Summary of Some Common Jarnac Language Constructs

Sample model definition:

```
p = defn cell
     var S1,S2;    // floating species
     ext X0,X1;    // boundary species (remain constant)

     // Declare each reaction and its associated rate law
     J1: X1 -> S1; k1*X0-k2*S2;
     J2: S1 -> S2; Vm*S1/(Km+S1);
     J3: S2 -> X1; k3*S2^n;
end;
```

// Initialize all the parameters and variables

```
p.X0=1.2;   p.X1=0.0;
p.S1=0.001; p.S2=0.001;
p.k1=1.2;   p.k2=5.6;
p.Vm=10.5;  p.Km=0.4;
p.k3=5.6;   p.n=3.5;
```

// Define some convenient variables if necessary

```
TimeStart=0;
TimeEnd=10;
NumberOfDataPoints=100;
```

// Perform the simulation

```
m = p.sim.eval(TimeStart, TimeEnd, NumberOfDataPoints,
       [<p.Time>, <p.S1>, <p.S2>, <p.J1>]);
```

// The above statement returns a matrix which we can graph

```
graph(m);
```

Reaction specifications, examples:

```
J1: 2 A -> B + C;       k1*A^2*B;
J2: 2 ADP -> ATP + AMP; k1*ADP^2 – k2*ATP*AMP;
```

Reactions can be named, eg J1 and J2 above, this allows the rate of a reaction to be made available.

Computing the Steady State

p.ss.eval ();

Loops and If/Then Statements

For-Loop :

```
for i=1 to 100 do
    begin
      <code>
    end;
```

If-statement:

```
if a == 4 then
   // begin-end only needed for multiple statements
   begin
     <code>
   end
else
   begin
     <code>
   end;
```

printing:

```
println "statement", x, y, z;
```

Matrix operations:

```
m1 = matrix (10,5);  // Construct a 10 x 5 matrix
v1 = matrix (10,1);  // Construct a column vector
v2 = matrix (1,6);   // a row vector

m2 = aug (m1,v1);    // Augment the columns of m1 and
                     // v2 together
m3 = augr (m1,v2);   // Augment the rows of m1 and
                     // v2 together

println m[3,4];      // Access a particular element

mt = tr (m);         // Compute the transpose
v = m[5];            // Copy an entire row
```

```
mt = tr (m);          // Use this trick to extract a
col2 = mt[2];         // column, in this case the 2nd
                      // column
```

Example of how to use the augr() function

```
p = defn newModel
   $Xo -> S1; k1*Xo;
   S1 -> $X1; k2*S1;
end;

p.k1 = 0.2;
p.k2 = 0.4;
p.Xo = 1;
p.S1 = 0.5;

// Simulate the first part up to 20 time units
m1 = p.sim.eval (0, 20, 100, [<p.time>, <p.S1>]);

// Perturb the concentration of S1 by 0.35 units
p.S1 = p.S1 + 0.35;

// Continue simulating from last end point
m2 = p.sim.eval (20, 50, 100, [<p.time>, <p.S1>]);

// Merge and plot the two halves of the simulation
graph (augr(m1, m2));
```

Other Useful Methods:

```
m = timeSlice (matrix, lowerTime, upperTime;
```

This call will take a matrix and returns all rows between lower and upper time bounds. It is assumed that the first column of the matrix contains the time variable with ascending values.

```
v = getColumn (matrix, index);
```

This function returns a single column from a matrix m at column index and returns the column as a vector.

```
m = getColumns (matrix, [3,4,1,2]);
```

This function extracts a set of columns from a matrix argument and the returns the columns in the form of a new matrix. The columns are selected from the list argument where elements of the list indicate column indices.

```
exportCSV (matrix)
```

or

```
exportCSV (matrix, separator)
```

Exports the matrix, m as CSV data. Depending on your computer setup, an appropriate application will be launched (eg. Excel, OpenOffice) to load the data. This is useful for quickly transfers a matrix of data to another application such as Excel or OpenOffice. The second version of exportCSV() allows one to specify the separator between data values. The default is to separate data values using ‘,’ but this may not always be appropriate.

Empty model:

```
p = defn model
end;
```

Including comment lines:

```
p = defn model
    // This is a comment
end;
```

Model with one reaction:

```
p = defn model
    S1 -> S2; k1*S1;
end;
```

Rate laws can be arbitrary expressions.

Initialize values:

```
p = defn model
    S1 -> S2; k1*S1;
end;
p.k1 = 1.2;
p.S1 = 10;
p.S2 = 0;
```

Additional Examples

Model with multiple reactions:

```
p = defn model
    S1 -> S2; k1*S1;
    S2 -> S3; k2*S2 – k3*S3;
    2 S3 -> S1; k4*S3^2;
    end;
```

Named reactions:

```
p = defn model
    J1: S1 -> S2; k1*S1;
    J2: S2 -> S3; k2*S2;
end;
```

Setting boundary species:

```
p = defn model
      // Use the '$' character to
      // indicate a boundary species
      // all other species a floating
      J1: $S1 -> S2; k1*S1;
      J2: S2 -> $S3; k2*S2;
end;
```

or alternatively explicitly declare boundary and floating species:

```
p = defn model
    ext S1, S3;
    var S2;

    J1: S1 -> S2; k1*S1;
    J2: S2 -> S3; k2*S2;
end;
```

Assignments in a model:

```
p = defn model
        v1 = k1*S1;
        v2 = k2*S2;

        J1: S1 -> S2; v1;
        J2: S2 -> S3; v2;
end;
```

Tricks: Specify differential equations

```
p = defn model
    ODE1: S1 -> $w;  k1*S1;
    ODE2: S2 -> $w;  k1*S1 - k2*S2;
    ODE3: S3 -> $w;  k2*S2;
end;
```

will generate the following ODEs:

$$\frac{dS1}{dt} = k1\ S1$$

$$\frac{dS2}{dt} = k1\ S1 - k2\ S2$$

$$\frac{dS3}{dt} = k3\ S3$$

Example of a published model in Jarnac script:

```
p = defn Jana_WolfGlycolysis

var Glucose, fructose_1_6_bisphosphate,
glyceraldehyde_3_phosphate, glycerate_3_phosphate,
pyruvate, Acetyladehyde, External_acetaldehyde, ATP, ADP, NAD,
NADH;

ext External_glucose, ethanol, Glycerol, Sink;

J0: External_glucose -> Glucose; J0_inputFlux;
J1:  Glucose + 2 ATP -> fructose_1_6_bisphosphate + 2 ADP;
J1_k1*Glucose*ATP*(1/(1+pow(ATP/J1_Ki,J1_n)));
J2:  fructose_1_6_bisphosphate -> glyceraldehyde_3_phosphate +
glyceraldehyde_3_phosphate; J2_k*fructose_1_6_bisphosphate;
J3:  glyceraldehyde_3_phosphate + NADH -> NAD + Glycerol;
J3_k*glyceraldehyde_3_phosphate*NADH;
J4: glyceraldehyde_3_phosphate + ADP + NAD -> ATP +
glycerate_3_phosphate + NADH;
(J4_kg*J4_kp*glyceraldehyde_3_phosphate*NAD*ADP-
J4_ka*J4_kk*glycerate_3_phosphate*ATP*NADH)/(J4_ka*NADH+J4_kp*ADP
);
J5:  glycerate_3_phosphate + ADP -> ATP + pyruvate;
J5_k*glycerate_3_phosphate*ADP;
J6:  pyruvate -> Acetyladehyde; J6_J6_k*pyruvate;
J7:  Acetyladehyde + NADH -> NAD + ethanol;
J7_k*Acetyladehyde*NADH;
J8:  Acetyladehyde -> External_acetaldehyde; J8_k1*Acetyladehyde-
J8_k2*External_acetaldehyde;
J9:  ATP -> ADP; J9_k*ATP;
J10:  External_acetaldehyde -> Sink; J10_k*External_acetaldehyde;

end;

p.External_glucose = 0;
p.ethanol = 0;
p.Glycerol = 0;
p.Sink = 0;
p.Glucose = 0;
p.fructose_1_6_bisphosphate = 0;
p.glyceraldehyde_3_phosphate = 0;
```

```
p.glycerate_3_phosphate = 0;
p.pyruvate = 0;
p.Acetyladehyde = 0;
p.External_acetaldehyde = 0;
p.ATP = 3;
p.ADP = 1;
p.NAD = 0.5;
p.NADH = 0.5;
p.J0_inputFlux = 50;
p.J1_k1 = 550;
p.J1_Ki = 1;
p.J1_n = 4;
p.J2_J2_k = 9.8;
p.J3_J3_k = 85.7;
p.J4_kg = 323.8;
p.J4_kp = 76411.1;
p.J4_ka = 57823.1;
p.J4_kk = 23.7;
p.J5_k = 80;
p.J6_k = 9.7;
p.J7_k = 2000;
p.J8_k1 = 375;
p.J8_k2 = 375;
p.J9_k = 28;
p.J10_J10_k = 80;
```