# EQN/Wind Fern

Version 7.5

## Developer’s Guide

Written by Scott Auge
and Contributors

sauge@amduus.com  scott_auge@yahoo.com

# Table of Contents

## History

One time on the peg (www.peg.com) there was a question about how to make software written with the Progress 4GL to accept a user's input of an algebraic expression and then reduces that expression down into a single decimal or integer number.

There was source to accomplish this, but it seemed to have gotten corrupted. (See http://peg.com/utilities/solver.html). Indeed a very disappointing event.

Since I was writing Denkh HTML Reporter at the time, I decided that I needed such a piece of software. And since there was a calling for it on the peg, the Progress 4GL world seemed to need such a piece of software. Also, I have always been interested in symbolic mathematics (Mathematica really turns my crank) – I was on a new mission.

The software began as a simple algebraic calculator. Send it an algebraic expression in a string and it would return a string representing the numeric result. (See BatCalcAlg.txt for more on it's capabilities!)

The idea was that software using the system would be able to keep equations in a parameter file or database table allowing the user to manipulate not just the numbers for a value in the system, but how to calculate that value to begin with! Imagine allowing a user to not only tweak numbers of an inventory level equation, but to configure the actual equation to determine inventory levels!

Soon it became obvious we should have the ability to have user defined formulas available to the application. Users needed some formula's that would be used in other formulas. Having a notation and ability for the system to know about all the formulas without programmer intervention was needed.

Users being users soon discovered that under some conditions values needed to be evaluated in one fashion and under other conditions they needed it evaluated in another fashion. Formulas are good for functional decomposition, but they were evaluated the same way all the time. Sometimes formulas needed to be evaluated multiple times with different values. User's needed the power to set a formula for a value under varied conditions, and so they needed a language to aid in when to calculate a value and when to define how to calculate a value.

Hence, here is the package. Be sure to check out the case studies on use of the software near the end of this document. They may give you ideas about features to add to your software you never knew possible!

Scott Auge
October 20, 2003

## License

Why do I readily release the source for this toolkit?

1. I want people to actually use it and maybe understand how to use it. Maybe even add to it or fix it.
2. I don't want an "Appgen incident" regarding the source. See http://www.newsforge.com/print.pl?sid=03/10/18/1814211 for more information about such things. Apparently a company providing accounting tool kits disappeared, leaving all the VARs and their customers high and dry.
3. Let people know what I can do.

Does it mean I don't want any money for it? Oh hell no. I want any kind of donation you would like to make. One way to measure it is perhaps half the rate it would have taken ya all to have made it yourselves.

Even if you're not donating, at least tell me how many people are on the application using this toolkit, and the number of companies that might be using it. It will help me measure interest in keeping this product alive. Even if I don't – hey – ya all got the source….

As of now, the tally is:

| | |
|---|---|
| Total Number of Users | 573 |
| Total Number of Companies | 6 |
| Total Number of Applications | 3 |

## Document Change Log

See README in the source co

de directory regarding changes to the software.

| Date | Author | Digest |
|---|---|---|
| 08/11/04 | Scott Auge | Changed DISPLAY to PUT in RunPrg.p so that same r-code can be used across multiple display devices. |
| October 20, 2003 | Scott Auge | Initial Document |
| November 7, 2003 | Scott Auge | Added section on if statement<br>Added section on streval statement |
| December 13, 2003 | Scott Auge | Added sections about error detecting |
| February 3, 2003 | Scott Auge | Information about how to use dates |
| April 14, 2004 | Scott Auge | Added information about using formula's in streval statement. |

| Contributors | |
|---|---|
| Marius Fiere | Conversion of EQN algebraic parser into persistent procedure with dynamic functions. See eqn/src/v9. *Not all changes to EQN since this conversion have been implemented in the persistent procedure. Perhaps someone will be willing to port that code changes into this code.* |
| ARC Systems | Keeping me employed so I can make this sort of code available. |

## Compatibility

EQN as tested by BatTestCalc.p has shown compatibility with:

Progress 4GL Version 8.3E
Progress 4GL Version 9.1C
Progress 4GL Version 9.1D
Webspeed 2.1
Webspeed 3.1C
Webspeed 3.1D

EQN can be used with the following interface types:

GUI
CHUI
Batch
WWW

EQN has been tested on the following operating systems:

Linux
Solaris
Windows 2000

NOT ALL PIECES OF EQN ARE V8 COMPATIBLE. THESE V8 INCOMPATIBLE SECTIONS ARE MENTIONED LATER IN THE DOCUMENTATION. IT DOES NOT MEAN YOU CANNOT USE IT WITH V8, JUST DON'T EXPECT THAT FEATURE TO WORK ON V8!

Pure V9 is in the /src/v9 subdirectory.

## Using EQN to calculate algebraic expressions

The 4GL programmer to compute out an algebraic expression can call upon EQN. If you remember from arithmetic, an algebraic expression has a certain order of operations regarding evaluation. This is opposed to something like reverse polish notation which computes out the result in a more linear fashion.

For example an algebraic expression is:

```
(2+3) * 4
```

It’s RPN version would be:

```
2 3 + 4 *
```

EQN does not *solve* an algebraic expression. It merely evaluates them. So something like:

```
(2 + x) * 4 = 16         EQN cannot solve algebraic expressions!
```

Simply will make the system puke!!!

It is best to call EQN via the Calc.p API.

An example follows:

```
/* Enter the expression to evaluate into a string */

ASSIGN cT = “(2 + 3) * 4”.

/* Call into EQN to evaluate via the Calc.p API */

RUN Calc.p (INPUT cT, OUTPUT cR).

/* Convert the result from CHAR to DECIMAL or INTEGER */

ASSIGN dR = DECIMAL (cR).
```

➡ Be sure to see Case Study One for a real world application of this use!

## Using EQN with formulas

Formulas can be considered placeholders for expressions. It gives a name to a calculation and describes how to compute it out.

Formulas can be composed of:

A number
An algebraic expression
A date token (See Dates)
Other formulas

All formula names must begin with a @ sign. No formula name may contain a space. Formula names can be made up of the letters of the alphabet, the numbers, and the _ symbol. The name can be of any length, but I suggest no more than 40 characters.

Some examples:

| Formula Name | Expression |
|---|---|
| @Revenue | 10000.00 |
| @Expenses | 5000.00 |
| @NetProfit | (@Revenue - @Expenses) / 90 |

Formula’s are inserted into the EQN system via the AddFormula.p API.

```
/* Insert formulas.  Note these ARE strings for numbers. */

RUN AddFormula.p (INPUT “@Revenue”, INPUT “10000.00”).
RUN AddFormula.p (INPUT “@Expenses”, INPUT “5000.00”).
RUN AddFormula.p (INPUT “@NetProfit”, INPUT “(@Revenue – @Expenses) / 90”).

/* Compute out a formula with Calc.p */

RUN Calc.p (“@NetProfit”, OUTPUT cResult).

/* cResult should be 55.55555 */
```

➡ Be sure to see Case Study Two for a real world application of this use!

## Using EQN's programming language

EQN has the ability to programmatically define and evaluate formulas in the Progress 4GL application. You can place these programs in a database table or a disk file and then execute them via the RunPrg.p API into EQN.

The programming language allows you to:

Define formulas
Evaluate formulas
Create sub-routines
Use date tokens
Branch to other locations in the program

When you are finished calling RunPrg.p, all the formula's created in the program will be available to your 4GL code via the Calc.p API.

In the source code for the API is a sub-directory called eqnprgs. Be sure to look some of those programs over to learn by example how to use the programming language.

### *Statement Format*

All statements end with a ;. The program is not case sensitive unless Progress is turned on to be case sensitive.

### *Comments*

Comments begin with // and end with a ;. For example:

```
// This is a comment EQN can understand;
```

### *Define*

```
Define [FormulaName]=[Formula Expression]
```

This defines a formula that eval would evaluate later one. It lets the programmer set up formulas that represent something on its own, but yet is a part of the whole.

It is equivalent to calling the AddFormula.p API.

Example:

```
define @b=@a + 5;
```

Will define a formula called @b. When @b is evaluated, the system will also evaluate @a, then add 5 to it.

➡ *Do not space around the equal sign! The parser is very simple and spacing is important!*

## *Eval*

This causes the program executor to actually evaluate an equation down to a numeric result. Any formulas are reduced to a number, and then that number is used in further calculations till the result is one decimal or integer number.

It is equivalent to calling the Calc.p API.

For example:

```
eval @Fee=0.10 * @BaseFee;
```

➡ *Do not space around the equal sign! The parser is very simple and spacing is important!*

## *Nop*

This stands for no-operation. These are used as placeholders for jump points accessed by jsr, jmp, and normal flow of program execution. For example:

```
nop ComputeInternal;
eval @Fee=0.10 * @BaseFee;
define @Msg=ComputeInternal;
brk;
```

## *Jmp*

This allows the user to conditionally jump to a new location in the program. These locations are identified by the label associated with a nop statement.

For example:

```
jmp BreakOutMySub @c = 10;
```

One can also jmp without a condition:

```
jmp BreakOutMySub;
```

## *Jsr*

This statement works very much like the jmp statement, but when the program executor encounters a rtn; statement, control will flow to the following statement of the jsr.

For example:

```
// Program to show jsr is working;
// Upon completion, @JumpCount should hold 2;

define @JumpCount=0;

jsr IncrJumpCount 1 = 1;
jsr IncrJumpCount 1 = 1;

brk;

nop IncrJumpCount;
eval @JumpCount=@JumpCount + 1;
rtn;
```

Should a jsr statement be at the end of a program, best practice is to follow it with a nop or a brk so the program executor has someplace to go after encountering a rtn statement.

The jsr statement can also be used without a condition:

```
jsr IncrJumpCount;
```

### *Brk*

This statement halts program execution and control returns to the Progress 4GL program.

Example:

```
define @Msg=if1 fired;
brk;
```

### *Rtn*

This statement returns the flow of execution to the statement following the most previous jsr statement.  Jsr's can be nested, and so there should be the same number of rtn; encountered as jsr; (unless a brk; is encountered!)

Example:

```
jsr MySub @c = 0;
brk;

nop MySub;
jmp BreakOutMySub @c = 10;
eval @c=@c + 1;
jsr MySub 1 = 1;

nop BreakOutMySub;
rtn;
```

## *list*

This statement allows a human readable listing of the formula's and their current standings. This will APPEND to any existing file. It is good for debugging.

Example:

```
list /tmp/formula.txt;
```

Results in:

cat tststreval1.list

```
Category             Name                            Setting
-------------------- ------------------------------ -----------------
Test                 @msg                              $34.00
```

## *export*

Since the formula's are actually in a temp table called Formula, the records can be exported and imported from disk files. This will over-write any existing files.

Example:

```
export /tmp/formula.export;
```

## *import*

Since the formula's are actually in a temp table called Formula, the records can be exported and imported from disk files.

Example:

```
import /tmp/formula.export;
```

## *jeqn*

This function allows you to call an EQN program module stored in EQNPrg. On release, the EQNPrg table is a temp-table made available in the eqnprgtbl.i file. One may want to make the table permanent in the database, or have someway of populating this table BEFORE executing an EQN program.

I/O is made via the shared formula table found in formulatbl.i. To make input and output values, merely create formula's and re-define formulas.

Example:

```
jeqn InitInventoryFormulas;
```

## *j4gl*

This routine allows your EQN program to call into a Progress 4GL program.

The called Progress program should include `formulatbl.i` if it needs to pass information to and from.

The Progress 4GL program needs to be in the PROPATH of the process using EQN.

This is a good way to create database or interface access from EQN to send and receive information from EQN into databases, files, or sockets.

Example:

```
j4gl DB2Formula.p;
```

## *load*

The load statement allows the programmer to load an EQN program from the disk file system.  Once a program is loaded, the jeqn statement can be used to actually execute that program.

For example:

```
// $Id: tstload.eqn,v 1.1 2003/10/28 03:08:36 sauge Exp $;

load eqnprgs/t5;

jeqn eqnprgs/t5;

list /tmp/tstload.list;
```

## *if*

This allows an if based control.  It should not be confused with the function if[] which evaluates based on a condition.

It's syntax is

```
if (conditional) expr;
```

where **conditional is the same for the function if[]** and expr is a single statement in the grammer of the EQN language.

Some examples are given below:

```
// Test if

define @msg=no;
define @msg1=no;

if (1 > 2) define @msg=yes;
if (1 < 2) jmp 1;

list /tmp/tsteqn2.list;
brk;

nop 1;
define @msg=Jump Worked!;

list /tmp/tsteqn2.list;
```

### *streval*

This statement needs to be executed on a V9 or better Progress installation.

It represents STRing EVALuation. You can use ALL of the Progress 4GL's functions with this statement. If you need to, you can call into Progress' built in 4GL functions with this statement.

Build 20040414* supports the use of formulas in the statement.

So if you wish to use it, you will need to use place holders and REPLACE on them before executing the code.

```
// test the new streval statement;
// one needs V9 to pull this off!;

streval @msg=string(integer(substring("1234", 3, 2)), "$>>>.99");

// Check formula substitution;

define @Text=This is some text;
streval @msg2=num-entries("@Text", " ");

list /tmp/tststreval1.list;
```

Note that formula substitution is a simple search and replace of a formula name with the formula's setting. See how @Text is between quotes as it is it's `define` does not include quotes to match the 4GL requirements. If it did have quotes in the setting, you would be able to place it as a normal variable.

### *Executing an EQN program from the 4GL*

To begin executing an EQN program from the 4GL, use the RunPrg.p API. This program is the program interpreter and will start executing the program as given to it, manipulating the formula's as needed by the program.

Remember that EQN programs can run into infinite loops just as 4GL programs can! Since it is still Progress underneath, you should be able to CTRL-C out of any infinite loops.

Any formula’s you create with AddFormula.p should be available to the RunPrg.p API to execute on. You should be able to use DelFormula.p to remove formulas from the system. Use Calc.p, iGetFormula, or dGetFormula to re-evaluate or pull up the value of a formula.

Example:

```
DEF VAR iMyInt AS INTEGER.

FIND EQNPrg NO-LOCK
WHERE EQNPrg.Name = “InitRevenue”
NO-ERROR.

RUN RunPrg.p (”, EQNPrg.PrgText).

ASSIGN iMyInt = iGetFormula(“@Revenue”).
```

➡ Be sure to see Case Study Three for a real world application of this use!

## Progress 4GL API Guide

### *Calc.p*

```
Calc.p (INPUT CHAR Expression, OUTPUT CHAR Result)
```

Given an algebraic expression, return it's evaluation.

### *AddFormula.p*

```
AddFormula.p (INPUT CHAR Category, INPUT CHAR FormulaName, INPUT CHAR
FormulaSetting)
```

This will insert a formula into the EQN system.

If the formula already exists, then it will be over-written with the new setting.

Category is for your use.

### *DelFormula.p*

```
DelFormula.p (INPUT CHAR Category, INPUT CHAR FormulaName)
```

Removes a formula from the EQN system.

Category is for your use.

### *iGetFormula()*

Normally the results of a formula are given via a call to Calc.p. In the EQN language, a formula can be defined and evaluated via the eval statement. In the 4GL a call to Calc.p after RunPrg.p causes significant overhead to simply return an already computed out number. This program is available to call up a formula with less overhead and returns an integer.

> Note the formula MUST ALREADY BE EVALUATED, NOT SIMPLY DEFINED to use this API.

```
DEF VAR iMyInt AS INTEGER.

FIND EQNPrg NO-LOCK
WHERE EQNPrg.Name = “InitRevenue”
NO-ERROR.

RUN RunPrg.p (”, EQNPrg.PrgText).
ASSIGN iMyInt = iGetFormula(“@Revenue”).
```

## dGetFormula()

Normally the results of a formula are given via a call to Calc.p. In the EQN language, a formula can be defined and evaluated via the eval statement. In the 4GL a call to Calc.p causes significant overhead to simply return an already computed out number. This program is available to call up a formula with less overhead and returns a decimal.

> Note the formula MUST ALREADY BE EVALUATED, NOT SIMPLY DEFINED to use this API.

```
DEF VAR dMyInt AS DECIMAL.

FIND EQNPrg NO-LOCK
WHERE EQNPrg.Name = “InitRevenue”
NO-ERROR.

RUN RunPrg.p (”, EQNPrg.PrgText).

ASSIGN dMyInt = dGetFormula(“@Revenue”).
```

## cGetFormula()

Normally the results of a formula are given via a call to Calc.p. In the EQN language, a formula can be defined and evaluated via the eval statement. In the 4GL a call to Calc.p causes significant overhead to simply return an already computed out number. This program is available to call up a formula with less overhead and returns a character version of the string associated with the formula.

> Note: This function can be useful for passing string data back and forth between programs. One can do this PROVIDED you don't try to evaluate it. Cuz it will just blow up. This is a way to get data into a place that a 4GL program called by j4gl can use to access information. It is also a good way to just get a definition of a formula back.

```
DEF VAR cMyChar AS CHARACTER.

FIND EQNPrg NO-LOCK
WHERE EQNPrg.Name = “InitRevenue”
NO-ERROR.

RUN RunPrg.p (”, EQNPrg.PrgText).

ASSIGN cMyChar = cGetFormula(“@Revenue”).
```

## AddEQNPrg.p

Adds an EQN program into the system for use by the `jeqn` statement.

```
AddEQNPrg.p (INPUT CHAR Name, INPUT CHAR PrgText)
```

### *DelEQNPrg.p*

Deletes an EQN program from the system for use by the `jeqn` statement.

```
DelEQNPrg.p (INPUT CHAR Name)
```

### *RunPrg.p*

```
RunPrg.p (INPUT CHAR Category, INPUT CHAR ProgramText)
```

Executes the code given in ProgramText.

Category is for your use.

## Built in Functions

### *Min*

Provide the minimum between x and y

Min[ x, y ]

### *Max*

Provide the maximum between x and y

Max[ x , y ]

### *Power*

Raise x to the power of y

Power[ x, y ]

### *if*

Provide a computational if.  Format reads as:

if[  conditional, true, false ]

where conditional is some condition
where true is the result to return when condition is true
where false is the result to return when condition is false

The conditional is of the syntax `x [>,<,>=,<=,==,!=] y [and,or] ….`

Some examples (from BatTestCalc.p/BatTestAlgCalc.p):

```
Evaluating  if[ (1+1-1) != 2,1,0 ]
if[ (1+1-1) != 2,1,0 ] = 1
Evaluating  if[ 1 != (1+1),1,0 ]
if[ 1 != (1+1),1,0 ] = 1
Evaluating  if[ 1 != 2, (2 - 1) ,0 ]
if[ 1 != 2, (2 - 1) ,0 ] = 1
Evaluating  if[ 1 != 2, (2 + 1) ,0 ]
if[ 1 != 2, (2 + 1) ,0 ] = 3
Evaluating  if[ 1 != 2, 1,0 ]
if[ 1 != 2, 1,0 ] = 1
Evaluating  if[ 1 != 2 and 1 = 1, 1,0 ]
if[ 1 != 2 and 1 = 1, 1,0 ] = 1
Evaluating  if[ 1 = 2 or 1 = 1, 1,0 ]
if[ 1 = 2 or 1 = 1, 1,0 ] = 1
Evaluating  if[ 1 = 2 and 1 = 1, 1,0 ]
if[ 1 = 2 and 1 = 1, 1,0 ] = 0
Evaluating  if[ 1 > 2 or 2 = 1, 1,0 ]
if[ 1 > 2 or 2 = 1, 1,0 ] = 0
```

Note that if can return a number other than 1 and 0.

## *Empty*

Test functions with empty arguments can exist

Empty[ ]

## Error Detection

So you tried out an expression, perhaps (4 + 6) / 0 and got a ? back as a result. As you know, division by zero is not allowed in mathematics, and so an error occurred.

The following errors are available in the system thus far:

| Error Description | Error Code |
|---|---|
| No Error | 000:No error encountered. |
| Unknown error | 001:Unknown has occurred. |
| Formula used but not defined | 100:Formula not found. |
| Mathematical division by zero | 200:Division by zero error. |

These error codes will be stored in the formula @ErrMsg as a means to universally make the message available to 4GL and Windfern programs.

Note the error message is prefixed by a number, followed by a ":" and then a human friendly explanation of the error. This is so that if you wish, you can program to check/display the error code as a whole, only the error code, or only the description of the error.

### *4GL API*

Version 6 of EQN/Windfern introduces some new error detection routines.

#### *ResetError.p*

Sets error to the 000 error message.

#### *cGetError()*

Returns the full error code and message as seen in the EQNErrCode.i file.

#### *cGetErrorCode()*

Returns the numeric value of the error code as a character string.

#### *cGetErrorDesc()*

Returns the human friendly description of the error as a character string.

## Using Dates in your EQN Programs and Expressions

It is now possible to use dates in your expressions and programs.

A date token is any entry in the program preceded by a # sign. For example, if you wished to note the date 12-23-2003 in your code, you would enter #12-23-2003.

This is accomplished by the system converts dates into their integer value before working with them. The full four digit year must be given. You can use a – or / as with Progress 4GL representations of dates. The format of the date should match the -mdy entry in your start-up.

There is a special token called #today, which will be the date at the time of execution of the program. Look into the 4GL TODAY function for more information about this token.

### *Using Dates in EQN Programs:*

Here is how the token would be used in an if statement in a program:

```
if (#12-23-2003 < #12-24-2003) jmp T3;
```

This says, if 12-23-2003 is less than 12-24-2003, then jump to marker T3.

Here is another example, where a date is used in a formula:

```
define @T3=#12-23-2003 + 1;

if (@T3 < #12-25-2003) jmp T4;
```

This will set the T3 formula to the 23rd of December +1. Upon execution of the if, it will be converted to the integer representation of December 24th and compared against the integer representation of December 35th. Since it is less, it will jump execution to marker T4.

### *Using Dates in Algebraic Expressions:*

As you can tell, dates can be used in algebraic expressions for computation also. Here is a section of code from BatTestCalc.p:

```
/* Date in a function test */

ASSIGN T = "if[#12-23-2003 > #12-24-2003, 1, 0]".
```

```
MESSAGE "Evaluating " T.
RUN Calc.p (INPUT T, OUTPUT U).
MESSAGE  T  "=" U SKIP.

ASSIGN T = "if[#12-23-2003 < #12-24-2003, 1, 0]".
MESSAGE "Evaluating " T.
RUN Calc.p (INPUT T, OUTPUT U).
MESSAGE  T  "=" U SKIP.

/* Date in () test */

ASSIGN T = "(if[#12-23-2003 < #12-24-2003, 1, 0] + 1) * 2".
MESSAGE "Evaluating " T.
RUN Calc.p (INPUT T, OUTPUT U).
MESSAGE  T  "=" U SKIP.

/* Date in a formula test */

ASSIGN T = "(if[#12-23-2003 < #12-24-2003, 1, 0] + 1) * 2".
RUN AddFormula.p ("", "@Date", T).
ASSIGN T = "@Date".
MESSAGE "Evaluating " T.
RUN Calc.p (INPUT T, OUTPUT U).
MESSAGE  T  "=" U SKIP.

/* Check that #today works */

ASSIGN T = "#today".
MESSAGE "Evaluating " T.
RUN Calc.p (INPUT T, OUTPUT U).
MESSAGE  T  "=" U SKIP.

/* Should return 1 */

ASSIGN T = "if[#today > #12-23-2003, 1, 0]".
MESSAGE "Evaluating " T.
RUN Calc.p (INPUT T, OUTPUT U).
MESSAGE  T  "=" U SKIP.
```

Which yields:

```
Evaluating  #12-23-2003
#12-23-2003 = 2452998
Evaluating  if[#12-23-2003 > #12-24-2003, 1, 0]
if[#12-23-2003 > #12-24-2003, 1, 0] = 0
Evaluating  if[#12-23-2003 < #12-24-2003, 1, 0]
if[#12-23-2003 < #12-24-2003, 1, 0] = 1
Evaluating  (if[#12-23-2003 < #12-24-2003, 1, 0] + 1) * 2
(if[#12-23-2003 < #12-24-2003, 1, 0] + 1) * 2 = 4
Evaluating  @Date
@Date = 4
Evaluating  #today
#today = 2453045
Evaluating  if[#today > #12-23-2003, 1, 0]
```

```
if[#today > #12-23-2003, 1, 0] = 1
```

## Implementation Hints:

### *How to add your own functions*

You can add your own functions by modifying the file AlgFunc.i.

If these functions are database oriented, that is looking up a value or the like; you may want to read the sections below. It is better to pre-populate the program text of formula's with values and then call into EQN to peform some work than to have it call into the database it's self.

Why? For one thing, one never knows what is going to happen with transactions. Another is action segment sizes should be kept in mind. EQN is very complicated and made up of multiple functions. If adding a function, one might want to have it call out to another procedure file or a persistent procedure to aid in handling action code segments.

The file AlgFunc.i is made up of two parts. One part allows the user to define the actual formula. These are at the beginning of the program. The other part relates a program token to that formula. That is near the end of the file.

One should expect one character argument to come in and it should return one character string as it's output. Functions requiring multiple arguments will get a comma delimited list as entered in the expression between the [ and ] of the function.

Any arguments that are themselves formulas or functions will be reduced before calling into your function. So remember – you will only be seeing numbers sent to your function.

To create a new function, lets say Min[], one would first create code for the function such as this:

```
FUNCTION fMin RETURNS CHARACTER (INPUT cArgs AS CHARACTER):

  RETURN STRING(MIN(DECIMAL(ENTRY(1,cArgs)),DECIMAL(ENTRY(2,cArgs)))).

END.
```

Note how the function is named fMin because Min is a Progress keyword. Note how the cArgs varible is parsed and it's results turned into the correct type for use with the Progress 4GL min function. It returns a string which will be substituted back into the system.

Now we need to relate the function to some token used in the EQN system. The token need not be the same name as the function.

There is a function called FuncEval. Add your AlgFuncTemplate.i entry near the others. Order is not important. The first argument is what should be searched for in the EQN expression. The other is the function name to calculate it's results with.

```
  {AlgFuncTemplate.i Min fMin}
```

### Making the EQNPrg temp-table permanent

One may want to make the EQNPrg temp table permanent. The easiest way to do this is to add a definition found in the eqnprgtbl.i file in the database. Then comment out the definition in the eqnprgtbl.i file. The system should then automatically start using the database version instead of the temp-table version.

### Making the Formula temp-table permanent

One should not make the Formula table permanent. There may be multiple people attempting to calculate out the same formulas, but with their own values in the equations. It is very likely different people will get the results of other people. It all depends on who executes which formula when. Temp-tables are unique to each process that created them, so there is some separation.

If there are a lot of common formula's that need to be managed and are common to all users, I would suggest something like this:

```
/* InitFormulas.p */

{FormulaTbl.i}

/* Transfer permanent formula's into user's formula work space. */

FOR EACH PermanentFormula NO-LOCK:

  RUN AddFormula.p ("",
                    PermanentFormula.Name,
                    PermanentFormula.Setting
                   ).

END.  /* FOR EACH PermanentFormula */
```

This way the common formulas are transferred to the user's formula work space. They will not be able to manipulate the default definitions programmatically, but in their own formula workspace.

One could do an initialization at the beginning of an EQN program with

```
j4gl InitFormulas.p;
```

which would actually run the above 4GL code loading default formula's into their workspace.

### Passing Data between EQN apps and Progress 4GL apps

There is the question of passing data between 4GL apps and EQN programs.

To spell it out, one would use the AddFormula.p API in 4GL code to pass a formula or value into an EQN app.

One would use iGetFormula, dGetFormula, cGetFormula, or Calc.p to retrieve a value from an EQN program.

Long enough you have {FormulaTbl.i} included in your program, you should have access to the formulas from both the 4GL and EQN programs.

Remember to NEW FormulaTbl.i at the top most programs, which is {FormulaTbl.i NEW} so the new shared temp table is created.

Normally one puts numbers or formula’s into a Formula via AddFormula.p.  One can also put in parameters for a 4GL app that is NOT numbers or a formula. One must be careful not to evaluate this “formula” because it will blow up the system.  But it is a good way to share ROWIDs, Part Numbers containing alpha/numeric characters, etc between 4GL programs reached via the EQN program.

## Case Studies

### *Case Study One – Using Algebraic Expressions in the Real World*

The Denkh HTML Reporter allows users to create reports against their databases much like Crystal Reports (only it is free!)

Since all calculations are rarely stored in the database, the report program needed a manner in which to calculate un-pre-determined algebraic expressions. For example, Denkh HTML Reporter needed a manner in order to calculate the total price for an order line. It handles calculations with the @calc() macro which sends an algebraic expression to EQN which resides under Denkh HTML Reporter.

The line in the HTML report reads as:

```
Total: @calc(@fld(OrderLine.Qty) * @fld(OrderLine.Price))
```

The @fld() macro's are replaced with their record values by HTML Reporter:

```
Total: @calc( 5 * 9.95 )
```

HTML Reporter has code which takes @calc and sends it to Calculate.p in the EQN package:

```
RUN Calculate.p ("5 * 9.95", OUTPUT cResult).
```

Finally the result is substituted for the @calc() macro by Denkh HTML Reporter in it's output:

```
Total: 49.95
```

### *Case Study Two – Using Formula's in the Real World*

There are times when a user might need to define a calculation that happens to be used in other calculations. Instead of having to repeatedly change a calculation as it is used in other calculations, EQN has the ability to look up formulas.

| | |
|---|---|
| @HrsPerWeek | 40 |
| @VacationWeeks | 2 |
| @WeeksPerYear | 52 |
| @HrsPerYear | @HrsPerWeek * (@WeeksPerYear - @VacationWeeks) |
| @PayPerYear | @HrsPerYear * 90.00 |

Each of these formulas are stored in parameter records denoting the configuration of the system. A 4GL program that would use these for its logic might read as follows:

```
/* Load user defined formulas into EQN */

FOR EACH Parms NO-LOCK
WHERE Parms.Application = “HR”
AND Parms.Group = “Formulas”:

  RUN AddFormula.p (“”, Parms.ParmName, Parms.ParmValue).

END.

/* Evaluate the formula we are interested in with EQN */

RUN Calc.p (“”, “@PayPerYear”, OUTPUT cPayPerYear).

/* Procede with the results in the Progress program */

ASSIGN dPayPerYear = DECIMAL(cPayPerYear).
```

## *Case Study Three – Using Programs in the Real World*

There are circumstances where a formula needs to change based on some other conditions. This is where the programming aspect of EQN comes in handy.

Take for example the following business rules regarding inventory:

1. If the items class is 4 and under minimum then order = item’s minimum – item’s current count + item’s needed.

2. Items of class 3 only need to keep a minimum stocking level.

3. All other items should be ordered as needed.

4. These rules may change regarding other inventory classes and calculations.

One can translate these business rule calculations with the following EQN program:

```
// InventoryOrderEQN;
// Determine how much of an item to order;
// Use shared variable or AddFormula.p to determine our part number;
// Then load up some working formulas and database values via the 4GL;
// @CurrentCount = item’s current count;
// @Class = item’s class number;
// @CurrentNeed = item’s current need;
// @MinimumStockingLevel = item’s minimum stocking level;

j4gl EQNLoadInventory.p;

// If we are ordering negative, then just order 0;

// Rule 1 – dealing with class 4 items;

nop Rule1;
jmp Rule2 @Class != 4;
eval @Order=@MinimumStockingLevel - @CurrentCount + @CurrentNeed;
jmp NoOrder @Order < 0;
brk;

// Rule 2 – dealing with class 3 items;

nop Rule2;
jmp Rule3 @Class != 3;
eval @Order=@MinimumStockingLevel - @CurrentCount;
jmp NoOrder @Order < 0;
brk;

// JIT order this part;
nop Rule3;
eval @Order=@CurrentNeed - @CurrentCount;
jmp NoOrder @Order < 0;
brk;

nop NoOrder;
define @Order=0;
brk;
```

One might think, well – I can do that with 4GL just fine! But what if they add another class of part or want to change an existing equation? All they need to do is pull up the parameter record, change the rules – wha la – the system acts differently with no 4GL coding. Business rule power is in the user’s hands!

So a section of a progress program for ordering inventory might read as:

```
/* Run through user defined rules for inventory ordering */

FIND EQNPrg NO-LOCK
WHERE EQNPrg.Name = “InventoryOrderEQN”
NO-ERROR.

/* Set shared variable for identifying part number */

ASSIGN scPartNum = Part.PartNo.

RUN RunPrg.p (”, EQNPrg.PrgTest).

ASSIGN iOrder = iGetFormula(“@Order”).
```

Remember the EQN program calls out to a progress program to aid it with some basic information about the part. That program was called EQNLoadInventory.p. Here would be the relevant sections:

```
/* EQNLoadInventory.p */
/* Called by an EQN program to populate some formulas it needs */

FIND Part NO-LOCK
WHERE Part.PartNo = scPartNum
NO-ERROR.

RUN AddFormula.p (“”, “@Class”, STRING(Part.Class)).
RUN AddFormula.p (“”, “@MinimumStockingLevel”, STRING
(Part.MinStckLevel)).

FIND CurInventory OF Part NO-LOCK.

RUN AddFormula.p (“”, “@CurrentCount”, STRING(CurInventory.CurCnt)).

FIND CurNeed OF Part NO-LOCK.

RUN AddFormula.p (“”, “@CurrentNeed”, STRING(CurNeed.Cnt)).
```

So the first section of progress coding would set a shared variable and run the EQN API RunPrg.p to run the EQN program.
The EQN Run-Time Engine would see it needs to execute a progress program (jprg EQNLoadInventory.p) and execute the second bit of code.
This code would look up the part, current inventory, and current need from the database and place them into formula's that the EQN program can work with.
The 4GL program ends and the EQN program continues on evaluating the order amount.
The original 4GL program then uses iGetFormula() to find out what the user defined order amount for that part should be.

Hopefully you can see the power of this programming language in terms of setting up rules for calculations, making decisions, and risk analysis.

## About Amduus™ Information Works, Inc.

Amduus™ Information Works, Inc. is the brainchild of Scott Auge. It was incorporated in 2001.

Someday it is going to be a big ass kicking company owning a marketplace and I will be hated and despised like Bill Gates. I will have a wad of cash in the bank and be worried about things like which new BMW or Jaguar I should buy.

I plan on ASPing some sort of software, once I find a way to sell the darn things. Anyone want to be a Manufacturers Rep for Amduus? Good stuff coming down the lane. Just need people in various parts of the country to knock on doors, configure for customers, and get some money! Do you want some money? You should write me: Scott Auge sauge@amduus.com.

Right now, it pretty much lets me nab contracts and people don't have to worry about the IRS calling me an employee. They are billed by Amduus and write the checks to Amduus. Amduus has it's own bank accounts. Scott gets lots of computer toys by expensing Amduus. Unfortunately Scott is not well paid by Amduus yet – but remembers the world domination plans!

Anyhow, Amduus Information Works, Inc releases all this software. So if your gonna be an idiot and sue someone over free and open software, Amduus is the entity to do so.