Config File Language Documentation

R. Bullock

The language used for configuration files is a very stripped-down version of C, with a few extensions. A config file is a sequence of statements. A statement may be either an assignment of a value to a variable, or a function or array definition. All statements end with a semicolon.

Comments

Either C or C++ style comments may be used. Unlike in the C language, C-style comments may be nested.

Scalars

Scalars (numbers) come in two types: integer and floating-point. The syntax for integers is as follows:

```
digit              : [0-9]

digits             : {digit}+

integer            : -?{digits}
```

That is, an integer is an optional minus sign followed by one or more digits. Integers are represented internally as signed 32-bit integer data types. Examples of valid integer constants are

```
0          1000          -98234
```

Floating-point numbers are represented internally as double precision data types. The syntax for a floating-point number is

```
exp                : (e|E)(-?){digits}

opt_exp            : {exp}?

floating_point     : -?{digits}{exp}
                   | -?"."{digits}{opt_exp}
                   | -?{digits}"."{opt_exp}
                   | -?{digits}"."{digits}{opt_exp}
```

Examples of valid floating-point constants are

```
1.        .002       3.14159      -.45        0.0
1e5       2E-3       .12e30       -1.e10      -1.234e-56
```

Floating-point constants *must* have either a decimal point or an exponent (or both) to distinguish them from integers. Note that one can use either a capital or a lower-case “e” in scientific notation.

Identifiers

Identifiers are used as variable, function, or array names. The syntax for identifiers is as follows:

```
letter             : [A-Z] | [a-z] | _

identifier         : letter (letter | digit)*
```

Roughly, an identifier is a letter followed by zero or more letters or digits. The underscore character “_” is considered to be a letter. Examples of valid identifiers are

```
x          XYZ       gamma_prime       z50
```

Expressions

This is where it gets complicated. Basically expressions are formed from numbers, identifiers, function evaluations, and array elements, and other expressions by applying the basic arithmetic operators. Examples of valid expressions are

```
      5          -x        3*y + z    f(x + 7.2)   a[5] + 9
  abc(x, 7)      x^3       f(g(y))     1/(x + 1)     x - zp
```

It's important to remember the difference between integer and floating-point arithmetic when writing expressions. As an example, `3.0/4.0` evaluates to `0.75`, but `3/4` evaluates to zero. Generally, if a binary arithmetic operator is given two integers as operands then integer arithmetic will be used, but if one or both operands are floating-point, then floating-point arithmetic is used.

An exception to this is the exponentiation operator "^". Floating-point arithmetic is *always* used for exponentiation, with the single exception that `a^2` is an integer if `a` is an integer.

Assignments

An assignment statement simply assigns a value to an identifier. The syntax is

```
identifier = expression ;
```

Examples of valid assignments are

```
x = 5;

voltage = sin(omega*t + phi);

grid_spacing = 22.0;   // km

value_50 = foo*bar;
```

All identifiers on the right hand side (RHS) of an assignment must have been previously defined.

Character strings are also allowed in assignments:

```
data_path = "/home/user/algorithm_output";
```

This is the only context in the language where character strings are allowed.

Functions

Functions can have up to ten arguments, and return scalar values. Examples of function definitions are

```
f(x) = 5*x + 4;

scale(horizontal, vertical) = f(horizontal)/f(vertical + 17);

d_scale(x, y, z, t) = scale(x, y)*scale(z, t);
```

Arguments to functions are dummy variables, local to the function. For example, the `x` in the definitions above has nothing to do with any identifier named `x` that may have been previously defined in the config file.

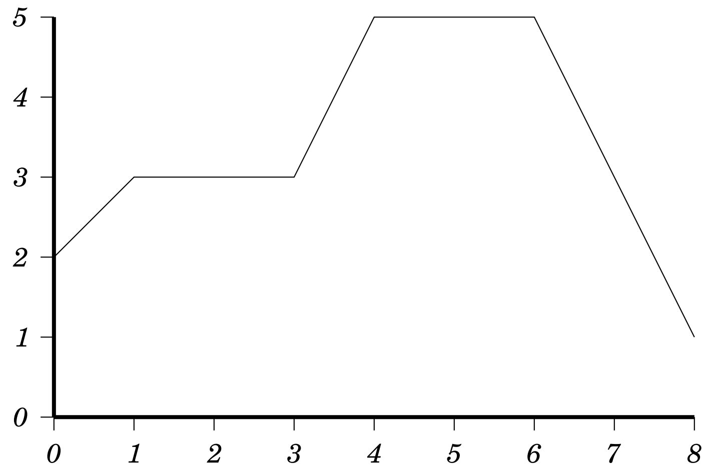
Not all arguments need be used in the expression for a function. The following definition is perfectly legal:

```
MY_func(x1, x2, x3, x4) = 5;
```

As with assignments, nothing (except for dummy arguments) can appear on the RHS that hasn't been previously defined.

Piecewise-Linear Functions

A piecewise-linear function is a function whose graph consists of straight line segments.



A function like this is specified by listing the corner points on it's graph, like so:

```
pwl_func = {

      ( 0.0, 2.0 )
      ( 1.0, 3.0 )
      ( 3.0, 3.0 )
      ( 4.0, 5.0 )
      ( 6.0, 5.0 )
      ( 8.0, 1.0 )

};
```

Once defined, the function can be used in assignments with the usual function-call syntax:

```
y_val = pwl_func(x^2 + 0.59);
```

Piecewise-linear functions of more that one variable are not yet implemented. I know how to do it, but haven't gotten around to it yet.

Piecewise-linear functions are constant outside their implied domain. In the above example, for instance, `pwl_func(x)` will evaluate to 2 if x is less than zero, and to one if x is greater than 8.

The x and y coordinates of points can be expressions. Notice also that the list of points is not separated by commas.

Built-In Functions

Many commonly used functions are built into the language, as a convenience to the user, and because many of them would be difficult or impossible for the user to define within the language.

`sin, cos, tan`

These are the usual trigonometric functions. They take one floating-point argument (in radians) and return a floating-point value.

`asin, acos, atan`

Inverse trigonometric functions. They take one floating-point argument and return a floating-point value (in radians).

`sind, cosd, tand`

These are the same as `sin`, `cos`, `tan` except the argument is in degrees.

`asind, acosd, atand`

These are the same as `asin`, `acos`, `atan` except the return value is in degrees.

`log, exp`

The natural logarithm and its inverse.

`log10, exp10`

Logarithm base 10 and its inverse. The expression `exp10(x)` is equivalent to `10^x`.

`sqrt`

Square root. The expression `sqrt(x)` is equivalent to `x^0.5`.

`abs`

Absolute value. If given an integer argument, an integer is returned. Otherwise the return value is floating-point.

`floor, ceil`

These are the same as the corresponding functions in the C math library. `floor` rounds a floating point argument down to an integer, while `ceil` rounds up. Return value is floating-point.

`nint`

Nearest integer function. It takes one floating-point argument and returns an integer value.

`sign`

Signum function. It returns an integer value of -1 if the argument is negative, +1 if the argument is positive, and 0 if the argument is zero.

`step`

Step function. It returns an integer value of 0 if the argument is negative, 1 otherwise.

`atan2`

Two-argument arctangent function. This works the same way as the one in the C math library. The expression `atan2(y, x)` gives the polar-coordinate angle of the point (x, y) in the correct quadrant. Note the reversal of the order of x and y in the argument list. Return value is in radians, in the range $-\pi$ to π.

`atan2d`

Same as `atan2` except return value is in degrees.

`arg`

A variation on the two-argument arctangent function that takes `x` as its first argument rather than `y`.

The expression `arg(x, y)` is equivalent to `atan2(y, x)`. Return value is in radians.

`argd`

Same as `arg` except return value is in degrees.

`min, max`

Minimum and maximum functions. They take two arguments. If both arguments are integers, an integer value is returned, otherwise a floating-point value is returned.

`mod`

Modulus function. Takes two arguments. If both arguments are integers, an integer value is returned, otherwise a floating-point value is returned. Examples: `mod(17, 5)` evaluates to `2` and `mod(17.3, 5)` evaluates to `2.3`. Unlike the similar " `%` " operator in C, the return value of `mod` is always non-negative. The expression `mod(a, 0)` evaluates to `a`.

Arrays

Arrays of dimension up to 10 are supported. A 1-dimensional array can be defined as follows:

```
a[] = [ 5, 4, 3, 2, 1 ];
```

For arrays of higher dimension, the size in each dimension must be explicitly declared:

```
r_values[2][3] = [
                    3, 4, 5,
                    6, 7, 8
                 ];
```

As in C, the last index varies fastest.

Once defined, array elemets can be used in expressions:

```
y = 5 + r_values[0][1];
```

Elements of arrays can be expressions. Array indices can be expressions — if the value of the expression is not an integer, it will be rounded to the nearest integer with `nint`. Array indices always start at zero.

Examples & Tips

Here are some examples of the language in use.

Commonly used mathematical constants are easily obtained:

```
pi = 4*atan(1);

e = exp(1);

golden_ratio = (1 + sqrt(5))/2;
```

Define whatever functions you need. For example, logarithm and exponential base 2 are not built in, but are easy to define:

```
log2(x) = log(x)/log(2);

exp2(x) = 2^x;
```

Remember to keep in mind the distinction between integer and floating-point arithmetic. Consider the following statements:

```
f(x) = x/10;

y = f(9.0);

z = f(9);
```

Then `y` will be a floating-point scalar with value `0.9`, but `z` will be an integer with value zero. If we had written instead

```
f(x) = x/10.0;
```

then both `y` and `z` would be floating-point scalars with value `0.9`.

The exponentiation operator "^" has highest precedence of all the binary arithmetic operators. Thus the expression `2*3^5` is interpreted as `2*(3^5)`, and `2^3*5` is interpreted as `(2^3)*5`. Use parenthesis whenever you're not sure about operator precedence. Liberal use of parenthesis will *not* slow down the evaluation of an expression. The two expressions `(a + b)/c` and `(((a + b)))/((((c))))` have *exactly the same* internal representation. Just make sure your parenthesis are *balanced*, please!

You can use expressions in piecewise-linear function definitions and in array definitions.

```
corner = 0.8;

g = {

      ( 0.0,    0.0 )
      ( corner, 1.0 )
      ( 1.0,    1.0 )

    };

a[] = [ 0.5, sin(x + y), atand(0.9) ];
```

You can also use expressions in array indices:

```
y = a[min(x, 2)];
```

You can define parameters that make things tunable. Suppose you have an interest map `imap` that is a function of distance (in grid units) on a 40-km grid. To use this on a different grid you could say:

```
grid_size_km = 10.5; //  size of this grid

scale_factor = 40.0/grid_size_km;

imap_new(dist) = imap(dist*scale_factor);
```