! Zone Inside Convection Algorithm
  ,                           ! Zone Outside Convection Algorithm
  Yes;                        ! Part of Total Floor Area
```

I hope you can take in how expressive that is. Write the minimal amount of data, then map a template function over it to build all your objects.

## Tutorial 4: Introducing Logic

If we want to have a Turing complete programming language, we need branching ability - which we accomplish in Neobem using *if expressions*.

Notice how I called it an *expression*. Like everything else, our if-else ability is an expression, meaning it can always be reduced to a new expression.

The syntax for the if expression is

```
if <expression> then <expression> else <expression>
```

The `else` is required because there must always be a result.

For this tutorial, let's use logic to determine which template we want to use for a given fan.

```
in_h2o_2_pa = \ in_h2o { in_h2o * 249.08891 }
cfm_2_m3s = \ cfm { cfm / 2118.88 }

variable_fan = \name cfm press {
Fan:VariableVolume,
  <name>,                ! Name
  ,                      ! Availability Schedule Name
  0.7,                   ! Fan Total Efficiency
  <in_h2o_2_pa(press)>,  ! Pressure Rise {Pa} (<press> inH2O)
  <cfm_2_m3s(cfm)>,      ! Maximum Flow Rate {m3/s} (<cfm> CFM)
  Fraction,              ! Fan Power Minimum Flow Rate Input Method
  0.25,                  ! Fan Power Minimum Flow Fraction
  ,                      ! Fan Power Minimum Air Flow Rate {m3/s}
  0.9,                   ! Motor Efficiency
  1.0,                   ! Motor In Airstream Fraction
  0,                     ! Fan Power Coefficient 1
  0,                     ! Fan Power Coefficient 2
  1,                     ! Fan Power Coefficient 3
  0,                     ! Fan Power Coefficient 4
  0,                     ! Fan Power Coefficient 5
  <name> Inlet,          ! Air Inlet Node Name
  <name> Outlet,         ! Air Outlet Node Name
  General;               ! End-Use Subcategory
}

constant_fan = \name cfm press {
Fan:ConstantVolume,
  <name>,                ! Name
  ,                      ! Availability Schedule Name
  0.7,                   ! Fan Total Efficiency
  <in_h2o_2_pa(press)>,  ! Pressure Rise {Pa} (<press> inH2O)
  <cfm_2_m3s(cfm)>,      ! Maximum Flow Rate {m3/s} (<cfm> CFM)
  0.9,                   ! Motor Efficiency
  1.0,                   ! Motor In Airstream Fraction
  <name> Inlet,          ! Air Inlet Node Name
  <name> Outlet,         ! Air Outlet Node Name
  General;               ! End-Use Subcategory
}


fans = [
    {
        'name': 'Fan 1',
        'type': 0,
        'press': 5,
        'cfm': 10000
    },
    {
        'name': 'Fan 2',
        'type': 1,
        'press': 6,
        'cfm': 20000
    }
]

which_fan = \fan {
if fan.'type' == 0 then
    constant_fan(fan.'name', fan.'cfm', fan.'press')
else
    variable_fan(fan.'name', fan.'cfm', fan.'press')
}

print map(fans, which_fan)
```

Notice here how we used an equality operator (==) to determine what function to use.

We also introduced some helper functions that do some conversion from IP to SI units. The units for EnergyPlus input files are base SI. In many cases this is not convenient. We can wrap values in functions, so that our entry is in the units we want, but the output correctly matches the energy simulation requirements.

## Tutorial 5: Importing From Other Files

Large software projects are not built in one monolithic file. Engineers break the code up into smaller pieces so that it is more manageable. Building simulation files can get quite large, thousands of lines of code, so it makes sense that we should be able to do the same thing in Neobem.

A small, but typical example of breaking out the input file in multiple files could be this.

**in.nbem**

```
import 'defaults.nbem'

print simulation_params()

import 'chillers.nbem'
```

where in the same directory as the `in.nbem` file is the two files

1. `defaults.nbem`
2. `chillers.nbem`

If the contents of those are

**defaults.nbem**

```
simulation_params = λ {
Version,9.4;

Timestep,6;

ZoneAirHeatBalanceAlgorithm,EulerMethod;
}

export (simulation_params)
```

**chillers.nbem**

```
chiller = λ unit_number {

chiller_name = 'Chiller ' + unit_number

tons_to_watts = λ tons { tons * 3516.8528 }

Chiller:ConstantCOP,
  <chiller_name>,            ! Name
  <tons_to_watts(1000)>,     ! Nominal Capacity {W} (1,000 tons)
  6,                         ! Nominal COP {W/W}
  autosize,                  ! Design Chilled Water Flow Rate {m3/s}
  autosize,                  ! Design Condenser Water Flow Rate {m3/s}
  <chiller_name> CHW Inlet,  ! Chilled Water Inlet Node Name
  <chiller_name> CHW Outlet, ! Chilled Water Outlet Node Name
  <chiller_name> CW Inlet,   ! Condenser Inlet Node Name
  <chiller_name> CW Outlet,  ! Condenser Outlet Node Name
  AirCooled,                 ! Condenser Type
  NotModulated,              ! Chiller Flow Mode
  1.0;                       ! Sizing Factor
}

print map([1, 2, 3], chiller)
```

then the resulting output would be:

```
Version, 9.4;

Timestep, 6;

ZoneAirHeatBalanceAlgorithm, EulerMethod;

Chiller:ConstantCOP,
  Chiller 1,                  ! Name
  3516852.8000000003,         ! Nominal Capacity {W} (1,000 tons)
  6,                          ! Nominal COP {W/W}
  autosize,                   ! Design Chilled Water Flow Rate {m3/s}
  autosize,                   ! Design Condenser Water Flow Rate {m3/s}
  Chiller 1 CHW Inlet,        ! Chilled Water Inlet Node Name
  Chiller 1 CHW Outlet,       ! Chilled Water Outlet Node Name
  Chiller 1 CW Inlet,         ! Condenser Inlet Node Name
  Chiller 1 CW Outlet,        ! Condenser Outlet Node Name
  AirCooled,                  ! Condenser Type
  NotModulated,               ! Chiller Flow Mode
  1.0;                        ! Sizing Factor

Chiller:ConstantCOP,
  Chiller 2,                  ! Name
  3516852.8000000003,         ! Nominal Capacity {W} (1,000 tons)
  6,                          ! Nominal COP {W/W}
  autosize,                   ! Design Chilled Water Flow Rate {m3/s}
  autosize,                   ! Design Condenser Water Flow Rate {m3/s}
  Chiller 2 CHW Inlet,        ! Chilled Water Inlet Node Name
  Chiller 2 CHW Outlet,       ! Chilled Water Outlet Node Name
  Chiller 2 CW Inlet,         ! Condenser Inlet Node Name
  Chiller 2 CW Outlet,        ! Condenser Outlet Node Name
  AirCooled,                  ! Condenser Type
  NotModulated,               ! Chiller Flow Mode
  1.0;                        ! Sizing Factor

Chiller:ConstantCOP,
  Chiller 3,                  ! Name
  3516852.8000000003,         ! Nominal Capacity {W} (1,000 tons)
  6,                          ! Nominal COP {W/W}
  autosize,                   ! Design Chilled Water Flow Rate {m3/s}
  autosize,                   ! Design Condenser Water Flow Rate {m3/s}
  Chiller 3 CHW Inlet,        ! Chilled Water Inlet Node Name
  Chiller 3 CHW Outlet,       ! Chilled Water Outlet Node Name
  Chiller 3 CW Inlet,         ! Condenser Inlet Node Name
  Chiller 3 CW Outlet,        ! Condenser Outlet Node Name
  AirCooled,                  ! Condenser Type
  NotModulated,               ! Chiller Flow Mode
  1.0;                        ! Sizing Factor
```

Neat, right? So perhaps the simplest way that Neobem can help you with your current files is by breaking them up into smaller pieces.

To note here:

- The string is a relative file path to the file to import.
- Notice the last line of the file `defaults.nbem`. It is an *export* statement, and it is required for other files to use the variables and functions that are defined.

  However, if the imported script generates any output, that is passed along no matter what - no `export` is required. This is what happens in the `chillers.nbem` file.
- By default, exported items will overwrite existing identifiers with the same name. You can get around this by *qualifying* the import using the `as` option.

  Using the previous example, we could have done:

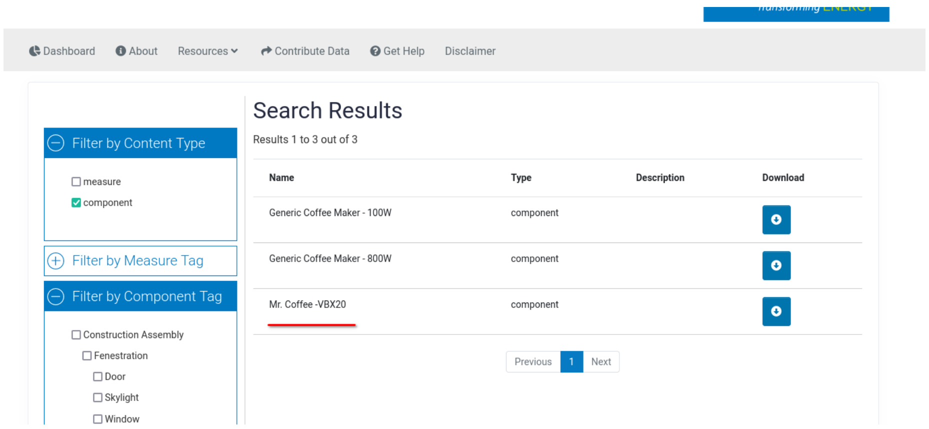
  **in.nbem**

  ```
  import 'defaults.nbem' as def

  print def@simulation_params()

  import 'chillers.nbem'
  ```

  The `as` option will put whatever string is specified as a prefix to all exports from the file, with an `@` sign between.

## Tutorial 6: Using the Building Component Library

The Building Component Library (BCL) is a library of “components” and “measures” to be used in energy models. It is hosted by NREL in conjunction with other National Laboratories from the United States. The data is hosted in GitHub repositories, searchable through a straightforward API.



Search Results

Results 1 to 3 out of 3

| Name | Type | Description | Download |
|---|---|---|---|
| Generic Coffee Maker - 100W | component | | |
| Generic Coffee Maker - 800W | component | | |
| Mr. Coffee -VBX20 | component | | |

Figure 2: BCL search results for coffee makers

Components contain the raw data for building up an energy model. Straight from the BCL, here is how they are described:

> The BCL contains components which are the building blocks of an energy model. They can represent physical characteristics of the building such as roofs, walls, and windows, or can refer to related operational information such as occupancy and equipment schedules and weather information. Each component is identified through a set of attributes that are specific to its type, as well as other metadata such as provenance information and associated files.

As the BCL is currently the de facto standard hosting entity for this type of information, it seemed pragmatic to dedicate *Neobem programming language syntax* to it.

## Using BCL Attributes

For example, let's imagine we are trying to model the loads in our zone, and we have a Mr. Coffee coffee maker. Using the BCL, we search for components with the tag `Appliance.Coffee Maker`.

I see the component exists, and I download it to check it out.

If I download that component, it will be a compressed zip file with a single directory and a file called `component.xml`. The `xml` file contains our data to be used in the energy model.

Here's the contents of the `component.xml` file for our coffee maker:

```xml
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="component.xsd">
 <schema_version/>
 <name>Mr. Coffee -VBX20</name>
 <uid>b87b9630-c2c7-012f-13bc-00ff10b04504</uid>
 <version_id>664faf5e-a91b-4b06-bc73-0c79c39c9afb</version_id>
 <xml_checksum/>
 <display_name/>
 <description/>
 <modeler_description/>
 <source>
  <manufacturer>Mr. Coffee</manufacturer>
  <model>VBX20</model>
  <serial_no/>
```

```
    <year>2012</year>
    <url>http://www.mrcoffee.com/Manuals/MANUALS/VBX20_43_65900646.PDF </url>
   </source>
   <tags>
    <tag>Appliance.Coffee Maker</tag>
   </tags>
   <attributes>
    <attribute>
     <name>Peak Power</name>
     <value>830.0</value>
     <datatype>float</datatype>
    </attribute>
   </attributes>
  </component>
```

Notice there are lots of fields and data available. We have a description of the manufacturer, URL reference, and a peak power consumption of 830 W.

Also notice that every component in the BCL has a *universally unique identifier (UUID)*, a 128 bit id that as you might expect, *uniquely identifies it.*

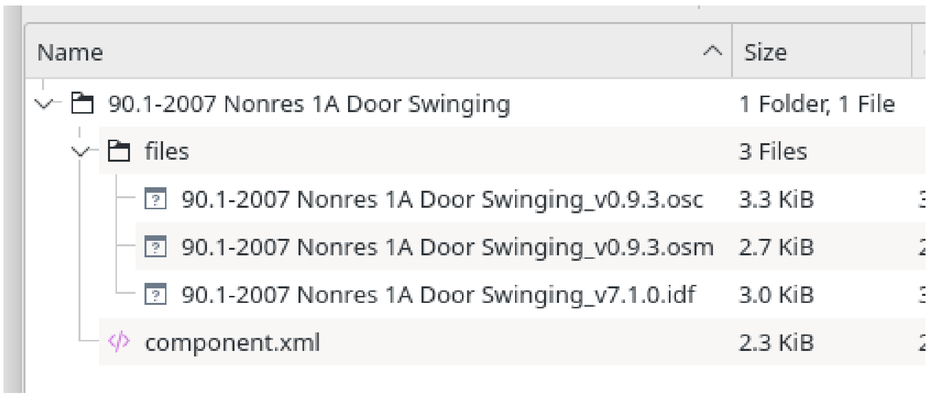
So we want to inject this data into our energy model. We can do that using the UUID.

Let's make an `ElectricEquipment` object to represent the heat gain from the coffee maker.

```
coffee_maker = bcl:b87b9630-c2c7-012f-13bc-00ff10b04504

ElectricEquipment,
  <coffee_maker.'source'.'manufacturer'> <coffee_maker.'source'.'model'>,  ! Name
  Kitchen,                    ! Zone or ZoneList or Space or SpaceList Name
  Coffee Maker Schedule,      ! Schedule Name
  EquipmentLevel,             ! Design Level Calculation Method
  <coffee_maker.'Peak Power'>,  ! Design Level {W}
  ,                           ! Watts per Zone Floor Area {W/m2}
  ,                           ! Watts per Person {W/person}
  0,                          ! Fraction Latent
  0,                          ! Fraction Radiant
  0,                          ! Fraction Lost
  General;                    ! End-Use Subcategory
```

The idf output from this is:

```
ElectricEquipment,
  Mr. Coffee VBX20,           ! Name
  Kitchen,                    ! Zone or ZoneList or Space or SpaceList Name
  Coffee Maker Schedule,      ! Schedule Name
  EquipmentLevel,             ! Design Level Calculation Method
  830,                        ! Design Level {W}
  ,                           ! Watts per Zone Floor Area {W/m2}
  ,                           ! Watts per Person {W/person}
  0,                          ! Fraction Latent
  0,                          ! Fraction Radiant
  0,                          ! Fraction Lost
```



| Name | Size |
|---|---|
| 90.1-2007 Nonres 1A Door Swinging | 1 Folder, 1 File |
| files | 3 Files |
| 90.1-2007 Nonres 1A Door Swinging_v0.9.3.osc | 3.3 KiB |
| 90.1-2007 Nonres 1A Door Swinging_v0.9.3.osm | 2.7 KiB |
| 90.1-2007 Nonres 1A Door Swinging_v7.1.0.idf | 3.0 KiB |
| component.xml | 2.3 KiB |

Figure 3: Door component contents

```
    General;                         ! End-Use Subcategory
```

The line

```
coffee_maker = bcl:b87b9630-c2c7-012f-13bc-00ff10b04504
```

sets the variable `coffee_maker` to a dictionary representing that component. It will match the schema found in the XML file. It also lifts all the `attributes` to the root of the dictionary. That's why the `<coffee_maker.'Peak Power'>` expression works. Otherwise you'd have to something extremely verbose to extract out the attribute, something like:

```
wattage = (coffee_maker.'attributes'.'attribute'
             ⊳ λ attr { attr.'name' == 'Peak Power' }
             → index(0)).'value'
```

and nobody wants to do stuff like that.

### Using BCL Files

For some components, idf objects will already be defined in an attached idf file. This is quite useful, as we can then simply import that file.

For example, let's look at the component with the UUID `2e613270-5ea8-0130-c85b-14109fdf0b37`, a non residential swinging door meeting ASHRAE Standard 90.1-2007 requirements for climate zone 1A.

If you download that component you'll see the `component.xml` file as before, but now there is a directory called `files`.

The contents looks like:

There are files for OpenStudio (`.osc` and `.osm`) and EnergyPlus (`.idf`).

When the component is loaded as a dictionary in Neobem, it will populate the URL to these files.

This makes including the file straightforward. An example would look like this:

```
import bcl:2e613270-5ea8-0130-c85b-14109fdf0b37.'idf'
```

Remember that the `import` statement in Neobem is simple and explicit. You give it a string URI, and Neobem downloads it as a Neobem file, executes it, and includes any idf output in the final result.

The expression `bcl:2e613270-5ea8-0130-c85b-14109fdf0b37` is a dictionary. We are then getting the value for the key `idf`. That key is a long string with the with the value of a URL (Breaking it up onto several lines so as to not ruin the documentation layout).

```
'https://bcl.nrel.gov/api/file?' +
'path=BuildingComponentLibrary-nrel-components/' +
'v0.3/90.1-2007 Nonres 1A Door Swinging/files/' +
'90.1-2007 Nonres 1A Door Swinging_v7.1.0.idf'
```

You can try out that endpoint directly in your browser by clicking the link here.

You could have manually put this URL in the `import` statement like:

```
import 'https://bcl.nrel.gov/api/file?' +
       'path=BuildingComponentLibrary-nrel-components/' +
       'v0.3/90.1-2007 Nonres 1A Door Swinging/files/' +
       '90.1-2007 Nonres 1A Door Swinging_v7.1.0.idf'
```

but using the UUID and the '`idf`' key expresses the intent more clearly and with fewer characters.

This will download the associated idf file, which has the contents:

```
! Type: Door:Swinging
! Description:
! Effective R-Value (SI): 0.2516
! Layer 1: METAL Door Medium 18Ga_1
! Layer 2: AIR
! Layer 3: METAL Door Medium 18Ga_2
Material,
  METAL Door Medium 18Ga_1,   !- Name
  Smooth,                     !- Roughness
  0.0013106,                  !- Thickness
  45.3149,                    !- Conductivity
  7833.03,                    !- Density
  502.08,                     !- Specific Heat
  0.8,                        !- Abs Thermal
  0.5,                        !- Abs Solar
  0.5;                        !- Abs Visible


Material:NoMass,
  AIR,                        !- Name
  Smooth,                     !- Roughness
  0.251489,                   !- Thermal Resistance
  0.8,                        !- Abs Thermal
  0.5,                        !- Abs Solar
  0.5;                        !- Abs Visible


Material,
  METAL Door Medium 18Ga_2,   !- Name
  Smooth,                     !- Roughness
  0.0013106,                  !- Thickness
  45.3149,                    !- Conductivity
  7833.03,                    !- Density
  502.08,                     !- Specific Heat
  0.8,                        !- Abs Thermal
  0.5,                        !- Abs Solar
  0.5;                        !- Abs Visible


Construction,
  90.1-2007 Nonres 1A Door Swinging,  !- Name
  METAL Door Medium 18Ga_1,   !- Layer 1
  AIR,                        !- Layer 2
  METAL Door Medium 18Ga_2;   !- Layer 3


! CostSource:
! InstallCost {$/m2}: 0.000
! MaterialCost {$/m2}: 0.000
! AnnualOM {$/m2}: 0.000
! MinorOM {$/m2}: 0.000
! MinorOMInt {years}: 1
! MajorOM {$/m2}: 0
! MajorOMInt {years}: 1
! ExpectedLife {years}: 0
! ResidualValue {$/m2}: 0.000
ComponentCost:LineItem,
  90.1-2007 Nonres 1A Door Swinging,  !- Name
  ,                           !- Type
  Construction,               !- Line Item Type
  90.1-2007 Nonres 1A Door Swinging,  !- Item Name
  ,                           !- Object End Use Key
  ,                           !- Cost per Each {$}
  0.000,                      !- Cost per Area {$/m2}
  ,                           !- Cost per Unit of Output Capacity {$/kW}
  ,                           !- Cost per Unit of Output Capacity per COP {$/kW}
  ,                           !- Cost per Volume {$/m3}
  ,                           !- Cost per Volume Rate {$/(m3/s)}
  ,                           !- Cost per Energy per Temperature Difference {$/(W/K)}
  ;                           !- Quantity
```

**Note**: In some of the idf files provided in the BCL, a ‘`Version`’ object is supplied. If multiple idf files were included, you would run into an EnergyPlus error for providing multiple ‘`Version`’ objects. Because of this, Neobem will automatically remove any ‘`Version`’ objects from the downloaded idf file.

## Automatic Formatting

If you have read through the tutorial section, you will have seen some unique characters allowed as part of the syntax. For example, you can use the `'λ'` character in function definitions, or `'✓'` for a literal true value.

But how do you easily enter this characters as you are initially programming? Plugins for your current favorite editor may arrive in the future. But there is a better solution.

The solution favored by Neobem is to use the Neobem transpiler itself to *automatically* format the files, including using the more aesthetically pleasing Unicode characters.

This is especially useful for formatting inline data tables. You can quickly define a table with sloppy syntax like (along with a function using the easy to type characters):

```
my_data =
---
'Header 1' | 'Header 2'
---
2 | 3
4| 5
---

my_function = \ param { if 'Having a formatter is great' == 'Yes' then true else false }
```

and then run the command `nbem -f sloppy_table_file.nbem` on your file, which will print out your freshly formatted file to standard output, which will look like:

```
my_data =
───────────┬───────────
'Header 1' │ 'Header 2'
───────────┼───────────
2          │ 3
4          │ 5
───────────┴───────────


my_function = λ param { if 'Having a formatter is great' == 'Yes' then ✓ else ✕ }
```

If you like the output, you can overwrite your current file using a syntax like (if using a POSIX-like shell like Bash):

```
nbem -f my_file.nbem > tmp && mv tmp my_file.nbem
```

This formatter will also take care of many other concerns, such as whitespace between tokens, trimming extra lines at the beginning and end of the file.

It is meant to be *opinionated*. There are many advantages to having a de-facto standard for file formatting.

1. All programmers get used to seeing the same style no matter what person or group wrote code.
2. No reason to worry about style arguments when working with a team.
3. No useless version control commits that only affect style, not content.
4. Can prototype fast and sloppy, and not spend valuable mental effort on worrying about irrelevant formatting details.

It makes most sense for the formatter to be part of the Neobem transpiler project since it can then use the exact same parsed AST (Abstract Syntax Tree) as the transpiler, leading to high quality output. As the currently sole author, it also makes sense that I define what I believe the standard should be for file formatting.

You can see more arguments for this approach from its inspiration: The Go programming language and `gofmt` (Blog on `gofmt`), and Prettier, an opinionated JavaScript/TypeScript formatter.

## Reference

### General

Whitespace is generally ignored, expect within the object definitions and comments.

When parsing the main file, there are 6 possibilities of what is being parsed:

1. An idf comment (ex: `! comment`)
2. An idf object (ex: `Version,9.4;`)
3. A variable declaration (ex: `name = <expression>`)
4. An `import` statement (ex: `import 'filename.nbem'`)
5. Print statement (ex: `print <expression>`)
6. A Neobem specific comment (ex: `# Neobem comment`)
7. A log statement (ex: `log myvar`)

idf comments and idf objects result in text being output to the final target.

The purpose of the `print` statement is to be able to evaluate an expression without naming it[4].

### Data Types

There are two primitive expression types, the same as EnergyPlus, along with the boolean True/False data type.

1. String expressions (or alpha in idf terminology)
2. Numeric expressions
3. Boolean expressions

There are 3 higher order expression types.

1. List expressions
2. Function expressions
3. Dictionary expressions

#### Strings

Strings are entered using single quotes “`'`”. You can escape the following characters with a backslash:

- `'\n'` for newline
- `'\r'` for carriage return
- `'\t'` for tab
- `'\''` for single quote
- `'\\'` for the backslash character itself

#### Boolean Literals

You can enter in a boolean literal in two ways each for True or False.

For true you can use `true` or ✓ (Unicode U+2713, Check Mark)

For false you can use `false` or ✗ (Unicode U+2717, Ballot X)

[4]The other main purpose of having a unique token before the expression is easier parsing of the file. Having lone expressions everywhere makes things tricky.

## Comments

There are two different types of comments in Neobem. One is the comment syntax used by the idf files, beginning with an exclamation point character (`!`). These comments are available to have portions replaced by expressions and they are always passed along on to the output.

The second comment type is internal to Neobem itself. These are like traditional comments in that they are parsed and then completely ignored. They are intended to aid the reader of the code.

So for example, given:

```
# This is an internal comment that won't appear in output. It won't
# have this <variable> replaced.
variable = 'Mitch'

! This is an IDF comment, with my name <variable> showing up.
```

the output is

```
! This is an IDF comment, with my name Mitch showing up.
```

## Replacements

Within object definitions, arbitrary expressions can be inserted using angle brackets '<' and '>'.

What is output *is a string representation of the expression.* As a reminder, all expressions must evaluate to one of the 6 types listed above (e.g. String expression, Numeric expression, List expression, etc.).

The representation of each is as follows:

- *String Expression*: The literal text that makes up the string. This is the most straightforward.
- *Numeric Expression*: The value of the numeric expression, currently using a default formatting (This will be updated in future versions to use a certain number of significant figures by default).
- *List Expression*: The string representation of each element, *separated by a comma.* This is critical, as it allows a list to be used to fill in the fields for objects in which the final length is not known. Examples would be the fields for floor coordinates.
- *Function Expression*: Nothing, the empty string `''`. This should rarely be used.
- *Dictionary Expression*: Returns the string representation of each expression for each member of the dictionary, separated by a comma. Note that there is currently no way to specify the order here, unlike the list expression, so this is rarely something that you would want to do.
- *Boolean Expression*: Returns either `True` or `False`, as one would reasonably expect.

The most common types that are expected to be used in replacements are the string expression, numeric expression, and list expression. The others are there for consistency, but the usage should be rare.

## Operators

### Plus '+'

- Addition for numeric types
- String concatenation for string types
  - `'Chiller ' + '1'` equals `'Chiller 1'`

  - If using '+' operator with a string and a numeric, the numeric is coerced to a string, and then the two strings are concatenated. So `'Chiller ' + 1` (notice no quotes around the number 1) becomes the string `'Chiller 1'` as you would expect.
- List concatenation for list types
  - `[1, 2, 3] + [4, 5]` equals `[1, 2, 3, 4, 5]`
- Dictionaries - concatenate/update
  - When two dictionaries are used with the `+` operator, the keys from both dictionaries are combined into a new dictionary. If both dictionaries have the same key, then the value from the *second* dictionary is used. This is how dictionaries can be "updated" or modified. An example is shown below.

```
original_dictionary = {
  'key': 'value',
  'second key': 10,
}

updated_dictionary = original_dictionary + {
  'other key': 'passed along',
  'second key': 20,
}

# This isn't a valid Version object, but go along with it.
Version,
  <updated_dictionary.'key'>,
  <updated_dictionary.'second key'>,
  <updated_dictionary.'other key'>;
```

Compiling the above code results in:

```
Version,
  value,
  20,
  passed along;
```

### Other Algebraic Operators

The following operators are only valid for numeric types.

- Exponentiation: '^'
- Minus '-'
- Multiplication '*'
- Division '/'

### Range Operator

The range operator can be used to quickly make a list of integer values. The syntax looks like:

```
list_normal = [1, 2, 3, 4, 5]
list_with_range_operator = 1..5
```

It is a numeric expression, followed by '..', then by a second numeric expression.

### Map and Filter Operator

Two important concepts in Neobem and functional programming in general are *filtering* and *mapping*. They are important enough that a dedicated operator in the syntax was created.

A filter takes a list and a function that returns true or false. What is returned is a list with only the elements that return true when put through the function. The filter operator is '|>', or a pipe followed by a greater than sign. You might remember it by thinking of the greater than sign as constricting a list, making it smaller from left to right. Or like a typical physical filter symbol, rotated counter-clockwise by 90°.

A map takes a list and a function and returns a new list of the same length in which each element has been transformed by the function. The operator is '|=' or a pipe followed by an equals sign. You can think of the horizontal lines in the equals sign being the individual elements being transformed some way 1:1.

Examples:

```
pump_template = λ pump {
  Pump:ConstantSpeed,
    <pump.'name'>,              ! Name
    <pump.'name'> Inlet,        ! Inlet Node Name
    <pump.'name'> Outlet,       ! Outlet Node Name
    <pump.'design flow rate'>,  ! Design Flow Rate {m3/s}
    179352,                     ! Design Pump Head {Pa}
    ,                           ! Design Power Consumption {W}
    0.9,                        ! Motor Efficiency
    0.0,                        ! Fraction of Motor Inefficiencies to Fluid Stream
    Continuous;                 ! Pump Control Type
}

pumps =
─────────┬──────────────────
'name'   │ 'design flow rate'
─────────┼──────────────────
'Pump 1' │ 0.5
'Pump 2' │ 1
'Pump 3' │ 2
─────────┴──────────────────

is_high_flow_pump = λ pump { pump.'design flow rate' > 0.75 }

# Filter for high flow pumps based on a property. Same as filter(high_flow_pumps, is_high_flow_pump)
# Results in [{ 'name': 'Pump 2', 'design flow rate': 1 }, { 'name': 'Pump 3', 'design flow rate': 2 }]
high_flow_pumps = pumps |> is_high_flow_pump

# Map the template across the resulting list. This is the same as: map(high_flow_pumps, pump_template)
print high_flow_pumps |= pump_template
```

### Pipe Operator

The pipe operator allows you to write functional code from left to right, like water flowing in a pipe. Many operations can be broken up into a series of transformations that can then be composed. The pipe operator ('->') is essentially syntax sugar for the following:

```
# Normal function call
var = function(parameter)

# Pipe function call
var = parameter -> function
```

When the function has more than one parameter, the *first* parameter can be piped. For example:

```
# Normal function call with multiple parameters
var = function(parameter1, parameter2)

# Using pipe
var = parameter1 -> function(parameter2)
```

Technically, what has happened is that Neobem allows something called *partial application* of a function. You are allowed to invoke a function specifying *1* less parameter than originally specified. This returns a new function that is a function of a single variable, the first parameter.

In this trivial example, not much has been gained. The real usefulness comes when multiple functions are chained. For example:

```
# Imagine fix_units, to_uppercase, and template are all functions
print obj -> fix_units -> to_uppercase -> template
# is more understandable than
print template(to_uppercase(fix_units(obj)))
# at least in my opinion
```

## Inline Data

Grammar for inline data tables in ANTLR form:

```
inline_table : INLINE_TABLE_BEGIN_END_SEP
               inline_table_header
               inline_table_header_separator
               inline_table_data_row+
               INLINE_TABLE_BEGIN_END_SEP ;

inline_table_header : STRING (INLINE_TABLE_COL_SEP STRING)* ;

inline_table_header_separator :
  INLINE_TABLE_BEGIN_END_SEP
  (INLINE_TABLE_COL_SEP INLINE_TABLE_BEGIN_END_SEP)* ;

inline_table_data_row : expression (INLINE_TABLE_COL_SEP expression)* ;

INLINE_TABLE_BEGIN_END_SEP :  [─┴┬_-] [─┴┬_-] [─┴┬_-] [─┴┬_-]* ;

INLINE_TABLE_COL_SEP : '|' | '│' | '┼';
```

In words, it's marked by 3 or more underscores, hyphens, or box drawing characters, then identifiers separated by pipes (|), a header separator row that has sections of 3 or more hyphens separated by pipes, then data in the form of expressions separated by pipes, finished by 3 or more underscores, hyphens, or box drawing characters.

So these are technically parsed the same:

```
zones =

-----------------
'name' | 'origin'
-------|---------
'Z1'   | 0
'Z2'   | 1

-----------------

zones = ___ 'name'|'origin'---'Z1'|0|'Z2'|1___
```

But I think it's obvious which one is easier for a *human* to parse.

## Mathematical Functions

A number of mathematical functions are built in.

- `abs(x)`: absolute value
- `acos(x)`: inverse cosine - return value is in radians
- `asin(x)`: inverse sine - return value is in radians
- `atan2(x, y)`: inverse tangent, returns the angle whose tangent is the quotient of two specified numbers.
- `ceiling(x)`: Returns the smallest integral value that is greater than or equal to the specified number.
- `cos(x)`: Returns the cosine of the angle specified in radians
- `floor(x)`: Returns the smallest integral value that is less than or equal to the specified number.
- `ln(x)`: Natural logarithm, logarithm with base $e$
- `log10(x)`: logarithm with base 10
- `log2(x)`: logarithm with base 2
- `mod(a, n)`: Returns the modulus of `a` divided by `n`. Uses *truncated* division (See Wikipedia) for details.
- `sin(x)`: Returns the sine of the angle specified in radians
- `sqrt(x)`: Returns the positive root of the value (always remember there are 2 roots!)
- `tan(x)`: Returns the tangent of the angle specified in radians

## Functions on Lists

- `length(list)`: number of elements in the list
- `head(list)`: Returns the first element of the list
- `tail(list)`: Returns all elements *except* the first element in the list
- `init(list)`: Returns all elements *except* the last element in the list
- `last(list)`: Returns the last element in the list
- `index(list, integer)`: Returns the element the specified index. The index is 0-based, so getting the first element of the list would be `index(list, 0)`. The index can also be negative to index from the end. `index(list, -1)` returns the last element of the list and `index(list, -2)` returns the second to last element.

## String Functions

- `join(list, seperator)`: Joins a list of strings together with a separator string.
  - Ex: `join(['a', 'b', 'c'], ', ')` results in `'a, b, c'`

- `contains(inputString, searchString)`: Returns a boolean representing whether the `searchString` is found in the `inputString`.
- `lower(inputString)`: Returns the input string with all characters converted to lowercase.
- `upper(inputString)`: Returns the input string with all characters converted to uppercase.

## Functions for Dictionaries

- `keys(dictionary)`: Returns a list of strings that are the keys to the dictionary.
- `has(dictionary, key)`: Returns a boolean representing whether the given dictionary has the string key as a member.

## Loading Data

- `load(expression)`: Load data from a file to a list of dictionaries.

  The single input parameter can either be a string expression or dictionary.

  If the input parameter is a string, it is a relative file path location. `..` represents the parent folder.

  If a dictionary is passed as the parameter, the members are used as options.

### Loading Delimited Text Data

The most straightforward file type to load is the delimited text file. To load, pass a dictionary with the following options:

1. `type`: This must be set to `'text'`.
2. `path`: Required. A string that is the relative path to the text file.
3. `has header`: Optional. A boolean that specifies whether the file has a header record or not. By default, this is `true`.
4. `delimiter`: Optional. A string that specifies the delimiter between fields. By default, this is a tab (`'\t'`) character.
5. `skip`. Optional. An integer number of lines to skip.

For example, with a text file like:

```
Non useful info to start:

Header 1|Header 2
Value1|Value2
Value3|Value4
```

It can be loaded like:

```
load_options = {
  'type': 'text',
  'path': 'path/to/file.txt',
  'skip': 2,
  'delimiter': '|'
}

data = load(load_options)
# data is same as:
data = [
  { 'Header 1': 'Value1', 'Header  2': 'Value2' },
  { 'Header 1': 'Value3', 'Header  2': 'Value4' },
]
```

### Loading from Excel

To load data from Excel, you can use a dictionary with the following options.

1. `type`: This must be set to `'Excel'`.
2. `path`: Required. A string that is the relative path to the Excel file.
3. `sheet`: Optional. A string that has the name of the sheet to pull the data from. If omitted, the first worksheet is used by default.
4. `range`: Optional. Can be specified in different forms.
    - `'A1:B2'` style. This is a string that specifies the complete range in normal Excel range syntax.
    - `'A1'` style. A single cell reference. Neobem will use the input as the upper left hand corner of the data table. It will read headers to the right until it reaches a blank cell. It then read down the table until it reaches a row in which every column in the table is empty.

An example:

```
load_options = {
    'type': 'Excel',
    'sheet': 'Data',
    'range': 'C10',
    'path': 'my_excel_data.xlsx'
}

print map(load(load_options), my_template)
```

### Loading JSON

To load JSON formatted data, you can use a dictionary with the following options.

1. `type`: Required. This must be set to `'JSON'`.
2. `path`: Required. A string that is the relative path to the JSON file.

The mapping between the different value types in JSON to the built in types for Neobem is straightforward, except for JSON's `null`. `null` values in JSON are currently converted to a string expression with the contents `'null'`.

| JSON values | Neobem types |
|---|---|
| object | Dictionary |
| array | List |
| string | String |
| number | Number |
| “true” | Boolean True |
| “false” | Boolean False |
| “null” | String `'null'` |

An example:

```
load_options = {
    'type': 'JSON',
    'path': '../some folder/data.json'
}

print map(load(load_options), my_template)
```

### Loading XML

In a similar manner to JSON, you can load XML passing in a dictionary with the following properties.

1. `type`: Required. This must be set to `'XML'`.
2. `path`: Required. A string that is the relative path to the XML file.

The mapping between XML and Neobem is only slightly lossy. An example will show the mapping best.

```
<root>
  <element attribute1="attribute value" iselement="true">Some text</element>
  <numbers>
    <list>1.5</list>
    <list>2.5</list>
    <list>3.5</list>
  </numbers>
</root>
```

These statements are the same:

```
root = load({ 'type': 'XML', 'path': 'path/to/example above.xml' })

root = {
  'element': {
    'attribute1': 'attribute value'
    'iselement': true
    'value': 'Some text'
  }

  'numbers': {
    'list': [
      { 'value': 1.5 },
      { 'value': 2.5 },
      { 'value': 3.5 },
    ]
  }

  'value': ''
}
```

The basics are that

1. A dictionary is returned.
2. Each child element is made as a property
3. If there are more than one of the same child element, the value for that property is a list of dictionaries.
4. Each element has a `'value'` property, which holds the text inside of a given element.

5. Booleans and numeric values are attempted to be parsed for all dictionary values, with a string type as the fall back.

Some tricky edge cases:

1. It's expected that there is a single root element, Neobem currently only loads the first element.
2. If you have an element such as

   ```
   <element>   some   <br></br>    text   </element>
   ```

   the `'value'` that you get out will be `'some text'`. Each run of text is trimmed of white space on both the start and end, then concatenated with a space.

## Functional Programming Functions

Several functions are staples of functional programming languages. Neobem has a few of the most important ones.

- `map(list, function)`: Returns a new list in which the `function` has been applied to each element.
  - EX: `map([1, 2, 3], \x { x + 2})` will equal `[3, 4, 5]`.
- `filter(list, function)`: Returns a new list in which each element is passed to the function provided, and only the ones in which the function result is `true` are returned.
  - EX: `filter([1, 2, 3, 4], \x { x < 3 })` will equal `[1, 2]`.
- `fold(list, function, initial_value)`: Applies a fold (aka reduce or aggregate) to the supplied list. The function parameter must take 2 arguments. The fold works left to right.
  - EX: `fold([1, 2, 3], \ x y { x + y}, 0)` will equal `6`.

## Other Functions

- `type(anything)`: Returns a string representing the type of the input parameter. Possibilities include: `'function'`, `'string'`, `'list'`, `'numeric'`, `'dictionary'`, and `'boolean'`.

## Let Expressions

Oftentimes, it is useful to be able to name a sub-calculation as part of evaluating an expression. This is especially the case when within if expressions where you can't put a variable declaration.

This is where 'let expressions' come into play. The syntax for a let expression is:

```
let var1 = expression [, var2 = expression]... in <expression>
```

Here's an example of using a let expression in defining a `filter` function.

```
filter = λ predicate list {
    if length(list) == 0 then [] else
    let elem = head(list),
        remaining = filter(predicate, tail(list))
    in
    if predicate(elem) then [elem] + remaining else remaining
}

filtered_list = filter(λ x { x < 2 }, [1, 2, 3, 4])
```

## Importing and Exporting

neobem has a fairly straightforward method of importing and exporting. The syntax for importing from another file is:

```
import <URI> [as prefix_identifier] [only (identifier1, identifier2, ...)]
```

The only portion that is required is the *expression* `URI`. Generally, this string is a relative path to a file on local machine. For example,

```
import 'utilities/my_utilities.nbem'
```

will import a 'my_utilities.nbem' file in the 'utilities' folder that is in the same directory as the executing script.

Notice that this is not necessarily a string, but any arbitrary *expression* that evaluates to a string. For example:

```
file_based_on_if_statement = if true then 'file1.nbem' else 'file2.nbem'
import file_based_on_if_statement
```

**Important note**: Paths are *relative from the script location, not from the current working directory of execution.* If this weren't the case, running `neobem` from different locations would affect the outcome.

```
nbem in.nbem
```

would be different from:

```
nbem sub_folder/in.nbem
```

The URI can also be a normal Internet URL. For example, you can test an example file from GitHub.

```
import 'example'
```

Important: *By default, identifiers that are imported are imported directly into the same namespace as the calling script, with the imported identifier overwriting any existing identifier.*

So for example:

```
my_template = \ value {
Schedule:Constant,
  Const <value>, ! Name
  ,              ! Schedule Type Limits Name
  <value>;       ! Hourly Value
}

import 'importfile.nbem'

print my_template(10)
```

where **importfile.nbem** has the contents

```
my_template = λ value {
  Material:AirGap,
    <value> Air Gap,          ! Name
    <value>;                  ! Thermal Resistance {m2-K/W}
}

export (my_template)
```

results in

```
Material:AirGap,
  10 Air Gap, ! Name
  10;         ! Thermal Resistance {m2-K/W}
```

To avoid conflicts, you can make use of the `as` and `only` options of importing.

The `as` option uses the specified identifier that follows as a prefix, with an '`@`' character between.

So if our example above instead used (note the `as my_import` option):

```
my_template = \ value {
Schedule:Constant,
  Const <value>, ! Name
  ,              ! Schedule Type Limits Name
  <value>;       ! Hourly Value
}

import 'importfile.nbem' as my_import

print my_template(10)
```

results in

```
Schedule:Constant,
  Const 10, ! Name
  ,         ! Schedule Type Limits Name
  10;       ! Hourly Value
```

If the imported function was desired, then it would be called like:

```
my_template = \ value {
Schedule:Constant,
  Const <value>, ! Name
  ,              ! Schedule Type Limits Name
  <value>;       ! Hourly Value
}

import 'importfile.nbem' as my_import

print my_import@my_template(10)
```

If only certain identifiers are desired to be imported, the `only (identifiers, ...)` Syntax can be used. It can be used in combination with the `as` option as well.

### Controlling What Gets Imported

When an import statement is called, the compiler moves to the imported file and serially executes all the statements contained within.

*Any object declaration in the imported file is passed on to the final result.* This is how you can import a normal idf file and have it pass along to the final output.

However, variable and function definitions are not passed along to the calling environment *unless they are exported.*

The syntax for exporting identifiers is:

```
export (identifier1, identifier2, ...)
```

For example, if the following file is imported:

```
my_template = \ name {
Zone,
  <name>,        ! Name
  0,             ! Direction of Relative North {deg}
  0,             ! X Origin {m}
  0,             ! Y Origin {m}
  0,             ! Z Origin {m}
  1,             ! Type
  1,             ! Multiplier
  autocalculate, ! Ceiling Height {m}
  autocalculate, ! Volume {m3}
  autocalculate, ! Floor Area {m2}
  ,              ! Zone Inside Convection Algorithm
  ,              ! Zone Outside Convection Algorithm
  Yes;           ! Part of Total Floor Area
}

const_schedule = \ value {
Schedule:Constant,
  Const <value>, ! Name
  ,              ! Schedule Type Limits Name
  <value>;       ! Hourly Value
}

Version,
    9.4;

export(const_schedule)
```

`Version, 9.4` will be printed to the output, but only `const_schedule` will be available for use from the script that has imported this file.

## Debugging

Currently, debugging is somewhat crude. As of now, there is no step by step debugger. What is available is a special keyword `log`. It can be invoked similarly to the `print` statement.

```
log expression
```

What the log statement does is evaluate the entire expression, and then prints a detailed representation of the expression to the *standard error* output stream. This allows for the debug information to normally be printed to the terminal, without ruining the final compiled idf file.

## Design Goals

- Be a superset of the idf input file format.

  Current building energy simulation programs should be valid. This lets users start small with existing input files.

- Expose simple programming concepts for those unfamiliar with programming.

  The core of the language is simple, variable assignments, functions, and logic.

  And while I love statically typed languages, it doesn't make sense for this domain of problems. Neobem is not meant to replace general purpose programming languages. It is a language designed specifically for generating comma separated data for input files. So there is no need to add verbosity to the language in the form of types.

- Be "Batteries" included.

  The user shouldn't expect to have to download additional packages for the program to be useful. Important functions should be built right into the language.

- Cross-platform.

  All users should be able to use tool regardless of OS. The Windows, Mac, and Linux experience should be first class.

- Once initial stability is reached, backwards compatibility should be taken *extremely* seriously.

  Projects in building construction last *years* and the buildings themselves last *decades*. You should expect to be able to run your simulation model years later and it should still work.

- Installation and getting started should feel easy. The same should be for the uninstall process.

  There should be efforts taken to make sure new users can get up and running quickly. This means offering convenient methods of installation - This may include:

  - Different flavors of package managers for Linux
  - Windows Store
  - `scoop`, `chocolately`, and the new Windows Package Manager `winget`
  - Standalone `.exe` installers
  - `brew` for MacOS
  - Documentation on building from source

  It may be difficult for me to accomplish this goal by myself, may need much help from other contributers!

- High quality documentation.

  Building energy modeling professionals shouldn't have to also be software engineers. The audience here may have never programmed before. Thorough, hand-crafted not generated, documentation is required.

- Provide surrounding tooling.

  For users who haven't spent years modifying their programming environment, it may not be a comfortable experience editing these text files in Notepad. We should:

1. Provide syntax highlighting/autocompletion files for various text editors.
2. Provide GUI wrappers around the program.
3. Provide an implementation of the Language Server Protocol

And to be clear, these are the goals and driving principles, it doesn't mean all this is finished already :)

## FAQ

### What's with the name?

The name actually has gone through several iterations. It began as idf-plus as a play on "EnergyPlus" and "C++". However, I realized that other input files, such as the format expected by the DOE-2 engine, could also use this, so I decided to remove the 'idf' portion from the name.

It then went to 'bemp', because I wanted something short, pronounceable, and it could stand for many things:

1. "Building Energy Modeling Programming" since that's what we're doing here.
2. A play on the ASHRAE certification "Building Energy Modeling Professional"

I didn't continue with this name since it was not unique, and searching for 'bemp' online comes up with the ASHRAE certification items along with other acronyms.

So its current name is Neobem. It's partially an ode to Neovim, the text editor that I will do most raw editing of idf files and now these input files. Plus, throwing 'Neo' in front of nouns makes things cooler, right? Searching the internet for Neobem didn't result in hits, so that should be helpful for people in the future searching for help.

The executable name is `nbem`, to match the expected extension for these files. I do realize this is going to have to be front and center in the documentation for those attempting to run `neobem.exe` and not finding it. But that's the price you pay for a unique name and reasonably short executable and file extension.

## Thank Yous and Inspiration

*If I have seen further it is by standing on the shoulders of Giants.*

A project such as this does not occur in a vacuum. I am indebted to many tools and resources that have made this possible.

First and foremost, the most thanks goes to ANTLR. This project would not exist without it. It makes designing grammars so straightforward that a mechanical engineer such as myself can do it. Anyone designing programming languages should give ANTLR a try.

While my personal vision for the language should be evident, I've stolen many good ideas from languages before me. Some inspirations include:

- The general purpose is analogous to Sass for compiling to CSS.
- The functional programming style comes from my experience with Haskell. The lambda grammar is inspired in part from Haskell.
- The logic operators `and` and `or` come from Python. I think that they are clearer than the usual `&&` and `||` and only require one extra character. I also prefer the snake case convention for identifiers.
- The dictionary grammar matches closely to that of JSON.