```
// Corecalc and Funcalc, spreadsheet implementations

// ----------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and Thomas S. Iversen

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ----------------------------------------------------------------------

// Coco/R grammar for spreadsheet formulas

// To build:
//   coco -namespace Corecalc Spreadsheet.ATG
// or
//   mono Coco.exe -namespace Corecalc Spreadsheet.ATG

//  - RaRefs in the R1C1 format
//  - The string concatenation operator &
//  - Numbers in scientific notation
//  - Sheetreferences in the style: [Alpha{Alpha}!]Raref
//  - ^ (power) operator (April 2006).
//  - Equality operator now "=" as in Excel, not "==" (Nov 2008)
//  - Functions may be called "LOG10" and similar (Nov 2008)
//  - Unary minus on factors now works (Nov 2008)
//  - Now accepts hard line breaks in quote cells (Aug 2011)
//  - Now accepts datetime strings "2009-05-20T00:00:00.000"
//  - Now accepts underscores and alphanumerics in function names
//  - Now accepts dots in function names

using System.Collections.Generic;

COMPILER CellContents

  private int col, row;
  private Workbook workbook;
  private Cell cell;
  private static System.Globalization.NumberFormatInfo numberFormat = null;

  static Parser() {
    // Set US/UK decimal point, regardless of culture
    System.Globalization.CultureInfo ci =
      System.Globalization.CultureInfo.InstalledUICulture;
    numberFormat = (System.Globalization.NumberFormatInfo)ci.NumberFormat.Clone();
    numberFormat.NumberDecimalSeparator = ".";
  }

  public Cell ParseCell(Workbook workbook, int col, int row) {
    this.workbook = workbook;
    this.col = col; this.row = row;
    Parse();
    return errors.count == 0 ? cell : null;
  }

/*----------------------------------------------------------------------*/
```



```
CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  uletter = letter + '_' + '.'.
  atoi   = "ABCDEFGHIabcdefghi".
  digit = "0123456789".
  Alpha = letter + digit.
  cr  = '\r'.
  lf  = '\n'.
  tab = '\t'.
  exclamation = '!'.
  dollar  = '$'.
  newLine = cr + lf.
  strchar    = ANY - '"' - '\\' - newLine.
  char       = ANY - '\\'.

TOKENS
  name       = uletter { (uletter | digit) } CONTEXT("(").
  number     =
               digit { digit }
               [                                /* optional fraction         */
                [ "." digit { digit }]          /* optional fractional digits */
                [ ( "E" | "e" )                 /* optional exponent         */
                  [ "+" | "-" ]                 /* optional exponent sign    */
                  digit { digit }
                ]
               ].
  datetime   = digit digit digit digit "-" digit digit "-" digit digit
               [ "T" digit digit ":" digit digit ":" digit digit [ "." { digit } ] ].
  sheetref   = Alpha { Alpha } exclamation.
  raref      = [ dollar ] letter [ dollar ] digit { digit }
             | [ dollar ] atoi letter [ dollar ] digit { digit }.
  xmlssraref11= "RC".
  xmlssraref12= "RC" digit { digit }.
  xmlssraref13= "RC[" ["+"|"-"] digit { digit } "]".
  xmlssraref21= "R" digit { digit } "C".
  xmlssraref22= "R" digit { digit } "C" digit {digit}.
  xmlssraref23= "R" digit { digit } "C[" ["+"|"-"] digit { digit } "]".
  xmlssraref31= "R[" ["+"|"-"] digit { digit } "]C".
  xmlssraref32= "R[" ["+"|"-"] digit { digit } "]C" digit { digit }.
  xmlssraref33= "R[" ["+"|"-"] digit { digit } "]C[" ["+"|"-"] digit { digit } "]".
  string     = '"' { strchar } '"'.
  quotecell  = "\'" { char }.

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO cr lf

IGNORE cr + lf + tab


PRODUCTIONS
/*----------------------------------------------------------------------*/
AddOp<out String op>
=                           (. op = "+"; .)
  ( '+'
  | '-'                     (. op = "-"; .)
  | '&'                     (. op = "&"; .)
  ).

LogicalOp<out String op>
=                           (. op = "="; .)
  ( "="
  | "<>"                    (. op = "<>"; .)
  | "<"                     (. op = "<";  .)
  | "<="                    (. op = "<="; .)
  | ">"                     (. op = ">";  .)
  | ">="                    (. op = ">="; .)
  ).

/*----------------------------------------------------------------------*/
Expr<out Expr e>            (. Expr e2; String op; e = null; .)
= LogicalTerm<out e>
```

```
  { LogicalOp<out op>
    LogicalTerm<out e2>  (. e = FunCall.Make(op, new Expr[] { e, e2 }); .)
  }
  .

LogicalTerm<out Expr e>  (. Expr e2; String op; e = null; .)
= Term<out e>
  { AddOp<out op>
    Term<out e2>         (. e = FunCall.Make(op, new Expr[] { e, e2 }); .)
  }
  .

/*--------------------------------------------------------------------------*/
Factor<out Expr e>       (. RARef r1, r2; Sheet s1 = null; double d;
                            bool sheetError = false; e = null; .)
= Application<out e>
  |
  (
   | sheetref            (. s1 = workbook[t.val.Substring(0,t.val.Length-1)];
                            if (s1 == null) sheetError = true;               .)
  )
   Raref<out r1> (       (. if (sheetError)
                              e = new Error(ErrorValue.refError);
                            else
                              e = new CellRef(s1, r1);        .)
    | ':' Raref<out r2>  (. if (sheetError)
                              e = new Error(ErrorValue.refError);
                            else
                              e = new CellArea(s1, r1, r2);   .)
   )
  | Number<out d>        (. e = new NumberConst(d);         .)
  | '-' Factor<out e>    (. if (e is NumberConst)
                              e = new NumberConst(-((NumberConst)e).value.value);
                            else
                              e = FunCall.Make("NEG", new Expr[] { e });
                         .)
  | string               (. e = new TextConst(t.val.Substring(1, t.val.Length-2)); .)
  | '(' Expr<out e> ')'
  .

/*--------------------------------------------------------------------------*/
PowFactor<out Expr e>    (. Expr e2; .)
= Factor<out e>
  { '^'
    Factor<out e2>       (. e = FunCall.Make("^", new Expr[] { e, e2 } ); .)
  }
  .
/*--------------------------------------------------------------------------*/

Raref<out RARef raref>   (. raref = null;.)
= raref                  (. raref = new RARef(t.val, col, row); .)
 | xmlssraref11           (. raref = new RARef(t.val); .)
 | xmlssraref12           (. raref = new RARef(t.val); .)
 | xmlssraref13           (. raref = new RARef(t.val); .)
 | xmlssraref21           (. raref = new RARef(t.val); .)
 | xmlssraref22           (. raref = new RARef(t.val); .)
 | xmlssraref23           (. raref = new RARef(t.val); .)
 | xmlssraref31           (. raref = new RARef(t.val); .)
 | xmlssraref32           (. raref = new RARef(t.val); .)
 | xmlssraref33           (. raref = new RARef(t.val); .)
.

/*--------------------------------------------------------------------------*/
Number<out double d>     (. d = 0.0;                      .)
= number                 (. d = double.Parse(t.val, numberFormat); .)
  .
/*--------------------------------------------------------------------------*/
Application<out Expr e>  (. String s; Expr[] es; e = null; .)
= Name<out s> '('
  ( ')'                  (. e = FunCall.Make(s.ToUpper(), new Expr[0]); .)
    | Exprs1<out es> ')' (. e = FunCall.Make(s.ToUpper(), es); .)
```



```
  )
.
/*--------------------------------------------------------------------------*/
Exprs1<out Expr[] es>    (. Expr e1, e2;
                            List<Expr> elist = new List<Expr>();
                         .)
= ( Expr<out e1>         (. elist.Add(e1); .)
    { (';' | ',') Expr<out e2>   (. elist.Add(e2); .)
    }
  )                      (. es = elist.ToArray();  .)
.
/*--------------------------------------------------------------------------*/
Name<out String s>
= name                   (. s = t.val; .)
.
/*--------------------------------------------------------------------------*/
MulOp<out String op>
=                        (. op = "*"; .)
  ( '*'
  | '/'                  (. op = "/"; .)
  ).
/*--------------------------------------------------------------------------*/
Term<out Expr e>         (. Expr e2; String op; .)
= PowFactor<out e>
  { MulOp<out op>
    PowFactor<out e2>    (. e = FunCall.Make(op, new Expr[] { e, e2 } ); .)
  }.
/*--------------------------------------------------------------------------*/
CellContents             (. Expr e; double d;     .)
= ( '=' Expr<out e>      (. this.cell = Formula.Make(workbook, e); .)
  | quotecell            (. this.cell = new QuoteCell(t.val.Substring(1)); .)
  | string               (. this.cell = new TextCell(t.val.Substring(1, t.val.Length-2)); .
)
  | datetime             (. long ticks = DateTime.Parse(t.val).Ticks;
                            double time = NumberValue.DoubleFromDateTimeTicks(ticks);
                            this.cell = new NumberCell(time);
                         .)
  | Number<out d>        (. this.cell = new NumberCell(d);  .)
  | '-' Number<out d>    (. this.cell = new NumberCell(-d);  .)
  ).
END CellContents.
```

```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2012 Thomas S. Iversen, Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;

namespace Corecalc.Benchmarks {
  public class Benchmarks {
    private TextWriter tw;
    public bool useLog;

    public Benchmarks(bool useLog) {
      this.useLog = useLog;
    }

    private TextWriter Tw {
      get {
        if (tw == null)
          tw = new StreamWriter("benchmark_results.txt");
        return tw;
      }
    }

    public void BenchmarkRecalculation(WorkbookForm wf, int runs) {
      BenchmarkWorkbook(wf, runs, "Workbook standard recalculation",
        wf.Workbook.Recalculate);
    }

    public void BenchmarkRecalculationFull(WorkbookForm wf, int runs) {
      BenchmarkWorkbook(wf, runs, "Workbook full recalculation",
        wf.Workbook.RecalculateFull);
    }

    public void BenchmarkRecalculationFullRebuild(WorkbookForm wf, int runs) {
      BenchmarkWorkbook(wf, runs, "Workbook full recalculation rebuild",
        wf.Workbook.RecalculateFullRebuild);
    }

    private void BenchmarkWorkbook(WorkbookForm wf, int runs, string benchmarkName, Func<lo
ng> benchmark) {
      Log("=== Benchmark workbook called: ");

      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Reset();
      stopwatch.Start();
      for (int i = 0; i < runs; i++)
        benchmark();
```



```
      stopwatch.Stop();
      double average = stopwatch.ElapsedMilliseconds / (double)runs;
      Log(String.Format("[{0}] Average of the {1} runs: {2:N2} ms",
                        benchmarkName, runs, average));
      wf.SetStatusLine((long)(average + 0.5));
    }

    //log both a flat file and console
    private void Log(string s) {
      if (useLog) {
        Console.WriteLine(s);
        Tw.WriteLine(s);
        Tw.Flush();
      }
    }
  }
}
```

```
// Corecalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.Reflection;

// Classes for representing absolute cell addresses,
// relative/absolute cell references, support ranges, and
// adjusted expressions and refererences

namespace Corecalc {
  /// <summary>
  /// A CellAddr is an absolute, zero-based (col, row) location in some sheet.
  /// Serializable because it is used in ClipboardCell, which must be serializable.
  /// </summary>
 [Serializable]
  public struct CellAddr : IEquatable<CellAddr> {
    public readonly int col, row;
    public static readonly CellAddr A1 = new CellAddr(0, 0);

    public CellAddr(int col, int row) {
      this.col = col;
      this.row = row;
    }

    public CellAddr(RARef cr, int col, int row) {
      this.col = cr.colAbs ? cr.colRef : cr.colRef + col;
      this.row = cr.rowAbs ? cr.rowRef : cr.rowRef + row;
    }

    // Turn an A1-format string into an absolute cell address
    public CellAddr(String a1Ref) : this(new RARef(a1Ref, 0, 0), 0, 0) { }

    // Return ulCa as upper left and lrCa as lower right of (ca1, ca2)
    public static void NormalizeArea(CellAddr ca1, CellAddr ca2,
                                     out CellAddr ulCa, out CellAddr lrCa) {
      int minCol = ca1.col, minRow = ca1.row,
          maxCol = ca2.col, maxRow = ca2.row;
      if (ca1.col > ca2.col) {
        minCol = ca2.col; maxCol = ca1.col;
      }
      if (ca1.row > ca2.row) {
        minRow = ca2.row; maxRow = ca1.row;
      }
      ulCa = new CellAddr(minCol, minRow);
      lrCa = new CellAddr(maxCol, maxRow);
```



```
    }

    public CellAddr(System.Drawing.Point p) {
      this.col = p.X;
      this.row = p.Y;
    }

    public CellAddr Offset(CellAddr offset) {
      return new CellAddr(this.col + offset.col, this.row + offset.row);
    }

    public bool Equals(CellAddr that) {
      return col == that.col && row == that.row;
    }

    public override int GetHashCode() {
      return 29 * col + row;
    }

    public override bool Equals(Object o) {
      return o is CellAddr && Equals((CellAddr)o);
    }

    public static bool operator==(CellAddr ca1, CellAddr ca2) {
      return ca1.col == ca2.col && ca1.row == ca2.col;
    }

    public static bool operator!=(CellAddr ca1, CellAddr ca2) {
      return ca1.col != ca2.col || ca1.row != ca2.col;
    }

    public override String ToString() { // A1 format only
      return ColumnName(col) + (row + 1);
    }

    public static String ColumnName(int col) {
      String name = "";
      while (col >= 26) {
        name = (char)('A' + col % 26) + name;
        col = col / 26 - 1;
      }
      return (char)('A' + col) + name;
    }
  }

  /// <summary>
  /// A FullCellAddr is an absolute cell address, including a non-null
  /// reference to a Sheet.
  /// </summary>
  public struct FullCellAddr : IEquatable<FullCellAddr> {
    public readonly Sheet sheet;    // non-null
    public readonly CellAddr ca;
    public static readonly Type type = typeof(FullCellAddr);
    public static readonly MethodInfo evalMethod = type.GetMethod("Eval");

    public FullCellAddr(Sheet sheet, CellAddr ca) {
      this.ca = ca;
      this.sheet = sheet;
    }

    public FullCellAddr(Sheet sheet, int col, int row)
      : this(sheet, new CellAddr(col, row)) { }

    public FullCellAddr(Sheet sheet, string A1Format)
      : this(sheet, new CellAddr(A1Format)) { }

    public bool Equals(FullCellAddr other) {
      return ca.Equals(other.ca) && sheet == other.sheet;
    }

    public override int GetHashCode() {
```

```
    return ca.GetHashCode() * 29 + sheet.GetHashCode();
  }

  public override bool Equals(Object o) {
    return o is FullCellAddr && Equals((FullCellAddr)o);
  }

  public static bool operator==(FullCellAddr fca1, FullCellAddr fca2) {
    return fca1.ca == fca2.ca && fca1.sheet == fca2.sheet;
  }

  public static bool operator!=(FullCellAddr fca1, FullCellAddr fca2) {
    return fca1.ca != fca2.ca || fca1.sheet != fca2.sheet;
  }

  public override string ToString() {
    return sheet.Name + "!" + ca;
  }

  public Value Eval() {
    Cell cell = sheet[ca];
    if (cell != null)
      return cell.Eval(sheet, ca.col, ca.row);
    else
      return null;
  }

  public bool TryGetCell(out Cell currentCell) {
    currentCell = sheet[ca];
    return currentCell != null;
  }
}

/// <summary>
/// A SupportSet represents the set of cells supported by a given cell;
/// ie. those cells that refer to it.
/// </summary>
public class SupportSet  {
  private readonly List<SupportRange> ranges = new List<SupportRange>();

  // Add suppSheet[suppCols, suppRows] except sheet[col,row] to support set
  public void AddSupport(Sheet sheet, int col, int row,
                         Sheet suppSheet, Interval suppCols, Interval suppRows)
  {
    SupportRange range = SupportRange.Make(suppSheet, suppCols, suppRows);
    // Console.WriteLine("{0} supports {1}", new FullCellAddr(sheet, col, row), range);
    // If RemoveCell removed something (giving true), it also added remaining ranges
    if (!range.RemoveCell(this, sheet, col, row))
      ranges.Add(range);
  }

  public void RemoveCell(Sheet sheet, int col, int row) {
    int i = 0, count = ranges.Count; // Process only original supportSet items
    while (i < count) {
      if (ranges[i].RemoveCell(this, sheet, col, row)) {
        ranges.RemoveAt(i);
        count--;
      } else
        i++;
    }
  }

  public void Add(SupportRange range) {
    ranges.Add(range);
  }

  public void ForEachSupported(Action<Sheet,int,int> act) {
    foreach (SupportRange range in ranges)
      range.ForEachSupported(act);
  }
}
```



```
/// <summary>
/// A SupportRange is a single supported cell or a supported cell area on a sheet.
/// </summary>
public abstract class SupportRange {
  public static SupportRange Make(Sheet sheet, Interval colInt, Interval rowInt) {
    if (colInt.min == colInt.max && rowInt.min == rowInt.max)
      return new SupportCell(sheet, colInt.min, rowInt.min);
    else
      return new SupportArea(sheet, colInt, rowInt);
  }

  // Remove cell sheet[col,row] from given support set, possibly adding smaller
  // support ranges at end of supportSet; if so return true
  public abstract bool RemoveCell(SupportSet set, Sheet sheet, int col, int row);

  public abstract void ForEachSupported(Action<Sheet,int,int> act);

  public abstract bool Contains(Sheet sheet, int col, int row);

  public abstract int Count { get; }
}

/// <summary>
/// A SupportCell is a single supported cell.
/// </summary>
public class SupportCell : SupportRange {
  public readonly Sheet sheet;
  public readonly int col, row;

  public SupportCell(Sheet sheet, int col, int row) {
    this.sheet = sheet;
    this.col = col;
    this.row = row;
  }

  public override bool RemoveCell(SupportSet set, Sheet sheet, int col, int row) {
    return Contains(sheet, col, row);
  }

  public override void ForEachSupported(Action<Sheet,int,int> act) {
    act(sheet, col, row);
  }

  public override bool Contains(Sheet sheet, int col, int row) {
    return this.sheet == sheet && this.col == col && this.row == row;
  }

  public override int Count {
    get { return 1; }
  }

  public override string ToString() {
    return new FullCellAddr(sheet, col, row).ToString();
  }
}

/// <summary>
/// A SupportArea is a supported  absolute cell area sheet[colInt, rowInt].
/// </summary>
public class SupportArea : SupportRange {
  private static readonly List<SupportArea> alreadyVisited
    = new List<SupportArea>();
  private static bool idempotentForeach;

  public readonly Interval colInt, rowInt;
  public readonly Sheet sheet;

  public SupportArea(Sheet sheet, Interval colInt, Interval rowInt) {
    this.sheet = sheet;
    this.colInt = colInt;
```

```
      this.rowInt = rowInt;
    }

    public override bool RemoveCell(SupportSet set, Sheet sheet, int col, int row) {
      if (Contains(sheet, col, row)) {
        // To exclude cell at sheet[col, row], split into up to 4 support ranges
        if (rowInt.min < row) // North, column above [col,row]
          set.Add(Make(sheet, new Interval(col, col), new Interval(rowInt.min, row-1)));
        if (row < rowInt.max) // South, column below [col,row]
          set.Add(Make(sheet, new Interval(col, col), new Interval(row+1, rowInt.max)));
        if (colInt.min < col) // West, block to the left of [col,row]
          set.Add(Make(sheet, new Interval(colInt.min, col-1), rowInt));
        if (col < colInt.max) // East, block to the right of [col,row]
          set.Add(Make(sheet, new Interval(col+1, colInt.max), rowInt));
        return true;
      } else
        return false;
    }

    public static bool IdempotentForeach {
      get { return idempotentForeach; }
      set {
        idempotentForeach = value;
        alreadyVisited.Clear();
      }
    }

    public override void ForEachSupported(Action<Sheet,int,int> act) {
      if (IdempotentForeach && this.Count > alreadyVisited.Count + 1) {
        for (int i = 0; i < alreadyVisited.Count; i++) {
          SupportArea old = alreadyVisited[i];
          if (this.Overlaps(old)) {
            SupportArea overlap = this.Overlap(old);
            if (overlap.Count == this.Count) { // contained in old
              return;
            } else if (overlap.Count == old.Count) { // contains old
              alreadyVisited[i] = this;
              ForEachExcept(overlap, act);
              return;
            } else if (this.colInt.Equals(old.colInt) && this.rowInt.Overlaps(old.rowInt))
{
              alreadyVisited[i] = new SupportArea(sheet, this.colInt, this.rowInt.Join(old.
rowInt));
              ForEachExcept(overlap, act);
              return;
            } else if (this.rowInt.Equals(old.rowInt) && this.colInt.Overlaps(old.colInt))
{
              alreadyVisited[i] = new SupportArea(sheet, this.colInt.Join(old.colInt), this
.rowInt);
              ForEachExcept(overlap, act);
              return;
            } else { // overlaps, but neither containment nor rectangular union
              alreadyVisited.Add(this);
              ForEachExcept(overlap, act);
              return;
            }
          }
        }
        // Large enough but no existing support set overlaps this one
        alreadyVisited.Add(this);
      }
      ForEachInArea(sheet, colInt, rowInt, act);
    }

    private void ForEachExcept(SupportArea overlap, Action<Sheet, int, int> act) {
      if (rowInt.min < overlap.rowInt.min) // North non-empty, columns above overlap
        ForEachInArea(sheet, overlap.colInt, new Interval(rowInt.min, overlap.rowInt.min -
1), act);
      if (overlap.rowInt.max < rowInt.max) // South non-empty, columns below overlap
        ForEachInArea(sheet, overlap.colInt, new Interval(overlap.rowInt.max + 1, rowInt.ma
x), act);
```



```
      if (colInt.min < overlap.colInt.min) // West non-empty, rows left of overlap
        ForEachInArea(sheet, new Interval(colInt.min, overlap.colInt.min - 1), rowInt, act)
;
      if (overlap.colInt.max < colInt.max) // East non-empty, rows right of overlap
        ForEachInArea(sheet, new Interval(overlap.colInt.max + 1, colInt.max), rowInt, act)
;
    }

    private static void ForEachInArea(Sheet sheet, Interval colInt, Interval rowInt,
                                      Action<Sheet, int, int> act)
    {
      for (int c = colInt.min; c <= colInt.max; c++)
        for (int r = rowInt.min; r <= rowInt.max; r++)
          act(sheet, c, r);
    }

    public override bool Contains(Sheet sheet, int col, int row) {
      return this.sheet == sheet && colInt.Contains(col) && rowInt.Contains(row);
    }

    public override int Count {
      get { return colInt.Length * rowInt.Length; }
    }

    public bool Overlaps(SupportArea that) {
      return this.sheet == that.sheet
          && this.colInt.Overlaps(that.colInt) && this.rowInt.Overlaps(that.rowInt);
    }

    public SupportArea Overlap(SupportArea that) {
      Debug.Assert(this.Overlaps(that)); // In particular, on same sheet
      return new SupportArea(sheet, this.colInt.Meet(that.colInt),
                                    this.rowInt.Meet(that.rowInt));
    }

    public override string ToString() {
      CellAddr ulCa = new CellAddr(colInt.min, rowInt.min),
               lrCa = new CellAddr(colInt.max, rowInt.max);
      return String.Format("{0}!{1}:{2}", sheet.Name, ulCa, lrCa);
    }
  }

  /// <summary>
  /// An Interval is a non-empty integer interval [min...max].
  /// </summary>
  public struct Interval : IEquatable<Interval> {
    public readonly int min, max;  // Assume min<=max

    public Interval(int min, int max) {
      Debug.Assert(min <= max);
      this.min = min;
      this.max = max;
    }

    public void ForEach(Action<int> act) {
      for (int i = min; i <= max; i++)
        act(i);
    }

    public bool Contains(int i) {
      return min <= i && i <= max;
    }

    public int Length {
      get { return max - min + 1; }
    }

    public bool Overlaps(Interval that) {
      return this.min <= that.min && that.min <= this.max
          || that.min <= this.min && this.min <= that.max;
    }
```

```
  // When the intervals overlap, this is their union:
  public Interval Join(Interval that) {
    return new Interval(Math.Min(this.min, that.min), Math.Max(this.max, that.max));
  }

  // When the intervals overlap, this is their intersection:
  public Interval Meet(Interval that) {
    return new Interval(Math.Max(this.min, that.min), Math.Min(this.max, that.max));
  }

  public bool Equals(Interval that) {
    return this.min == that.min && this.max == that.max;
  }
}

/// <summary>
/// A RARef is a relative or absolute cell reference (sheet unspecified).
/// </summary>
public sealed class RARef : IEquatable<RARef> {
  public readonly bool colAbs, rowAbs;  // True=absolute, False=relative
  public readonly int colRef, rowRef;

  public RARef(bool colAbs, int colRef, bool rowAbs, int rowRef) {
    this.colAbs = colAbs; this.colRef = colRef;
    this.rowAbs = rowAbs; this.rowRef = rowRef;
  }

  // Parse "$A$3" to (true,0,true,2) and so on; relative references
  // must be adjusted to the (col,row) in which the reference occurs.

  public RARef(String a1Ref, int col, int row) {
    /* RC denotes "this" cell in MS XMLSS format; needed for area support */
    if (a1Ref.Equals("RC", StringComparison.CurrentCultureIgnoreCase)) {
      colAbs = false; rowAbs = false; colRef = 0; rowRef = 0;
    } else {
      int i = 0;
      if (i < a1Ref.Length && a1Ref[i] == '$') {
        colAbs = true;
        i++;
      }
      int val = -1;
      while (i < a1Ref.Length && IsAToZ(a1Ref[i])) {
        val = (val + 1) * 26 + AToZValue(a1Ref[i]);
        i++;
      }
      colRef = colAbs ? val : val - col;
      if (i < a1Ref.Length && a1Ref[i] == '$') {
        rowAbs = true;
        i++;
      }
      val = ParseInt(a1Ref, ref i);
      rowRef = (rowAbs ? val : val - row) - 1;
    }
  }

  // Parse XMLSS R1C1 notation from well-formed string (checked by parser)

  public RARef(String r1c1) {
    int i = 0;
    rowAbs = true;
    colAbs = true;
    if (i < r1c1.Length && r1c1[i] == 'R')
      i++;
    if (i < r1c1.Length && r1c1[i] == '[') {
      rowAbs = false;
      i++;
    }
    int val = ParseInt(r1c1, ref i);
    if (rowAbs && val == 0)
      rowAbs = false;
    this.rowRef = rowAbs ? val - 1 : val;
    if (i < r1c1.Length && r1c1[i] == ']')
      i++;
    if (i < r1c1.Length && r1c1[i] == 'C')
      i++;
    if (i < r1c1.Length && r1c1[i] == '[') {
      colAbs = false;
      i++;
    }
    val = ParseInt(r1c1, ref i);
    if (i < r1c1.Length && r1c1[i] == ']')
      i++;
    if (colAbs && val == 0)
      colAbs = false;
    this.colRef = colAbs ? val - 1 : val;
  }

  // Parse possibly signed decimal integer
  private static int ParseInt(String s, ref int i) {
    int val = 0;
    bool negative = false;
    if (i < s.Length && (s[i] == '-' || s[i] == '+')) {
      negative = s[i] == '-';
      i++;
    }
    val = 0;
    while (i < s.Length && Char.IsDigit(s[i])) {
      val = val * 10 + (s[i] - '0');
      i++;
    }
    return negative ? -val : val;
  }

  private static bool IsAToZ(char c) {
    return 'a' <= c && c <= 'z' || 'A' <= c && c <= 'Z';
  }

  private static int AToZValue(char c) {
    return (c - 'A') % 32;
  }

  // Absolute address of ref
  public CellAddr Addr(int col, int row) {
    return new CellAddr(this, col, row);
  }

  // Insert N new rowcols before rowcol R>=0, when we're at rowcol r
  public Adjusted<RARef> InsertRowCols(int R, int N, int r, bool insertRow) {
    int newRef;
    int upper;
    if (insertRow) {   // Insert N rows before row R; we're at row r
      InsertRowCols(R, N, r, rowAbs, rowRef, out newRef, out upper);
      RARef rarefNew = new RARef(colAbs, colRef, rowAbs, newRef);
      return new Adjusted<RARef>(rarefNew, upper, rowRef == newRef);
    } else {           // Insert N columns before column R; we're at column r
      InsertRowCols(R, N, r, colAbs, colRef, out newRef, out upper);
      RARef rarefNew = new RARef(colAbs, newRef, rowAbs, rowRef);
      return new Adjusted<RARef>(rarefNew, upper, colRef == newRef);
    }
  }

  // Insert N new rowcols before rowcol R>=0, when we're at rowcol r
  private static void InsertRowCols(int R, int N, int r,
                                    bool rcAbs, int rcRef,
                                    out int newRc, out int upper) {
    if (rcAbs) {
      if (rcRef >= R) {
        // Absolute ref to cell after inserted         // Case (Ab)
        newRc = rcRef + N;
        upper = int.MaxValue;
      } else {
```

```
        // Absolute ref to cell before inserted       // Case (Aa)
        newRc = rcRef;
        upper = int.MaxValue;
      }
    } else // Relative reference
      if (r >= R) {
        if (r + rcRef < R) {
          // Relative ref from after inserted rowcols to cell before them
          newRc = rcRef - N;                          // Case (Rab)
          upper = R - rcRef;
        } else {
          // Relative ref from after inserted rowcols to cell after them
          newRc = rcRef;                              // Case (Rbb)
          upper = int.MaxValue;
        }
      } else // r < R
        if (r + rcRef >= R) {
          // Relative ref from before inserted rowcols to cell after them
          newRc = rcRef + N;                          // Case (Rba)
          upper = R;
        } else {
          // Relative ref from before inserted rowcols to cell before them
          newRc = rcRef;                              // Case (Raa)
          upper = Math.Min(R, R - rcRef);
        }
  }

  // Clone and move (when the containing formula is moved, not copied)
  public RARef Move(int deltaCol, int deltaRow) {
    return new RARef(colAbs, colAbs ? colRef : colRef + deltaCol,
                     rowAbs, rowAbs ? rowRef : rowRef + deltaRow);
  }

  // Does this raref at (col, row) refer inside the sheet?
  public bool ValidAt(int col, int row) {
    CellAddr ca = new CellAddr(this, col, row);
    return 0 <= ca.col && 0 <= ca.row;
  }

  public String Show(int col, int row, Formats fo) {
    switch (fo.RefFmt) {
      case Formats.RefType.A1:
        CellAddr ca = new CellAddr(this, col, row);
        return (colAbs ? "$" : "") + CellAddr.ColumnName(ca.col)
             + (rowAbs ? "$" : "") + (ca.row + 1);
      case Formats.RefType.R1C1:
        return "R" + RelAbsFormat(rowAbs, rowRef, 1)
             + "C" + RelAbsFormat(colAbs, colRef, 1);
      case Formats.RefType.C0R0:
        return "C" + RelAbsFormat(colAbs, colRef, 0)
             + "R" + RelAbsFormat(rowAbs, rowRef, 0);
      default:
        throw new ImpossibleException("Unknown reference format");
    }
  }

  private static String RelAbsFormat(bool abs, int offset, int origo) {
    if (abs)
      return (offset + origo).ToString();
    else if (offset == 0)
      return "";
    else
      return "[" + (offset > 0 ? "+" : "") + offset.ToString() + "]";
  }

  public bool Equals(RARef that) {
    return that != null && this.colAbs == that.colAbs && this.rowAbs == that.rowAbs
      && this.colRef == that.colRef && this.rowRef == that.rowRef;
  }

  public override int GetHashCode() {
```



```
      return (((colAbs ? 1 : 0) + (rowAbs ? 2 : 0)) + colRef * 4) * 37 + rowRef;
    }
  }

  /// <summary>
  /// An Adjusted<T> represents an adjusted expression or
  /// a relative/absolute ref, for use in method InsertRowCols.
  /// </summary>
  /// <typeparam name="T">The type of adjusted entity: Expr or RaRef.</typeparam>
  public struct Adjusted<T> {
    public readonly T e;          // The adjusted Expr or RaRef
    public readonly int upper;    // ... invalid for rows >= upper
    public readonly bool same;    // Adjusted is identical to original

    public Adjusted(T e, int upper, bool same) {
      this.e = e;
      this.upper = upper;
      this.same = same;
    }

    public Adjusted(T e) : this(e, int.MaxValue, true) { }
  }
}
```

```
// Funcalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.IO;                        // MemoryStream, Stream
using System.Text;
using System.Diagnostics;

namespace Corecalc {

  /// <summary>
  /// The recalculation state of a Formula cell.
  /// </summary>
  public enum CellState { Dirty, Enqueued, Computing, Uptodate }

  /// <summary>
  /// A Cell is what populates one position of a Sheet, if anything.
  /// Some cells (ConstCells) are immutable and so it seems the same
  /// cell could appear multiple places in a workbook, but since cells
  /// have metadata, such as the support set, which varies from position to position, that
doesn't work.
  /// </summary>
  public abstract class Cell : IDepend {
    // The CellRanges in the support set may overlap; null means empty set
    private SupportSet supportSet;

    public abstract Value Eval(Sheet sheet, int col, int row);

    public abstract Cell MoveContents(int deltaCol, int deltaRow);

    public abstract void InsertRowCols(Dictionary<Expr, Adjusted<Expr>> adjusted,
              Sheet modSheet, bool thisSheet,
              int R, int N, int r, bool doRows);

    public abstract void ResetCellState();

    // Mark the cell dirty, for subsequent evaluation
    public abstract void MarkDirty();

    // Mark cell, dirty if non-empty
    public static void MarkCellDirty(Sheet sheet, int col, int row) {
      // Console.WriteLine("MarkDirty({0})", new FullCellAddr(sheet, col, row));
      Cell cell = sheet[col, row];
      if (cell != null)
        cell.MarkDirty();
    }

    // Enqueue this cell for evaluation
```



```
    public abstract void EnqueueForEvaluation(Sheet sheet, int col, int row);

    // Enqueue the cell at sheet[col, row] for evaluation, if non-null
    public static void EnqueueCellForEvaluation(Sheet sheet, int col, int row) {
      Cell cell = sheet[col, row];
      if (cell != null)
        cell.EnqueueForEvaluation(sheet, col, row); // Add if not already added, etc
    }

    // Show computed value; overridden in Formula and ArrayFormula to show cached value
    public virtual String ShowValue(Sheet sheet, int col, int row) {
      Value v = Eval(sheet, col, row);
      return v != null ? v.ToString() : "";
    }

    // Show constant or formula or array formula
    public abstract String Show(int col, int row, Formats fo);

    // Parse string to cell contents at (col, row) in given workbook
    public static Cell Parse(String text, Workbook workbook, int col, int row) {
      if (!String.IsNullOrWhiteSpace(text)) {
        Scanner scanner = new Scanner(IO.IOFormat.MakeStream(text));
        Parser parser = new Parser(scanner);
        return parser.ParseCell(workbook, col, row);      // May be null
      } else
        return null;
    }

    // Add the support range to the cell, avoiding direct self-support at sheet[col,row]
    public void AddSupport(Sheet sheet, int col, int row,
                           Sheet suppSheet, Interval suppCols, Interval suppRows)
    {
      if (supportSet == null)
        supportSet = new SupportSet();
      supportSet.AddSupport(sheet, col, row, suppSheet, suppCols, suppRows);
    }

    // Remove sheet[col,row] from the support sets of cells that this cell refers to
    public abstract void RemoveFromSupportSets(Sheet sheet, int col, int row);

    // Remove sheet[col,row] from this cell's support set
    public void RemoveSupportFor(Sheet sheet, int col, int row) {
      if (supportSet != null)
        supportSet.RemoveCell(sheet, col, row);
    }

    // Overridden in ArrayFormula?
    public virtual void ForEachSupported(Action<Sheet, int, int> act) {
      if (supportSet != null)
        supportSet.ForEachSupported(act);
    }

    // Use at manual cell update, and only if the oldCell is never used again
    public void TransferSupportTo(ref Cell newCell) {
      if (supportSet != null) {
        newCell = newCell ?? new BlankCell();
        newCell.supportSet = supportSet;
      }
    }

    // Add to support sets of all cells referred to from this cell, when
    // the cell appears in the block supported[col..col+cols-1, row..row+rows-1]
    public abstract void AddToSupportSets(Sheet supported, int col, int row, int cols, int
rows);

    // Clear the cell's support set; in ArrayFormula also clear the supportSetUpdated flag
    public virtual void ResetSupportSet() {
      supportSet = null;
    }

    public abstract void ForEachReferred(Sheet sheet, int col, int row, Action<FullCellAddr
```

```
> act);

    // True if the expression in the cell is volatile
    public abstract bool IsVolatile { get; }

    // Clone cell (supportSet, state fields) but not its sharable contents
    public abstract Cell CloneCell(int col, int row);

    public abstract void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn);
  }

  /// <summary>
  /// A ConstCell is a cell that contains a constant only.  Its value is
  /// immutable, yet it cannot in general be shared between sheet positions
  /// because these may have different supported sets and other metadata.
  /// </summary>
  abstract class ConstCell : Cell {
    public override Cell MoveContents(int deltaCol, int deltaRow) {
      return this;
    }

    public override void InsertRowCols(Dictionary<Expr, Adjusted<Expr>> adjusted,
               Sheet modSheet, bool thisSheet,
               int R, int N, int r, bool doRows) { }

    public override void AddToSupportSets(Sheet supported, int col, int row, int cols, int
rows) { }

    public override void RemoveFromSupportSets(Sheet sheet, int col, int row) { }

    public override void ForEachReferred(Sheet sheet, int col, int row, Action<FullCellAddr
> act)
    { }

    public override void MarkDirty() {
      ForEachSupported(MarkCellDirty);
    }

    // A (newly edited) constant cell should not be enqueued, but its support set should
    public override void EnqueueForEvaluation(Sheet sheet, int col, int row) {
      ForEachSupported(EnqueueCellForEvaluation);
    }

    public override void ResetCellState() { }

    public override bool IsVolatile {
      get { return false; }
    }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) { }
  }

  /// <summary>
  /// A NumberCell is a cell containing a floating-point constant.
  /// </summary>
  sealed class NumberCell : ConstCell {
    public readonly NumberValue value;        // Non-null

    public NumberCell(double d) {
      Debug.Assert(!Double.IsNaN(d) && !Double.IsInfinity(d));
      value = (NumberValue)NumberValue.Make(d);
    }

    private NumberCell(NumberCell cell) {
      this.value = cell.value;
    }

    public override Value Eval(Sheet sheet, int col, int row) {
      return value;
    }
```



```
    public override String Show(int col, int row, Formats fo) {
      return value.value.ToString();
    }

    public override Cell CloneCell(int col, int row) {
      return new NumberCell(this);
    }
  }

  /// <summary>
  /// A QuoteCell is a cell containing a single-quoted string constant.
  /// </summary>
  sealed class QuoteCell : ConstCell {
    public readonly TextValue value;          // Non-null

    public QuoteCell(String s) {
      Debug.Assert(s != null);
      this.value = TextValue.Make(s);         // No interning
    }

    private QuoteCell(QuoteCell cell) {
      this.value = cell.value;
    }

    public override Value Eval(Sheet sheet, int col, int row) {
      return value;
    }

    public override String Show(int col, int row, Formats fo) {
      return "'" + value.value;
    }

    public override Cell CloneCell(int col, int row) {
      return new QuoteCell(this);
    }
  }

  /// <summary>
  /// A TextCell is a cell containing a double-quoted string constant.
  /// </summary>
  sealed class TextCell : ConstCell {
    public readonly TextValue value;          // Non-null

    public TextCell(String s) {
      Debug.Assert(s != null);
      this.value = TextValue.Make(s);         // No interning
    }

    private TextCell(TextCell cell) {
      this.value = cell.value;
    }

    public override Value Eval(Sheet sheet, int col, int row) {
      return value;
    }

    public override String Show(int col, int row, Formats fo) {
      return "\"" + value.value + "\"";
    }

    public override Cell CloneCell(int col, int row) {
      return new TextCell(this);
    }
  }

  /// <summary>
  /// A BlankCell is a blank cell, used only to record a blank cell's support set.
  /// </summary>
  sealed class BlankCell : ConstCell {
    public override Value Eval(Sheet sheet, int col, int row) {
      return null;
```

```
    }

    public override String Show(int col, int row, Formats fo) {
      return "";
    }

    public override Cell CloneCell(int col, int row) {
      return new BlankCell();
    }
  }

  /// <summary>
  /// A Formula is a non-null caching expression contained in one cell.
  /// </summary>
  sealed class Formula : Cell {
    public readonly Workbook workbook;  // Non-null
    private Expr e;                     // Non-null
    public CellState state;             // Initially Dirty
    private Value v;                    // Up to date if state==Uptodate

    public Formula(Workbook workbook, Expr e) {
      Debug.Assert(workbook != null);
      Debug.Assert(e != null);
      this.workbook = workbook;
      this.e = e;
      this.state = CellState.Uptodate;
    }

    public static Formula Make(Workbook workbook, Expr e) {
      if (e == null)
        return null;
      else
        return new Formula(workbook, e);
    }

    // FIXME: Adequate for moving one cell, but block moves and row/column
    // inserts should avoid the duplication and unsharing of expressions.
    public override Cell MoveContents(int deltaCol, int deltaRow) {
      return new Formula(workbook, e.Move(deltaCol, deltaRow));
    }

    // Evaluate cell's expression if necessary and cache its value;
    // also enqueue supported cells for evaluation if we use support graph
    public override Value Eval(Sheet sheet, int col, int row) {
      switch (state) {
        case CellState.Uptodate:
          break;
        case CellState.Computing:
          FullCellAddr culprit = new FullCellAddr(sheet, col, row);
          String msg = String.Format("### CYCLE in cell {0} formula {1}",
                                     culprit, Show(col, row, workbook.format));
          throw new CyclicException(msg, culprit);
        case CellState.Dirty:
        case CellState.Enqueued:
          state = CellState.Computing;
          v = e.Eval(sheet, col, row);
          state = CellState.Uptodate;
          if (workbook.UseSupportSets)
            ForEachSupported(EnqueueCellForEvaluation);
          break;
      }
      return v;
    }

    public override void InsertRowCols(Dictionary<Expr, Adjusted<Expr>> adjusted,
                Sheet modSheet, bool thisSheet,
                int R, int N, int r, bool doRows) {
      Adjusted<Expr> ae;
      if (adjusted.ContainsKey(e) && r < adjusted[e].upper)
        // There is a valid cached adjusted expression
        ae = adjusted[e];
```



```
      else {
        // Compute a new adjusted expression and insert into the cache
        ae = e.InsertRowCols(modSheet, thisSheet, R, N, r, doRows);
        Console.WriteLine("Making new adjusted at rowcol" + r
                          + "; upper = " + ae.upper);
        if (ae.same) { // For better sharing, reuse unadjusted e if same
          ae = new Adjusted<Expr>(e, ae.upper, ae.same);
          Console.WriteLine("Reusing expression");
        }
        adjusted[e] = ae;
      }
      Debug.Assert(r < ae.upper, "Formula.InsertRowCols");
      e = ae.e;
    }

    public Value Cached {
      get { return v; }
    }

    public override void MarkDirty() {
      if (state != CellState.Dirty) {
        state = CellState.Dirty;
        ForEachSupported(MarkCellDirty);
      }
    }

    public override void EnqueueForEvaluation(Sheet sheet, int col, int row) {
      if (state == CellState.Dirty) { // Not Computing or Enqueued or Uptodate
        state = CellState.Enqueued;
        sheet.workbook.AddToQueue(sheet, col, row);
      }
    }

    public override void AddToSupportSets(Sheet supported, int col, int row, int cols, int
rows) {
      e.AddToSupportSets(supported, col, row, cols, rows);
    }

    public override void RemoveFromSupportSets(Sheet sheet, int col, int row) {
      e.RemoveFromSupportSets(sheet, col, row);
    }

    // Reset recomputation flags, eg. after a circularity has been found
    public override void ResetCellState() {
      state = CellState.Dirty;
    }

    public override void ForEachReferred(Sheet sheet, int col, int row, Action<FullCellAddr
> act) {
      e.ForEachReferred(sheet, col, row, act);
    }

    public override Cell CloneCell(int col, int row) {
      return new Formula(workbook, e.CopyTo(col, row));
    }

    public override bool IsVolatile {
      get { return e.IsVolatile; }
    }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      e.DependsOn(here, dependsOn);
    }

    public override String ShowValue(Sheet sheet, int col, int row) {
      // Use cached value, do not call Eval, as there might be a cycle!
      return v != null ? v.ToString() : "";
    }

    public override String Show(int col, int row, Formats fo) {
      return "=" + e.Show(col, row, 0, fo);
```

```
    }

    // Slight abuse of the cell state, used when detecting formula copies
    public bool Visited {
      get { return state == CellState.Uptodate; }
      set { state = value ? CellState.Uptodate : CellState.Dirty; }
    }

    public Expr Expr {
      get { return e; }
    }
  }

  /// <summary>
  /// An ArrayFormula is a cached array formula shared among several
  /// cells, each cell accessing one part of the result array.  Several
  /// ArrayFormula cells share one CachedArrayFormula cell; evaluation
  /// of one cell will evaluate the formula and cache its (array) value
  /// for the other cells to use.
  /// </summary>
  sealed class ArrayFormula : Cell {
    public readonly CachedArrayFormula caf;  // Non-null
    private readonly CellAddr ca;            // Cell's location within array value

    public ArrayFormula(CachedArrayFormula caf, CellAddr ca) {
      this.caf = caf; this.ca = ca;
    }

    public ArrayFormula(CachedArrayFormula caf, int col, int row)
      : this(caf, new CellAddr(col, row)) { }

    public override Value Eval(Sheet sheet, int col, int row) {
      Value v = caf.Eval();
      if (v is ArrayValue)
        return (v as ArrayValue)[ca];
      else if (v is ErrorValue)
        return v;
      else
        return ErrorValue.Make("#ERR: Not array");
    }

    public bool Contains(int col, int row) {
      return caf.ulCa.col <= col && col <= caf.lrCa.col
          && caf.ulCa.row <= row && row <= caf.lrCa.row;
    }

    public override Cell MoveContents(int deltaCol, int deltaRow) {
      // FIXME: loses sharing of the CachedArrayFormula; but then again
      // an array formula should never be moved cell by cell, but in its
      // entirety, and in one go.
      return new ArrayFormula(caf.MoveContents(deltaCol, deltaRow), ca);
    }

    public override void InsertRowCols(Dictionary<Expr, Adjusted<Expr>> adjusted,
                Sheet modSheet, bool thisSheet,
                int R, int N, int r, bool doRows) {
      // FIXME: Implement, make sure to update underlying formula only once
      throw new NotImplementedException("Insertions that move array formulas");
    }

    public override String ShowValue(Sheet sheet, int col, int row) {
      // Use the underlying cached value, do not call Eval, there might be a cycle!
      Value v = caf.CachedArray;
      if (v is ArrayValue) {
        Value element = (v as ArrayValue)[ca];
        return element != null ? element.ToString() : "";
      } else if (v is ErrorValue)
        return v.ToString();
      else
        return ErrorValue.Make("#ERR: Not array").ToString();
    }
```



```
    public override void MarkDirty() {
      switch (caf.formula.state) {
        case CellState.Uptodate:
          caf.formula.MarkDirty();
          ForEachSupported(MarkCellDirty);
          // caf.formula will call MarkDirty on the array formula's display
          // cells, except this one, so must mark its dependents explicitly.
          break;
        case CellState.Dirty:
          ForEachSupported(MarkCellDirty);
          break;
      }
    }

    public override void EnqueueForEvaluation(Sheet sheet, int col, int row) {
      switch (caf.formula.state) {
        case CellState.Dirty:
          caf.Eval();
          ForEachSupported(EnqueueCellForEvaluation);
          // caf.formula will call EnqueueForEvaluation on the array formula's
          // display cells, except this one, so must enqueue its dependents.
          break;
        case CellState.Uptodate:
          ForEachSupported(EnqueueCellForEvaluation);
          break;
      }
    }

    public override void ResetCellState() {
      caf.formula.ResetCellState();
    }

    public override void ResetSupportSet() {
      caf.ResetSupportSet();
      base.ResetSupportSet();
    }

    public override void AddToSupportSets(Sheet supported, int col, int row, int cols, int
rows) {
      caf.UpdateSupport(supported);
    }

    public override void RemoveFromSupportSets(Sheet sheet, int col, int row) {
      caf.RemoveFromSupportSets(sheet, col, row);
    }

    public override void ForEachReferred(Sheet sheet, int col, int row, Action<FullCellAddr
> act) {
      caf.ForEachReferred(act);
    }

    public override Cell CloneCell(int col, int row) {
      // Not clear how to copy an array formula.  It does make sense; see Excel.
      throw new System.NotImplementedException();
    }

    public override bool IsVolatile {
      get { return caf.formula.IsVolatile; }
    }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      // It seems that this could uselessly be called on every cell
      // that shares the formula, but this will not happen on function sheets
      caf.formula.DependsOn(here, dependsOn);
    }

    public override String Show(int col, int row, Formats fo) {
      return "{" + caf.Show(caf.formulaCol, caf.formulaRow, fo) + "}";
    }
  }
```

```
/// <summary>
/// A CachedArrayFormula is shared between multiple array cells.
/// It contains a (caching) array-valued formula, the original cell
/// address of that formula, and the upper left and lower right corners
/// of the array.
/// </summary>
sealed class CachedArrayFormula {
  public readonly Formula formula;             // Non-null
  public readonly Sheet sheet;                 // Sheet containing the array formulas
  public readonly int formulaCol, formulaRow;  // Location of formula entry
  public readonly CellAddr ulCa, lrCa;         // Corners of array formula
  private bool supportAdded, supportRemoved;   // Referred cells' support sets up to date

  // Invariant: Every cell within the display area sheet[ulCa, lrCa] is an
  // ArrayFormula whose CachedArrayFormula instance is this one.

  public CachedArrayFormula(Formula formula, Sheet sheet, int formulaCol, int formulaRow,
                            CellAddr ulCa, CellAddr lrCa)
  {
    if (formula == null)
      throw new Exception("CachedArrayFormula arguments");
    else {
      this.formula = formula;
      this.sheet = sheet;
      this.formulaCol = formulaCol;
      this.formulaRow = formulaRow;
      this.ulCa = ulCa;
      this.lrCa = lrCa;
      this.supportAdded = this.supportRemoved = false;
    }
  }

  // Evaluate expression if necessary
  public Value Eval() {
    return formula.Eval(sheet, formulaCol, formulaRow);
  }

  public CachedArrayFormula MoveContents(int deltaCol, int deltaRow) {
    // FIXME: Unshares the formula, shouldn't ...
    return new CachedArrayFormula((Formula)formula.MoveContents(deltaCol, deltaRow),
      sheet, formulaCol, formulaRow, ulCa, lrCa);
  }

  public Value CachedArray {
    get { return formula.Cached; }
  }

  public void ResetSupportSet() {
    // Do NOT clear the underlying formula's support set; it will not be recreated
    supportAdded = false;
  }

  public void UpdateSupport(Sheet supported) {
    // Update the support sets of cells referred from an array formula only once
    if (!supportAdded) {
      formula.AddToSupportSets(supported, formulaCol, formulaRow, 1, 1);
      supportAdded = true;
    }
  }

  public void RemoveFromSupportSets(Sheet sheet, int col, int row) {
    // Update the support sets of cells referred from an array formula only once
    if (!supportRemoved) {
      formula.RemoveFromSupportSets(sheet, col, row);
      supportRemoved = true;
    }
  }

  public void ForEachReferred(Action<FullCellAddr> act) {
    formula.ForEachReferred(sheet, formulaCol, formulaRow, act);
```



```
    }

    public String Show(int col, int row, Formats fo) {
      return formula.Show(col, row, fo);
    }
  }
}
```

```
// Corecalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;

// Class Expr and its subclasses are used to recursively build formulas

namespace Corecalc {
  // ----------------------------------------------------------------

  /// <summary>
  /// An Expr is an expression that may appear in a Formula cell.
  /// </summary>
  public abstract class Expr : IDepend {
    // Update cell references when containing cell is moved (not copied)
    public abstract Expr Move(int deltaCol, int deltaRow);

    // Invalidate off-sheet references when containing cell is copied (not moved)
    public abstract Expr CopyTo(int col, int row);

    // Evaluate expression as if at cell address sheet[col, row]
    public abstract Value Eval(Sheet sheet, int col, int row);

    internal abstract void VisitorCall(IExpressionVisitor visitor);

    // Insert N new rowcols before rowcol R>=0, when we're at rowcol r
    public abstract Adjusted<Expr> InsertRowCols(Sheet modSheet, bool thisSheet,
                                                 int R, int N, int r, bool doRows);

    // Apply refAct once to each CellRef in expression, and areaAct once to each CellArea
    internal abstract void VisitRefs(RefSet refSet, Action<CellRef> refAct,
                                                    Action<CellArea> areaAct);

    // Increase the support sets of all cells referred from this expression, when
    // the expression appears in the block supported[col..col+cols-1, row..row+rows-1]
    internal void AddToSupportSets(Sheet supported, int col, int row, int cols, int rows) {
      VisitRefs(new RefSet(),
        (CellRef cellRef)   => cellRef.AddToSupport(supported, col, row, cols, rows),
        (CellArea cellArea) => cellArea.AddToSupport(supported, col, row, cols, rows));
    }

    // Remove sheet[col, row] from the support sets of cells referred from this expression
    public void RemoveFromSupportSets(Sheet sheet, int col, int row) {
      ForEachReferred(sheet, col, row, // Remove sheet[col,row] from support set at fca
        delegate(FullCellAddr fca) {
          Cell cell;
```



```
          if (fca.TryGetCell(out cell)) // Will be non-null if support correctly added
            cell.RemoveSupportFor(sheet, col, row);
        });
    }

    // Apply act, once only, to the full cell address of each cell referred from expression
    public void ForEachReferred(Sheet sheet, int col, int row, Action<FullCellAddr> act) {
      VisitRefs(new RefSet(),
        (CellRef cellRef)  => act(cellRef.GetAbsoluteAddr(sheet, col, row)),
        (CellArea areaRef) => areaRef.ApplyToFcas(sheet, col, row, act));
    }

    // Call dependsOn(fca) on all cells fca referred from expression, with multiplicity.
    // Cannot be implemented in terms of VisitRefs, which visits only once.
    public abstract void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn);

    // True if expression textually contains a call to a volatile function
    public abstract bool IsVolatile { get; }

    // Show contents as expression
    public abstract String Show(int col, int row, int ctxpre, Formats fo);
  }

  /// <summary>
  /// A Const expression is a constant, immutable and sharable.
  /// </summary>
  abstract class Const : Expr {
    public static Const Make(Value value) {
      if (value is NumberValue)
        return new NumberConst((value as NumberValue).value);
      else if (value is TextValue)
        return new TextConst((value as TextValue).value);
      else
        return new ValueConst(value);
    }

    public override Expr Move(int deltaCol, int deltaRow) {
      return this;
    }

    // Any expression can be copied with sharing
    public override Expr CopyTo(int col, int row) {
      return this;
    }

    public override Adjusted<Expr> InsertRowCols(Sheet modSheet, bool thisSheet,
                                                 int R, int N, int r, bool doRows) {
      return new Adjusted<Expr>(this);
    }

    internal override void VisitRefs(RefSet refSet, Action<CellRef> refAct, Action<CellArea
> areaAct)
    { }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) { }

    public override bool IsVolatile {
      get { return false; }
    }
  }

  /// <summary>
  /// A NumberConst is a constant number-valued expression.
  /// </summary>
  class NumberConst : Const {
    public readonly NumberValue value;

    public NumberConst(double d) {
      Debug.Assert(!Double.IsNaN(d) && !Double.IsInfinity(d));
      value = (NumberValue)NumberValue.Make(d);
    }
```

```
  public override Value Eval(Sheet sheet, int col, int row) {
    return value;
  }

  internal override void VisitorCall(IExpressionVisitor visitor) {
    visitor.CallVisitor(this);
  }

  public override String Show(int col, int row, int ctxpre, Formats fo) {
    return value.ToString();
  }
}

/// <summary>
/// A TextConst is a constant string-valued expression.
/// </summary>
class TextConst : Const {
  public readonly TextValue value;

  public TextConst(String s) {
    value = TextValue.MakeInterned(s);
  }

  public override Value Eval(Sheet sheet, int col, int row) {
    return value;
  }

  internal override void VisitorCall(IExpressionVisitor visitor) {
    visitor.CallVisitor(this);
  }

  public override String Show(int col, int row, int ctxpre, Formats fo) {
    return "\"" + value + "\"";
  }
}

/// <summary>
/// A ValueConst is an arbitrary constant valued expression, used only
/// for partial evaluation; there is no corresponding formula source syntax.
/// </summary>
class ValueConst : Const {
  public readonly Value value;

  public ValueConst(Value value) {
    this.value = value;
  }

  public override Value Eval(Sheet sheet, int col, int row) {
    return value;
  }

  internal override void VisitorCall(IExpressionVisitor visitor) {
    visitor.CallVisitor(this);
  }

  public override String Show(int col, int row, int ctxpre, Formats fo) {
    return "ValueConst[" + value + "]";
  }
}

/// <summary>
/// An Error expression represents a static error, e.g. invalid cell reference.
/// </summary>
class Error : Const {
  private readonly String error;
  public readonly ErrorValue value;
  public static readonly Error refError = new Error(ErrorValue.refError);

  public Error(String msg) : this(ErrorValue.Make(msg)) { }
```



```
  public Error(ErrorValue value) {
    this.value = value;
    this.error = this.value.ToString();
  }

  public override Value Eval(Sheet sheet, int col, int row) {
    return value;
  }

  public override String Show(int col, int row, int ctxpre, Formats fo) {
    return error;
  }

  internal override void VisitorCall(IExpressionVisitor visitor) {
    visitor.CallVisitor(this);
  }
}

/// <summary>
/// A FunCall expression is an operator application such as 1+$A$4 or a function
/// call such as RAND() or SIN(4*A$7) or SUM(B4:B52; 3) or IF(A1; A2; 1/A1).
/// </summary>
class FunCall : Expr {
  public readonly Function function;   // Non-null
  public readonly Expr[] es;           // Non-null, elements non-null

  private FunCall(String name, params Expr[] es)
    : this(Function.Get(name), es) { }

  private FunCall(Function function, Expr[] es) {
    // Assert: function != null, all es[i] != null
    this.function = function;
    this.es = es;
  }

  public static Expr Make(String name, Expr[] es) {
    Function function = Function.Get(name);
    if (function == null)
      function = Function.MakeUnknown(name);
    for (int i = 0; i < es.Length; i++)
      if (es[i] == null)
        es[i] = new Error("#SYNTAX");
    if (name == "SPECIALIZE" && es.Length > 1)
      return new FunCall("SPECIALIZE", Make("CLOSURE", es));
    else
      return new FunCall(function, es);
  }

  // Arguments are passed unevaluated to cater for non-strict IF

  public override Value Eval(Sheet sheet, int col, int row) {
    return function.Applier(sheet, es, col, row);
  }

  public override Expr Move(int deltaCol, int deltaRow) {
    Expr[] newEs = new Expr[es.Length];
    for (int i = 0; i < es.Length; i++)
      newEs[i] = es[i].Move(deltaCol, deltaRow);
    return new FunCall(function, newEs);
  }

  // Can be copied with sharing if arguments can
  public override Expr CopyTo(int col, int row) {
    bool same = true;
    Expr[] newEs = new Expr[es.Length];
    for (int i = 0; i < es.Length; i++) {
      newEs[i] = es[i].CopyTo(col, row);
      same &= Object.ReferenceEquals(newEs[i], es[i]);
    }
    return same ? this : new FunCall(function, newEs);
  }
```

```
    public override Adjusted<Expr> InsertRowCols(Sheet modSheet, bool thisSheet,
                                                 int R, int N, int r, bool doRows)
    {
      Expr[] newEs = new Expr[es.Length];
      int upper = int.MaxValue;
      bool same = true;
      for (int i = 0; i < es.Length; i++) {
        Adjusted<Expr> ae
          = es[i].InsertRowCols(modSheet, thisSheet, R, N, r, doRows);
        upper = Math.Min(upper, ae.upper);
        same = same && ae.same;
        newEs[i] = ae.e;
      }
      return new Adjusted<Expr>(new FunCall(function, newEs), upper, same);
    }

    // Show infixed operators as infix and without excess parentheses

    public override String Show(int col, int row, int ctxpre, Formats fo) {
      StringBuilder sb = new StringBuilder();
      int pre = function.fixity;
      if (pre == 0) { // Not operator
        sb.Append(function.name).Append("(");
        for (int i = 0; i < es.Length; i++) {
          if (i > 0)
            sb.Append(",");
          sb.Append(es[i].Show(col, row, 0, fo));
        }
        sb.Append(")");
      } else { // Operator.  Assume es.Length is 1 or 2
        if (es.Length == 2) {
          // If precedence lower than context, add parens
          if (pre < ctxpre)
            sb.Append("(");
          sb.Append(es[0].Show(col, row, pre, fo));
          sb.Append(function.name);
          // Only higher precedence right operands avoid parentheses
          sb.Append(es[1].Show(col, row, pre + 1, fo));
          if (pre < ctxpre)
            sb.Append(")");
        } else if (es.Length == 1) {
          sb.Append(function.name == "NEG" ? "-" : function.name);
          sb.Append(es[0].Show(col, row, pre, fo));
        } else
          throw new ImpossibleException("Operator not unary or binary");
      }
      return sb.ToString();
    }

    internal override void VisitRefs(RefSet refSet, Action<CellRef> refAct, Action<CellArea
> areaAct)
    {
      foreach (Expr e in es)
        e.VisitRefs(refSet, refAct, areaAct);
    }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      foreach (Expr e in es)
        e.DependsOn(here, dependsOn);
    }

    public override bool IsVolatile {
      get {
        if (function.IsVolatile(es))
          return true;
        foreach (Expr e in es)
          if (e.IsVolatile)
            return true;
        return false;
      }
```



```
    }

    internal override void VisitorCall(IExpressionVisitor visitor) {
      visitor.CallVisitor(this);
    }
  }

  /// <summary>
  /// A CellRef expression refers to a single cell, eg.
  /// is A1 or $A1 or A$1 or $A$1 or Sheet1!A1.
  /// </summary>
  class CellRef : Expr, IEquatable<CellRef> {
    public readonly RARef raref;
    public readonly Sheet sheet;    // non-null if sheet-absolute

    public CellRef(Sheet sheet, RARef raref) {
      this.sheet = sheet;
      this.raref = raref;
    }

    // Evaluate cell ref by evaluating the cell referred to

    public override Value Eval(Sheet sheet, int col, int row) {
      CellAddr ca = raref.Addr(col, row);
      Cell cell = (this.sheet ?? sheet)[ca];
      return cell == null ? null : cell.Eval(sheet, ca.col, ca.row);
    }

    public FullCellAddr GetAbsoluteAddr(Sheet sheet, int col, int row) {
      return new FullCellAddr(this.sheet ?? sheet, raref.Addr(col, row));
    }

    public FullCellAddr GetAbsoluteAddr(FullCellAddr fca) {
      return GetAbsoluteAddr(fca.sheet, fca.ca.col, fca.ca.row);
    }

    // Clone and move (when the containing formula is moved, not copied!)
    public override Expr Move(int deltaCol, int deltaRow) {
      return new CellRef(sheet, raref.Move(deltaCol, deltaRow));
    }

    // Can be copied with sharing iff reference is within sheet
    public override Expr CopyTo(int col, int row) {
      if (raref.ValidAt(col, row))
        return this;
      return
        Error.refError;
    }

    public override Adjusted<Expr> InsertRowCols(Sheet modSheet, bool thisSheet,
                                                 int R, int N, int r,
                                                 bool doRows) {
      if (sheet == modSheet || sheet == null && thisSheet) {
        Adjusted<RARef> adj = raref.InsertRowCols(R, N, r, doRows);
        return new Adjusted<Expr>(new CellRef(sheet, adj.e), adj.upper, adj.same);
      } else
        return new Adjusted<Expr>(this);
    }

    internal void AddToSupport(Sheet supported, int col, int row, int cols, int rows)
    {
      Sheet referredSheet = this.sheet ?? supported;
      int ca = raref.colRef, ra = raref.rowRef;
      int r1 = row, r2 = row + rows - 1, c1 = col, c2 = col + cols - 1;
      Interval referredCols, referredRows;
      Func<int, Interval> supportedCols, supportedRows;
      RefAndSupp(raref.colAbs, ca, c1, c2, out referredCols, out supportedCols);
      RefAndSupp(raref.rowAbs, ra, r1, r2, out referredRows, out supportedRows);
      // Outer iteration is made over the shorter interval for efficiency
      if (referredCols.Length < referredRows.Length)
        referredCols.ForEach(c => {
```

```
          Interval suppCols = supportedCols(c);
          referredRows.ForEach(r =>
            referredSheet.AddSupport(c, r, supported, suppCols, supportedRows(r)));
        });
      else
        referredRows.ForEach(r => {
          Interval suppRows = supportedRows(r);
          referredCols.ForEach(c =>
            referredSheet.AddSupport(c, r, supported, supportedCols(c), suppRows));
        });
    }

    // This uses the notation from the book's analysis of the row aspect of support sets
    private static void RefAndSupp(bool abs, int ra, int r1, int r2,
        out Interval referred, out Func<int, Interval> supported) {
      if (abs) { // case abs
        referred = new Interval(ra, ra);
        supported = r => new Interval(r1, r2);
      } else {   // case rel
        referred = new Interval(r1 + ra, r2 + ra);
        supported = r => new Interval(r - ra, r - ra);
      }
    }

    internal override void VisitRefs(RefSet refSet, Action<CellRef> refAct, Action<CellArea
> areaAct)
    {
      if (!refSet.SeenBefore(this))
        refAct(this);
    }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      dependsOn(GetAbsoluteAddr(here));
    }

    public override bool IsVolatile {
      get { return false; }
    }

    public bool Equals(CellRef that) {
      return this.raref.Equals(that.raref);
    }

    public override int GetHashCode() {
      return raref.GetHashCode();
    }

    public override String Show(int col, int row, int ctxpre, Formats fo) {
      String s = raref.Show(col, row, fo);
      return sheet == null ? s : sheet.Name + "!" + s;
    }

    internal override void VisitorCall(IExpressionVisitor visitor) {
      visitor.CallVisitor(this);
    }
  }

  /// <summary>
  /// A CellArea expression refers to a rectangular cell area, eg. is
  /// A1:C4 or $A$1:C4 or A1:$C4 or Sheet1!A1:C4
  /// </summary>
  class CellArea : Expr, IEquatable<CellArea> {
    // It would be desirable to store the cell area in normalized form,
    // with ul always to the left and above lr; but this cannot be done
    // because it may be mixed relative/absolute as in $A1:$A$5, which
    // when copied from B1 to B10 is no longer normalized.
    private readonly RARef ul, lr;  // upper left, lower right
    public readonly Sheet sheet;    // non-null if sheet-absolute

    public CellArea(Sheet sheet,
                    bool ulColAbs, int ulColRef, bool ulRowAbs, int ulRowRef,
```



```
                    bool lrColAbs, int lrColRef, bool lrRowAbs, int lrRowRef)
      : this(sheet,
             new RARef(ulColAbs, ulColRef, ulRowAbs, ulRowRef),
             new RARef(lrColAbs, lrColRef, lrRowAbs, lrRowRef)) {
    }

    public CellArea(Sheet sheet, RARef ul, RARef lr) {
      this.sheet = sheet;
      this.ul = ul;
      this.lr = lr;
    }

    // Evaluate cell area by returning an array view of it

    public override Value Eval(Sheet sheet, int col, int row) {
      return MakeArrayView(sheet, col, row);
    }

    public ArrayView MakeArrayView(FullCellAddr fca) {
      return MakeArrayView(fca.sheet, fca.ca.col, fca.ca.row);
    }

    public ArrayView MakeArrayView(Sheet sheet, int col, int row) {
      CellAddr ulCa = ul.Addr(col, row), lrCa = lr.Addr(col, row);
      ArrayView view = ArrayView.Make(ulCa, lrCa, this.sheet ?? sheet);
      // Forcing the evaluation of all cells in an array view value.
      // TODO: Doing this repeatedly, as in ManyDependents.xml, is costly
      for (int c = 0; c < view.Cols; c++)
        for (int r = 0; r < view.Rows; r++) {
          // Value ignore = view[c, r];
        }
      return view;
    }

    public void ApplyToFcas(Sheet sheet, int col, int row, Action<FullCellAddr> act) {
      CellAddr ulCa = ul.Addr(col, row), lrCa = lr.Addr(col, row);
      ArrayView.Make(ulCa, lrCa, this.sheet ?? sheet).Apply(act);
    }

    // Clone and move (when the containing formula is moved, not copied)
    public override Expr Move(int deltaCol, int deltaRow) {
      return new CellArea(sheet,
                          ul.Move(deltaCol, deltaRow),
                          lr.Move(deltaCol, deltaRow));
    }

    // Can copy cell area with sharing iff corners are within sheet
    public override Expr CopyTo(int col, int row) {
      if (ul.ValidAt(col, row) && lr.ValidAt(col, row))
        return this;
      else
        return Error.refError;
    }

    public override Adjusted<Expr> InsertRowCols(Sheet modSheet, bool thisSheet,
                                                 int R, int N, int r,
                                                 bool doRows) {
      if (sheet == modSheet || sheet == null && thisSheet) {
        Adjusted<RARef> ulNew = ul.InsertRowCols(R, N, r, doRows),
                        lrNew = lr.InsertRowCols(R, N, r, doRows);
        int upper = Math.Min(ulNew.upper, lrNew.upper);
        return new Adjusted<Expr>(new CellArea(sheet, ulNew.e, lrNew.e),
                                  upper, ulNew.same && lrNew.same);
      } else
        return new Adjusted<Expr>(this);
    }

    internal void AddToSupport(Sheet supported, int col, int row, int cols, int rows) {
      Sheet referredSheet = this.sheet ?? supported;
      Interval referredRows, referredCols;
      Func<int, Interval> supportedRows, supportedCols;
```

```
      int ra = ul.rowRef, rb = lr.rowRef, r1 = row, r2 = row + rows - 1;
      RefAndSupp(ul.rowAbs, lr.rowAbs, ra, rb, r1, r2, out referredRows, out supportedRows)
;
      int ca = ul.colRef, cb = lr.colRef, c1 = col, c2 = col + cols - 1;
      RefAndSupp(ul.colAbs, lr.colAbs, ca, cb, c1, c2, out referredCols, out supportedCols)
;
      // Outer iteration should be over the shorter interval for efficiency
      if (referredCols.Length < referredRows.Length)
        referredCols.ForEach(c => {
          Interval suppCols = supportedCols(c);
          referredRows.ForEach(r =>
            referredSheet.AddSupport(c, r, supported, suppCols, supportedRows(r)));
        });
      else
        referredRows.ForEach(r => {
          Interval suppRows = supportedRows(r);
          referredCols.ForEach(c =>
            referredSheet.AddSupport(c, r, supported, supportedCols(c), suppRows));
        });
    }

    // This uses notation from the book's discussion of the row dimension of support sets,
    // works equally well for the column dimension.
    // Assumes r1 <= r2 but nothing about the order of ra and rb
    private static void RefAndSupp(bool ulAbs, bool lrAbs, int ra, int rb, int r1, int r2,
        out Interval referred, out Func<int, Interval> supported) {
      if (ulAbs) {
        if (lrAbs) { // case abs-abs
          SortInts(ref ra, ref rb);
          referred = new Interval(ra, rb);
          supported = r => new Interval(r1, r2);
        } else {     // case abs-rel
          referred = new Interval(Math.Min(ra, r1 + rb), Math.Max(ra, r2 + rb));
          supported = r => ra < r ? new Interval(Math.Max(r1, r - rb), r2)
                         : ra > r ? new Interval(r1, Math.Min(r2, r - rb))
                                  : new Interval(r1, r2);
        }
      } else {
        if (lrAbs) { // case rel-abs
          referred = new Interval(Math.Min(r1 + ra, rb), Math.Max(r2 + ra, rb));
          supported = r => rb > r ? new Interval(r1, Math.Min(r2, r - ra))
                         : rb < r ? new Interval(Math.Max(r1, r - ra), r2)
                                  : new Interval(r1, r2);
        } else {     // case rel-rel
          SortInts(ref ra, ref rb);
          referred = new Interval(r1 + ra, r2 + rb);
          supported = r => new Interval(Math.Max(r1, r - rb), Math.Min(r2, r - ra));
        }
      }
    }

    private static void SortInts(ref int a, ref int b) {
      if (a > b) {
        int tmp = a; a = b; b = tmp;
      }
    }

    internal override void VisitRefs(RefSet refSet, Action<CellRef> refAct, Action<CellArea
> areaAct)
    {
      if (!refSet.SeenBefore(this))
        areaAct(this);
    }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      ApplyToFcas(here.sheet, here.ca.col, here.ca.row, dependsOn);
    }

    public override bool IsVolatile {
      get { return false; }
    }
```



```
    public bool Equals(CellArea that) {
      return that != null && this.ul.Equals(that.ul) && this.lr.Equals(that.lr);
    }

    public override int GetHashCode() {
      return lr.GetHashCode() * 511 + ul.GetHashCode();
    }

    public override String Show(int col, int row, int ctxpre, Formats fo) {
      String s = ul.Show(col, row, fo) + ":" + lr.Show(col, row, fo);
      return sheet == null ? s : sheet.Name + "!" + s;
    }

    internal override void VisitorCall(IExpressionVisitor visitor) {
      visitor.CallVisitor(this);
    }
  }

  /// <summary>
  /// An IExpressionVisitor is used to traverse the Expr abstract syntax used in formulas.
  /// </summary>
  interface IExpressionVisitor {
    void CallVisitor(NumberConst numbConst);
    void CallVisitor(TextConst textConst);
    void CallVisitor(ValueConst valueConst);
    void CallVisitor(Error expr);
    void CallVisitor(FunCall funCall);
    void CallVisitor(CellRef cellRef);
    void CallVisitor(CellArea cellArea);
  }

  /// <summary>
  /// A RefSet is a set of CellRefs and CellAreas already seen by a VisitRefs visitor.
  /// </summary>
  internal class RefSet {
    private readonly HashSet<CellRef> cellRefsSeen = new HashSet<CellRef>();
    private readonly HashSet<CellArea> cellAreasSeen = new HashSet<CellArea>();

    public void Clear() {
      cellRefsSeen.Clear();
      cellAreasSeen.Clear();
    }

    public bool SeenBefore(CellRef cellRef) {
      return !cellRefsSeen.Add(cellRef);
    }

    public bool SeenBefore(CellArea cellArea) {
      return !cellAreasSeen.Add(cellArea);
    }
  }
}
```

```
// Funcalc, spreadsheet with functions
// --------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// --------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Text;
using System.Diagnostics;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A CGExpr represents an expression as used in sheet-defined functions,
  /// suitable for code generation and partial evaluation.
  /// </summary>
  public abstract class CGExpr : CodeGenerate, IDepend {
    /// <summary>
    ///  The entry point for compilation, called from ProgramLines.
    ///  The generated code must leave a Value on the stack top.
    /// </summary>
    public abstract void Compile();

    /// <summary>
    /// Compile expression that is expected to evaluate to a number, leaving
    /// a float64 on the stack top and avoiding wrapping where possible.
    /// If result is an error, produce a NaN whose 32 least significant bits
    /// give that error's int index into the ErrorValue.errorTable arraylist.
    /// </summary>
    public abstract void CompileToDoubleOrNan();

    // General version in terms of CompileToDoubleOrNan.
    // Should be overridden in CGNumberConst, CGTextConst, CGError, CGComposite ...
    /// <summary>
    /// Compile expression that is expected to evaluate to a proper (finite
    /// and non-NaN) number; generate code to test whether it is actually a
    /// proper number and then execute the code generated by ifProper, or
    /// else execute the code generated by ifOther.
    /// </summary>
    /// <param name="ifProper">Generates code for the case where the expression
    /// evaluates to a proper number; the generated code expects to find the value as
    /// an unwrapped proper float64 on the stack top.</param>
    /// <param name="ifOther"></param>
    public virtual void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      CompileToDoubleOrNan();
      ilg.Emit(OpCodes.Stloc, testDouble);
      ilg.Emit(OpCodes.Ldloc, testDouble);
      ilg.Emit(OpCodes.Call, isInfinityMethod);
      ilg.Emit(OpCodes.Brtrue, ifOther.GetLabel(ilg));
      ilg.Emit(OpCodes.Ldloc, testDouble);
```



```
      ilg.Emit(OpCodes.Call, isNaNMethod);
      ilg.Emit(OpCodes.Brtrue, ifOther.GetLabel(ilg));
      ilg.Emit(OpCodes.Ldloc, testDouble);
      ifProper.Generate(ilg);
      if (!ifOther.Generated) {
        Label endLabel = ilg.DefineLabel();
        ilg.Emit(OpCodes.Br, endLabel);
        ifOther.Generate(ilg);
        ilg.MarkLabel(endLabel);
      }
    }

    /// <summary>
    /// Compiles an expression as a condition, that can be true (if non-zero) or
    /// false (if zero) or other (if +/-infinity or NaN).  If possible, avoids
    /// computing and pushing a value and then testing it, instead performing
    /// comparisons directly on arguments, or even statically.  This implementation
    /// is a general version in terms of CompileToDoubleProper.  Should be overridden
    /// in CGNumberConst, CGTextConst, CGError, CGIf, CGComparison, ...
    /// </summary>
    /// <param name="ifTrue">Generates code for the true branch</param>
    /// <param name="ifFalse">Generates code for the false branch</param>
    /// <param name="ifOther">Generates code for the other (neither true nor
    /// false) branch</param>
    public virtual void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      CompileToDoubleProper(
        new Gen(delegate {
        ilg.Emit(OpCodes.Ldc_R8, 0.0);
        ilg.Emit(OpCodes.Beq, ifFalse.GetLabel(ilg));
        ifTrue.Generate(ilg);
        if (!ifFalse.Generated) {
          Label endLabel = ilg.DefineLabel();
          ilg.Emit(OpCodes.Br, endLabel);
          ifFalse.Generate(ilg);
          ilg.MarkLabel(endLabel);
        }
      }),
      ifOther);
    }

    /// <summary>
    /// Specialize the expression with respect to the values/residual expressions in pEnv.
    /// </summary>
    /// <param name="pEnv">Maps cell addresses to already-specialized expressions</param>
    /// <returns>The specialized expression</returns>

    public abstract CGExpr PEval(PEnv pEnv, bool hasDynamicControl);

    /// <summary>
    /// Update the evaluation conditions (in the evalConds dictionary) for every cell
    /// referenced from this expression, assuming that this expression itself
    /// has evaluation condition evalCond.
    /// </summary>
    /// <param name="evalCond"></param>
    /// <param name="evalConds"></param>
    public abstract void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches);

    /// <summary>
    /// Called on a sheet-defined function's output
    /// cell expression to record that that expression and some of its
    /// subexpressions are in tail position.  Only overridden in CGIf,
    /// CGChoose and CGSdfCall (but not CGApply).
    /// </summary>
    public virtual void NoteTailPosition() { }

    /// <summary>
    /// Returns true if the expression's evaluation could
    /// be recursive, have side effects, or take a long time.
    /// </summary>
```

```
    /// <param name="bound">The bound on non-serious expression size</param>
    public virtual bool IsSerious(ref int bound) {
      return 0 > bound--;
    }

    public abstract void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn);

    /// <summary>
    /// Update the numberUses bag with the number of number-uses of each full cell
    /// address in this expression.
    /// </summary>
    /// <param name="typ">The expected type of this expression</param>
    /// <param name="numberUses">A hashbag noting for each full cell address
    /// the number of times it is used as a NumberValue.</param>
    public abstract void CountUses(Typ typ, HashBag<FullCellAddr> numberUses);

    public abstract Typ Type();

    public bool Is(double d) {
      return this is CGNumberConst && (this as CGNumberConst).number.value == d;
    }
  }

  /// <summary>
  /// A CGCellRef is a reference to a single cell on this function sheet.
  /// </summary>
  public class CGCellRef : CGExpr {
    private readonly FullCellAddr cellAddr;
    private readonly Variable var;

    public CGCellRef(FullCellAddr cellAddr, Variable var) {
      this.cellAddr = cellAddr;
      this.var = var;
    }

    public override void Compile() {
      var.EmitLoad(ilg);
      if (var.Type == Typ.Number)
        WrapDoubleToNumberValue();
      // In other cases, there's no need to wrap the variable's contents
    }

    public override void CompileToDoubleOrNan() {
      Variable doubleVar;
      if (NumberVariables.TryGetValue(cellAddr, out doubleVar))
        doubleVar.EmitLoad(ilg);
      else {
        if (var.Type == Typ.Value) {
          var.EmitLoad(ilg);
          UnwrapToDoubleOrNan();
        } else if (var.Type == Typ.Number)
          var.EmitLoad(ilg);
        else // A variable of a type not convertible to a float64, so ArgTypeError
          LoadErrorNan(ErrorValue.argTypeError);
      }
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      return pEnv[cellAddr];
    }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) {
      PathCond old;
      if (evalConds.TryGetValue(cellAddr, out old))
        evalConds[cellAddr] = old.Or(evalCond);
      else
        evalConds[cellAddr] = evalCond;
    }
```



```
    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      dependsOn(cellAddr);
    }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
      if (typ == Typ.Number)
        numberUses.Add(this.cellAddr);
    }

    public override string ToString() {
      return var.Name;
    }

    public override Typ Type() {
      return var.Type;
    }
  }

  /// <summary>
  /// A CGCachedExpr holds a number-valued expression and its cached value;
  /// used in evaluation conditions.
  /// </summary>
  public class CGCachedExpr : CGExpr {
    public readonly CGExpr expr;
    private readonly CachedAtom cachedAtom;
    private LocalBuilder cacheVariable;  // The cache localvar (double)
    private int generateCount = 0;       // Number of times emitted by CachedAtom.ToCGExpr
    private readonly int cacheNumber;    // Within the current SDF, for diagnostics only

    private const int uninitializedBits = -1;
    private static readonly double uninitializedNan = ErrorValue.MakeNan(uninitializedBits)
;
    private static readonly MethodInfo
      doubleToInt64BitsMethod = typeof(System.BitConverter).GetMethod("DoubleToInt64Bits");

    public CGCachedExpr(CGExpr expr, CachedAtom cachedAtom, List<CGCachedExpr> caches) {
      this.expr = expr;
      this.cachedAtom = cachedAtom;
      this.cacheNumber = caches.Count;
      caches.Add(this);
    }

    public override void Compile() {
      CompileToDoubleOrNan();
      ilg.Emit(OpCodes.Call, NumberValue.makeMethod);
    }

    public override void CompileToDoubleOrNan() {
      if (IsCacheNeeded)
        EmitCacheAccess();
      else
        expr.CompileToDoubleOrNan();
    }

    public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      if (IsCacheNeeded)
        // Call CompileToDoubleProper via base class CompileCondition
        base.CompileCondition(ifTrue, ifFalse, ifOther);
      else
        expr.CompileCondition(ifTrue, ifFalse, ifOther);
    }

    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      if (IsCacheNeeded)
        // Call CompileToDoubleOrNan via base class CompileToDoubleProper
        base.CompileToDoubleProper(ifProper, ifOther);
      else
        expr.CompileToDoubleProper(ifProper, ifOther);
    }

    public void IncrementGenerateCount() {
```

```
      generateCount++;
    }

    // Cache only if used in PathCond and worth caching
    private bool IsCacheNeeded {
      get { return generateCount > 0 && !(expr is CGCellRef) && !(expr is CGConst); }
    }

    /// <summary>
    /// This must be called, at most once, before any use of the cache is generated.
    /// </summary>
    public void EmitCacheInitialization() {
      if (IsCacheNeeded) {
        Debug.Assert(cacheVariable == null);  // Has not been called before
        cacheVariable = ilg.DeclareLocal(typeof(double));
        // Console.WriteLine("Emitted {0} in localvar {1}", this, cacheVariable);
        ilg.Emit(OpCodes.Ldc_R8, uninitializedNan);
        ilg.Emit(OpCodes.Stloc, cacheVariable);
      }
    }

    /// <summary>
    /// Generate code for each use of the cache.
    /// </summary>
    private void EmitCacheAccess() {
      Debug.Assert(cacheVariable != null); // EmitCacheInitialization() has been called
      ilg.Emit(OpCodes.Ldloc, cacheVariable);
      ilg.Emit(OpCodes.Call, isNaNMethod);
      Label endLabel = ilg.DefineLabel();
      ilg.Emit(OpCodes.Brfalse, endLabel);  // Already computed, non-NaN result
      ilg.Emit(OpCodes.Ldloc, cacheVariable);
      ilg.Emit(OpCodes.Call, doubleToInt64BitsMethod);
      ilg.Emit(OpCodes.Conv_I4);
      ilg.Emit(OpCodes.Ldc_I4, uninitializedBits);
      ilg.Emit(OpCodes.Ceq);
      ilg.Emit(OpCodes.Brfalse, endLabel);  // Already computed, NaN result
      expr.CompileToDoubleOrNan();
      // ilg.EmitWriteLine("Filled " + this);
      ilg.Emit(OpCodes.Stloc, cacheVariable);
      ilg.MarkLabel(endLabel);
      ilg.Emit(OpCodes.Ldloc, cacheVariable);
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      // Partial evaluation may encounter a cached expression in conditionals
      // etc, and should simply partially evaluate the expression inside the cache.
      // No duplication of cached expression happens in the residual program because
      // a cached expression appears only once in the regular code, and no occurrence
      // from evaluation conditions is added to the residual program; see ComputeCell.PEval
.
      // New evaluation conditions are added later; see ProgramLines.CompileToDelegate.
      return expr.PEval(pEnv, hasDynamicControl);
    }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) {
      throw new ImpossibleException("CGCachedExpr.EvalCond");
    }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      expr.DependsOn(here, dependsOn);
    }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
      expr.CountUses(typ, numberUses);
    }

    public override string ToString() {
      return "CACHE#" + cacheNumber + "[" + expr + "]";
    }
```



```
    public override Typ Type() {
      return Typ.Number;
    }
  }

  /// <summary>
  /// A CGComposite represents an expression that may have subexpressions,
  /// such as an arithmetic operation, a call to a built-in function
  /// (including IF, AND, OR, RAND, LN, EXP, ...) or to a sheet-defined or
  /// external function.
  /// </summary>
  public abstract class CGComposite : CGExpr {
    protected readonly CGExpr[] es;

    protected CGComposite(CGExpr[] es) {
      this.es = es;
    }

    public static CGExpr Make(String name, CGExpr[] es) {
      name = name.ToUpper();
      // This switch should agree with the function table in Functions.cs
      switch (name) {
        case "+": return new CGArithmetic2(OpCodes.Add, name, es);
        case "*": return new CGArithmetic2(OpCodes.Mul, name, es);
        case "-": return new CGArithmetic2(OpCodes.Sub, name, es);
        case "/": return new CGArithmetic2(OpCodes.Div, name, es);

        case "=":  return CGEqual.Make(es);
        case "<>": return CGNotEqual.Make(es);
        case ">":  return new CGGreaterThan(es);
        case "<=": return new CGLessThanOrEqual(es);
        case "<":  return new CGLessThan(es);
        case ">=": return new CGGreaterThanOrEqual(es);

        // Non-strict, or other special treatment
        case "AND": return new CGAnd(es);
        case "APPLY": return new CGApply(es);
        case "CHOOSE": return new CGChoose(es);
        case "CLOSURE": return new CGClosure(es);
        case "ERR": return new CGError(es);
        case "EXTERN": return new CGExtern(es);
        case "IF": return new CGIf(es);
        case "NA":
          if (es.Length == 0)
            return new CGError(ErrorValue.naError);
          else
            return new CGError(ErrorValue.argCountError);
        case "NEG": return new CGNeg(es);
        case "NOT": return new CGNot(es);
        case "OR": return new CGOr(es);
        case "PI":
          if (es.Length == 0)
            return new CGNumberConst(NumberValue.PI);
          else
            return new CGError(ErrorValue.argCountError);
        case "VOLATILIZE":
          if (es.Length == 1)
            return es[0];
          else
            return new CGError(ErrorValue.argCountError);
        default:
          // The general case for most built-in functions with unspecific argument types
          FunctionInfo functionInfo;
          if (FunctionInfo.Find(name, out functionInfo))
            return new CGFunctionCall(functionInfo, es);
          else { // May be a sheet-defined function
            SdfInfo sdfInfo = SdfManager.GetInfo(name);
            if (sdfInfo != null)
              return new CGSdfCall(sdfInfo, es);
            else
```

```
            return new CGError(ErrorValue.nameError);
          }
        }
      }

      protected abstract Typ GetInputTypWithoutLengthCheck(int pos);

      public Typ GetInputTyp(int pos) {
        if (0 <= pos && pos < Arity)
          return GetInputTypWithoutLengthCheck(pos);
        else
          return Typ.Value;
      }

      public CGExpr[] PEvalArgs(PEnv pEnv, CGExpr r0, bool hasDynamicControl) {
        CGExpr[] res = new CGExpr[es.Length];
        res[0] = r0;
        for (int i = 1; i < es.Length; i++)
          res[i] = es[i].PEval(pEnv, hasDynamicControl);
        return res;
      }

      public override bool IsSerious(ref int bound) {
        bool serious = base.IsSerious(ref bound);
        for (int i = 0; !serious && i < es.Length; i++)
          serious = es[i].IsSerious(ref bound);
        return serious;
      }

      public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
        foreach (CGExpr e in es)
          e.DependsOn(here, dependsOn);
      }

      protected String FormatAsCall(String name) {
        return FunctionValue.FormatAsCall(name, es);
      }

      public abstract int Arity { get; }
    }

    /// <summary>
    /// A CGIf represents an application of the IF built-in function.
    /// </summary>
    public class CGIf : CGComposite {
      public CGIf(CGExpr[] es) : base(es) { }

      // This handles compilation of IF(e0,e1,e2) in a Value context
      public override void Compile() {
        if (es.Length != 3)
          LoadErrorValue(ErrorValue.argCountError);
        else
          es[0].CompileCondition(
            new Gen(delegate { es[1].Compile(); }),
            new Gen(delegate { es[2].Compile(); }),
            new Gen(delegate {
            ilg.Emit(OpCodes.Ldloc, testDouble);
            WrapDoubleToNumberValue();
          }));
      }

      // This handles compilation of 5 + IF(e0,e1,e2) and such
      public override void CompileToDoubleOrNan() {
        if (es.Length != 3)
          LoadErrorValue(ErrorValue.argCountError);
        else
          es[0].CompileCondition(
            new Gen(delegate { es[1].CompileToDoubleOrNan(); }),
            new Gen(delegate { es[2].CompileToDoubleOrNan(); }),
            new Gen(delegate { ilg.Emit(OpCodes.Ldloc, testDouble); }));
      }
```



```
      // This handles compilation of 5 > IF(e0,e1,e2) and such
      public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
        if (es.Length != 3) {
          SetArgCountErrorNan();
          ifOther.Generate(ilg);
        } else
          es[0].CompileCondition(
            new Gen(delegate { es[1].CompileToDoubleProper(ifProper, ifOther); }),
            new Gen(delegate { es[2].CompileToDoubleProper(ifProper, ifOther); }),
            ifOther);
      }

      // This handles compilation of IF(IF(e00,e01,e02), e1, e2) and such
      public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
        if (es.Length != 3) {
          SetArgCountErrorNan();
          ifOther.Generate(ilg);
        } else
          es[0].CompileCondition(
          new Gen(delegate { es[1].CompileCondition(ifTrue, ifFalse, ifOther); }),
          new Gen(delegate { es[2].CompileCondition(ifTrue, ifFalse, ifOther); }),
          ifOther);
      }

      public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
        CGExpr r0 = es[0].PEval(pEnv, hasDynamicControl);
        if (r0 is CGNumberConst) {
          if ((r0 as CGNumberConst).number.value != 0.0)
            return es[1].PEval(pEnv, hasDynamicControl);
          else
            return es[2].PEval(pEnv, hasDynamicControl);
        } else
          return new CGIf(PEvalArgs(pEnv, r0, true));
      }

      public override string ToString() {
        return FormatAsCall("IF");
      }

      public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                    List<CGCachedExpr> caches) {
        if (es.Length == 3) {
          CachedAtom atom = new CachedAtom(es[0], caches);
          es[0].EvalCond(evalCond, evalConds, caches);
          es[0] = atom.cachedExpr;
          es[1].EvalCond(evalCond.And(atom), evalConds, caches);
          es[2].EvalCond(evalCond.AndNot(atom), evalConds, caches);
        }
      }

      public override void NoteTailPosition() {
        if (es.Length == 3) {
          es[1].NoteTailPosition();
          es[2].NoteTailPosition();
        }
      }

      public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
        if (es.Length == 3) {
          es[0].CountUses(Typ.Number, numberUses);
          es[1].CountUses(typ, numberUses);
          es[2].CountUses(typ, numberUses);
        }
      }

      public override Typ Type() {
        if (es.Length != 3)
          return Typ.Error;
        else
```

```
      return Lub(es[1].Type(), es[2].Type());
  }

  protected override Typ GetInputTypWithoutLengthCheck(int pos) {
    if (pos == 0)
      return Typ.Number;
    else
      return Typ.Value;
  }

  public override int Arity { get { return 3; } }
}

/// <summary>
/// A CGChoose represents an application of the CHOOSE built-in function.
/// </summary>
public class CGChoose : CGComposite {
  public CGChoose(CGExpr[] es) : base(es) { }

  public override void Compile() {
    es[0].CompileToDoubleProper(new Gen(delegate {
      Label endLabel = ilg.DefineLabel();
      Label[] labels = new Label[es.Length - 1];
      for (int i = 1; i < es.Length; i++)
        labels[i - 1] = ilg.DefineLabel();
      ilg.Emit(OpCodes.Conv_I4);
      ilg.Emit(OpCodes.Ldc_I4, 1);
      ilg.Emit(OpCodes.Sub);
      ilg.Emit(OpCodes.Switch, labels);
      LoadErrorValue(ErrorValue.valueError);
      for (int i = 1; i < es.Length; i++) {
        ilg.Emit(OpCodes.Br, endLabel);
        ilg.MarkLabel(labels[i - 1]);
        es[i].Compile();
      }
      ilg.MarkLabel(endLabel);
    }),
    GenLoadErrorValue(ErrorValue.argTypeError)
    );
  }

  public override void CompileToDoubleOrNan() {
    es[0].CompileToDoubleProper(new Gen(delegate {
      Label endLabel = ilg.DefineLabel();
      Label[] labels = new Label[es.Length - 1];
      for (int i = 1; i < es.Length; i++)
        labels[i - 1] = ilg.DefineLabel();
      ilg.Emit(OpCodes.Conv_I4);
      ilg.Emit(OpCodes.Ldc_I4, 1);
      ilg.Emit(OpCodes.Sub);
      ilg.Emit(OpCodes.Switch, labels);
      LoadErrorNan(ErrorValue.valueError);
      for (int i = 1; i < es.Length; i++) {
        ilg.Emit(OpCodes.Br, endLabel);
        ilg.MarkLabel(labels[i - 1]);
        es[i].CompileToDoubleOrNan();
      }
      ilg.MarkLabel(endLabel);
    }),
    GenLoadErrorNan(ErrorValue.argTypeError)
    );
  }

  public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
    CGExpr r0 = es[0].PEval(pEnv, hasDynamicControl);
    if (r0 is CGNumberConst) {
      int index = (int)((r0 as CGNumberConst).number.value);
      if (index < 1 || index >= es.Length)
        return new CGError(ErrorValue.valueError);
      else
        return es[index].PEval(pEnv, hasDynamicControl);
```



```
      } else
        return new CGChoose(PEvalArgs(pEnv, r0, true /* has dynamic control */));
    }

    public override string ToString() {
      return FormatAsCall("CHOOSE");
    }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) {
      if (es.Length >= 1) {
        CachedAtom atom = new CachedAtom(es[0], caches);
        CGCachedExpr cached = atom.cachedExpr;
        es[0].EvalCond(evalCond, evalConds, caches);
        es[0] = cached;
        for (int i = 1; i < es.Length; i++) {
          CGExpr iConst = CGConst.Make(i);
          CGExpr cond = new CGEqual(new CGExpr[] { cached, iConst });
          es[i].EvalCond(evalCond.And(new CachedAtom(cond, caches)), evalConds, caches);
        }
      }
    }

    public override void NoteTailPosition() {
      for (int i = 1; i < es.Length; i++)
        es[i].NoteTailPosition();
    }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
      es[0].CountUses(Typ.Number, numberUses);
      for (int i = 1; i < es.Length; i++)
        es[i].CountUses(typ, numberUses);
    }

    public override Typ Type() {
      Typ result = Typ.Error;
      for (int i = 1; i < es.Length; i++)
        result = Lub(result, es[i].Type());
      return result;
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      if (pos == 0)
        return Typ.Number;
      else
        return Typ.Value;
    }

    public override int Arity { get { return es.Length; } }
  }

  /// <summary>
  /// A CGAnd represents a variadic AND operation, strict only
  /// in its first argument, although that's not Excel semantics.
  /// </summary>
  public class CGAnd : CGComposite {
    public CGAnd(CGExpr[] es) : base(es) { }

    public override void Compile() {
      CompileCondition(
        new Gen(delegate { ilg.Emit(OpCodes.Ldsfld, NumberValue.oneField); }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldsfld, NumberValue.zeroField); }),
        GenLoadTestDoubleErrorValue()
        );
    }

    public override void CompileToDoubleOrNan() {
      CompileCondition(
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 1.0); }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 0.0); }),
```

```
        new Gen(delegate { ilg.Emit(OpCodes.Ldloc, testDouble); }));
    }

    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      CompileCondition(
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 1.0); ifProper.Generate(ilg); }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 0.0); ifProper.Generate(ilg); }),
        ifOther);
    }

    public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      for (int i = es.Length - 1; i >= 0; i--) {
        // These declarations are needed to capture rvalues rather than lvalues:
        CGExpr ei = es[i];
        Gen localIfTrue = ifTrue;
        ifTrue = new Gen(delegate { ei.CompileCondition(localIfTrue, ifFalse, ifOther); });
      }
      ifTrue.Generate(ilg);
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      List<CGExpr> res = new List<CGExpr>();
      for (int i = 0; i < es.Length; i++) {
        CGExpr ri = es[i].PEval(pEnv, hasDynamicControl || res.Count > 0);
        if (ri is CGNumberConst) {
          // A FALSE operand makes the AND false; a TRUE operand can be ignored
          if ((ri as CGNumberConst).number.value == 0.0)
            return new CGNumberConst(NumberValue.ZERO);
        } else
          res.Add(ri);
      }
      // The residual AND consists of the non-constant operands, if any
      if (res.Count == 0)
        return new CGNumberConst(NumberValue.ONE);
      else
        return new CGAnd(res.ToArray());
    }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) {
      for (int i = 0; i < es.Length; i++) {
        es[i].EvalCond(evalCond, evalConds, caches);
        if (SHORTCIRCUIT_EVALCONDS && i != es.Length - 1) {
          // Take short-circuit evaluation into account for precision
          CachedAtom atom = new CachedAtom(es[i], caches);
          evalCond = evalCond.And(atom);
          es[i] = atom.cachedExpr;
        }
      }
    }

    public override string ToString() {
      return FormatAsCall("AND");
    }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
      foreach (CGExpr e in es)
        e.CountUses(Typ.Number, numberUses);
    }

    public override Typ Type() {
      return Typ.Number;
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      return Typ.Number;
    }

    public override int Arity { get { return es.Length; } }
  }
```



```
  /// <summary>
  /// A CGOr represents a variadic OR operation, strict only
  /// in its first argument, although that's not Excel semantics.
  /// </summary>
  public class CGOr : CGComposite {
    public CGOr(CGExpr[] es) : base(es) { }

    public override void Compile() {
      CompileCondition(
        new Gen(delegate { ilg.Emit(OpCodes.Ldsfld, NumberValue.oneField); }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldsfld, NumberValue.zeroField); }),
        GenLoadTestDoubleErrorValue()
        );
    }

    public override void CompileToDoubleOrNan() {
      CompileCondition(
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 1.0); }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 0.0); }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldloc, testDouble); }));
    }

    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      CompileCondition(
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 1.0); ifProper.Generate(ilg); }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldc_R8, 0.0); ifProper.Generate(ilg); }),
        ifOther);
    }

    public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      for (int i = es.Length - 1; i >= 0; i--) {
        // These declarations are needed to capture rvalues rather than lvalues:
        CGExpr ei = es[i];
        Gen localIfFalse = ifFalse;
        ifFalse = new Gen(delegate { ei.CompileCondition(ifTrue, localIfFalse, ifOther); })
;
      }
      ifFalse.Generate(ilg);
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      List<CGExpr> res = new List<CGExpr>();
      for (int i = 0; i < es.Length; i++) {
        CGExpr ri = es[i].PEval(pEnv, hasDynamicControl || res.Count > 0);
        if (ri is CGNumberConst) {
          // A TRUE operand makes the OR true; a FALSE operand can be ignored
          if ((ri as CGNumberConst).number.value != 0.0)
            return new CGNumberConst(NumberValue.ONE);
        } else
          res.Add(ri);
      }
      // The residual OR consists of the non-constant operands, if any
      if (res.Count == 0)
        return new CGNumberConst(NumberValue.ZERO);
      else
        return new CGOr(res.ToArray());
    }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) {
      for (int i = 0; i < es.Length; i++) {
        es[i].EvalCond(evalCond, evalConds, caches);
        if (SHORTCIRCUIT_EVALCONDS && i != es.Length - 1) {
          // Take short-circuit evaluation into account for precision
          CachedAtom atom = new CachedAtom(es[i], caches);
          evalCond = evalCond.AndNot(atom);
          es[i] = atom.cachedExpr;
        }
      }
    }
```

```
    }

    public override string ToString() {
      return FormatAsCall("OR");
    }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
      foreach (CGExpr e in es)
        e.CountUses(Typ.Number, numberUses);
    }

    public override Typ Type() {
      return Typ.Number;
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      return Typ.Number;
    }

    public override int Arity { get { return es.Length; } }
  }

  /// <summary>
  /// A CGStrictOperation is a composite expression that evaluates all
  /// its subexpressions.
  /// </summary>
  public abstract class CGStrictOperation : CGComposite {
    // For partial evaluation; null for CGApply, CGExtern, CGSdfCall:
    public readonly Applier applier;

    public CGStrictOperation(CGExpr[] es, Applier applier)
      : base(es) {
      this.applier = applier;
    }

    public CGExpr[] PEvalArgs(PEnv pEnv, bool hasDynamicControl) {
      CGExpr[] res = new CGExpr[es.Length];
      for (int i = 0; i < es.Length; i++)
        res[i] = es[i].PEval(pEnv, hasDynamicControl);
      return res;
    }

    public bool AllConstant(CGExpr[] res) {
      for (int i = 0; i < res.Length; i++)
        if (!(res[i] is CGConst))
          return false;
      return true;
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      CGExpr[] res = PEvalArgs(pEnv, hasDynamicControl);
      // If all args are constant then evaluate else residualize:
      if (AllConstant(res)) {
        Expr[] es = new Expr[res.Length];
        for (int i = 0; i < res.Length; i++)
          es[i] = Const.Make((res[i] as CGConst).Value);
        // Use the interpretive implementation's applier on a fake sheet
        // and fake cell coordinates, but constant argument expressions:
        return CGConst.Make(applier(null, es, -1, -1));
      } else
        return Residualize(res);
    }

    public abstract CGExpr Residualize(CGExpr[] res);

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) {
      for (int i = 0; i < es.Length; i++)
        es[i].EvalCond(evalCond, evalConds, caches);
    }
```



```
    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
      for (int i = 0; i < es.Length; i++)
        es[i].CountUses(GetInputTyp(i), numberUses);
    }
  }

  /// <summary>
  /// A CGFunctionCall is a call to a fixed-arity or variable-arity
  /// strict built-in function.
  /// </summary>
  public class CGFunctionCall : CGStrictOperation {
    protected readonly FunctionInfo functionInfo;

    public CGFunctionCall(FunctionInfo functionInfo, CGExpr[] es)
      : base(es, functionInfo.applier) {
      this.functionInfo = functionInfo;
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      // Volatile functions must be residualized
      if (functionInfo.name == "NOW" || functionInfo.name == "RAND")
        return Residualize(PEvalArgs(pEnv, hasDynamicControl));
      else
        return base.PEval(pEnv, hasDynamicControl);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return new CGFunctionCall(functionInfo, res);
    }

    public override void Compile() {
      Gen success =
          new Gen(delegate {
            ilg.Emit(OpCodes.Call, functionInfo.methodInfo);
            if (functionInfo.signature.retType == Typ.Number)
              WrapDoubleToNumberValue();
          });
      if (Arity < 0) { // Variable arity
        CompileToValueArray(es.Length, 0, es);
        success.Generate(ilg);
      } else if (es.Length != Arity)
        LoadErrorValue(ErrorValue.argCountError);
      else {
        // TODO: ifOther should probably load error from testValue instead?
        Gen ifOther = GenLoadErrorValue(ErrorValue.argTypeError);
        CompileArgumentsAndApply(es, success, ifOther);
      }
    }

    public override void CompileToDoubleOrNan() {
      Gen success =
          new Gen(delegate {
            ilg.Emit(OpCodes.Call, functionInfo.methodInfo);
            if (functionInfo.signature.retType != Typ.Number)
              UnwrapToDoubleOrNan();
          });
      if (Arity < 0) { // Variable arity
        CompileToValueArray(es.Length, 0, es);
        success.Generate(ilg);
      } else if (es.Length != Arity)
        LoadErrorNan(ErrorValue.argCountError);
      else {
        // TODO: ifOther should probably load error from testValue instead?
        Gen ifOther = GenLoadErrorNan(ErrorValue.argTypeError);
        CompileArgumentsAndApply(es, success, ifOther);
      }
    }

    // Special case for the argumentless and always-proper
    // functions RAND and NOW, often used in conditions
```

```
    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      if (es.Length == 0 && (functionInfo.name == "RAND" || functionInfo.name == "NOW"))
{
        ilg.Emit(OpCodes.Call, functionInfo.methodInfo);
        ifProper.Generate(ilg);
      } else
        base.CompileToDoubleProper(ifProper, ifOther);
    }

    // Generate code to evaluate all argument expressions, including the receiver es[1]
    // if the method is an instance method, and convert their values to .NET types.

    private void CompileArgumentsAndApply(CGExpr[] es, Gen ifSuccess, Gen ifOther) {
      int argCount = es.Length;
      // The error continuations must pop the arguments computed so far.
      Gen[] errorCont = new Gen[argCount];
      if (argCount > 0)
        errorCont[0] = ifOther;
      for (int i = 1; i < argCount; i++) {
        int ii = i; // Capture lvalue -- do NOT inline!
        errorCont[ii] = new Gen(delegate {
          ilg.Emit(OpCodes.Pop);
          errorCont[ii - 1].Generate(ilg);
        });
      }
      // Generate code, backwards, to evaluate argument expressions and
      // convert to the .NET method's argument types
      for (int i = argCount - 1; i >= 0; i--) {
        // These local vars capture rvalue rather than lvalue -- do NOT inline them!
        CGExpr ei = es[i];
        Gen localSuccess = ifSuccess;
        Typ argType = functionInfo.signature.argTypes[i];
        Gen ifError = errorCont[i];
        if (argType == Typ.Number)
          ifSuccess = new Gen(delegate {
               ei.CompileToDoubleOrNan();
               localSuccess.Generate(ilg);
             });
        else if (argType == Typ.Function)
          ifSuccess = new Gen(delegate {
               ei.Compile();
               CheckType(FunctionValue.type, localSuccess, ifError);
             });
        else if (argType == Typ.Array)
          ifSuccess = new Gen(delegate {
               ei.Compile();
               CheckType(ArrayValue.type, localSuccess, ifError);
             });
        else if (argType == Typ.Text)
          ifSuccess = new Gen(delegate {
              ei.Compile();
              CheckType(TextValue.type, localSuccess, ifError);
            });
        else // argType.Value -- TODO: neglects to propagate ErrorValue from argument
          ifSuccess = new Gen(delegate {
              ei.Compile();
              localSuccess.Generate(ilg);
            });
      }
      ifSuccess.Generate(ilg);
    }

    public override bool IsSerious(ref int bound) {
      return functionInfo.isSerious || base.IsSerious(ref bound);
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      return Arity < 0 ? Typ.Value : functionInfo.signature.argTypes[pos];
    }

    public override Typ Type() {
```



```
      return functionInfo.signature.retType;
    }

    public override int Arity {
      get { return functionInfo.signature.Arity; }
    }

    public override string ToString() {
      return FormatAsCall(functionInfo.name);
    }
  }

  /// <summary>
  /// A CGSdfCall is a call to a sheet-defined function.
  /// </summary>
  public class CGSdfCall : CGStrictOperation {
    private readonly SdfInfo sdfInfo;
    private bool isInTailPosition = false;

    public CGSdfCall(SdfInfo sdfInfo, CGExpr[] es)
      : base(es, null) {
      this.sdfInfo = sdfInfo;
    }

    public override void Compile() {
      if (es.Length != sdfInfo.arity)
        LoadErrorValue(ErrorValue.argCountError);
      else {
        ilg.Emit(OpCodes.Ldsfld, SdfManager.sdfDelegatesField);
        ilg.Emit(OpCodes.Ldc_I4, sdfInfo.index);
        ilg.Emit(OpCodes.Ldelem_Ref);
        ilg.Emit(OpCodes.Castclass, sdfInfo.MyType);
        for (int i = 0; i < es.Length; i++)
          es[i].Compile();
        if (isInTailPosition)
          ilg.Emit(OpCodes.Tailcall);
        ilg.Emit(OpCodes.Call, sdfInfo.MyInvoke);
        if (isInTailPosition)
          ilg.Emit(OpCodes.Ret);
      }
    }

    public override void CompileToDoubleOrNan() {
      Compile();
      UnwrapToDoubleOrNan();
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      CGExpr[] res = PEvalArgs(pEnv, hasDynamicControl);
      // The static argument positions are those that have args[i] != naError
      Value[] args = new Value[res.Length];
      bool anyStatic = false, allStatic = true;
      for (int i = 0; i < res.Length; i++)
        if (res[i] is CGConst) {
          args[i] = (res[i] as CGConst).Value;
          anyStatic = true;
        } else {
          args[i] = ErrorValue.naError; // Generalize to dynamic
          allStatic = false;
        }
      if (!hasDynamicControl) {
        // This would be wrong, because the called function might contain
        // volatile functions and in that case should residualize:
        // if (allStatic)       // If all arguments static, just call the SDF
        //  return CGConst.Make(sdfInfo.Apply(args));
        // else
        if (anyStatic)  // Specialize if there are static arguments
          return Specialize(res, args);
        // Do nothing if all arguments are dynamic
      } else {
        // If under dynamic control reduce to longest static prefix
```

```
        // where the argument values agree.
        // TODO: This is wrong -- should always specialize when the call is not
        // recursive
        ICollection<Value[]> pending = SdfManager.PendingSpecializations(sdfInfo.name);
        Value[] maxArray = null;
        int maxCount = 0;
        foreach (Value[] vs in pending) {
          int agree = AgreeCount(vs, args);
          if (agree > maxCount) {
            maxCount = agree;
            maxArray = vs;
          }
        }
        if (maxCount > 0) {
          SetNaInArgs(maxArray, args);
          return Specialize(res, args);
        }
      }
      return new CGSdfCall(sdfInfo, res);
    }

    private static int AgreeCount(Value[] xs, Value[] ys) {
      Debug.Assert(xs.Length == ys.Length);
      int count = 0;
      for (int i = 0; i < xs.Length; i++)
        if (xs[i] != ErrorValue.naError && ys[i] != ErrorValue.naError && xs[i].Equals(ys[i
]))
          count++;
      return count;
    }

    private static void SetNaInArgs(Value[] xs, Value[] args) {
      Debug.Assert(xs.Length == args.Length);
      for (int i = 0; i < xs.Length; i++)
        if (!xs[i].Equals(args[i]))
          args[i] = ErrorValue.naError; // Generalize to dynamic
    }

    private CGSdfCall Specialize(CGExpr[] res, Value[] args) {
      FunctionValue fv = new FunctionValue(sdfInfo, args);
      SdfInfo residualSdf = SdfManager.SpecializeAndCompile(fv);
      CGExpr[] residualArgs = new CGExpr[fv.Arity];
      int j = 0;
      for (int i = 0; i < args.Length; i++)
        if (args[i] == ErrorValue.naError)
          residualArgs[j++] = res[i];
      return new CGSdfCall(residualSdf, residualArgs);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      throw new ImpossibleException("CGSdfCall.Residualize");
    }

    public override bool IsSerious(ref int bound) {
      return true;
    }

    public override string ToString() {
      return FormatAsCall(sdfInfo.name);
    }

    public override void NoteTailPosition() {
      isInTailPosition = true;
    }

    public override Typ Type() {
      return Typ.Value;
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      return Typ.Value;
```



```
    }

    public override int Arity { get { return sdfInfo.arity; } }
  }

  /// <summary>
  /// A CGApply is a call to the APPLY function; its first argument must
  /// evaluate to a function value (closure).
  /// </summary>
  class CGApply : CGStrictOperation {
    public CGApply(CGExpr[] es) : base(es, null) { }

    public override void Compile() {
      if (es.Length < 1)
        LoadErrorValue(ErrorValue.argCountError);
      else {
        es[0].Compile();
        CheckType(FunctionValue.type,
          new Gen(delegate {
            int arity = es.Length - 1;
            // Don't check arity here; it is done elsewhere
            // Compute and push additional arguments
            for (int i = 1; i < es.Length; i++)
              es[i].Compile();
            // Call the appropriate CallN method on the FunctionValue
            ilg.Emit(OpCodes.Call, FunctionValue.callMethods[arity]);
          }),
        GenLoadErrorValue(ErrorValue.argTypeError));
      }
    }

    public override void CompileToDoubleOrNan() {
      Compile();
      UnwrapToDoubleOrNan();
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      // When FunctionValue is known, reduce to a CGSdfCall node.
      // Don't actually call the function (even on constant arguments); could loop.
      CGExpr[] res = PEvalArgs(pEnv, hasDynamicControl);
      if (res[0] is CGValueConst) {
        FunctionValue fv = (res[0] as CGValueConst).Value as FunctionValue;
        if (fv != null) {
          CGExpr[] args = new CGExpr[fv.args.Length];
          int j = 1;
          for (int i = 0; i < args.Length; i++)
            if (fv.args[i] != ErrorValue.naError)
              args[i] = CGConst.Make(fv.args[i]);
            else
              args[i] = res[j++];
          return new CGSdfCall(fv.sdfInfo, args);
        } else
          return new CGError(ErrorValue.argCountError);
      } else
        return new CGApply(res);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      throw new ImpossibleException("CGApply.Residualize");
    }

    public override bool IsSerious(ref int bound) {
      return true;
    }

    public override string ToString() {
      return FormatAsCall("APPLY");
    }

    public override void NoteTailPosition() {
      // We currently do not try to optimize tail calls in APPLY
```

```
      // isInTailPosition = true;
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      switch (pos) {
        // Could infer the expected argument types for a more precise function type:
        case 0: return Typ.Function;
        default: return Typ.Value;
      }
    }

    public override int Arity { get { return es.Length; } }

    public override Typ Type() {
      // An SDF in general returns a Value
      return Typ.Value;
    }
  }

  /// <summary>
  /// A CGExtern is a call to EXTERN, ie. an external .NET function.
  /// </summary>
  class CGExtern : CGStrictOperation {
    // If ef==null then errorValue is set to the error that occurred
    // during lookup, and the other fields are invalid
    private readonly ExternalFunction ef;
    private readonly ErrorValue errorValue;
    private readonly Typ resType;
    private readonly Typ[] argTypes;

    private static readonly ISet<Type>
      signed32 = new HashSet<Type>(),
      unsigned32 = new HashSet<Type>(),
      numeric = new HashSet<Type>();

    static CGExtern() {
      signed32.Add(typeof(System.Int32));
      signed32.Add(typeof(System.Int16));
      signed32.Add(typeof(System.SByte));
      unsigned32.Add(typeof(System.UInt32));
      unsigned32.Add(typeof(System.UInt16));
      unsigned32.Add(typeof(System.Byte));
      numeric.Add(typeof(System.Double));
      numeric.Add(typeof(System.Single));
      numeric.Add(typeof(System.Int64));
      numeric.Add(typeof(System.UInt64));
      numeric.Add(typeof(System.Boolean));
      numeric.UnionWith(signed32);
      numeric.UnionWith(unsigned32);
    }

    public CGExtern(CGExpr[] es)
      : base(es, null) {
      if (es.Length < 1)
        errorValue = ErrorValue.argCountError;
      else {
        CGTextConst nameAndSignatureConst = es[0] as CGTextConst;
        if (nameAndSignatureConst == null)
          errorValue = ErrorValue.argTypeError;
        else {
          try {
            // This retrieves the method from cache, or creates it:
            ef = ExternalFunction.Make(nameAndSignatureConst.value.value);
            if (ef.arity != es.Length - 1) {
              ef = null;
              errorValue = ErrorValue.argCountError;
            } else {
              resType = FromType(ef.ResType);
              argTypes = new Typ[ef.arity];
              for (int i = 0; i < argTypes.Length; i++)
                argTypes[i] = FromType(ef.ArgType(i));
```



```
            }
          } catch (Exception exn)  // Covers a multitude of sins
            {
              errorValue = ErrorValue.Make(exn.Message);
            }
        }
      }
    }

    private static Typ FromType(Type t) {
      if (numeric.Contains(t))
        return Typ.Number;
      else if (t == typeof(System.String))
        return Typ.Text;
      else
        return Typ.Value;
    }

    public override void Compile() {
      if (ef == null)
        LoadErrorValue(errorValue);
      else {
        // If argument evaluation is successful, call the external function
        // and convert its result to Value; if unsuccessful, return ArgTypeError
        Gen success;
        // First some return type special cases, avoid boxing:
        if (ef.ResType == typeof(System.Double))
          success = new Gen(delegate {
            ef.EmitCall(ilg);
            ilg.Emit(OpCodes.Call, NumberValue.makeMethod);
          });
        else if (numeric.Contains(ef.ResType))
          success = new Gen(delegate {
            ef.EmitCall(ilg);
            ilg.Emit(OpCodes.Conv_R8);
            ilg.Emit(OpCodes.Call, NumberValue.makeMethod);
          });
        else if (ef.ResType == typeof(char))
          success = new Gen(delegate {
            ef.EmitCall(ilg);
            ilg.Emit(OpCodes.Call, TextValue.fromNakedCharMethod);
          });
        else if (ef.ResType == typeof(void))
          success = new Gen(delegate {
              ef.EmitCall(ilg);
              ilg.Emit(OpCodes.Ldsfld, TextValue.voidField);
          });
        else
          success = new Gen(delegate {
            ef.EmitCall(ilg);
            if (ef.ResType.IsValueType)
              ilg.Emit(OpCodes.Box, ef.ResType);
            ilg.Emit(OpCodes.Call, ef.ResConverter.Method);
          });
        Gen ifOther = GenLoadErrorValue(ErrorValue.argTypeError);
        CompileArgumentsAndApply(es, success, ifOther);
      }
    }

    public override void CompileToDoubleOrNan() {
      if (ef == null)
        ilg.Emit(OpCodes.Ldc_R8, errorValue.ErrorNan);
      else {
        // sestoft: This is maybe correct
        Gen ifOther = GenLoadErrorNan(ErrorValue.argTypeError);
        if (ef.ResType == typeof(System.Double)) {
          // If argument evaluation is successful, call external function
          // and continue with ifDouble; otherwise continue with ifOther
          Gen success = new Gen(delegate { ef.EmitCall(ilg); });
          CompileArgumentsAndApply(es, success, ifOther);
        } else if (numeric.Contains(ef.ResType)) {
```

```
        // If argument evaluation is successful, call external function, convert
        // to float64
        Gen success =
          new Gen(delegate {
            ef.EmitCall(ilg);
            ilg.Emit(OpCodes.Conv_R8);
          });
        CompileArgumentsAndApply(es, success, ifOther);
      } else // Result type cannot be converted to a float64
        ifOther.Generate(ilg);
    }
  }

  // Generate code to evaluate all argument expressions, including the receiver es[1]
  // if the method is an instance method, and convert their values to .NET types.

  private void CompileArgumentsAndApply(CGExpr[] es, Gen ifSuccess, Gen ifOther) {
    int argCount = es.Length - 1;
    // The error continuations must pop the arguments computed so far:
    Gen[] errorCont = new Gen[argCount];
    if (argCount > 0)
      errorCont[0] = ifOther;
    for (int i = 1; i < argCount; i++) {
      int ii = i; // Capture lvalue -- do NOT inline!
      errorCont[ii] = new Gen(delegate {
        ilg.Emit(OpCodes.Pop);
        errorCont[ii - 1].Generate(ilg);
      });
    }
    // Generate code, backwards, to evaluate argument expressions and
    // convert to external method's argument types
    for (int i = argCount - 1; i >= 0; i--) {
      // These local vars capture rvalue rather than lvalue -- do NOT inline them!
      CGExpr ei = es[i + 1];
      Gen localSuccess = ifSuccess;
      int argIndex = i;
      Type argType = ef.ArgType(i);
      Gen ifError = errorCont[i];
      // First some special cases to avoid boxing:
      if (argType == typeof(System.Double))
        ifSuccess = new Gen(
           delegate {
             ei.CompileToDoubleOrNan();
             localSuccess.Generate(ilg);
           });
      else if (argType == typeof(System.Single))
        ifSuccess = new Gen(
           delegate {
             ei.CompileToDoubleOrNan();
             ilg.Emit(OpCodes.Conv_R4);
             localSuccess.Generate(ilg);
           });
      else if (signed32.Contains(argType))
        ifSuccess = new Gen(
           delegate {
             ei.CompileToDoubleProper(
               new Gen(delegate {
               ilg.Emit(OpCodes.Conv_I4);
               localSuccess.Generate(ilg);
             }),
               ifError);
           });
      else if (unsigned32.Contains(argType))
        ifSuccess = new Gen(
           delegate {
             ei.CompileToDoubleProper(
               new Gen(delegate {
               ilg.Emit(OpCodes.Conv_U4);
               localSuccess.Generate(ilg);
             }),
               ifError);
```



```
           });
      else if (argType == typeof(System.Int64))
        ifSuccess = new Gen(
           delegate {
             ei.CompileToDoubleProper(
               new Gen(delegate {
               ilg.Emit(OpCodes.Conv_I8);
               localSuccess.Generate(ilg);
             }),
               ifError);
           });
      else if (argType == typeof(System.UInt64))
        ifSuccess = new Gen(
           delegate {
             ei.CompileToDoubleProper(
               new Gen(delegate {
                ilg.Emit(OpCodes.Conv_U8);
                localSuccess.Generate(ilg);
             }),
               ifError);
           });
      else if (argType == typeof(System.Boolean))
        ifSuccess = new Gen(
            delegate {
              ei.CompileToDoubleProper(
                 new Gen(delegate {
                  ilg.Emit(OpCodes.Ldc_R8, 0.0);
                  ilg.Emit(OpCodes.Ceq);
                  localSuccess.Generate(ilg);
              }),
                 ifError);
            });
      else if (argType == typeof(System.Char))
        ifSuccess = new Gen(
            delegate {
              ei.Compile();
              ilg.Emit(OpCodes.Call, TextValue.toNakedCharMethod);
              localSuccess.Generate(ilg);
            });
      else if (argType == typeof(System.String))
        ifSuccess = new Gen(
          delegate {
            ei.Compile();
            UnwrapToString(localSuccess, ifError);
          });
      else // General cases: String[], double[], double[,], ...
        ifSuccess = new Gen(
          delegate {
            ei.Compile();
            ilg.Emit(OpCodes.Call, ef.ArgConverter(argIndex).Method);
            if (argType.IsValueType)  // must unbox wrapped value type, but this is too s
imple-minded
              ilg.Emit(OpCodes.Unbox, argType);
            localSuccess.Generate(ilg);
          });
    }
    ifSuccess.Generate(ilg);
  }

  public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
    // Always residualize; external function could have effects or be volatile
    return new CGExtern(PEvalArgs(pEnv, hasDynamicControl));
  }

  public override CGExpr Residualize(CGExpr[] res) {
    throw new ImpossibleException("CGExtern.Residualize");
  }

  public override bool IsSerious(ref int bound) {
    return true;
  }
```

```
    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      if (ef == null)
        return Typ.Value;
      switch (pos) {
        case 0:
          return Typ.Text;
        default:
          return argTypes[pos - 1];
      }
    }

    public override int Arity { get { return es.Length; } }

    public override Typ Type() {
      if (ef == null)
        return Typ.Value;
      else
        return resType;  // From external function signature
    }
  }

  /// <summary>
  /// A CGNormalCellArea is a reference to a single cell on a normal data sheet.
  /// </summary>
  public class CGNormalCellArea : CGExpr {
    private readonly int index;  // If negative, array reference is illegal

    private static readonly MethodInfo getArrayViewMethod
      = typeof(CGNormalCellArea).GetMethod("GetArrayView");
    private static readonly ValueTable<ArrayView> valueTable
      = new ValueTable<ArrayView>();

    public CGNormalCellArea(ArrayView array) {
      if (array.sheet.IsFunctionSheet)
        this.index = -1;              // Illegal cell area
      else
        this.index = valueTable.GetIndex(array);
    }

    public override void Compile() {
      if (index >= 0) {
        ilg.Emit(OpCodes.Ldc_I4, index);
        ilg.Emit(OpCodes.Call, getArrayViewMethod);
      } else
        LoadErrorValue(ErrorValue.Make("#FUNERR: Range on function sheet"));
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      return this;
    }

    public override void CompileToDoubleOrNan() {
      GenLoadErrorNan(ErrorValue.argTypeError);
    }

    // Used only (through reflection and) from generated code
    public static ArrayView GetArrayView(int number) {
      return valueTable[number];
    }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) { }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      // We do not track dependencies on cells or areas on ordinary sheets
    }

    public override Typ Type() {
      return Typ.Array;
```



```
    }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) { }
  }

  /// <summary>
  /// A CGArithmetic1 is an application of a one-argument numeric-valued
  /// built-in function.
  /// </summary>
  abstract public class CGArithmetic1 : CGStrictOperation {
    public CGArithmetic1(CGExpr[] es, Applier applier) : base(es, applier) { }

    public override void Compile() {
      CompileToDoubleOrNan();
      WrapDoubleToNumberValue();
    }

    public override Typ Type() { return Typ.Number; }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      return Typ.Number;
    }

    public override int Arity { get { return 1; } }
  }

  /// <summary>
  /// A CGNot is an application of the NOT built-in function.
  /// </summary>
  public class CGNot : CGArithmetic1 {
    private static readonly Applier notApplier = Function.Get("NOT").Applier;

    public CGNot(CGExpr[] es) : base(es, notApplier) { }

    public override void CompileToDoubleOrNan() {
      // sestoft: Seems a bit redundant?
      CompileToDoubleProper(
        new Gen(delegate { }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldloc, testDouble); }));
    }

    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      if (es.Length != Arity) {
        SetArgCountErrorNan();
        ifOther.Generate(ilg);
      } else
        es[0].CompileToDoubleProper(
            new Gen(delegate {
              ilg.Emit(OpCodes.Ldc_R8, 0.0);
              ilg.Emit(OpCodes.Ceq);
              ilg.Emit(OpCodes.Conv_R8);
              ifProper.Generate(ilg);
            }),
            ifOther);
    }

    public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      es[0].CompileCondition(ifFalse, ifTrue, ifOther);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return new CGNot(res);
    }

    public override string ToString() {
      return FormatAsCall("NOT");
    }
  }

  /// <summary>
  /// A CGNeg is an application of the numeric negation operation.
```

```
  /// </summary>
  public class CGNeg : CGArithmetic1 {
    private static readonly Applier negApplier = Function.Get("NEG").Applier;

    public CGNeg(CGExpr[] es) : base(es, negApplier) { }

    public override void CompileToDoubleOrNan() {
      es[0].CompileToDoubleOrNan();
      ilg.Emit(OpCodes.Neg);
    }

    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      if (es.Length != Arity) {
        SetArgCountErrorNan();
        ifOther.Generate(ilg);
      } else
        es[0].CompileToDoubleProper(
            new Gen(delegate {
              ilg.Emit(OpCodes.Neg);
              ifProper.Generate(ilg);
            }),
            ifOther);
    }

    // -x is true/false/infinite/NaN if and only if x is:
    public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      es[0].CompileCondition(ifTrue, ifFalse, ifOther);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return new CGNeg(res);
    }

    public override string ToString() {
      return "-" + es[0];
    }
  }

  /// <summary>
  /// A CGArithmetic2 is an application of a two-argument numeric-valued
  /// built-in operator or function.
  /// </summary>
  public class CGArithmetic2 : CGStrictOperation {
    public readonly OpCode opCode;
    public readonly String op;

    public CGArithmetic2(OpCode opCode, String op, CGExpr[] es)
      : base(es, Function.Get(op).Applier) {
      this.opCode = opCode;
      this.op = op;
    }

    // Reductions such as 0*e==>0 are a bit dubious when you
    // consider that e could evaluate to ArgType error or similar:
    public CGExpr Make(CGExpr[] es) {
      if (es.Length == 2) {
        if (op == "+" && es[0].Is(0))
          return es[1]; // 0+e = e
        else if ((op == "+" || op == "-") && es[1].Is(0))
          return es[0]; // e+0 = e-0 = e
        else if (op == "-" && es[0].Is(0))
          return new CGNeg(new CGExpr[] { es[1] }); // 0-e = -e
        else if (op == "*" && (es[0].Is(0) || es[1].Is(0)))
          return new CGNumberConst(NumberValue.ZERO); // 0*e = e*0 = 0 (**)
        else if (op == "*" && es[0].Is(1))
          return es[1]; // 1*e = e
        else if ((op == "*" || op == "/") && es[1].Is(1))
          return es[0]; // e*1 = e/1 = e
        else if (op == "^" && (es[0].Is(1) || es[1].Is(1)))
          return es[0]; // e^1 = e and also 1^e = 1 (IEEE)
        else if (op == "^" && es[1].Is(0))
```



```
          return new CGNumberConst(NumberValue.ONE); // e^0 = 1 (IEEE)
      }
      return new CGArithmetic2(opCode, op, es);
    }

    public override void Compile() {
      CompileToDoubleOrNan();
      WrapDoubleToNumberValue();
    }

    public override void CompileToDoubleOrNan() {
      es[0].CompileToDoubleOrNan();
      es[1].CompileToDoubleOrNan();
      ilg.Emit(opCode);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return Make(res);
//      return new CGArithmetic2(opCode, op, res);
    }

    public override Typ Type() {
      return Typ.Number;
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      return Typ.Number;
    }

    public override int Arity { get { return 2; } }

    public override string ToString() {
      if (es.Length == 2)
        return "(" + es[0] + op + es[1] + ")";
      else
        return "Err: ArgCount";
    }
  }

  /// <summary>
  /// A CGComparison is a comparison operation that takes two operands,
  /// both numeric for now, and produces
  /// the outcome true (1.0) or false (0.0), or in case either operand
  /// evaluates to NaN or +/-infinity, it produces that NaN or a plain NaN.
  /// </summary>
  public abstract class CGComparison : CGStrictOperation {
    public CGComparison(CGExpr[] es, Applier applier) : base(es, applier) { }

    // These are used in a template method pattern
    // Emit instruction to compare doubles, leaving 1 on stack if true:
    protected abstract void GenCompareDouble();
    // Emit instructions to compare doubles, jump to target if false:
    protected abstract void GenDoubleFalseJump(Label target);
    protected abstract String Name { get; }

    public override void Compile() {
      CompileToDoubleProper(new Gen(delegate { WrapDoubleToNumberValue(); }),
                            GenLoadTestDoubleErrorValue());
    }

    // A comparison always evaluates to a double; if it cannot evaluate to
    // a proper double, it evaluates to an infinity or NaN.
    // This code seems a bit redundant: It leaves the double on the stack top,
    // whether or not it is proper?  On the other hand, it can be improved only
    // by basically duplicating the CompileToDoubleProper method's body.
    // Doing so might avoid a jump to a jump or similar, though.
    public override void CompileToDoubleOrNan() {
      CompileToDoubleProper(
        new Gen(delegate { }),
        new Gen(delegate { ilg.Emit(OpCodes.Ldloc, testDouble); }));
    }
```

```
  // A comparison evaluates to a proper double only if both operands do
  public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
    es[0].CompileToDoubleProper(
      new Gen(delegate {
        es[1].CompileToDoubleProper(
          new Gen(delegate {
            GenCompareDouble();
            ilg.Emit(OpCodes.Conv_R8);
            ifProper.Generate(ilg);
          }),
          new Gen(delegate {
            ilg.Emit(OpCodes.Pop);
            ifOther.Generate(ilg);
          }));
      }),
      ifOther);
  }

  // This override combines the ordering predicate and the conditional jump
  public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
    es[0].CompileToDoubleProper(
      new Gen(delegate {
        es[1].CompileToDoubleProper(
          new Gen(delegate {
            GenDoubleFalseJump(ifFalse.GetLabel(ilg));
            ifTrue.Generate(ilg);
            if (!ifFalse.Generated) {
              Label endLabel = ilg.DefineLabel();
              ilg.Emit(OpCodes.Br, endLabel);
              ifFalse.Generate(ilg);
              ilg.MarkLabel(endLabel);
            }
          }),
          new Gen(delegate {
            ilg.Emit(OpCodes.Pop);
            ifOther.Generate(ilg);
          }));
      }),
      ifOther);
  }

  public override string ToString() {
    return es[0] + Name + es[1];
  }

  public override Typ Type() {
    return Typ.Number;
  }

  protected override Typ GetInputTypWithoutLengthCheck(int pos) {
    return Typ.Number;
  }

  public override int Arity { get { return 2; } }
}

// TODO: Comparisons should also be made to work for strings etc.  Static type
// information should be exploited when available, and when not,
// a general comparer should be generated.  For doubles, we should
// check that we're consistent with respect to unordered comparison.

/// <summary>
/// A CGGreaterThan is a comparison e1 > e2.
/// </summary>
public class CGGreaterThan : CGComparison {
  private static readonly Applier gtApplier = Function.Get(">").Applier;

  public CGGreaterThan(CGExpr[] es) : base(es, gtApplier) { }

  protected override void GenCompareDouble() {
```



```
    ilg.Emit(OpCodes.Cgt);
  }

  protected override void GenDoubleFalseJump(Label target) {
    ilg.Emit(OpCodes.Ble, target);
  }

  public override CGExpr Residualize(CGExpr[] res) {
    return new CGGreaterThan(res);
  }

  protected override string Name { get { return ">"; } }
}

/// <summary>
/// A CGLessThan is a comparison e1 < e2.
/// </summary>
public class CGLessThan : CGComparison {
  private static readonly Applier ltApplier = Function.Get("<").Applier;

  public CGLessThan(CGExpr[] es) : base(es, ltApplier) { }

  protected override void GenCompareDouble() {
    ilg.Emit(OpCodes.Clt);
  }

  protected override void GenDoubleFalseJump(Label target) {
    ilg.Emit(OpCodes.Bge, target);
  }

  public override CGExpr Residualize(CGExpr[] res) {
    return new CGLessThan(res);
  }

  protected override string Name { get { return "<"; } }
}

/// <summary>
/// A CGLessThanOrEqual is a comparison e1 <= e2.
/// </summary>
public class CGLessThanOrEqual : CGComparison {
  private static readonly Applier leApplier = Function.Get("<=").Applier;

  public CGLessThanOrEqual(CGExpr[] es) : base(es, leApplier) { }

  protected override void GenCompareDouble() {
    // OpCodes.Not negates all int bits and doesn't work here!
    ilg.Emit(OpCodes.Cgt);
    ilg.Emit(OpCodes.Ldc_I4_0);
    ilg.Emit(OpCodes.Ceq);
  }

  protected override void GenDoubleFalseJump(Label target) {
    ilg.Emit(OpCodes.Bgt, target);
  }

  public override CGExpr Residualize(CGExpr[] res) {
    return new CGLessThanOrEqual(res);
  }

  protected override string Name { get { return "<="; } }
}

/// <summary>
/// A CGGreaterThanOrEqual is a comparison e1 >= e2.
/// </summary>
public class CGGreaterThanOrEqual : CGComparison {
  private static readonly Applier geApplier = Function.Get(">=").Applier;

  public CGGreaterThanOrEqual(CGExpr[] es) : base(es, geApplier) { }
```

```
    protected override void GenCompareDouble() {
      ilg.Emit(OpCodes.Clt);
      ilg.Emit(OpCodes.Ldc_I4_0);
      ilg.Emit(OpCodes.Ceq);
    }

    protected override void GenDoubleFalseJump(Label target) {
      ilg.Emit(OpCodes.Blt, target);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return new CGGreaterThanOrEqual(res);
    }

    protected override string Name { get { return ">="; } }
  }

  /// <summary>
  /// A CGEqual is a comparison e1 = e2.
  /// </summary>
  public class CGEqual : CGComparison {
    private static readonly Applier eqApplier = Function.Get("=").Applier;

    public CGEqual(CGExpr[] es) : base(es, eqApplier) { }

    public static CGExpr Make(CGExpr[] es) {
      if (es.Length == 2) {
        if (es[0].Is(0))        // 0.0=e1 ==> NOT(e1)
          return new CGNot(new CGExpr[] { es[1] });
        else if (es[1].Is(0))   // e0=0.0 ==> NOT(e0)
          return new CGNot(new CGExpr[] { es[0] });
      }
      return new CGEqual(es);
    }

    protected override void GenCompareDouble() {
      ilg.Emit(OpCodes.Ceq);
    }

    protected override void GenDoubleFalseJump(Label target) {
      ilg.Emit(OpCodes.Bne_Un, target);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return Make(res);
    }

    protected override string Name { get { return "="; } }
  }

  /// <summary>
  /// A CGNotEqual is a comparison e1 <> e2.
  /// </summary>
  public class CGNotEqual : CGComparison {
    private CGNotEqual(CGExpr[] es) : base(es, Function.Get("<>").Applier) { }

    public static CGExpr Make(CGExpr[] es) {
      if (es.Length == 2) {
        if (es[0].Is(0))        // 0.0<>e1 ==> AND(e1)
          return new CGAnd(new CGExpr[] { es[1] });
        else if (es[1].Is(0))   // e0<>0.0 ==> AND(e0)
          return new CGAnd(new CGExpr[] { es[0] });
      }
      return new CGNotEqual(es);
    }

    protected override void GenCompareDouble() {
      ilg.Emit(OpCodes.Ceq);
      ilg.Emit(OpCodes.Ldc_I4_0);
      ilg.Emit(OpCodes.Ceq);
    }
```



```
    protected override void GenDoubleFalseJump(Label target) {
      ilg.Emit(OpCodes.Beq, target);
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return Make(res);
    }

    protected override string Name { get { return "<>"; } }
  }

  /// <summary>
  /// A CGConst is a constant expression.
  /// </summary>
  public abstract class CGConst : CGExpr {
    public static CGConst Make(double d) {
      return Make(NumberValue.Make(d));
    }

    public static CGConst Make(Value v) {
      if (v is NumberValue)
        return new CGNumberConst((v as NumberValue));
      else if (v is TextValue)
        return new CGTextConst(v as TextValue);
      else if (v is ErrorValue)
        return new CGError((v as ErrorValue));
      else
        return new CGValueConst(v);
    }

    public abstract Value Value { get; }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      return this;
    }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) { }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) { }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) { }
  }

  /// <summary>
  /// A CGTextConst is a constant text-valued expression.
  /// </summary>
  public class CGTextConst : CGConst {
    public readonly TextValue value;
    private readonly int index;

    public CGTextConst(TextValue value) {
      this.value = value;
      this.index = TextValue.GetIndex(value.value);
    }

    public override void Compile() {
      if (value.value == "")
        ilg.Emit(OpCodes.Ldsfld, TextValue.emptyField);
      else {
        ilg.Emit(OpCodes.Ldc_I4, index);
        ilg.Emit(OpCodes.Call, TextValue.fromIndexMethod);
      }
    }

    public override void CompileToDoubleOrNan() {
      LoadErrorNan(ErrorValue.argTypeError);
    }
```

```
    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      ifOther.Generate(ilg);
    }

    // In Excel, a text used in a conditional produces the error #VALUE! -- mostly
    public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      ifOther.Generate(ilg);
    }

    public override Value Value {
      get { return value; }
    }

    public override Typ Type() {
      return Typ.Text;
    }
  }

  /// <summary>
  /// A CGNumberConst is a constant number-valued expression.
  /// </summary>
  public class CGNumberConst : CGConst {
    public readonly NumberValue number;

    public CGNumberConst(NumberValue number) {
      this.number = number;
    }

    public override void Compile() {
      // sestoft: Would be better to load the NumberValue from this instance,
      // but then it would need to be stored somewhere (e.g. in a global Value
      // array from which it can be loaded).  See notes.txt.
      if (number.value == 0.0)
        ilg.Emit(OpCodes.Ldsfld, NumberValue.zeroField);
      else if (number.value == 1.0)
        ilg.Emit(OpCodes.Ldsfld, NumberValue.oneField);
      else if (number.value == Math.PI)
        ilg.Emit(OpCodes.Ldsfld, NumberValue.piField);
      else {
        ilg.Emit(OpCodes.Ldc_R8, number.value);
        WrapDoubleToNumberValue();
      }
    }

    public override void CompileToDoubleOrNan() {
      ilg.Emit(OpCodes.Ldc_R8, number.value);
    }

    public override void CompileToDoubleProper(Gen ifProper, Gen ifOther) {
      if (double.IsInfinity(number.value) || double.IsNaN(number.value)) {
        ilg.Emit(OpCodes.Ldc_R8, number.value);
        ilg.Emit(OpCodes.Stloc, testDouble);
        ifOther.Generate(ilg);
      } else {
        ilg.Emit(OpCodes.Ldc_R8, number.value);
        ifProper.Generate(ilg);
      }
    }

    public override void CompileCondition(Gen ifTrue, Gen ifFalse, Gen ifOther) {
      if (Double.IsInfinity(number.value) || Double.IsNaN(number.value)) {
        ilg.Emit(OpCodes.Ldc_R8, number.value);
        ilg.Emit(OpCodes.Stloc, testDouble);
        ifOther.Generate(ilg);
      } else if (number.value != 0)
        ifTrue.Generate(ilg);
      else
        ifFalse.Generate(ilg);
    }

    public override Value Value {
```



```
      get { return number; }
    }

    public override string ToString() {
      return number.value.ToString();
    }

    public override Typ Type() {
      return Typ.Number;
    }
  }

  /// <summary>
  /// A CGError is a constant error-valued expression, such as an
  /// illegal cell reference arising from deletion of rows or columns
  /// or the copying of relative references.
  /// </summary>
  public class CGError : CGConst {
    private readonly ErrorValue errorValue;

    public CGError(ErrorValue errorValue) {
      this.errorValue = errorValue;
    }

    public CGError(String message) : this(ErrorValue.Make(message))
    { }

    // This is used to implement the ERR function
    public CGError(CGExpr[] es) {
      if (es.Length != 1)
        errorValue = ErrorValue.argCountError;
      else {
        CGTextConst messageConst = es[0] as CGTextConst;
        if (messageConst == null)
          errorValue = ErrorValue.argTypeError;
        else
          errorValue = ErrorValue.Make("#ERR: " + messageConst.value.value);
      }
    }

    public override void Compile() {
      LoadErrorValue(errorValue);
    }

    public override void CompileToDoubleOrNan() {
      ilg.Emit(OpCodes.Ldc_R8, errorValue.ErrorNan);
    }

    public override Value Value {
      get { return errorValue; }
    }

    public override string ToString() {
      return errorValue.message;
    }

    public override Typ Type() {
      return Typ.Error;
    }
  }

  /// <summary>
  /// A CGValueConst is a general constant-valued expression; arises
  /// from partial evaluation only.
  /// </summary>
  public class CGValueConst : CGConst {
    public readonly int index;

    private static readonly MethodInfo loadValueConstMethod
      = typeof(CGValueConst).GetMethod("LoadValueConst");
    private static readonly ValueTable<Value> valueTable
```

```
      = new ValueTable<Value>();

    public CGValueConst(Value value) {
      this.index = valueTable.GetIndex(value);
    }

    public override void Compile() {
      ilg.Emit(OpCodes.Ldc_I4, index);
      ilg.Emit(OpCodes.Call, loadValueConstMethod);
    }

    public override void CompileToDoubleOrNan() {
      LoadErrorNan(ErrorValue.argTypeError);
    }

    // Used only (through reflection and) from generated code
    public static Value LoadValueConst(int index) {
      return valueTable[index];
    }

    public override Value Value {
      get { return valueTable[index]; }
    }

    public override string ToString() {
      return String.Format("CGValueConst({0}) at {1}", valueTable[index], index);
    }

    public override Typ Type() {
      return Typ.Value;
    }
  }

  /// <summary>
  /// A CGClosure is an application of the CLOSURE built-in function.
  /// </summary>
  public class CGClosure : CGStrictOperation {
    private static readonly Applier closureApplier = Function.Get("CLOSURE").Applier;

    public CGClosure(CGExpr[] es)
      : base(es, closureApplier) { }

    public override void Compile() {
      int argCount = es.Length - 1;
      if (es.Length < 1)
        LoadErrorValue(ErrorValue.argCountError);
      else if (es[0] is CGTextConst) {
        String name = (es[0] as CGTextConst).value.value;
        SdfInfo sdfInfo = SdfManager.GetInfo(name);
        if (sdfInfo == null)
          LoadErrorValue(ErrorValue.nameError);
        else if (argCount != 0 && argCount != sdfInfo.arity)
          LoadErrorValue(ErrorValue.argCountError);
        else {
          ilg.Emit(OpCodes.Ldc_I4, sdfInfo.index);
          CompileToValueArray(argCount, 1, es);
          ilg.Emit(OpCodes.Call, FunctionValue.makeMethod);
        }
      } else {
        es[0].Compile();
        CheckType(FunctionValue.type,
          new Gen(delegate {
            CompileToValueArray(argCount, 1, es);
            ilg.Emit(OpCodes.Call, FunctionValue.furtherApplyMethod);
          }),
          GenLoadErrorValue(ErrorValue.argTypeError));
      }
    }

    public override CGExpr Residualize(CGExpr[] res) {
      return new CGClosure(res);
```



```
    }

    public override int Arity { get { return es.Length; } }

    public override Typ Type() {
      return Typ.Function;
    }

    protected override Typ GetInputTypWithoutLengthCheck(int pos) {
      return Typ.Value;
    }

    public override void CompileToDoubleOrNan() {
      // TODO: Is this right?
      LoadErrorNan(ErrorValue.argTypeError);
    }
  }

  /// <summary>
  /// A CGNormalCellRef is a reference to a cell in an ordinary sheet, not
  /// a function sheet; must be evaluated each time the function is called.
  /// </summary>
  public class CGNormalCellRef : CGExpr {
    private readonly int index; // If negative, cell ref is illegal

    private static readonly MethodInfo getAddressMethod
      = typeof(CGNormalCellRef).GetMethod("GetAddress");
    private static readonly ValueTable<FullCellAddr> valueTable
      = new ValueTable<FullCellAddr>();

    public CGNormalCellRef(FullCellAddr cellAddr) {
      if (cellAddr.sheet.IsFunctionSheet)
        this.index = -1;             // Illegal cell reference
      else
        this.index = valueTable.GetIndex(cellAddr);
    }

    public override void Compile() {
      if (index >= 0) {
        ilg.Emit(OpCodes.Ldc_I4, index);
        ilg.Emit(OpCodes.Call, getAddressMethod);
        // HERE
        ilg.Emit(OpCodes.Stloc, tmpFullCellAddr);
        ilg.Emit(OpCodes.Ldloca, tmpFullCellAddr);
        ilg.Emit(OpCodes.Call, FullCellAddr.evalMethod);
      } else
        LoadErrorValue(ErrorValue.Make("#FUNERR: Ref to other function sheet"));
    }

    public override CGExpr PEval(PEnv pEnv, bool hasDynamicControl) {
      return this;
    }

    public static FullCellAddr GetAddress(int number) {
      return valueTable[number];
    }

    public override void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) { }

    public override void EvalCond(PathCond evalCond, IDictionary<FullCellAddr, PathCond> ev
alConds,
                                  List<CGCachedExpr> caches) { }

    public override void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      // We do not track dependencies on cells on ordinary sheets
    }

    public override Typ Type() {
      // We don't infer cell types in ordinary sheets, so assume the worst
      return Typ.Value;
    }
```

```
  public override void CompileToDoubleOrNan() {
    Compile();
    UnwrapToDoubleOrNan();
  }
}

/// <summary>
/// A Gen object is a code generator that avoids generating the same code
/// multiple times, and also to some extent avoids generating jumps to jumps.
/// </summary>
public class Gen {
  private readonly Action generate;
  private Label? label;
  private bool generated;  // Invariant: generated implies label.HasValue

  public Gen(Action generate) {
    this.generate = generate;
    label = null;
    generated = false;
  }

  // A Generate object has a unique label
  public Label GetLabel(ILGenerator ilg) {
    if (!label.HasValue)
      label = ilg.DefineLabel();
    return label.Value;
  }

  public bool Generated { get { return generated; } }

  public void Generate(ILGenerator ilg) {
    if (generated)
      ilg.Emit(OpCodes.Br, GetLabel(ilg));
    else {
      ilg.MarkLabel(GetLabel(ilg));
      generated = true;
      generate();
    }
  }
}

/// <summary>
/// A PEnv is an environment for partial evaluation.
/// </summary>
public class PEnv : Dictionary<FullCellAddr, CGExpr> { }

/// <summary>
/// A FunctionInfo is a table of built-in functions: each function's
/// signature, MethodInfo object, and other information.
/// </summary>
public class FunctionInfo {
  public readonly String name;             // For lookup and display
  public readonly MethodInfo methodInfo;   // For code generation
  public readonly Signature signature;     // For arg. compilation
  public readonly bool isSerious;          // Cache it or not?
  public readonly Applier applier;         // For specialization

  private static readonly IDictionary<String, FunctionInfo>
    functions = new Dictionary<String, FunctionInfo>();

  // Some signatures for fixed-arity built-in functions; return type first
  private static readonly Signature
    numToNum = new Signature(Typ.Number, Typ.Number),
    numNumToNum = new Signature(Typ.Number, Typ.Number, Typ.Number),
    unitToNum = new Signature(Typ.Number),
    valuesToNum = new Signature(Typ.Number, null),  // Variable arity
    valuesToValue = new Signature(Typ.Value, null),  // Variable arity
    valueToNum = new Signature(Typ.Number, Typ.Value),
    valueNumNumToValue = new Signature(Typ.Value, Typ.Value, Typ.Number, Typ.Number),
    valueValueToNum = new Signature(Typ.Number, Typ.Value, Typ.Value),
```



```
    valueToValue = new Signature(Typ.Value, Typ.Value),
    valueValueToValue = new Signature(Typ.Value, Typ.Value, Typ.Value),
    valueValueValueToValue = new Signature(Typ.Value, Typ.Value, Typ.Value, Typ.Value),
    valueNumNumNumNumToValue = new Signature(Typ.Value, Typ.Value, Typ.Number, Typ.Number
, Typ.Number, Typ.Number);

  // Fixed-arity built-in functions that do not require special treatment.
  // Those that require special treatment are in CGComposite.Make.
  static FunctionInfo() {
    MakeMath1("Abs");
    MakeMath1("Acos");
    MakeMath1("Asin");
    MakeMath1("Atan");
    MakeFunction("ATAN2", "ExcelAtan2", numNumToNum, false);
    MakeFunction("AVERAGE", "Average", valuesToNum, true);
    MakeFunction("BENCHMARK", "Benchmark", valueValueToValue, true);
    MakeFunction("CEILING", "ExcelCeiling", numNumToNum, false);
    MakeFunction("COLMAP", "ColMap", valueValueToValue, true);
    MakeFunction("COLUMNS", "Columns", valueToNum, false);
    MakeFunction("&", "ExcelConcat", valueValueToValue, false);
    MakeFunction("CONSTARRAY", "ConstArray", valueValueValueToValue, true);
    MakeMath1("Cos");
    MakeFunction("COUNTIF", "CountIf", valueValueToValue, true);
    MakeFunction("EQUAL", "Equal", valueValueToNum, false);
    MakeMath1("Exp");
    MakeFunction("FLOOR", "ExcelFloor", numNumToNum, false);
    MakeFunction("HARRAY", "HArray", valuesToValue, true);
    MakeFunction("HCAT", "HCat", valuesToValue, true);
    MakeFunction("HSCAN", "HScan", valueValueValueToValue, true);
    MakeFunction("INDEX", "Index", valueNumNumToValue, false);
    MakeFunction("ISARRAY", "IsArray", valueToNum, false);
    MakeFunction("ISERROR", "IsError", valueToNum, false);
    MakeMath1("LN", "Log");
    MakeMath1("LOG", "Log10");
    MakeMath1("LOG10", "Log10");
    MakeFunction("MAP", "Map", valuesToValue, true);
    MakeFunction("MAX", "Max", valuesToNum, true);
    MakeFunction("MIN", "Min", valuesToNum, true);
    MakeFunction("MOD", "ExcelMod", numNumToNum, false);
    MakeFunction("NOW", "ExcelNow", unitToNum, false);
    MakeFunction("^", "ExcelPow", numNumToNum, false);
    MakeFunction("RAND", "ExcelRand", unitToNum, false);
    MakeFunction("REDUCE", "Reduce", valueValueValueToValue, true);
    MakeFunction("ROUND", "ExcelRound", numNumToNum, false);
    MakeFunction("ROWMAP", "RowMap", valueValueToValue, true);
    MakeFunction("ROWS", "Rows", valueToNum, false);
    MakeFunction("SIGN", "Sign", numToNum, false);
    MakeMath1("Sin");
    MakeFunction("SLICE", "Slice", valueNumNumNumNumToValue, false);
    MakeFunction("SPECIALIZE", "Specialize", valueToValue, true);
    MakeMath1("Sqrt");
    MakeFunction("SUM", "Sum", valuesToNum, true);
    MakeFunction("SUMIF", "SumIf", valueValueToValue, true);
    MakeFunction("TABULATE", "Tabulate", valueValueValueToValue, true);
    MakeMath1("Tan");
    MakeFunction("TRANSPOSE", "Transpose", valueToValue, true);
    MakeFunction("VARRAY", "VArray", valuesToValue, true);
    MakeFunction("VCAT", "VCat", valuesToValue, true);
    MakeFunction("VSCAN", "VScan", valueValueValueToValue, true);
  }

  public FunctionInfo(String name, MethodInfo methodInfo, Applier applier, Signature sign
ature, bool isSerious) {
    this.name = name.ToUpper();
    this.methodInfo = methodInfo;
    this.applier = applier;
    this.signature = signature;
    functions[this.name] = this; // For Funsheet
    this.isSerious = isSerious;
  }
```

```
    private static FunctionInfo MakeMath1(String funcalcName, String mathName) {
      return new FunctionInfo(funcalcName.ToUpper(),
        typeof(Math).GetMethod(mathName, new Type[] { typeof(double) }),
        Function.Get(funcalcName.ToUpper()).Applier,
        numToNum,
        isSerious: false);
    }

    private static FunctionInfo MakeMath1(String name) {
      return MakeMath1(name, name);
    }

    private static FunctionInfo MakeFunction(String ssName, MethodInfo method,
                                             Signature signature, bool isSerious) {
      return new FunctionInfo(ssName,
        method,
        Function.Get(ssName.ToUpper()).Applier,
        signature,
        isSerious);
    }

    private static FunctionInfo MakeFunction(String ssName, String implementationName,
                                             Signature signature, bool isSerious) {
      return new FunctionInfo(ssName,
        Function.type.GetMethod(implementationName),
        Function.Get(ssName.ToUpper()).Applier,
        signature,
        isSerious);
    }

    public static bool Find(String name, out FunctionInfo functionInfo) {
      return functions.TryGetValue(name, out functionInfo);
    }
  }

  /// <summary>
  /// A Signature describes the return type and argument types of a built-in function.
  /// </summary>
  public class Signature {
    public readonly Typ retType;
    public readonly Typ[] argTypes;   // null means variadic

    public Signature(Typ retType, params Typ[] argTypes) {
      this.retType = retType;
      this.argTypes = argTypes;
    }

    public int Arity {
      get { return argTypes != null ? argTypes.Length : -1; }
    }
  }
}
```



```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Text;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A CellsUsedInFunctions maps each function sheet cell to the
  /// names of sheet-defined functions using that cell, and conversely,
  /// maps each name of a sheet-defined function to the cells that
  /// function uses.  There should be a single instance, as a static field
  /// in class SdfManager.
  /// </summary>
  public class CellsUsedInFunctions {
    public readonly Dictionary<FullCellAddr, HashSet<string>> addressToFunctionList
      = new Dictionary<FullCellAddr, HashSet<string>>();
    private readonly Dictionary<string, List<FullCellAddr>> functionToAddressList
      = new Dictionary<string, List<FullCellAddr>>();
    // Bags, because multiple sheet-defined functions may use the same cells as
    // input cell or as output cell
    public readonly HashBag<FullCellAddr>
      inputCellBag = new HashBag<FullCellAddr>(),
      outputCellBag = new HashBag<FullCellAddr>();

    public void Clear() {
      addressToFunctionList.Clear();
      functionToAddressList.Clear();
      inputCellBag.Clear();
      outputCellBag.Clear();
    }

    internal ICollection<string> GetFunctionsUsingAddresses(ICollection<FullCellAddr> fcas)
 {
      ISet<string> affectedFunctions = new SortedSet<string>();
      foreach (FullCellAddr fca in fcas) {
        HashSet<string> names;
        if (fca.sheet.IsFunctionSheet && addressToFunctionList.TryGetValue(fca, out names))
          affectedFunctions.UnionWith(names);
      }
      return affectedFunctions;
    }

    internal void AddFunction(SdfInfo info, ICollection<FullCellAddr> addrs) {
      HashSet<FullCellAddr> inputCellSet = new HashSet<FullCellAddr>();
      inputCellSet.UnionWith(info.inputCells);
      foreach (FullCellAddr addr in addrs)
        if (!inputCellSet.Contains(addr))
```

```
          AddCellToFunction(info.name, addr);
      List<FullCellAddr> addrsList = new List<FullCellAddr>(addrs);
      functionToAddressList[info.name] = addrsList;
      inputCellBag.AddAll(info.inputCells);
      outputCellBag.Add(info.outputCell);
    }

    internal void RemoveFunction(SdfInfo info) {
      inputCellBag.RemoveAll(info.inputCells);
      outputCellBag.Remove(info.outputCell);
      List<FullCellAddr> addresses;
      if (!functionToAddressList.TryGetValue(info.name, out addresses))
        return;
      foreach (FullCellAddr addr in addresses)
        addressToFunctionList.Remove(addr);
      functionToAddressList.Remove(info.name);
    }

    private void AddCellToFunction(String info, FullCellAddr addr) {
      HashSet<String> names;
      if (!addressToFunctionList.TryGetValue(addr, out names)) {
        names = new HashSet<String>();
        addressToFunctionList[addr] = names;
      }
      names.Add(info);
    }
  }
}
```



```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Text;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A Typ is the type of a spreadsheet value, used for return type
  /// and argument types of built-in functions.
  /// </summary>
  public enum Typ { Error, Number, Text, Array, Function, Value };

  /// <summary>
  /// Class CodeGenerate holds all global methods and state for code
  /// generation from sheet-defined functions.
  /// </summary>
  public abstract class CodeGenerate {
    /// <summary>
    /// When true, the evaluation conditions take into account that AND
    /// and OR have short-circuit evaluation.  This gives more accurate
    /// but more complex evaluation conditions, which may be undesirable.
    /// </summary>
    public readonly bool SHORTCIRCUIT_EVALCONDS = false;

    /// <summary>
    /// The IL generator for a single function body, used by the
    /// CGExpr compile functions.  Set by the Initialize method,
    /// which must be called before any compilation methods.
    /// </summary>
    public static ILGenerator ilg;

    /// <summary>
    /// These temporaries are used in code that tests whether a Value
    /// is a NumberValue, whether a double is proper, and so on
    /// </summary>
    protected static LocalBuilder testValue;
    protected static LocalBuilder testDouble;
    protected static LocalBuilder tmpFullCellAddr;

    protected static readonly MethodInfo isInfinityMethod
      = typeof(double).GetMethod("IsInfinity", new Type[] { typeof(double) });
    protected static readonly MethodInfo isNaNMethod
      = typeof(double).GetMethod("IsNaN", new Type[] { typeof(double) });

    /// <summary>
```

```
    /// Maps cell address to a variable that holds the cell's value
    /// as a double, if any such variable exists
    /// </summary>
    private static Dictionary<FullCellAddr, Variable> numberVariables;

    /// <summary>
    /// Initializes the CodeGenerator infrastructure, binds the ILGenerator,
    /// creates temporaries; resets evaluation condition caches.
    /// Ready to compile a new SDF.
    /// </summary>
    /// <param name="ilg">The ILGenerator used by all compile methods</param>
    public static void Initialize(ILGenerator ilg) {
      CodeGenerate.ilg = ilg;
      numberVariables = new Dictionary<FullCellAddr, Variable>();
      testDouble = ilg.DeclareLocal(typeof(double));
      testValue = ilg.DeclareLocal(Value.type);
      tmpFullCellAddr = ilg.DeclareLocal(FullCellAddr.type);
    }

    protected static void LoadErrorValue(ErrorValue error) {
      ilg.Emit(OpCodes.Ldc_I4, error.index);
      ilg.Emit(OpCodes.Call, ErrorValue.fromIndexMethod);
    }

    protected static void LoadErrorNan(ErrorValue error) {
      ilg.Emit(OpCodes.Ldc_R8, error.ErrorNan);
    }

    protected static Gen GenLoadErrorValue(ErrorValue error) {
      return new Gen(delegate { LoadErrorValue(error); });
    }

    protected static Gen GenLoadErrorNan(ErrorValue error) {
      return new Gen(delegate { LoadErrorNan(error); });
    }

    // The code generated by this method expects that the local var testDouble
    // contains a NaN or Inf representing the error.
    protected static Gen GenLoadTestDoubleErrorValue() {
      return new Gen(delegate {
        ilg.Emit(OpCodes.Ldloc, testDouble);
        WrapDoubleToNumberValue();
      });
    }

    protected static void SetArgCountErrorNan() {
      ilg.Emit(OpCodes.Ldc_R8, ErrorValue.argCountError.ErrorNan);
      ilg.Emit(OpCodes.Stloc, testDouble);
    }

    public static Dictionary<FullCellAddr, Variable> NumberVariables {
      get { return numberVariables; }
    }

    /// <summary>
    /// Generate code to check the type of the stack top value, and leave it
    /// there if it is of the expected type t followed by success code; else
    /// jump to failure code.
    /// </summary>
    /// <param name="t">The expected type of the stack top value.</param>
    /// <param name="ifType">Generate success code -- the value is of the expected type.</p
aram>
    /// <param name="ifOther">Generate failure code -- the value is not of the expected typ
e.</param>
    protected void CheckType(Type t, Gen ifType, Gen ifOther) {
      ilg.Emit(OpCodes.Stloc, testValue);
      ilg.Emit(OpCodes.Ldloc, testValue);
      ilg.Emit(OpCodes.Isinst, t);
      ilg.Emit(OpCodes.Brfalse, ifOther.GetLabel(ilg));
      ilg.Emit(OpCodes.Ldloc, testValue);
      ifType.Generate(ilg);
```



```
      if (!ifOther.Generated) {
        Label endLabel = ilg.DefineLabel();
        ilg.Emit(OpCodes.Br, endLabel);
        ifOther.Generate(ilg);
        ilg.MarkLabel(endLabel);
      }
    }

    protected void UnwrapToString(Gen ifString, Gen ifError) {
      CheckType(TextValue.type,
          new Gen(delegate {
            ilg.Emit(OpCodes.Ldfld, TextValue.valueField);
            ifString.Generate(ilg);
          }),
        ifError);
    }

    // Convert NumberValue to the enclosed float64, convert ErrorValue to its
    // NaN representation, and convert anything else to the NaN for ArgTypeError
    protected void UnwrapToDoubleOrNan() {
      ilg.Emit(OpCodes.Call, Value.toDoubleOrNanMethod);
    }

    public static void WrapDoubleToNumberValue() {
      ilg.Emit(OpCodes.Call, NumberValue.makeMethod);
    }

    /// <summary>
    /// Compute least upper bound of two types in the Typ type lattice.
    /// </summary>
    /// <returns>The least upper bound of the two argument types.</returns>
    protected Typ Lub(Typ typ1, Typ typ2) {
      if (typ1 == typ2) { return typ1; } else {
        switch (typ1) {
          case Typ.Error:
            return typ2;
          case Typ.Number:
          case Typ.Text:
          case Typ.Array:
          case Typ.Function:
            return typ2 == Typ.Error ? typ1 : Typ.Value;
          case Typ.Value:
            return Typ.Value;
          default:
            throw new ImpossibleException("Lub(Typ, Typ)");
        }
      }
    }

    /// <summary>
    /// Generate code to allocate an array vs of type Value[] and with length arity,
    /// then evaluate each argument expression es[i] to a Value and storing it
    /// in vs[i].  Leaves vs on the stack top.
    /// </summary>
    /// <param name="arity">The total number of arguments.</param>
    /// <param name="offset">The number of given argument expressions to ignore.</param>
    /// <param name="es">The given (partial) argument expressions.</param>
    internal static void CompileToValueArray(int arity, int offset, CGExpr[] es) {
      CreateValueArray(arity);
      CompileExpressionsAndStore(offset, es);
    }

    /// <summary>
    /// Generates code to evaluate each expression es[sourceOffset,...] and
    /// store it into array vs[0,...] on the stack top.
    /// Assumes an array vs of type Value[] is on the stack, and leaves array vs there.
    /// </summary>
    /// <param name="offset">The number of given argument expressions to ignore</param>
    /// <param name="es">The given (partial) argument expressions</param>
    internal static void CompileExpressionsAndStore(int sourceOffset, CGExpr[] es) {
      for (int i = sourceOffset; i < es.Length; i++) {
```

```
        ilg.Emit(OpCodes.Dup);
        ilg.Emit(OpCodes.Ldc_I4, i - sourceOffset);
        es[i].Compile();
        ilg.Emit(OpCodes.Stelem_Ref);
      }
    }

    /// <summary>
    /// Generates code to allocate a Value array, as in new Value[arity].
    /// </summary>
    /// <param name="arity">The size of the new array.</param>
    internal static void CreateValueArray(int arity) {
      ilg.Emit(OpCodes.Ldc_I4, arity);
      ilg.Emit(OpCodes.Newarr, Value.type);
    }
  }
}
```



```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Text;
using Corecalc;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A DependencyGraph is a graph representation of the dependencies between the function
sheet
  /// cells involved in a sheet-defined function.  Records the set
  /// nodesPrecedents[fca] of cells directly referred by the formula in cell fca, as
  /// well as the set nodesDependents[fca] of the cells directly referring to cell fca.
  /// Also determines whether the sheet-defined function references any volatile cells,
  /// including volatile cells on normal sheets.
  /// </summary>
  public class DependencyGraph {
    /// <summary>
    /// Map from a cell to its dependents: the cells that directly depend on it.
    /// </summary>
    private readonly Dictionary<FullCellAddr, HashSet<FullCellAddr>> nodesDependents;

    /// <summary>
    /// Map from a cell to its precedents: the cells on which it directly depends.
    /// </summary>
    private readonly Dictionary<FullCellAddr, HashSet<FullCellAddr>> nodesPrecedents;

    public readonly FullCellAddr outputCell;
    public readonly FullCellAddr[] inputCells;
    public readonly HashSet<FullCellAddr> inputCellSet;
    // Maps a full cell address to the Formula or CGExpr at that address
    private readonly Func<FullCellAddr, IDepend> getNode;

    public DependencyGraph(FullCellAddr outputCell, FullCellAddr[] inputCells, Func<FullCel
lAddr, IDepend> getNode) {
      this.outputCell = outputCell;
      this.nodesDependents = new Dictionary<FullCellAddr, HashSet<FullCellAddr>>();
      this.nodesPrecedents = new Dictionary<FullCellAddr, HashSet<FullCellAddr>>();
      this.inputCells = inputCells;
      this.inputCellSet = new HashSet<FullCellAddr>();
      this.inputCellSet.UnionWith(inputCells);
      this.getNode = getNode;
      CollectPrecedents();
    }

    public HashSet<FullCellAddr> GetDependents(FullCellAddr fca) {
      return nodesDependents[fca];
```

```
    }

    /// <summary>
    /// Make "precedent" a precedent of "node", and
    /// make "node" a dependent of "precedent".
    /// If the required sets of precedents resp. dependents do not exist,
    /// create them and associate them with "node" resp. "precedent".
    /// </summary>
    /// <exception cref="CyclicException">If a static cycle exists</exception>
    public bool AddPrecedentDependent(FullCellAddr precedent, FullCellAddr node) {
      HashSet<FullCellAddr> precedents;
      if (nodesPrecedents.TryGetValue(node, out precedents)) {
        try {
          precedents.Add(precedent);
        } catch (ArgumentException) {
          //Happens if the element already exists: there is a cycle
          throw new CyclicException("Static cycle through cell " + precedent, precedent);
        }
      } else {
        precedents = new HashSet<FullCellAddr>();
        precedents.Add(precedent);
        nodesPrecedents.Add(node, precedents);
      }

      HashSet<FullCellAddr> dependents;
      if (nodesDependents.TryGetValue(precedent, out dependents)) {
        if (!dependents.Add(node))
          //Happens if the element already exists: there is a cycle
          throw new CyclicException("Static cycle through cell " + node, node);
        else
          return true;
      } else {
        dependents = new HashSet<FullCellAddr>();
        dependents.Add(node);
        nodesDependents.Add(precedent, dependents);
        return false;
      }
    }

    /// <summary>
    /// Tests whether transitiveDependents contains any cell transitively dependent on node
.
    /// </summary>
    /// <param name="node"></param>
    /// <param name="possibleDependents"></param>
    /// <returns></returns>

    public bool HasDependent(FullCellAddr node, ICollection<FullCellAddr> transitiveDepende
nts) {
      HashSet<FullCellAddr> visitedCells = new HashSet<FullCellAddr>();
      return HasDependentHelper(node, transitiveDependents, visitedCells);
    }

    private bool HasDependentHelper(FullCellAddr node, ICollection<FullCellAddr> transitive
Dependents,
        HashSet<FullCellAddr> visitedCells) {
      HashSet<FullCellAddr> dependents;
      if (nodesDependents.TryGetValue(node, out dependents)) {
        foreach (FullCellAddr addr in transitiveDependents)
          if (dependents.Contains(addr))
            return true;
        foreach (FullCellAddr addr in dependents)
          if (visitedCells.Add(addr)) // returns true only first time
            if (HasDependentHelper(addr, transitiveDependents, visitedCells))
              return true;
      }
      return false;
    }

    /// <summary>
    /// Tests whether possiblePrecedents contains any cell that node
```



```
    /// transitively depends on.
    /// </summary>
    /// <param name="node"></param>
    /// <param name="transitivePrecedents"></param>
    /// <returns></returns>

    public bool HasPrecedent(FullCellAddr node, ICollection<FullCellAddr> transitivePrecede
nts) {
      HashSet<FullCellAddr> visitedCells = new HashSet<FullCellAddr>();
      return HasPrecedentHelper(node, transitivePrecedents, visitedCells);
    }

    private bool HasPrecedentHelper(FullCellAddr node, ICollection<FullCellAddr> transitive
Precedents, HashSet<FullCellAddr> visitedCells) {
      HashSet<FullCellAddr> precedents;
      if (nodesPrecedents.TryGetValue(node, out precedents)) {
        foreach (FullCellAddr addr in transitivePrecedents)
          if (precedents.Contains(addr))
            return true;
        foreach (FullCellAddr addr in precedents)
          if (visitedCells.Add(addr))
            if (HasPrecedentHelper(addr, transitivePrecedents, visitedCells))
              return true;
      }
      return false;
    }

    internal bool GetPrecedents(FullCellAddr node, out HashSet<FullCellAddr> precedents) {
      return nodesPrecedents.TryGetValue(node, out precedents);
    }

    /// <summary>
    /// Build the (two-way) dependency graph whose nodes are the
    /// output cell and all cells (on the same function sheet) on
    /// which it transitively depends.
    /// </summary>
    /// <exception cref="CyclicException"></exception>
    public void CollectPrecedents() {
      GetTransitivePrecedents(outputCell);
    }

    public IList<FullCellAddr> GetAllNodes() {
      // The full set of precedent cells is the output cell plus all cells
      // on which something depends -- true, but a funny way to compute it.
      IList<FullCellAddr> result = new List<FullCellAddr>();
      foreach (FullCellAddr fca in nodesDependents.Keys)
        result.Add(fca);
      result.Add(outputCell);
      return result;
    }

    /// <summary>
    /// Find all transitive precedents, that is, cells
    /// that this cell transitively depends on.
    /// The cell thisFca is within the function sheet being translated.
    /// Cells outside the function sheet are not traced.
    /// </summary>
    /// <param name="thisFca"></param>
    /// <exception cref="CyclicException"></exception>
    private void GetTransitivePrecedents(FullCellAddr thisFca) {
      ISet<FullCellAddr> precedents = new HashSet<FullCellAddr>();
      IDepend node = getNode(thisFca);
      if (node != null)
        node.DependsOn(thisFca, delegate(FullCellAddr fca) { precedents.Add(fca); });

      // Now precedents is the set of cells directly referred from
      // the Expr in cell thisFca; that is, that cell's direct precedents

      foreach (FullCellAddr addr in precedents)
        // Trace dependencies only from cells on this sheet,
        // and don't trace precedents of input cells
```

```
        if (addr.sheet == thisFca.sheet)
          if (!AddPrecedentDependent(addr, thisFca) && !inputCellSet.Contains(addr))
            GetTransitivePrecedents(addr);
    }

    /// <summary>
    /// Build a list of all the Formula and ArrayFormula cells that the outputCell
    /// depends on, in calculation order.
    /// </summary>
    /// <returns>Cells sorted in calculation order; output cell last.</returns>
    public IList<FullCellAddr> PrecedentOrder() {
      HashList<FullCellAddr> sorted = new HashList<FullCellAddr>();
      AddNode(sorted, outputCell);
      return sorted.ToArray();
    }

    private void AddNode(HashList<FullCellAddr> sorted, FullCellAddr node) {
      HashSet<FullCellAddr> precedents;
      if (GetPrecedents(node, out precedents)) {
        Cell cell;
        foreach (FullCellAddr precedent in precedents)
          // By including only non-input Formula and ArrayFormula cells, we avoid that
          // constant cells get stored in local variables.  The result will not contain
          // constants cells (and will contain an input cell only if it is also the
          // output cell), so must the raw graph to find all cells belonging to an SDF.
          if (!sorted.Contains(precedent)
            && precedent.TryGetCell(out cell)
            && (cell is Formula || cell is ArrayFormula)
            && !inputCellSet.Contains(precedent))
            AddNode(sorted, precedent);
      }
      sorted.Add(node); // Last in HashList
    }
  }
}
```



```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A CGExpressionBuilder is an expression visitor that builds a CGExpr
  /// from the expression.
  /// </summary>
  class CGExpressionBuilder : IExpressionVisitor {
    private readonly Dictionary<FullCellAddr, Variable> addressToVariable;
    private FullCellAddr thisFca; // The cell containing the Expr being translated
    private CGExpr result;        // The result of the compilation is left here

    private CGExpressionBuilder(Dictionary<FullCellAddr, Variable> addressToVariable,
                                FullCellAddr addr)
    {
      thisFca = addr;
      this.addressToVariable = addressToVariable;
    }

    public static CGExpr BuildExpression(FullCellAddr addr,
      Dictionary<FullCellAddr, Variable> addressToVariable)
    {
      Cell cell;
      if (!addr.TryGetCell(out cell))
        return new CGTextConst(TextValue.EMPTY);
      else if (cell is NumberCell)
        return new CGNumberConst(((NumberCell)cell).value);
      else if (cell is TextCell)
        return new CGTextConst(((TextCell)cell).value);
      else if (cell is QuoteCell)
        return new CGTextConst(((QuoteCell)cell).value);
      else if (cell is BlankCell)
        return new CGError("#FUNERR: Blank cell in function");
      else if (cell is Formula) {
        // Translate the expr relative to its containing cell at addr
        CGExpressionBuilder cgBuilder = new CGExpressionBuilder(addressToVariable, addr);
        Expr expr = ((Formula)cell).Expr;
        expr.VisitorCall(cgBuilder);
        return cgBuilder.result;
      } else if (cell is ArrayFormula)
        return new CGError("#FUNERR: Array formula in function");
      else
        throw new ImpossibleException("BuildExpression: " + cell);
```

```
    }

    public void CallVisitor(CellArea cellArea) {
      result = new CGNormalCellArea(cellArea.MakeArrayView(thisFca));
    }

    public void CallVisitor(CellRef cellRef) {
      FullCellAddr cellAddr = cellRef.GetAbsoluteAddr(thisFca);
      if (cellAddr.sheet != thisFca.sheet)
        // Reference to other sheet, hopefully a normal sheet
        result = new CGNormalCellRef(cellAddr);
      else if (this.addressToVariable.ContainsKey(cellAddr))
        // Reference to a cell that has already been computed in a local variable
        result = new CGCellRef(cellAddr, this.addressToVariable[cellAddr]);
      else // Inline the cell's formula's expression
        result = BuildExpression(cellAddr, addressToVariable);
    }

    public void CallVisitor(FunCall funCall) {
      CGExpr[] expressions = new CGExpr[funCall.es.Length];
      for (int i = 0; i < funCall.es.Length; i++) {
        funCall.es[i].VisitorCall(this);
        expressions[i] = result;
      }
      result = CGComposite.Make(funCall.function.name, expressions);
    }

    public void CallVisitor(Error error) {
      result = new CGError(error.value);
    }

    public void CallVisitor(NumberConst numbConst) {
      result = new CGNumberConst(numbConst.value);
    }

    public void CallVisitor(TextConst textConst) {
      result = new CGTextConst(textConst.value);
    }

    public void CallVisitor(ValueConst valueConst) {
      result = new CGValueConst(valueConst.value);
    }
  }
}
```



```
ï»¿// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006–2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A PathCond represents an evaluation condition.
  /// </summary>
  public abstract class PathCond : IEquatable<PathCond> {
    public static readonly PathCond FALSE = new Disj();
    public static readonly PathCond TRUE = new Conj();

    public abstract PathCond And(CachedAtom expr);
    public abstract PathCond AndNot(CachedAtom expr);
    public abstract PathCond Or(PathCond other);

    public abstract bool Is(bool b);
    public abstract CGExpr ToCGExpr();
    public abstract bool Equals(PathCond other);

    protected static PathCond[] AddItem(IEnumerable<PathCond> set, PathCond item) {
      HashList<PathCond> result = new HashList<PathCond>();
      result.AddAll(set);
      result.Add(item);
      return result.ToArray();
    }

    protected static String FormatInfix(String op, IEnumerable<PathCond> conds) {
      bool first = true;
      StringBuilder sb = new StringBuilder();
      sb.Append("(");
      foreach (PathCond p in conds) {
        if (!first)
          sb.Append(op);
        first = false;
        sb.Append(p);
      }
      return sb.Append(")").ToString();
    }
  }

  /// <summary>
  /// A CachedAtom represents a (possibly negated) expression from a source spreadsheet
  /// formula.  That expression is cached, in the sense that it will be
  /// evaluated at most once and its value stored in a local variable.
```

```
  /// For this to work, all copies and negations of a CachedAtom must have
  /// the same enclosed cachedExpr.
  /// </summary>
  public class CachedAtom : PathCond {
    public readonly CGCachedExpr cachedExpr;
    public readonly bool negated; // True if represents NOT(cachedExpr)

    public CachedAtom(CGExpr cond, List<CGCachedExpr> caches) {
      this.cachedExpr = new CGCachedExpr(cond, this, caches);
      this.negated = false;
    }

    private CachedAtom(CachedAtom atom, bool negated) {
      this.cachedExpr = atom.cachedExpr;
      this.negated = negated;
    }

    public CachedAtom Negate() {
      return new CachedAtom(this, !negated);
    }

    public override PathCond And(CachedAtom cond) {
      if (this.EqualsNega(cond))
        return FALSE;
      else
        return Conj.Make(this, cond);
    }

    public override PathCond AndNot(CachedAtom cond) {
      if (this.Equals(cond))
        return FALSE;
      else
        return Conj.Make(this, cond.Negate());
    }

    public override PathCond Or(PathCond other) {
      if (this.EqualsNega(other))
        return TRUE;
      else if (other is Conj || other is Disj)
        // TODO: This doesn't preserve order of disjuncts:
        return other.Or(this);
      else
        return Disj.Make(this, other);
    }

    public override bool Is(bool b) {
      return false;
    }

    public override CGExpr ToCGExpr() {
      cachedExpr.IncrementGenerateCount();
      if (negated)
        return new CGNot(new CGExpr[] { cachedExpr });
      else
        return cachedExpr;
    }

    public override bool Equals(PathCond other) {
      CachedAtom atom = other as CachedAtom;
      return atom != null && cachedExpr.Equals(atom.cachedExpr) && negated == atom.negated;
    }

    public override int GetHashCode() {
      return cachedExpr.GetHashCode() * 2 + (negated ? 1 : 0);
    }

    public bool EqualsNega(PathCond other) {
      CachedAtom atom = other as CachedAtom;
      return atom != null && cachedExpr.Equals(atom.cachedExpr) && negated != atom.negated;
    }
```



```
    public override string ToString() {
      if (negated)
        return "NOT(" + cachedExpr.ToString() + ")";
      else
        return cachedExpr.ToString();
    }
  }

  /// <summary>
  /// A Conj is conjunction of ordered path conditions.
  /// </summary>
  public class Conj : PathCond {
    internal readonly HashList<PathCond> conds;

    public Conj(params PathCond[] conds) {
      this.conds = new HashList<PathCond>();
      this.conds.AddAll(conds);
    }

    public static PathCond Make(params PathCond[] conjs) {
      HashList<PathCond> result = new HashList<PathCond>();
      foreach (PathCond conj in conjs)
        if (conj.Is(false))
          return FALSE;
        else if (!conj.Is(true))
          result.Add(conj);
      if (result.Count == 0)
        return TRUE;
      else if (result.Count == 1)
        return result.Single();
      else
        return new Conj(result.ToArray());
    }

    public override PathCond And(CachedAtom cond) {
      if (conds.Contains(cond.Negate()))
        return FALSE;
      else
        return Make(AddItem(this.conds, cond));
    }

    public override PathCond AndNot(CachedAtom cond) {
      if (conds.Contains(cond))
        return FALSE;
      else
        return Make(AddItem(this.conds, cond.Negate()));
    }

    public override PathCond Or(PathCond other) {
      if (this.Is(true) || other.Is(true))
        return TRUE;
      else if (this.Is(false))
        return other;
      else if (other.Is(false))
        return this;
      else if (conds.Contains(other))
        // Reduce ((p1 & ... & pn)) | pi to pi
        return other;
      else if (other is Disj)
        // TODO: This doesn't preserve order of disjuncts:
        return other.Or(this);
      else if (other is Conj) {
        if ((other as Conj).conds.Contains(this))
          // Reduce (pi | (p1 & ... & pn)) to pi
          return this;
        else {
          HashList<PathCond> intersect = HashList<PathCond>.Intersection(this.conds, (other
as Conj).conds);
          if (intersect.Count > 0) {
            // Reduce (p1 & ... & pn & q1 & ... & qm) | (p1 & ... & pn & r1 & ... & rk)
            // to (p1 & ... & pn & (q1 & ... & qm | r1 & ... & rk).
```

```
            // The pi go in intersect, qi in thisRest, and ri in otherRest.
            HashList<PathCond> thisRest = HashList<PathCond>.Difference(this.conds, intersect);
            HashList<PathCond> otherRest = HashList<PathCond>.Difference((other as Conj).conds, intersect);
            // This recursion terminates because thisRest is smaller than this.conds
            intersect.Add(Conj.Make(thisRest.ToArray()).Or(Conj.Make(otherRest.ToArray())));
            return Conj.Make(intersect.ToArray());
          } else
            return Disj.Make(AddItem(this.conds, other));
        }
      } else
        return Disj.Make(this, other);
    }

    public override bool Is(bool b) { // Conj() is TRUE, and Conj(b) is b
      return b && conds.Count == 0 || conds.Count == 1 && conds.Single().Is(b);
    }

    public override CGExpr ToCGExpr() {
      if (conds.Count == 1)
        return conds.Single().ToCGExpr();
      else
        return new CGAnd(conds.Select(cond => cond.ToCGExpr()).ToArray());
    }

    public override bool Equals(PathCond other) {
      return other is Conj && conds.UnsequencedEquals((other as Conj).conds);
    }

    public override int GetHashCode() {
      int result = 0;
      foreach (PathCond cond in conds)
        result = 37 * result + cond.GetHashCode();
      return result;
    }

    public override string ToString() {
      if (conds.Count == 0)
        return "TRUE";
      else
        return FormatInfix(" && ", conds);
    }
  }

  /// <summary>
  /// A Disj is a disjunction of ordered path conditions.
  /// </summary>
  public class Disj : PathCond {
    private readonly HashList<PathCond> conds;

    public Disj(params PathCond[] conds) {
      this.conds = new HashList<PathCond>();
      this.conds.AddAll(conds);
    }

    public static PathCond Make(params PathCond[] disjs) {
      HashList<PathCond> result = new HashList<PathCond>();
      foreach (PathCond disj in disjs)
        if (disj.Is(true))
          return TRUE;
        else if (!disj.Is(false))
          result.Add(disj);
      if (result.Count == 0)
        return FALSE;
      else if (result.Count == 1)
        return result.Single();
      else
        return new Disj(result.ToArray());
    }
```



```
    public override PathCond And(CachedAtom cond) {
      return Conj.Make(this, cond);
      // Alternatively, weed out disjuncts inconsistent with the condition,
      // using an order-preserving version of this code:
      // HashSet<PathCond> result = new HashSet<PathCond>();
      // result.AddAll(conds);
      // result.Filter(disj => !(disj is NegAtom && (disj as NegAtom).cond.Equals(cond)
      // || disj is Conj && (disj as Conj).conds.Contains(new NegAtom(cond))));
      // return Conj.Make(Make(result.ToArray(), new Atom(cond));
    }

    public override PathCond AndNot(CachedAtom cond) {
      return Conj.Make(this, cond.Negate());
    }

    public override PathCond Or(PathCond other) {
      if (other is CachedAtom && conds.Contains((other as CachedAtom).Negate()))
        // Reduce Or(OR(...,e,...), NOT(e)) and Or(OR(...,NOT(e),...), e) to TRUE
        return TRUE;
      else if (other is Disj) {
        HashList<PathCond> result = new HashList<PathCond>();
        result.AddAll(conds);
        foreach (PathCond cond in (other as Disj).conds) {
          if (cond is CachedAtom && conds.Contains((cond as CachedAtom).Negate()))
            // Reduce Or(OR(...,e,...),OR(...,NOT(e),...)) to TRUE
            // and    Or(OR(...,NOT(e),...),OR(...,e,...)) to TRUE
            return TRUE;
          result.Add(cond);
        }
        return Disj.Make(result.ToArray());
      } else if (other is Conj) {
        if ((other as Conj).conds.Contains(this))
          // Reduce (pi | (p1 & ... & pn)) to pi
          return this;
        else
          foreach (PathCond cond in (other as Conj).conds)
            if (conds.Contains(cond))
              // Reduce (p1 | ... | pn) | (... & pi & ...) to (p1 | ... | pn)
              return this;
        return Disj.Make(AddItem(this.conds, other));
      } else
        return Disj.Make(AddItem(this.conds, other));
    }

    public override bool Is(bool b) { // Disj() is FALSE, and Disj(b) is b
      return !b && conds.Count == 0 || conds.Count == 1 && conds.Single().Is(b);
    }

    public override CGExpr ToCGExpr() {
      if (conds.Count == 1)
        return conds.Single().ToCGExpr();
      else
        return new CGOr(conds.Select(cond => cond.ToCGExpr()).ToArray());
    }

    public override bool Equals(PathCond other) {
      return other is Disj && conds.UnsequencedEquals((other as Disj).conds);
    }

    public override int GetHashCode() {
      int result = 0;
      foreach (PathCond cond in conds)
        result = 37 * result + cond.GetHashCode();
      return result;
    }

    public override string ToString() {
      if (conds.Count == 0)
        return "FALSE";
      else
```

```
      return FormatInfix("‖", conds);
    }
  }
}
```



```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Reflection.Emit;
using System.Text;
using System.Diagnostics;
using System.Linq;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A ProgramLines object holds the cells involved in the definitions of
  /// a particular sheet-defined function, including its output cell, its input
  /// cells, all intermediate cells as a topologically sorted list of ComputeCells,
  /// and further data used during code generation and partial evaluation.
  /// </summary>
  public class ProgramLines : CodeGenerate {
    private readonly FullCellAddr outputCell;
    private readonly FullCellAddr[] inputCells;
    // Code to unwrap input cells to float64 when expedient:
    private readonly List<UnwrapInputCell> unwrapInputCells;
    // Code to evaluate all cells in order, ending with the output cell:
    private readonly List<ComputeCell> programList;
    // Map intermediate cell (address) to the formula in that cell:
    // Invariant: fcaToComputeCell[fca].cellAddr == fca
    private readonly IDictionary<FullCellAddr, ComputeCell> fcaToComputeCell;
    // The following dictionary maps both input cells and intermediate cells:
    // Invariant: fcaToComputeCell[fca].var == addressToVariable[fca] when both defined
    public readonly Dictionary<FullCellAddr, Variable> addressToVariable;
    // The list of CGCachedExpr used in evaluation conditions of this program
    private readonly List<CGCachedExpr> caches = new List<CGCachedExpr>();

    public ProgramLines(FullCellAddr outputCell, FullCellAddr[] inputCells) {
      this.outputCell = outputCell;
      this.inputCells = inputCells;
      programList = new List<ComputeCell>();
      fcaToComputeCell = new Dictionary<FullCellAddr, ComputeCell>();
      unwrapInputCells = new List<UnwrapInputCell>();
      addressToVariable = new Dictionary<FullCellAddr, Variable>();
      for (short i = 0; i < inputCells.Length; i++) {
        FullCellAddr addr = inputCells[i];
        addressToVariable.Add(addr, new LocalArgument(addr.ToString(), Typ.Value, i));
      }
    }

    public static Delegate CreateSdfDelegate(SdfInfo sdfInfo, DependencyGraph dpGraph, ILis
t<FullCellAddr> cellList) {
      Debug.Assert(sdfInfo.inputCells == dpGraph.inputCells);
```

```
      Debug.Assert(sdfInfo.outputCell == dpGraph.outputCell);
      ProgramLines program = new ProgramLines(sdfInfo.outputCell, sdfInfo.inputCells);
      program.AddComputeCells(dpGraph, cellList);
      // TODO: This is not the final program, so order may not respect eval cond dependenci
es!
      sdfInfo.Program = program;  // Save ComputeCell list for later partial evaluation
      return program.CompileToDelegate(sdfInfo);
    }

    public Delegate CompileToDelegate(SdfInfo sdfInfo) {
      Debug.Assert(sdfInfo.Program == this);  // Which is silly, lots of redundancy
      // Create dynamic method with signature: Value CGMethod(Value, Value, ...) in class F
unction:
      DynamicMethod method = new DynamicMethod("CGMethod", Value.type, sdfInfo.MyArgumentTy
pes,
                                               Function.type, true);
      ILGenerator ilg = method.GetILGenerator();
      CodeGenerate.Initialize(ilg);
      sdfInfo.Program.EvalCondReorderCompile();
      ilg.Emit(OpCodes.Ret);
      return method.CreateDelegate(sdfInfo.MyType);
    }

    /// <summary>
    /// Compiles the topologically sorted list of Expr to a list (program)
    /// of ComputeCells, encapsulating CGExprs.  Builds a map from cellAddr to
    /// local variable ids, for compiling sheet-internal cellrefs to ldloc instructions.
    /// </summary>
    public void AddComputeCells(DependencyGraph dpGraph, IList<FullCellAddr> cellList) {
      Debug.Assert(dpGraph.outputCell == cellList[cellList.Count - 1]);
      CGExpr outputExpr;
      if (cellList.Count == 0 || cellList.Count == 1 && dpGraph.inputCellSet.Contains(cellL
ist.Single()))
        // The output cell is also an input cell; load it:
        outputExpr = new CGCellRef(dpGraph.outputCell, addressToVariable[dpGraph.outputCell
]);
      else {
        // First process all non-output cells, and ignore all input cells:
        foreach (FullCellAddr cellAddr in cellList) {
          if (cellAddr.Equals(dpGraph.outputCell))
            continue;
          HashSet<FullCellAddr> dependents = dpGraph.GetDependents(cellAddr);
          int minUses = dependents.Count;
          if (minUses == 1) {
            FullCellAddr fromFca = dependents.First();
            minUses = Math.Max(minUses, GetCount(fromFca, cellAddr));
          }
          // Now if minUses==1 then there is at most one use of the cell at cellAddr,
          // and no local variable is needed.  Otherwise, allocate a local variable:
          if (minUses > 1) {
            CGExpr newExpr = CGExpressionBuilder.BuildExpression(cellAddr, addressToVariabl
e);
            Variable var = new LocalVariable(cellAddr.ToString(), newExpr.Type());
            AddComputeCell(cellAddr, new ComputeCell(newExpr, var, cellAddr));
          }
        }
        // Then process the output cell:
        outputExpr = CGExpressionBuilder.BuildExpression(dpGraph.outputCell, addressToVaria
ble);
      }
      // Add the output cell expression last, without a variable to bind it to; hence the n
ull,
      // also indicating that (only) the output cell is in tail position:
      AddComputeCell(dpGraph.outputCell, new ComputeCell(outputExpr, null, dpGraph.outputCe
ll));
    }

    public void AddComputeCell(FullCellAddr fca, ComputeCell ccell) {
      programList.Add(ccell);
      fcaToComputeCell.Add(fca, ccell);
      if (ccell.var != null)
```



```
        addressToVariable.Add(fca, ccell.var);
    }

    public void Compile() {
      foreach (UnwrapInputCell uwic in unwrapInputCells)
        uwic.Compile();
      foreach (ComputeCell expr in programList)
        expr.Compile();
    }

    public FullCellAddr[] ResidualInputs(FunctionValue fv) {
      // The residual input cells are those that have input value NA
      FullCellAddr[] residualInputs = new FullCellAddr[fv.Arity];
      int j = 0;
      for (int i = 0; i < fv.args.Length; i++)
        if (fv.args[i] == ErrorValue.naError)
          residualInputs[j++] = inputCells[i];
      return residualInputs;
    }

    // Partially evaluate the programList with respect to the given static inputs,
    // producing a new ProgramLines object.
    public ProgramLines PEval(Value[] args, FullCellAddr[] residualInputs) {
      PEnv pEnv = new PEnv();
      // Map static input cells to their constant values:
      for (int i = 0; i < args.Length; i++)
        pEnv[inputCells[i]] = CGConst.Make(args[i]);
      ProgramLines residual = new ProgramLines(outputCell, residualInputs);
      // PE-time environment PEnv maps each residual input cell address to the delegate arg
ument:
      for (int i = 0; i < residualInputs.Length; i++) {
        FullCellAddr input = residualInputs[i];
        pEnv[input] = new CGCellRef(input, residual.addressToVariable[input]);
      }
      // Process the given function's compute cells in dependency order, output last:
      foreach (ComputeCell ccell in programList) {
        ComputeCell rCcell = ccell.PEval(pEnv);
        if (rCcell != null)
          residual.AddComputeCell(ccell.cellAddr, rCcell);
      }
      residual = residual.PruneZeroUseCells();
      return residual;
    }

    private ProgramLines PruneZeroUseCells() {
      // This is slightly more general than necessary, since we know that the
      // new order of FullCellAddrs could be embedded in the old one.  So it would suffice
      // to simply count the number of uses of each FullCellAddr rather than do this sort.
      DependencyGraph dpGraph = new DependencyGraph(outputCell, inputCells, GetComputeCell)
;
      IList<FullCellAddr> prunedList = dpGraph.PrecedentOrder();
      ProgramLines prunedProgram = new ProgramLines(outputCell, inputCells);
      foreach (FullCellAddr cellAddr in prunedList)
        prunedProgram.AddComputeCell(cellAddr, GetComputeCell(cellAddr));
      return prunedProgram;
    }

    /// CodeGenerate.Initialize(ilg) must be called first.
    public void EvalCondReorderCompile() {
      ComputeEvalConds();
      // Re-sort the expressions to reflect new dependencies
      // introduced by evaluation conditions
      DependencyGraph augmentedGraph
        = new DependencyGraph(outputCell, inputCells, GetComputeCell);
      IList<FullCellAddr> augmentedList = augmentedGraph.PrecedentOrder();
      ProgramLines finalProgram = new ProgramLines(outputCell, inputCells);
      foreach (FullCellAddr cellAddr in augmentedList)
        finalProgram.AddComputeCell(cellAddr, GetComputeCell(cellAddr));
      // This relies on all pathconds having been generated at this point:
      EmitCacheInitializations();
      finalProgram.CreateUnwrappedNumberCells();
```

```
      finalProgram.Compile();
    }

    /// <summary>
    /// Insert code to unwrap the computed value of a cell, if the cell
    /// has type Value but is referred to as a Number more than once.
    /// Also register the unwrapped version of the variable
    /// in the NumberVariables dictionary.
    /// CodeGenerate.Initialize(ilg) must be called first.
    /// </summary>
    public void CreateUnwrappedNumberCells() {
      HashBag<FullCellAddr> numberUses = CountNumberUses();
      foreach (KeyValuePair<FullCellAddr, int> numberUseCount in numberUses.ItemMultiplicit
ies()) {
        FullCellAddr fca = numberUseCount.Key;
        if (numberUseCount.Value >= 2 && addressToVariable[fca].Type == Typ.Value) {
          Variable numberVar = new LocalVariable(fca + "_number", Typ.Number);
          ComputeCell ccell;
          if (fcaToComputeCell.TryGetValue(fca, out ccell)) // fca is ordinary computed cel
l
            ccell.NumberVar = numberVar;
          else // fca is an input cell
            unwrapInputCells.Add(new UnwrapInputCell(addressToVariable[fca], numberVar));
          NumberVariables.Add(fca, numberVar);
        }
      }
    }

    /// <summary>
    /// Count number of references from cell at fromFca to cell address toFca
    /// </summary>
    private static int GetCount(FullCellAddr fromFca, FullCellAddr toFca) {
      int count = 0;
      Cell fromCell;
      if (fromFca.TryGetCell(out fromCell))
        fromCell.DependsOn(fromFca,
          delegate(FullCellAddr fca) { if (toFca.Equals(fca)) count++; });
      return count;
    }

    public FullCellAddr[] InputCells {
      get { return inputCells; }
    }

    public HashBag<FullCellAddr> CountNumberUses() {
      var numberUses = new HashBag<FullCellAddr>();
      foreach (ComputeCell ccell in programList)
        ccell.CountUses(ccell.Type, numberUses);
      return numberUses;
    }

    public override string ToString() {
      StringBuilder sb = new StringBuilder();
      int counter = 0;
      foreach (ComputeCell expr in programList)
        sb.Append(counter++).Append(": ").AppendLine(expr.ToString());
      return sb.ToString();
    }

    public void ComputeEvalConds() {
      const int THRESHOLD = 30;
      // Compute evaluation condition for each cell
      IDictionary<FullCellAddr, PathCond> evalConds = new Dictionary<FullCellAddr, PathCond
>();
      evalConds[outputCell] = PathCond.TRUE;
      // The outputCell is also the first ccell processed below
      for (int i = programList.Count - 1; i >= 0; i--) {
        ComputeCell ccell = programList[i];
        int bound = THRESHOLD;
        bool isSerious = ccell.expr.IsSerious(ref bound);
        PathCond evalCond = evalConds[ccell.cellAddr];
```



```
        // Console.WriteLine("evalConds[{0}{1}] = {2}\n", casv.cellAddr, isSerious ? "" : "
:TRIVIAL", evalCond);
        if (isSerious && !evalCond.Is(true)) {
          Console.WriteLine("Setting EvalCond[{0}] = {1}", ccell.cellAddr, evalCond);
          ccell.EvalCond = evalCond.ToCGExpr();
        }
        ccell.expr.EvalCond(evalCond, evalConds, caches);
      }
    }

    /// CodeGenerate.Initialize(ilg) must be called first.
    public void EmitCacheInitializations() {
      foreach (CGCachedExpr cache in caches)
        cache.EmitCacheInitialization();
    }

    public ComputeCell GetComputeCell(FullCellAddr fca) {
      return fcaToComputeCell[fca];
    }
  }

  /// <summary>
  /// A ComputeCell represents a (intermediate or output) cell of a sheet-defined
  /// function and its associated IL local variable (if intermediate cell), the
  /// cell's evaluation condition, if any, and a double-type IL local variable if
  /// the cell is known to be number-valued.
  /// </summary>
  public class ComputeCell : CodeGenerate, IDepend {
    public readonly CGExpr expr;
    public readonly Variable var; // Null only for the unique output cell
    public readonly FullCellAddr cellAddr;
    private CGExpr evalCond;      // If non-null, then conditional evaluation
    private Variable numberVar;   // If non-null, unwrap to this Number variable
    // If var==null then numberVar==null

    public ComputeCell(CGExpr expr, Variable var, FullCellAddr cellAddr) {
      this.expr = expr;
      this.var = var;
      // The output cell's expression is in tail position:
      if (var == null)
        this.expr.NoteTailPosition();
      this.cellAddr = cellAddr;
      this.numberVar = null;
    }

    public CGExpr EvalCond {
      get { return evalCond; }
      set { this.evalCond = value; }
    }

    public Variable NumberVar {
      set { this.numberVar = value; }
    }

    /// <summary>
    /// Generate code to compute the expression's value and storing it in the IL
    /// local variable; possibly to unwrap to a number variable; and possibly under
    /// the control of an evaluation condition evalCond.
    /// </summary>
    public virtual void Compile() {
      EvalCondCompile(delegate {
        if (var != null && var.Type == Typ.Number)
          expr.CompileToDoubleOrNan();
        else
          expr.Compile();
        if (var != null)
          var.EmitStore(ilg);
        if (numberVar != null) {
          Debug.Assert(var.Type == Typ.Value);
          var.EmitLoad(ilg);
          UnwrapToDoubleOrNan();
```

```
          numberVar.EmitStore(ilg);
        }
      });
    }

    protected void EvalCondCompile(Action compile) {
      if (evalCond != null)
        evalCond.CompileToDoubleProper(
          new Gen(delegate {
            Label endLabel = ilg.DefineLabel();
            ilg.Emit(OpCodes.Ldc_R8, 0.0);
            ilg.Emit(OpCodes.Beq, endLabel);
            compile();
            ilg.MarkLabel(endLabel);
          }),
          new Gen(delegate { }));
      else
        compile();
    }

    public virtual void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn) {
      expr.DependsOn(here, dependsOn);
      if (evalCond != null)
        evalCond.DependsOn(here, dependsOn);
    }

    public void CountUses(Typ typ, HashBag<FullCellAddr> numberUses) {
      expr.CountUses(typ, numberUses);
      if (evalCond != null)
        evalCond.CountUses(Typ.Number, numberUses);
    }

    // Returns residual ComputeCell or null if no cell needed
    public ComputeCell PEval(PEnv pEnv) {
      CGExpr rCond = null;
      if (evalCond != null) // Never the case for an output cell
        rCond = evalCond.PEval(pEnv, false /* not dynamic control */);
      if (rCond is CGNumberConst)
        if ((rCond as CGNumberConst).number.value != 0.0)
          rCond = null;     // eval cond constant TRUE, discard eval cond
        else
          return null;      // eval cond constant FALSE, discard entire compute cell
      // If residual eval cond is not TRUE then expr has dynamic control
      CGExpr rExpr = expr.PEval(pEnv, rCond != null);
      if (rExpr is CGConst && var != null) {
        // If cell's value is constant and it is not an output cell just put in PEnv
        pEnv[cellAddr] = rExpr;
        return null;
      } else {
        // Else create fresh local variable for the residual cell, and make
        // PEnv map cell address to that local variable:
        Variable newVar = var != null ? var.Fresh() : null;
        pEnv[cellAddr] = new CGCellRef(cellAddr, newVar);
        ComputeCell result = new ComputeCell(rExpr, newVar, cellAddr);
        // result.EvalCond = rCond;  // Don't save residual eval cond, we compute it accura
tely later...
        return result;
      }
    }

    public Typ Type {
      // For an output cell (only) the expr.Type() may be Number whereas
      // the enclosing cell naturally has type Value.  This makes a difference
      // if that cell contains IF(..., A1, 17) because then A1 appears in a
      // context that expects a Number.  But this is so only if A1 has type number,
      // and hence no unwrapping is needed.  Hence it seems safe to always use
      // var.Type instead of recomputing expr.Type().
      get { return var != null ? var.Type : Typ.Value; }
      // Alternatively: return expr.Type();
    }
```



```
    public override string ToString() {
      StringBuilder sb = new StringBuilder();
      if (evalCond != null)
        sb.AppendFormat("if ({0}) {\n", evalCond);
      sb.AppendFormat("{0} = {1}", (var != null ? var.Name : "<output>"), expr);
      if (numberVar != null)
        sb.AppendFormat("\n{0} = UnwrapToDoubleOrNan{1}", numberVar.Name, var.Name);
      if (evalCond != null)
        sb.AppendFormat("\n}");
      return sb.ToString();
    }
  }

  /// <summary>
  /// An UnwrapInputCell represents the action, in a program list, to
  /// unwrap an input cell inputVar of type Value to a numberVar of type Number.
  /// </summary>
  public class UnwrapInputCell : CodeGenerate {
    public readonly Variable inputVar, numberVar;

    public UnwrapInputCell(Variable inputVar, Variable numberVar) {
      this.inputVar = inputVar;
      this.numberVar = numberVar;
    }

    public void Compile() {
      Debug.Assert(inputVar.Type == Typ.Value);
      Debug.Assert(numberVar.Type == Typ.Number);
      inputVar.EmitLoad(ilg);
      UnwrapToDoubleOrNan();
      numberVar.EmitStore(ilg);
    }

    public override string ToString() {
      return String.Format("{0} = UnwrapToDoubleOrNan({1})", numberVar.Name, inputVar.Name);
    }
  }
}
```

```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using System.Diagnostics;
using System.Linq;

namespace Corecalc.Funcalc {
  /// <summary>
  /// SfdManager holds the catalog of sheet defined functions.
  /// </summary>
  static class SdfManager {
    // Static tree dictionary to map an SDF name to an SdfInfo object.
    private static readonly IDictionary<String, SdfInfo> sdfNameToInfo
      = new SortedDictionary<String, SdfInfo>();
    // Static arrays to map an SDF index to a callable SdfDelegate and SdfInfo.
    // Poorer modularity than if we used a private ArrayList, but faster SDF calls.
    // Invariant: sdfInfos[i]==null || sdfInfos[i].index==i
    // Invariant: sdfDelegates.Length == sdfInfos.Length >= 1
    public static Delegate[] sdfDelegates = new Delegate[1];
    private static SdfInfo[] sdfInfos = new SdfInfo[1];
    private static int nextIndex = 0;

    // For generating code to call an SDF, in CGSdfCall and CGApply
    internal static readonly FieldInfo sdfDelegatesField
      = typeof(SdfManager).GetField("sdfDelegates");

    // For discovering edits to sheet-defined functions, and for coloring them
    public static readonly CellsUsedInFunctions cellToFunctionMapper
      = new CellsUsedInFunctions();

    // For caching partially evaluated sheet-defined functions
    private static readonly IDictionary<FunctionValue, SdfInfo> specializations
      = new Dictionary<FunctionValue, SdfInfo>();

    public static void ResetTables() {
      foreach (SdfInfo info in sdfNameToInfo.Values)
        Function.Remove(info.name);
      sdfNameToInfo.Clear();
      sdfDelegates = new Delegate[1];
      sdfInfos = new SdfInfo[1];
      nextIndex = 0;
      cellToFunctionMapper.Clear();
      specializations.Clear();
    }
```



```
    public static void ShowIL(SdfInfo info) {
      MethodInfo mif = sdfDelegates[info.index].Method;
      ClrTest.Reflection.MethodBodyViewer viewer = new ClrTest.Reflection.MethodBodyViewer(
);
      viewer.Text = info.ToString();
      viewer.SetMethodBase(mif);
      viewer.ShowDialog();
    }

    public static void CreateFunction(string name, FullCellAddr outPutCell, List<FullCellAd
dr> inputCells) {
      CreateFunction(name, outPutCell, inputCells.ToArray());
    }

    public static void CreateFunction(string name, FullCellAddr outputCell, FullCellAddr[]
inputCells) {
      name = name.ToUpper();
      // If the function exists, with the same input and output cells, keep it.
      // If it is a placeholder, overwrite its applier; if its input and output
      // cells have changed, recreate it (including its SdfInfo record).
      Function oldFunction = Function.Get(name);
      if (oldFunction != null)
        if (!oldFunction.IsPlaceHolder)
          return;
      // Registering the function before compilation allows it to call itself recursively
      SdfInfo sdfInfo = Register(outputCell, inputCells, name);
      // Console.WriteLine("Compiling {0} as #{1}", name, info.index);
      Update(sdfInfo, CompileSdf(sdfInfo));
      if (oldFunction != null) // ... and is not a placeholder
        oldFunction.UpdateApplier(sdfInfo.Apply, sdfInfo.IsVolatile);
      else
        new Function(name, sdfInfo.Apply, isVolatile: sdfInfo.IsVolatile);
    }

    /// <summary>
    /// Compiles entry code and body of a sheet-defined function
    /// </summary>
    /// <param name="info">The SdfInfo object describing the function</param>
    /// <returns></returns>
    private static Delegate CompileSdf(SdfInfo info) {
      // Build dependency graph containing all cells needed by the output cell
      DependencyGraph dpGraph = new DependencyGraph(info.outputCell, info.inputCells,
        delegate(FullCellAddr fca) { return fca.sheet[fca.ca]; });
      // Topologically sort the graph in calculation order; leave out constants
      IList<FullCellAddr> cellList = dpGraph.PrecedentOrder();
      info.SetVolatility(cellList);
      // Convert each Expr into a CGExpr while preserving order.  Inline single-use express
ions
      cellToFunctionMapper.AddFunction(info, dpGraph.GetAllNodes());
      return ProgramLines.CreateSdfDelegate(info, dpGraph, cellList);
    }

    /// <summary>
    /// Create a residual (sheet-defined) function for the given FunctionValue.
    /// As a side effect, register the new SDF and cache it in a dictionary mapping
    /// function values to residual SDFs.
    /// </summary>
    /// <param name="fv">The function value to specialize</param>
    /// <returns></returns>
    public static SdfInfo SpecializeAndCompile(FunctionValue fv) {
      SdfInfo residualSdf;
      if (!specializations.TryGetValue(fv, out residualSdf)) {
        FullCellAddr[] residualInputCells = fv.sdfInfo.Program.ResidualInputs(fv);
        String name = String.Format("{0}#{1}", fv, nextIndex);
        Console.WriteLine("Created residual function {0}", name);
        // Register before partial evaluation to enable creation of call cycles
        residualSdf = Register(fv.sdfInfo.outputCell, residualInputCells, name);
        specializations.Add(fv, residualSdf);
        ProgramLines residual = fv.sdfInfo.Program.PEval(fv.args, residualInputCells);
        residualSdf.Program = residual;
        Update(residualSdf, residual.CompileToDelegate(residualSdf));
```

```
      }
      return residualSdf;
    }

    // TODO: This may be inefficient when we have many specialized functions.
    // Perhaps maintain a more specialized data structure, such as a dictionary
    // from function name to dictionary from value list to specialization.
    public static List<Value[]> PendingSpecializations(String name) {
      List<Value[]> result = new List<Value[]>();
      foreach (FunctionValue fv in specializations.Keys)
        // TODO: Should we only consider pending (fv.sdfInfo.Program==null) functions?
        if (fv.sdfInfo.name == name)
          result.Add(fv.args);
      // Recipient is expected not to update the Value arrays
      return result;
    }

    // Register, unregister, update and look up the SDF tables

    /// <summary>
    /// Allocate an index for a new SDF, but do not bind its SdfDelegate
    /// </summary>
    /// <param name="outputCell"></param>
    /// <param name="inputCells"></param>
    /// <param name="name"></param>
    /// <returns></returns>
    public static SdfInfo Register(FullCellAddr outputCell, FullCellAddr[] inputCells, stri
ng name) {
      name = name.ToUpper();
      SdfInfo sdfInfo = GetInfo(name);
      if (sdfInfo == null) { // New SDF, register it
        sdfInfo = new SdfInfo(outputCell, inputCells, name, nextIndex++);
        Debug.Assert(sdfInfo.index == nextIndex - 1);
        sdfNameToInfo[name] = sdfInfo;
        if (sdfInfo.index >= sdfDelegates.Length) {
          Debug.Assert(sdfDelegates.Length == sdfInfos.Length);
          // Reallocate sdfDelegates array
          Delegate[] newSdfs = new Delegate[2 * sdfDelegates.Length];
          Array.Copy(sdfDelegates, newSdfs, sdfDelegates.Length);
          sdfDelegates = newSdfs;
          // Reallocate sdfInfos array
          SdfInfo[] newSdfInfos = new SdfInfo[2 * sdfInfos.Length];
          Array.Copy(sdfInfos, newSdfInfos, sdfInfos.Length);
          sdfInfos = newSdfInfos;
        }
        sdfInfos[sdfInfo.index] = sdfInfo;
        // Update SDF function listbox if created and visible
        GUI.SdfForm sdfForm = System.Windows.Forms.Application.OpenForms["sdf"] as GUI.SdfF
orm;
        if (sdfForm != null && sdfForm.Visible) {
          sdfForm.PopulateFunctionListBox(false);
          sdfForm.PopulateFunctionListBox(name);
          sdfForm.Invalidate();
        }
      }
      return sdfInfo;
    }

    private static void Update(SdfInfo info, Delegate method) {
      sdfDelegates[info.index] = method;
    }

    private static readonly ErrorValue errorDeleted
      = ErrorValue.Make("#FUNERR: Function deleted");
    private static readonly Delegate[] sdfDeleted
      = { (Func<Value>)(delegate { return errorDeleted; }),
          (Func<Value,Value>)(delegate { return errorDeleted; }),
          (Func<Value,Value,Value>)(delegate { return errorDeleted; }),
          (Func<Value,Value,Value,Value>)(delegate { return errorDeleted; }),
          (Func<Value,Value,Value,Value,Value>)(delegate { return errorDeleted; }),
          (Func<Value,Value,Value,Value,Value,Value>)(delegate { return errorDeleted; }),
```



```
          (Func<Value,Value,Value,Value,Value,Value,Value>)(delegate { return errorDeleted;
 }),
          (Func<Value,Value,Value,Value,Value,Value,Value,Value>)(delegate { return errorDe
leted; }),
          (Func<Value,Value,Value,Value,Value,Value,Value,Value,Value>)(delegate { return e
rrorDeleted; }),
          (Func<Value,Value,Value,Value,Value,Value,Value,Value,Value,Value>)(delegate { re
turn errorDeleted; }),
        };

    private static void Unregister(SdfInfo info) {
      sdfNameToInfo.Remove(info.name);
      sdfDelegates[info.index] = sdfDeleted[info.arity];
      sdfInfos[info.index] = null;
    }

    public static SdfInfo GetInfo(String name) {
      SdfInfo info;
      if (sdfNameToInfo.TryGetValue(name.ToUpper(), out info))
        return info;
      else
        return null;
    }

    public static SdfInfo GetInfo(int sdfIndex) {
      return sdfInfos[sdfIndex];
    }

    public static IEnumerable<SdfInfo> GetAllInfos() {
      return sdfNameToInfo.Values;
    }

    /// <summary>
    /// Regenerates the indicated SDF delegates
    /// </summary>
    /// <param name="methods"></param>
    public static void Regenerate(IEnumerable<string> methods) {
      foreach (string s in methods)
        Regenerate(s);
    }

    public static void Regenerate(String name) {
      SdfInfo info = GetInfo(name);
      if (info != null)
        Regenerate(info);
    }

    public static void RegenerateAll() {
      foreach (SdfInfo info in GetAllInfos())
        Regenerate(info);
    }

    /// <summary>
    /// Generate code again for the given SDF with unchanged input and output cells,
    /// recreate the corresponding Function, and update the delegate table entry
    /// </summary>
    /// <param name="info"></param>
    // TODO: This should *not* be applied to SDF's resulting from partial evaluation.
    public static void Regenerate(SdfInfo info) {
      cellToFunctionMapper.RemoveFunction(info);
      // Removing the SDF from Function enables CreateFunction to overwrite it
      Function.Remove(info.name);
      // Rebuild and add it back
      CreateFunction(info.name, info.outputCell, info.inputCells);
    }

    public static void DeleteFunction(string methodName) {
      SdfInfo info = GetInfo(methodName);
      if (info != null) {
        Unregister(info);
        cellToFunctionMapper.RemoveFunction(info);
```

```
        Function.Remove(methodName);
      }
    }

    internal static string[] CheckForModifications(List<FullCellAddr> changedCells) {
      return cellToFunctionMapper.GetFunctionsUsingAddresses(changedCells).ToArray();
    }
  }

  /// <summary>
  /// An SdfInfo instance represents a compiled sheet-defined function (SDF).
  /// </summary>
  public class SdfInfo {
    public readonly FullCellAddr outputCell;
    public readonly FullCellAddr[] inputCells;
    public readonly string name;           // Always upper case
    public readonly int index;             // Index into SdfManager.sdfDelegates
    public readonly int arity;
    public ProgramLines Program { get; set; }
    public bool IsVolatile { get; private set; }

    private static readonly Type[] sdfDelegateType
      = { typeof(Func<Value>),
          typeof(Func<Value,Value>),
          typeof(Func<Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value,Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value,Value,Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value,Value,Value,Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value,Value,Value,Value,Value,Value,Value>),
          typeof(Func<Value,Value,Value,Value,Value,Value,Value,Value,Value,Value,Value>) }
;
    private static readonly MethodInfo[] sdfDelegateInvokeMethods;
    private static readonly Type[][] argumentTypes;

    public static readonly FieldInfo indexField = typeof(SdfInfo).GetField("index");

    static SdfInfo() {
      sdfDelegateInvokeMethods = new MethodInfo[sdfDelegateType.Length];
      for (int i = 0; i < sdfDelegateType.Length; i++)
        sdfDelegateInvokeMethods[i] = sdfDelegateType[i].GetMethod("Invoke");
      argumentTypes = new Type[sdfDelegateType.Length][];
      for (int i = 0; i < argumentTypes.Length; i++) {
        argumentTypes[i] = new Type[i];
        for (int j = 0; j < i; j++)
          argumentTypes[i][j] = Value.type;
      }
    }

    internal SdfInfo(FullCellAddr outputCell, FullCellAddr[] inputCells,
                     string name, int index) {
      this.outputCell = outputCell;
      this.inputCells = inputCells;
      this.name = name.ToUpper();
      this.index = index;
      this.arity = inputCells.Length;
    }

    public Type[] MyArgumentTypes {
      get { return argumentTypes[arity]; }
    }

    public Type MyType {
      get { return sdfDelegateType[arity]; }
    }

    public static Type SdfDelegateType(int n) {
      return sdfDelegateType[n];
    }
```



```
    public MethodInfo MyInvoke {
      get { return sdfDelegateInvokeMethods[arity]; }
    }

    /// <summary>
    /// Determine whether the sheet-defined function involves any volatile cells.
    /// Does not track volatility of normal cells or normal cell areas referred to.
    /// </summary>
    /// <param name="fcas">The set of function-sheet cells making up the function.</param>
    public void SetVolatility(IEnumerable<FullCellAddr> fcas) {
      bool isVolatile = false;
      foreach (FullCellAddr fca in fcas) {
        Cell cell = fca.sheet[fca.ca];
        if (cell != null && cell.IsVolatile) {
          isVolatile = true;
          break;
        }
      }
      this.IsVolatile = isVolatile;
    }

    public Value Apply(Value[] aa) {
      switch (arity) {
        case 0: return Call0();
        case 1: return Call1(aa[0]);
        case 2: return Call2(aa[0], aa[1]);
        case 3: return Call3(aa[0], aa[1], aa[2]);
        case 4: return Call4(aa[0], aa[1], aa[2], aa[3]);
        case 5: return Call5(aa[0], aa[1], aa[2], aa[3], aa[4]);
        case 6: return Call6(aa[0], aa[1], aa[2], aa[3], aa[4], aa[5]);
        case 7: return Call7(aa[0], aa[1], aa[2], aa[3], aa[4], aa[5], aa[6]);
        case 8: return Call8(aa[0], aa[1], aa[2], aa[3], aa[4], aa[5], aa[6], aa[7]);
        case 9: return Call9(aa[0], aa[1], aa[2], aa[3], aa[4], aa[5], aa[6], aa[7], aa[8])
;
        default: return ErrorValue.tooManyArgsError;
      }
    }

    // These don't seem to be used reflexively (for code generation)

    public Value Call0() {
      if (arity == 0)
        return ((Func<Value>)(SdfManager.sdfDelegates[index]))();
      else
        return ErrorValue.argCountError;
    }

    public Value Call1(Value v0) {
      if (arity == 1)
        return ((Func<Value, Value>)(SdfManager.sdfDelegates[index]))(v0);
      else
        return ErrorValue.argCountError;
    }

    public Value Call2(Value v0, Value v1) {
      if (arity == 2)
        return ((Func<Value, Value, Value>)(SdfManager.sdfDelegates[index]))(v0, v1);
      else
        return ErrorValue.argCountError;
    }

    public Value Call3(Value v0, Value v1, Value v2) {
      if (arity == 3)
        return ((Func<Value, Value, Value, Value>)(SdfManager.sdfDelegates[index]))(v0, v1,
 v2);
      else
        return ErrorValue.argCountError;
    }

    public Value Call4(Value v0, Value v1, Value v2, Value v3) {
```

```
      if (arity == 4)
        return ((Func<Value, Value, Value, Value, Value>)(SdfManager.sdfDelegates[index]))(
v0, v1, v2, v3);
      else
        return ErrorValue.argCountError;
    }

    public Value Call5(Value v0, Value v1, Value v2, Value v3, Value v4) {
      if (arity == 5)
        return ((Func<Value, Value, Value, Value, Value, Value>)(SdfManager.sdfDelegates[in
dex]))(v0, v1, v2, v3, v4);
      else
        return ErrorValue.argCountError;
    }

    public Value Call6(Value v0, Value v1, Value v2, Value v3, Value v4, Value v5) {
      if (arity == 6)
        return ((Func<Value, Value, Value, Value, Value, Value, Value>)(SdfManager.sdfDeleg
ates[index]))(v0, v1, v2, v3, v4, v5);
      else
        return ErrorValue.argCountError;
    }

    public Value Call7(Value v0, Value v1, Value v2, Value v3, Value v4, Value v5, Value v6
) {
      if (arity == 7)
        return ((Func<Value, Value, Value, Value, Value, Value, Value, Value>)(SdfManager.s
dfDelegates[index]))(v0, v1, v2, v3, v4, v5, v6);
      else
        return ErrorValue.argCountError;
    }

    public Value Call8(Value v0, Value v1, Value v2, Value v3, Value v4, Value v5, Value v6
, Value v7) {
      if (arity == 8)
        return ((Func<Value, Value, Value, Value, Value, Value, Value, Value, Value>)(SdfMa
nager.sdfDelegates[index]))(v0, v1, v2, v3, v4, v5, v6, v7);
      else
        return ErrorValue.argCountError;
    }

    public Value Call9(Value v0, Value v1, Value v2, Value v3, Value v4, Value v5, Value v6
, Value v7, Value v8) {
      if (arity == 9)
        return ((Func<Value, Value, Value, Value, Value, Value, Value, Value, Value, Value>
)(SdfManager.sdfDelegates[index]))(v0, v1, v2, v3, v4, v5, v6, v7, v8);
      else
        return ErrorValue.argCountError;
    }

    public Value Apply(Sheet sheet, Expr[] es, int col, int row) {
      // Arity is checked in the SdfInfo.CallN methods
      switch (es.Length) {
        case 0:
          return Call0();
        case 1:
          return Call1(es[0].Eval(sheet, col, row));
        case 2:
          return Call2(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row));
        case 3:
          return Call3(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row),
                       es[2].Eval(sheet, col, row));
        case 4:
          return Call4(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row),
                       es[2].Eval(sheet, col, row), es[3].Eval(sheet, col, row));
        case 5:
          return Call5(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row),
                       es[2].Eval(sheet, col, row), es[3].Eval(sheet, col, row),
                       es[4].Eval(sheet, col, row));
        case 6:
          return Call6(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row),
```



```
                       es[2].Eval(sheet, col, row), es[3].Eval(sheet, col, row),
                       es[4].Eval(sheet, col, row), es[5].Eval(sheet, col, row));
        case 7:
          return Call7(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row),
                       es[2].Eval(sheet, col, row), es[3].Eval(sheet, col, row),
                       es[4].Eval(sheet, col, row), es[5].Eval(sheet, col, row),
                       es[6].Eval(sheet, col, row));
        case 8:
          return Call8(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row),
                       es[2].Eval(sheet, col, row), es[3].Eval(sheet, col, row),
                       es[4].Eval(sheet, col, row), es[5].Eval(sheet, col, row),
                       es[6].Eval(sheet, col, row), es[7].Eval(sheet, col, row));
        case 9:
          return Call9(es[0].Eval(sheet, col, row), es[1].Eval(sheet, col, row),
                       es[2].Eval(sheet, col, row), es[3].Eval(sheet, col, row),
                       es[4].Eval(sheet, col, row), es[5].Eval(sheet, col, row),
                       es[6].Eval(sheet, col, row), es[7].Eval(sheet, col, row),
                       es[8].Eval(sheet, col, row));
        default:
          return ErrorValue.Make("#FUNERR: Too many arguments");
      }
    }

    public override string ToString() {
      return String.Format("FUN {0} AT #{1}", name, index);
    }
  }
}
```

```
ï»¿// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using System.Reflection.Emit;
using System.Linq;

namespace Corecalc.Funcalc {
  /// <summary>
  /// An SdfType represents and parses signatures of sheet-defined functions
  /// as well as .NET Class Library (external) methods.
  /// </summary>
  public abstract class SdfType {
    public abstract Type GetDotNetType();

    /// <summary>
    /// A SigToken is a lexical used in the signature of an external function.
    /// </summary>
    private abstract class SigToken {
      public readonly static SigToken
        LPAR = new LPar(),
        RPAR = new RPar(),
        LBRK = new LBrk(),
        LBRE = new LBre(),
        EOF = new Eof();
    }

    private class LPar : SigToken { }   // (
    private class RPar : SigToken { }   // )
    private class LBrk : SigToken { }   // [
    private class LBre : SigToken { }   // {
    private class Eof  : SigToken { }

    /// <summary>
    /// A TypenameToken represents a type name in a signature.
    /// </summary>
    private class TypenameToken : SigToken {
      public readonly String typename;
      public TypenameToken(String typename) {
        this.typename = typename;
      }
      public override string ToString() {
        return typename;
      }
    }
```



```
    /// <summary>
    /// An ErrorToken represents an error in lexical analysis of a signature.
    /// </summary>
    private class ErrorToken : SigToken {
      public readonly String message;
      public ErrorToken(String message) {
        this.message = message;
      }
    }

    // ParseType: from stream of signature tokens to a Type object.

    public static SdfType ParseType(String signature) {
      IEnumerator<SigToken> tokens = Scanner(signature);
      tokens.MoveNext();  // There's at least Eof in the stream
      SdfType res = ParseOneType(tokens);
      if (tokens.Current is Eof)
        return res;
      else
        throw new SigParseException("Extraneous characters in signature");
    }

    // Before the call, tokens.Current is the first token of the type
    // to parse; and after the call it is the first token after that type.

    private static SdfType ParseOneType(IEnumerator<SigToken> tokens) {
      if (tokens.Current is LPar) {
        tokens.MoveNext();
        return ParseFunctionSignature(tokens);
      } else if (tokens.Current is LBrk) {
        tokens.MoveNext();
        return ParseArraySignature(tokens, 1);
      } else if (tokens.Current is LBre) {
        tokens.MoveNext();
        return ParseArraySignature(tokens, 2);
      } else if (tokens.Current is TypenameToken) {
        TypenameToken token = tokens.Current as TypenameToken;
        tokens.MoveNext();
        switch (token.typename) {
          case "Z": return new SimpleType(typeof(System.Boolean));
          case "C": return new SimpleType(typeof(System.Char));
          case "B": return new SimpleType(typeof(System.SByte));
          case "b": return new SimpleType(typeof(System.Byte));
          case "S": return new SimpleType(typeof(System.Int16));
          case "s": return new SimpleType(typeof(System.UInt16));
          case "I": return new SimpleType(typeof(System.Int32));
          case "i": return new SimpleType(typeof(System.UInt32));
          case "J": return new SimpleType(typeof(System.Int64));
          case "j": return new SimpleType(typeof(System.UInt64));
          case "F": return new SimpleType(typeof(System.Single));
          case "D":
          case "N": return new SimpleType(typeof(System.Double));
          case "M": return new SimpleType(typeof(System.Decimal));
          case "V": return new SimpleType(Value.type);
          case "W": return new SimpleType(typeof(void));
          case "T": return new SimpleType(typeof(System.String));
          case "O": return new SimpleType(typeof(System.Object));
          default:
            return new SimpleType(ExternalFunction.FindType(token.typename));
        }
      } else if (tokens.Current is Eof)
        throw new SigParseException("Unexpected end of signature");
      else
        throw new SigParseException("Unexpected token " + tokens.Current);
    }

    private static SdfType ParseFunctionSignature(IEnumerator<SigToken> tokens) {
      List<SdfType> arguments = new List<SdfType>();
      while (!(tokens.Current is Eof) && !(tokens.Current is RPar))
        arguments.Add(ParseOneType(tokens));
      if (tokens.Current is RPar)
```

```
      tokens.MoveNext();
    else
      throw new SigParseException("Unexpected end of function signature");
    SdfType returntype = ParseOneType(tokens);
    return new FunctionType(arguments.ToArray(), returntype);
  }

  private static SdfType ParseArraySignature(IEnumerator<SigToken> tokens, int dim) {
    if (tokens.Current is Eof)
      throw new SigParseException("Unexpected end of function signature");
    SdfType elementtype = ParseOneType(tokens);
    return new ArrayType(elementtype, dim);
  }

  /// <summary>
  /// A SigParseException signals an error during parsing of a function signature.
  /// </summary>
  public class SigParseException : Exception {
    public SigParseException(String message) : base(message) { }
  }

  // Scanner: from signature string to stream of signature tokens

  private static IEnumerator<SigToken> Scanner(String signature) {
    int i = 0;
    while (i < signature.Length) {
      char ch = signature[i];
      switch (ch) {
        case 'Z':
        case 'C':
        case 'B':
        case 'b':
        case 'S':
        case 's':
        case 'I':
        case 'i':
        case 'J':
        case 'j':
        case 'D':
        case 'N':
        case 'F':
        case 'M':
        case 'V':
        case 'W':
        case 'T':
        case 'O':
          yield return new TypenameToken(signature.Substring(i, 1));
          break;
        case 'L': // For instance, LSystem.Text.StringBuilder;
          i++;
          int start = i;
          while (i < signature.Length && signature[i] != ';')
            i++;
          // Now signature[i]==';' or i == signature.Length
          if (i < signature.Length)
            yield return new TypenameToken(signature.Substring(start, i - start));
          else
            yield return new ErrorToken("Unterminated class name");
          break;
        case '(':
          yield return SigToken.LPAR;
          break;
        case ')':
          yield return SigToken.RPAR;
          break;
        case '[':
          yield return SigToken.LBRK;
          break;
        case '{':
          yield return SigToken.LBRE;
          break;
```



```
        default:
          yield return new ErrorToken("Illegal character '" + ch + "'");
          break;
      }
      i++;
    }
    yield return SigToken.EOF;
    yield break;
  }
}


/// <summary>
/// A SimpleType is simple (non-composite) type such as Double, String, StringBuilder.
/// </summary>
public class SimpleType : SdfType {
  public readonly Type type;

  public SimpleType(Type type) {
    this.type = type;
  }

  public override Type GetDotNetType() {
    return type;
  }

  public override string ToString() {
    return type.ToString();
  }
}

/// <summary>
/// A FunctionType is the type of a sheet-defined function,
/// such as Number * Text -> Value.
/// </summary>
public class FunctionType : SdfType {
  public readonly SdfType[] arguments;
  public readonly SdfType returntype;

  public FunctionType(SdfType[] arguments, SdfType returntype) {
    this.arguments = arguments;
    this.returntype = returntype;
  }

  public Type[] ArgumentDotNetTypes() {
    Type[] res = new Type[arguments.Length];
    for (int i = 0; i < arguments.Length; i++)
      res[i] = arguments[i].GetDotNetType();
    return res;
  }

  public override Type GetDotNetType() {
    throw new Exception("Function type not allowed");
  }

  public override String ToString() {
    StringBuilder sb = new StringBuilder();
    sb.Append("(");
    if (arguments.Length > 0)
      sb.Append(arguments[0]);
    for (int i = 1; i < arguments.Length; i++)
      sb.Append(" * ").Append(arguments[i]);
    sb.Append(" -> ").Append(returntype).Append(")");
    return sb.ToString();
  }
}

/// <summary>
/// An ArrayType is a CLI array type such as String[] or String[,]
/// or a spreadsheet array type such as Number array.
/// </summary>
```

```
public class ArrayType : SdfType {
  public readonly SdfType elementtype;
  public readonly int dim;

  public ArrayType(SdfType elementtype, int dim) {
    this.elementtype = elementtype;
    this.dim = dim;
  }

  public override Type GetDotNetType() {
    if (dim == 1) // Special case, see System.Type.MakeArrayType(int)
      return elementtype.GetDotNetType().MakeArrayType();
    else
      return elementtype.GetDotNetType().MakeArrayType(dim);
  }

  public override String ToString() {
    return elementtype + (dim == 1 ? "[]" : dim == 2 ? "[,]" : "UNHANDLED");
  }
}

/// <summary>
/// An ExternalFunction represents an external .NET function, conversion
/// of its arguments from spreadsheet Values to .NET values, conversion of
/// its result value in the opposite direction, its .NET MethodInfo for
/// invoking it, its .NET signature, and more.
/// </summary>
class ExternalFunction {
  private readonly Func<Value, Object>[] argConverters;
  private readonly Func<Object, Value> resConverter;
  private readonly MethodBase mcInfo;     // MethodInfo or ConstructorInfo
  private readonly Type[] argTypes;       // Does not include receiver type
  private readonly Object[] argValues;    // Reused from call to call
  private readonly Type resType;          // Result type
  private readonly bool isStatic;
  private readonly Type recType;                        // Receiver type
  private readonly Func<Value, Object> recConverter;  // Receiver converter
  public readonly int arity;              // Includes receiver if !isStatic

  // Invariant: argTypes.Length == argConverters.Length == argValues.Length
  // Invariant: if isStatic then arity==argTypes.Length else arity==argTypes.Length+1
  // Invariant: if !isStatic then recType != null and recConverter != null
  // Invariant: argConverters[i] == toObjectConverter[argTypes[i]]
  // Invariant: recConverter == null || recConverter = toObjectConverter[recType]
  // Invariant: resConverter == fromObjectConverter[resType]

  // Mapping .NET types to argument and result converters: Value <-> Object
  private static readonly IDictionary<Type, Func<Value, Object>> toObjectConverter
    = new Dictionary<Type, Func<Value, Object>>();
  private static IDictionary<Type, Func<Object, Value>> fromObjectConverter
    = new Dictionary<Type, Func<Object, Value>>();

  static ExternalFunction() {
    // Initialize tables of conversions
    // From Funcalc type to .NET type, for argument converters
    toObjectConverter.Add(typeof(System.Int64), NumberValue.ToInt64);
    toObjectConverter.Add(typeof(System.Int32), NumberValue.ToInt32);
    toObjectConverter.Add(typeof(System.Int16), NumberValue.ToInt16);
    toObjectConverter.Add(typeof(System.SByte), NumberValue.ToSByte);
    toObjectConverter.Add(typeof(System.UInt64), NumberValue.ToUInt64);
    toObjectConverter.Add(typeof(System.UInt32), NumberValue.ToUInt32);
    toObjectConverter.Add(typeof(System.UInt16), NumberValue.ToUInt16);
    toObjectConverter.Add(typeof(System.Byte), NumberValue.ToByte);
    toObjectConverter.Add(typeof(System.Double), NumberValue.ToDouble);
    toObjectConverter.Add(typeof(System.Single), NumberValue.ToSingle);
    toObjectConverter.Add(typeof(System.Boolean), NumberValue.ToBoolean);
    toObjectConverter.Add(typeof(System.String), TextValue.ToString);
    toObjectConverter.Add(typeof(System.Char),   TextValue.ToChar);
    toObjectConverter.Add(typeof(System.Object), Value.ToObject);
    toObjectConverter.Add(typeof(System.Double[]), ArrayValue.ToDoubleArray1D);
    toObjectConverter.Add(typeof(System.Double[,]), ArrayValue.ToDoubleArray2D);
```



```
    toObjectConverter.Add(typeof(System.String[]), ArrayValue.ToStringArray1D);

    // From .NET type to Funcalc type, for result converters
    fromObjectConverter.Add(typeof(System.Int64), NumberValue.FromInt64);
    fromObjectConverter.Add(typeof(System.Int32), NumberValue.FromInt32);
    fromObjectConverter.Add(typeof(System.Int16), NumberValue.FromInt16);
    fromObjectConverter.Add(typeof(System.SByte), NumberValue.FromSByte);
    fromObjectConverter.Add(typeof(System.UInt64), NumberValue.FromUInt64);
    fromObjectConverter.Add(typeof(System.UInt32), NumberValue.FromUInt32);
    fromObjectConverter.Add(typeof(System.UInt16), NumberValue.FromUInt16);
    fromObjectConverter.Add(typeof(System.Byte), NumberValue.FromByte);
    fromObjectConverter.Add(typeof(System.Double), NumberValue.FromDouble);
    fromObjectConverter.Add(typeof(System.Single), NumberValue.FromSingle);
    fromObjectConverter.Add(typeof(System.Boolean), NumberValue.FromBoolean);
    fromObjectConverter.Add(typeof(System.String), TextValue.FromString);
    fromObjectConverter.Add(typeof(System.Char),   TextValue.FromChar);
    fromObjectConverter.Add(typeof(System.Object), ObjectValue.Make);
    fromObjectConverter.Add(typeof(System.String[]), ArrayValue.FromStringArray1D);
    fromObjectConverter.Add(typeof(System.Double[]), ArrayValue.FromDoubleArray1D);
    fromObjectConverter.Add(typeof(System.Double[,]), ArrayValue.FromDoubleArray2D);
    fromObjectConverter.Add(typeof(void), Value.MakeVoid);
  }

  // For caching the result of nameAndSignature lookups
  private static readonly IDictionary<String, ExternalFunction> cache
   = new Dictionary<String, ExternalFunction>();

  public static ExternalFunction Make(String nameAndSignature) {
    ExternalFunction res;
    if (!cache.TryGetValue(nameAndSignature, out res)) {
      res = new ExternalFunction(nameAndSignature);
      cache.Add(nameAndSignature, res);
    }
    return res;
  }

  private ExternalFunction(String nameAndSignature) {
    int firstParen = nameAndSignature.IndexOf('(');
    if (firstParen <= 0 || firstParen == nameAndSignature.Length - 1)
      throw new Exception("#ERR: Ill-formed name and signature");
    isStatic = nameAndSignature[firstParen - 1] == '$';
    String name = nameAndSignature.Substring(0, isStatic ? firstParen - 1 : firstParen);
    String signature = nameAndSignature.Substring(firstParen);
    int lastDot = name.LastIndexOf('.');
    if (lastDot <= 0 || lastDot == name.Length - 1)
      throw new Exception("#ERR: Ill-formed .NET method name");
    String typeName = name.Substring(0, lastDot);
    String methodName = name.Substring(lastDot + 1);
    // Experimental: Search appdomain's assemblies
    Type declaringType = FindType(typeName);
    if (declaringType == null)
      throw new Exception("#ERR: Unknown .NET type " + typeName);
    FunctionType ft = SdfType.ParseType(signature) as FunctionType;
    if (ft == null)
      throw new Exception("#ERR: Ill-formed .NET method signature");
    argTypes = ft.ArgumentDotNetTypes();
    resType = ft.returntype.GetDotNetType();
    if (methodName == "new" && isStatic)
      mcInfo = declaringType.GetConstructor(argTypes);
    else
      mcInfo = declaringType.GetMethod(methodName, argTypes);
    if (mcInfo == null)
      throw new Exception("#ERR: Unknown .NET method");
    argConverters = new Func<Value, Object>[argTypes.Length];
    for (int i = 0; i < argTypes.Length; i++)
      argConverters[i] = GetToObjectConverter(argTypes[i]);
    resConverter = GetFromObjectConverter(ft.returntype.GetDotNetType());
    if (isStatic)
      arity = argTypes.Length;
    else {
      arity = argTypes.Length + 1;
```

```
      recType = declaringType;
      recConverter = GetToObjectConverter(recType);
    }
    argValues = new Object[argTypes.Length];   // Allocate once, reuse at calls
  }

  private static Func<Value, Object> GetToObjectConverter(Type typ) {
    if (toObjectConverter.ContainsKey(typ))
      return toObjectConverter[typ];
    else
      return ObjectValue.ToObject;
      //return delegate(Value v) {
      //  ObjectValue o = v as ObjectValue;
      //  return o != null && typ.IsInstanceOfType(o.value) ? o.value : null;
      //};
  }

  private static Func<Object, Value> GetFromObjectConverter(Type typ) {
    if (fromObjectConverter.ContainsKey(typ))
      return fromObjectConverter[typ];
    else
      return ObjectValue.Make;
      //return delegate(Object o) {
      //  return typ.IsInstanceOfType(o) ? ObjectValue.Make(o) : ErrorValue.argTypeError;
      //};
  }

  // Experimental: Search for type by name in some known assemblies

  private static readonly String[] assmNames = new String[] {
    "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089",
    "System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089",
    // This is for testing EXTERN functions:
    "Externals, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
  };

  public static Type FindType(String typeName) {
    Type declaringType = null;
    foreach (String assmName in assmNames) {
      Assembly assm = Assembly.Load(assmName);
      // Console.WriteLine(assm);
      declaringType = assm.GetType(typeName);
      if (declaringType != null)
        break;
    }
    return declaringType;
  }

  // Called from the EXTERN function applier in interpreted sheets.

  public Value Call(Value[] vs) {
    if (vs.Length != arity)
      return ErrorValue.argCountError;
    Object receiver;
    if (isStatic) {
      receiver = null;
      for (int i = 0; i < vs.Length; i++)
        argValues[i] = argConverters[i](vs[i]);
    } else {
      receiver = recConverter(vs[0]);
      for (int i = 1; i < vs.Length; i++)
        argValues[i - 1] = argConverters[i - 1](vs[i]);
    }
    if (mcInfo is ConstructorInfo)
      return resConverter((mcInfo as ConstructorInfo).Invoke(argValues));
    else
      return resConverter(mcInfo.Invoke(receiver, argValues));
  }

  // These are used by the SDF code generator in CGExtern
```



```
    public void EmitCall(ILGenerator ilg) {
      if (mcInfo is ConstructorInfo)
        ilg.Emit(OpCodes.Newobj, mcInfo as ConstructorInfo);
      else if (isStatic)
        ilg.Emit(OpCodes.Call, mcInfo as MethodInfo);
      else
        ilg.Emit(OpCodes.Call, mcInfo as MethodInfo);
    }

    public Type ArgType(int i) {
      return isStatic ? argTypes[i] : i > 0 ? argTypes[i - 1] : recType;
    }

    public Func<Value, Object> ArgConverter(int i) {
      return isStatic ? argConverters[i] : i > 0 ? argConverters[i - 1] : recConverter;
    }

    public Type ResType {
      get { return resType; }
    }

    public Func<Object, Value> ResConverter {
      get { return resConverter; }
    }
  }
}
```

```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Reflection.Emit;
using System.Text;

namespace Corecalc.Funcalc {
  /// <summary>
  /// A Variable represents a cell of a sheet-defined function, whether
  /// an argument (input cell) or a computed cell (intermediate or output cell).
  /// </summary>
  public abstract class Variable : IEquatable<Variable> {
    private readonly string name;
    private readonly Typ type;

    public Variable(String name, Typ type) {
      this.name = name;
      this.type = type;
    }

    public bool Equals(Variable that) {
      return this == that;
    }

    public override bool Equals(Object obj) {
      return Equals(obj as Variable);
    }

    public override int GetHashCode() {
      return base.GetHashCode();
    }

    public abstract void EmitLoad(ILGenerator ilg);

    public abstract void EmitStore(ILGenerator ilg);

    public abstract Variable Fresh();

    public String Name {
      get { return name; }
    }

    public Typ Type {
      get { return type; }
    }
  }
```



```
  /// <summary>
  /// A LocalVariable represents a .NET IL variable holding a computed
  /// (intermediate or output) cell of a sheet-defined function.
  /// </summary>
  public class LocalVariable : Variable {
    private LocalBuilder localBuilder;  // null until var emitted

    public LocalVariable(String name, Typ type)
      : base(name, type) { }

    private LocalBuilder GetLocalBuilder(ILGenerator ilg) {
      if (localBuilder == null) {
        if (Type == Typ.Number)
          this.localBuilder = ilg.DeclareLocal(typeof(double));
        else
          this.localBuilder = ilg.DeclareLocal(Value.type);
      }
      return localBuilder;
    }

    public override void EmitLoad(ILGenerator ilg) {
      ilg.Emit(OpCodes.Ldloc, GetLocalBuilder(ilg));
    }

    public override void EmitStore(ILGenerator ilg) {
      ilg.Emit(OpCodes.Stloc, GetLocalBuilder(ilg));
    }

    public override Variable Fresh() {
      return new LocalVariable(Name, Type);
    }
  }

  /// <summary>
  /// A LocalArgument represents a .NET IL function argument holding
  /// an input of a sheet-defined function.
  /// </summary>
  public class LocalArgument : Variable {
    private readonly short argumentNumber;

    public LocalArgument(String name, Typ type, short argumentNumber)
      : base(name, type) {
      this.argumentNumber = argumentNumber;
    }

    public override void EmitLoad(ILGenerator ilg) {
      ilg.Emit(OpCodes.Ldarg, argumentNumber);
    }

    public override void EmitStore(ILGenerator ilg) {
      throw new ImpossibleException("LocalArgument.EmitStore unexpected");
    }

    public override Variable Fresh() {
      throw new ImpossibleException("LocalArgument.Fresh()");
    }
  }
}
```

```
// Corecalc, a spreadsheet core implementation

// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Text;
using Corecalc.Funcalc;    // ExternalFunction, SdfInfo, SdfManager ...
using System.Diagnostics;
using System.Reflection;   // TargetInvocationException
using System.Linq;

namespace Corecalc {
  /// <summary>
  /// A Function represents a built-in function or operator, or a sheet-defined function.
  /// </summary>
  public class Function {
    public readonly String name;
    public Applier Applier { get; private set; }
    public readonly int fixity;                       // If non-zero: operator, precedence
    public bool IsPlaceHolder { get; private set; }  // May be overwritten by an SDF
    private bool isVolatile;                          // True for RAND, NOW, some SDFs
    private static readonly IDictionary<String, Function> table;

    public static readonly Type type = typeof(Function);  // Used in IL code generation

    private static readonly Random random = new Random();

    // Used by the SDF machinery to update a placeholder function
    public void UpdateApplier(Applier applier, bool isVolatile) {
      Debug.Assert(IsPlaceHolder);
      this.Applier = applier;
      this.IsPlaceHolder = false;
      this.isVolatile = isVolatile;
    }

    // The following methods are used also from compiled SDFs:

    public static double Average(Value[] vs) {
      // May consider whether empty cells and texts should just
      // be ignored instead of given ArgTypeError.
      // We're using Kahan's accurate sum algorithm; see Goldberg 1991.
      double S = 0.0, C = 0.0;
      int count = 0;
      foreach (Value outerV in vs)
        outerV.Apply(delegate(Value v) {
          double Y = NumberValue.ToDoubleOrNan(v) - C, T = S + Y;
          C = (T - S) - Y;
          S = T;
```



```
          count++;
        });
      return S / count;
    }

    public static Value Benchmark(Value v0, Value v1) {
      if (v0 is ErrorValue) return v0;
      if (v1 is ErrorValue) return v1;
      FunctionValue fv = v0 as FunctionValue;
      NumberValue n1 = v1 as NumberValue;
      if (fv == null || n1 == null)
        return ErrorValue.argTypeError;
      int count = (int)(n1.value);
      if (count <= 0)
        return ErrorValue.numError;
      // The following replicates some of fun.Call0(), to lift
      // array unwrapping (although not arity checks and a cast)
      // out of the timing loops
      if (fv.Arity != 0)
        return ErrorValue.argCountError;
      SdfInfo sdfInfo = fv.sdfInfo;
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Reset();
      stopwatch.Start();
      switch (sdfInfo.arity) {
        case 0:
          for (int i = count; i > 0; i--)
            sdfInfo.Call0();
          break;
        case 1: {
            Value arg0 = fv.args[0];
            for (int i = count; i > 0; i--)
              sdfInfo.Call1(arg0);
            break;
          }
        case 2: {
            Value arg0 = fv.args[0], arg1 = fv.args[1];
            for (int i = count; i > 0; i--)
              sdfInfo.Call2(arg0, arg1);
            break;
          }
        case 3: {
            Value arg0 = fv.args[0], arg1 = fv.args[1], arg2 = fv.args[2];
            for (int i = count; i > 0; i--)
              sdfInfo.Call3(arg0, arg1, arg2);
            break;
          }
        case 4: {
            Value arg0 = fv.args[0], arg1 = fv.args[1],
              arg2 = fv.args[2], arg3 = fv.args[3];
            for (int i = count; i > 0; i--)
              sdfInfo.Call4(arg0, arg1, arg2, arg3);
            break;
          }
        case 5: {
            Value arg0 = fv.args[0], arg1 = fv.args[1], arg2 = fv.args[2],
              arg3 = fv.args[3], arg4 = fv.args[4];
            for (int i = count; i > 0; i--)
              sdfInfo.Call5(arg0, arg1, arg2, arg3, arg4);
            break;
          }
        case 6: {
            Value arg0 = fv.args[0], arg1 = fv.args[1], arg2 = fv.args[2],
              arg3 = fv.args[3], arg4 = fv.args[4], arg5 = fv.args[5];
            for (int i = count; i > 0; i--)
              sdfInfo.Call6(arg0, arg1, arg2, arg3, arg4, arg5);
            break;
          }
        case 7: {
            Value arg0 = fv.args[0], arg1 = fv.args[1], arg2 = fv.args[2],
              arg3 = fv.args[3], arg4 = fv.args[4], arg5 = fv.args[5],
```

```
            arg6 = fv.args[6];
          for (int i = count; i > 0; i--)
            sdfInfo.Call7(arg0, arg1, arg2, arg3, arg4, arg5, arg6);
          break;
        }
      case 8: {
          Value arg0 = fv.args[0], arg1 = fv.args[1], arg2 = fv.args[2],
            arg3 = fv.args[3], arg4 = fv.args[4], arg5 = fv.args[5],
            arg6 = fv.args[6], arg7 = fv.args[7];
          for (int i = count; i > 0; i--)
            sdfInfo.Call8(arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7);
          break;
        }
      case 9: {
          Value arg0 = fv.args[0], arg1 = fv.args[1], arg2 = fv.args[2],
            arg3 = fv.args[3], arg4 = fv.args[4], arg5 = fv.args[5],
            arg6 = fv.args[6], arg7 = fv.args[7], arg8 = fv.args[8];
          for (int i = count; i > 0; i--)
            sdfInfo.Call9(arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8);
          break;
        }
      default:
        return ErrorValue.tooManyArgsError;
    }
    stopwatch.Stop();
    return NumberValue.Make((1E6 * stopwatch.ElapsedMilliseconds) / count);
  }

  public static Value ColMap(Value v0, Value v1) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if (v0 is FunctionValue && v1 is ArrayValue) {
      FunctionValue fv = v0 as FunctionValue;
      ArrayValue arr = v1 as ArrayValue;
      if (fv.Arity != arr.Rows)
        return ErrorValue.argCountError;
      Value[,] result = new Value[arr.Cols, 1];
      Value[] arguments = new Value[arr.Rows];
      for (int c = 0; c < arr.Cols; c++) {
        for (int r = 0; r < arr.Rows; r++)
          arguments[r] = arr[c, r];
        result[c, 0] = fv.Apply(arguments);
      }
      return new ArrayExplicit(result);
    } else
      return ErrorValue.argTypeError;
  }

  public static double Columns(Value v0) {
    if (v0 is ErrorValue) return (v0 as ErrorValue).ErrorNan;
    ArrayValue v0arr = v0 as ArrayValue;
    if (v0arr != null)
      return v0arr.Cols;
    else
      return ErrorValue.argTypeError.ErrorNan;
  }

  public static Value ConstArray(Value v0, Value v1, Value v2) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if (v2 is ErrorValue) return v2;
    if (v1 is NumberValue && v2 is NumberValue) {
      int rows = (int)(v1 as NumberValue).value,
          cols = (int)(v2 as NumberValue).value;
      if (0 <= rows && 0 <= cols) {
        Value[,] result = new Value[cols, rows];
        for (int c = 0; c < cols; c++)
          for (int r = 0; r < rows; r++)
            result[c, r] = v0;
        return new ArrayExplicit(result);
      } else
```



```
        return ErrorValue.Make("#ERR: Size");
    } else
      return ErrorValue.argTypeError;
  }

  public static Value CountIf(Value v0, Value v1) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    FunctionValue fv = v0 as FunctionValue;
    if (fv == null)
      return ErrorValue.argTypeError;
    if (fv.Arity != 1)
      return ErrorValue.argCountError;
    double count = 0.0;
    v1.Apply(delegate(Value v) {
      if (!Double.IsNaN(count)) {
        double condition = NumberValue.ToDoubleOrNan(fv.Call1(v));
        if (Double.IsNaN(condition))
          count = condition; // Hack: Error propagation from predicate
        else if (condition != 0)
          count++;
      }
    });
    return NumberValue.Make(count);
  }

  public static double Equal(Value e1, Value e2) {
    if (e1 is ErrorValue) return (e1 as ErrorValue).ErrorNan;
    if (e2 is ErrorValue) return (e2 as ErrorValue).ErrorNan;
    return e1 == e2 || e1 != null && e1.Equals(e2) ? 1.0 : 0.0;
  }

  public static double ExcelAtan2(double x, double y) {
    return Math.Atan2(y, x);      // Note swapped arguments
  }

  public static double ExcelCeiling(double d, double signif) {
    return signif * Math.Ceiling(d / signif);
  }

  public static Value ExcelConcat(Value v0, Value v1) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if ((v0 is TextValue || v0 is NumberValue) && (v1 is TextValue || v1 is NumberValue))
{
      String s0 = v0.ToString(), s1 = v1.ToString();
      // Avoid creating new String or TextValue objects when possible
      if (s0 == "")
        return v1 as TextValue ?? TextValue.Make(s1);
      else if (s1 == "")
        return v0 as TextValue ?? TextValue.Make(s0);
      else
        return TextValue.Make(s0 + s1);
    } else
      return ErrorValue.argTypeError;
  }

  public static double ExcelFloor(double d, double signif) {
    return signif * Math.Floor(d / signif);
  }

  public static double ExcelMod(double x, double y) {
    return x - y * Math.Floor(x / y);
  }

  public static double ExcelNow() {
    return NumberValue.DoubleFromDateTimeTicks(DateTime.Now.Ticks);
  }

  public static double ExcelPow(double x, double y) {
    // Necessary because MS .NET and Mono Math.Pow is wrong
```

```
    return double.IsNaN(x) ? x : double.IsNaN(y) ? y : Math.Pow(x, y);
  }

  public static double ExcelRand() {
    return random.NextDouble();
  }

  public static double ExcelRound(double d, double digits) {
    if (Double.IsNaN(digits)) return digits;
    int idigits = (int)digits; // Truncation towards zero is correct
    if (idigits >= 0)
      return Math.Round(d, idigits, MidpointRounding.AwayFromZero);
    else {
      double scale = Math.Pow(10, -idigits);
      return scale * Math.Round(d / scale, 0);
    }
  }

  public static Value HArray(Value[] vs) {
    Value[,] result = new Value[vs.Length, 1];
    for (int c = 0; c < vs.Length; c++)
      if (vs[c] is ErrorValue)
        return vs[c];
      else
        result[c, 0] = vs[c];
    return new ArrayExplicit(result);
  }

  // Make array as horizontal (side-by-side) concatenation of arguments' columns
  public static Value HCat(Value[] vs) {
    int rows = 0, cols = 0;
    foreach (Value v in vs)
      if (v is ErrorValue)
        return v;
      else if (v is ArrayValue) {
        rows = Math.Max(rows, (v as ArrayValue).Rows);
        cols += (v as ArrayValue).Cols;
      } else {
        rows = Math.Max(rows, 1);
        cols += 1;
      }
    foreach (Value v in vs)
      if (v is ArrayValue && (v as ArrayValue).Rows != rows)
        return ErrorValue.Make("#ERR: Row counts differ");
    Value[,] result = new Value[cols, rows];
    int nextCol = 0;
    foreach (Value v in vs)
      if (v is ArrayValue) {
        ArrayValue arr = v as ArrayValue;
        for (int c = 0; c < arr.Cols; c++) {
          for (int r = 0; r < rows; r++)
            result[nextCol, r] = arr[c, r];
          nextCol++;
        }
      } else {
        for (int r = 0; r < rows; r++)
          result[nextCol, r] = v;
        nextCol++;
      }
    return new ArrayExplicit(result);
  }

  // Return horizontal array with nv+1 columns cv, fv(cv), fv(fv(cv)), ...
  public static Value HScan(Value v0, Value v1, Value v2) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if (v2 is ErrorValue) return v2;
    FunctionValue fv = v0 as FunctionValue;
    ArrayValue cv = v1 as ArrayValue;
    NumberValue nv = v2 as NumberValue;
    if (fv == null || cv == null || nv == null)
```



```
      return ErrorValue.argTypeError;
    else {
      int n = (int)nv.value;
      int rows = cv.Rows;
      if (n < 0 || rows == 0 || cv.Cols != 1)
        return ErrorValue.Make("#ERR: Argument value in HSCAN");
      else {
        Value[,] result = new Value[n+1, rows];
        for (int r=0; r<rows; r++)
          result[0, r] = cv[0, r];
        for (int c=1; c<=n; c++) {
          cv = fv.Call1(cv) as ArrayValue;
          if (cv == null)
            return ErrorValue.argTypeError;
          if (cv.Cols != 1 || cv.Rows != rows)
            return ErrorValue.Make("#ERR: Result shape in HSCAN");
          for (int r=0; r<rows; r++)
            result[c, r] = cv[0, r];
        }
        return new ArrayExplicit(result);
      }
    }
  }

  public static Value Index(Value v0, double r, double c) {
    if (v0 is ErrorValue) return v0;
    if (Double.IsNaN(r)) return NumberValue.Make(r);
    if (Double.IsNaN(c)) return NumberValue.Make(c);
    ArrayValue arr = v0 as ArrayValue;
    if (arr != null)
      return arr.Index(r, c);
    else
      return ErrorValue.argTypeError;
  }

  public static double IsArray(Value v0) {
    if (v0 is ErrorValue) return (v0 as ErrorValue).ErrorNan;
    if (v0 == null)
      return ErrorValue.argTypeError.ErrorNan;
    else
      return v0 is ArrayValue ? 1.0 : 0.0;
  }

  // This is Excel ISERROR; Excel ISERR does not consider #N/A an error
  public static double IsError(Value v0) {
    return v0 as ErrorValue != null ? 1.0 : 0.0;
  }

  // Generalized n-argument map
  public static Value Map(Value[] vs) {
    int n = vs.Length - 1;
    if (n < 1)
      return ErrorValue.argCountError;
    if (vs[0] is ErrorValue) return vs[0];
    FunctionValue fv = vs[0] as FunctionValue;
    if (fv == null)
      return ErrorValue.argTypeError;
    if (fv.Arity != n)
      return ErrorValue.argCountError;
    ArrayValue[] arrs = new ArrayValue[n];
    for (int i = 0; i < n; i++) {
      Value vi = vs[i + 1];
      if (vi is ArrayValue)
        arrs[i] = vi as ArrayValue;
      else if (vi is ErrorValue)
        return vi;
      else
        return ErrorValue.argTypeError;
    }
    int cols = arrs[0].Cols, rows = arrs[0].Rows;
    for (int i=1; i<n; i++)
```

```
      if (arrs[i].Cols != cols || arrs[i].Rows != rows)
        return ErrorValue.Make("#ERR: Array shapes differ");
    Value[] args = new Value[n];
    Value[,] result = new Value[cols, rows];
    for (int c = 0; c < cols; c++)
      for (int r = 0; r < rows; r++) {
        for (int i = 0; i < n; i++)
          args[i] = arrs[i][c, r];
        result[c, r] = fv.Apply(args);
      }
    return new ArrayExplicit(result);
  }

  public static double Max(Value[] vs) {
    double result = Double.NegativeInfinity;
    foreach (Value outerV in vs)
      outerV.Apply(delegate(Value v) {
        result = Math.Max(result, NumberValue.ToDoubleOrNan(v));
      });
    return result;
  }

  public static double Min(Value[] vs) {
    double result = Double.PositiveInfinity;
    foreach (Value outerV in vs)
      outerV.Apply(delegate(Value v) {
        result = Math.Min(result, NumberValue.ToDoubleOrNan(v));
      });
    return result;
  }

  public static Value Reduce(Value v0, Value v1, Value v2) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if (v2 is ErrorValue) return v2;
    if (v0 is FunctionValue && v2 is ArrayValue) {
      FunctionValue fv = v0 as FunctionValue;
      ArrayValue arr = v2 as ArrayValue;
      if (fv.Arity != 2)
        return ErrorValue.argCountError;
      Value result = v1;
      for (int r = 0; r < arr.Rows; r++)
        for (int c = 0; c < arr.Cols; c++)
          result = fv.Call2(result, arr[c, r]);
      return result;
    } else
      return ErrorValue.argTypeError;
  }

  public static Value RowMap(Value v0, Value v1) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if (v1 is ArrayValue && v0 is FunctionValue) {
      ArrayValue arr = v1 as ArrayValue;
      FunctionValue fv = v0 as FunctionValue;
      if (fv.Arity != arr.Cols)
        return ErrorValue.argCountError;
      Value[,] result = new Value[1, arr.Rows];
      Value[] arguments = new Value[arr.Cols];
      for (int r = 0; r < arr.Rows; r++) {
        for (int c = 0; c < arr.Cols; c++)
          arguments[c] = arr[c, r];
        result[0, r] = fv.Apply(arguments);
      }
      return new ArrayExplicit(result);
    } else
      return ErrorValue.argTypeError;
  }

  public static double Rows(Value v0) {
    if (v0 is ErrorValue) return (v0 as ErrorValue).ErrorNan;
```



```
    ArrayValue v0arr = v0 as ArrayValue;
    if (v0arr != null)
      return v0arr.Rows;
    else
      return ErrorValue.argTypeError.ErrorNan;
  }

  public static double Sign(double x) {
    return Double.IsNaN(x) ? x : (double)Math.Sign(x);
  }

  public static Value Slice(Value v0, double r1, double c1, double r2, double c2) {
    if (v0 is ErrorValue) return v0;
    if (Double.IsNaN(r1)) return NumberValue.Make(r1);
    if (Double.IsNaN(c1)) return NumberValue.Make(c1);
    if (Double.IsNaN(r2)) return NumberValue.Make(r2);
    if (Double.IsNaN(c2)) return NumberValue.Make(c2);
    ArrayValue arr = v0 as ArrayValue;
    if (arr != null)
      return arr.Slice(r1, c1, r2, c2);
    else
      return ErrorValue.argTypeError;
  }

  public static Value Specialize(Value v0) {
    if (v0 is ErrorValue) return v0;
    FunctionValue fv = v0 as FunctionValue;
    if (fv != null)
      if (fv.args.All(v => v == ErrorValue.naError))
        return fv;
      else
        return new FunctionValue(SdfManager.SpecializeAndCompile(fv), null);
    else
      return ErrorValue.argTypeError;
  }

  public static double Sum(Value[] vs) {
    // May consider whether empty cells and texts should just
    // be ignored instead of giving ArgTypeError.
    // We're using Kahan's accurate sum algorithm; see Goldberg 1991.
    double S = 0.0, C = 0.0;
    foreach (Value outerV in vs)
      outerV.Apply(delegate(Value v) {
        double Y = NumberValue.ToDoubleOrNan(v) - C, T = S + Y;
        C = (T - S) - Y;
        S = T;
      });
    return S;
  }

  public static double SumNew(Value[] vs) {
    // Lower-functionality slightly-higher performance SUM.
    // May consider whether empty cells and texts should just
    // be ignored instead of giving ArgTypeError.
    // We're using Kahan's accurate sum algorithm; see Goldberg 1991.
    double S = 0.0, C = 0.0;
    foreach (Value outerV in vs)
      if (outerV is ArrayValue) {
        ArrayValue arr = outerV as ArrayValue;
        int cols = arr.Cols, rows = arr.Rows;
        for (int c = 0; c < cols; c++)
          for (int r = 0; r < rows; r++) {
            Value v = arr[c, r];
            if (v != null) { // Only non-blank cells contribute
              double Y = NumberValue.ToDoubleOrNan(v) - C, T = S + Y;
              C = (T - S) - Y;
              S = T;
            }
          }
      }
      else if (outerV != null) {
```

```
        double Y = NumberValue.ToDoubleOrNan(outerV) - C, T = S + Y;
        C = (T - S) - Y;
        S = T;
      }
    return S;
  }

  public static Value SumIf(Value v0, Value v1) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    FunctionValue fv = v0 as FunctionValue;
    if (fv == null)
      return ErrorValue.argTypeError;
    if (fv.Arity != 1)
      return ErrorValue.argCountError;
    // We're using Kahan's accurate sum algorithm; see Goldberg 1991.
    double S = 0.0, C = 0.0;
    v1.Apply(delegate(Value v) {
        if (!Double.IsNaN(S)) {
          double condition = NumberValue.ToDoubleOrNan(fv.Call1(v));
          if (Double.IsNaN(condition))
            S = condition; // Error propagation from predicate
          else if (condition != 0) {
            double Y = NumberValue.ToDoubleOrNan(v) - C, T = S + Y;
            C = (T - S) - Y;
            S = T;
          }
        }
      });
    return NumberValue.Make(S);
  }

  public static Value Tabulate(Value v0, Value v1, Value v2) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if (v2 is ErrorValue) return v2;
    if (v0 is FunctionValue && v1 is NumberValue && v2 is NumberValue) {
      FunctionValue fv = v0 as FunctionValue;
      if (fv.Arity != 2)
        return ErrorValue.argCountError;
      int rows = (int)(v1 as NumberValue).value,
        cols = (int)(v2 as NumberValue).value;
      if (0 <= rows && 0 <= cols) {
        Value[,] result = new Value[cols, rows];
        for (int c = 0; c < cols; c++)
          for (int r = 0; r < rows; r++)
            result[c, r] = fv.Call2(NumberValue.Make(r + 1), NumberValue.Make(c + 1));
        return new ArrayExplicit(result);
      } else
        return ErrorValue.Make("#ERR: Size");
    } else
      return ErrorValue.argTypeError;
  }

  public static Value Transpose(Value v0) {
    if (v0 is ErrorValue) return v0;
    ArrayValue v0arr = v0 as ArrayValue;
    if (v0arr != null) {
      int cols = v0arr.Rows, rows = v0arr.Cols;
      Value[,] result = new Value[cols, rows];
      for (int c = 0; c < cols; c++)
        for (int r = 0; r < rows; r++)
          result[c, r] = v0arr[r, c];
      return new ArrayExplicit(result);
    } else
      return ErrorValue.argTypeError;
  }

  public static Value VArray(Value[] vs) {
    Value[,] result = new Value[1, vs.Length];
    for (int r = 0; r < vs.Length; r++)
```



```
      if (vs[r] is ErrorValue)
        return vs[r];
      else
        result[0, r] = vs[r];
    return new ArrayExplicit(result);
  }

  // Make array as vertical concatenation (stack) of the arguments' rows
  public static Value VCat(Value[] vs) {
    int rows = 0, cols = 0;
    foreach (Value v in vs)
      if (v is ErrorValue)
        return v;
      else if (v is ArrayValue) {
        cols = Math.Max(cols, (v as ArrayValue).Cols);
        rows += (v as ArrayValue).Rows;
      } else {
        cols = Math.Max(cols, 1);
        rows += 1;
      }
    foreach (Value v in vs)
      if (v is ArrayValue && (v as ArrayValue).Cols != cols)
        return ErrorValue.Make("#ERR: Column counts differ");
    Value[,] result = new Value[cols, rows];
    int nextRow = 0;
    foreach (Value v in vs)
      if (v is ArrayValue) {
        ArrayValue arr = v as ArrayValue;
        for (int r = 0; r < arr.Rows; r++) {
          for (int c = 0; c < cols; c++)
            result[c, nextRow] = arr[c, r];
          nextRow++;
        }
      } else {
        for (int c = 0; c < cols; c++)
          result[c, nextRow] = v;
        nextRow++;
      }
    return new ArrayExplicit(result);
  }

  // Return vertical array with nv+1 rows rv, fv(rv), fv(fv(rv)), ...
  public static Value VScan(Value v0, Value v1, Value v2) {
    if (v0 is ErrorValue) return v0;
    if (v1 is ErrorValue) return v1;
    if (v2 is ErrorValue) return v2;
    FunctionValue fv = v0 as FunctionValue;
    ArrayValue rv = v1 as ArrayValue;
    NumberValue nv = v2 as NumberValue;
    if (fv == null || rv == null || nv == null)
      return ErrorValue.argTypeError;
    else {
      int n = (int)nv.value;
      int cols = rv.Cols;
      if (n < 0 || cols == 0 || rv.Rows != 1)
        return ErrorValue.Make("#ERR: Argument value in VSCAN");
      else {
        Value[,] result = new Value[cols,n+1];
        for (int c=0; c<cols; c++)
          result[c, 0] = rv[c, 0];
        for (int r=1; r<=n; r++) {
          rv = fv.Call1(rv) as ArrayValue;
          if (rv == null)
            return ErrorValue.argTypeError;
          if (rv.Rows != 1 || rv.Cols != cols)
            return ErrorValue.Make("#ERR: Result shape in VSCAN");
          for (int c=0; c<cols; c++)
            result[c, r] = rv[c, 0];
        }
        return new ArrayExplicit(result);
      }
```

```
    }
  }

  // ----- End of SDF-callable built-in functions -----

  public bool IsVolatile(Expr[] es) {
    // A partial application is volatile if the underlying function is
    if (name == "CLOSURE") {
      if (es.Length > 0 && es[0] is TextConst) {
        String sdfName = (es[0] as TextConst).value.value;
        SdfInfo sdfInfo = SdfManager.GetInfo(sdfName);
        return sdfInfo != null && sdfInfo.IsVolatile;
      } else
        return false;
    } else
      return isVolatile;
  }

  private static Value Closure(Sheet sheet, Expr[] es, int col, int row) {
    // First argument may be a (constant) function name or a FunctionValue
    if (es.Length < 1)
      return ErrorValue.argCountError;
    int argCount = es.Length - 1;
    Value[] arguments = new Value[argCount];
    for (int i = 1; i < es.Length; i++) {
      Value vi = es[i].Eval(sheet, col, row);
      if (vi == null)
        return ErrorValue.argTypeError;
      arguments[i - 1] = vi;
    }
    if (es[0] is TextConst) {
      String name = (es[0] as TextConst).value.value;
      SdfInfo sdfInfo = SdfManager.GetInfo(name);
      if (sdfInfo == null)
        return ErrorValue.nameError;
      if (argCount != 0 && argCount != sdfInfo.arity)
        return ErrorValue.argCountError;
      return new FunctionValue(sdfInfo, arguments);
    } else {
      Value v0 = es[0].Eval(sheet, col, row);
      if (v0 is FunctionValue)  // Further application of a partial application
        return (v0 as FunctionValue).FurtherApply(arguments);
      else if (v0 is ErrorValue)
        return v0;
      else
        return ErrorValue.argTypeError;
    }
  }

  // Auxiliary for EXTERN and EXTERNVOLATILE
  private static Value CallExtern(Sheet sheet, Expr[] es, int col, int row) {
    if (es.Length < 1)
      return ErrorValue.argCountError;
    TextConst nameAndSignatureConst = es[0] as TextConst;
    if (nameAndSignatureConst == null)
      return ErrorValue.argTypeError;
    try {
      // This retrieves the method from cache, or creates it:
      ExternalFunction ef = ExternalFunction.Make(nameAndSignatureConst.value.value);
      Value[] values = new Value[es.Length - 1];
      for (int i = 0; i < values.Length; i++)
        values[i] = es[i + 1].Eval(sheet, col, row);
      return ef.Call(values);
    } catch (TargetInvocationException exn)  // From external method
    {
      return ErrorValue.Make(exn.InnerException.Message);
    } catch (Exception exn)  // Covers a multitude of sins
    {
      return ErrorValue.Make("#EXTERN:" + exn.Message);
    }
  }
```



```
  // Get a Function by name
  public static Function Get(String name) {
    Function result;
    if (table.TryGetValue(name.ToUpper(), out result))
      return result;
    else
      return null;
  }

  internal static bool Exists(string name) {
    return table.ContainsKey(name.ToUpper());
  }

  internal static void Remove(string name) {
    table.Remove(name);
  }

  // Populate table of functions.  Corresponding data for sheet-defined
  // functions are in FunctionInfo.functions and in a CGComposite switch.
  static Function() {
    table = new Dictionary<String, Function>();
    // <fun> : unit -> number
    new Function("NOW", MakeNumberFunction(ExcelNow),
                 isVolatile: true);
    new Function("PI", MakeConstant(NumberValue.PI));
    new Function("RAND", MakeNumberFunction(ExcelRand),
                 isVolatile: true);
    // <fun> : unit -> #NA error
    new Function("NA", MakeConstant(ErrorValue.naError));
    // <fun> : number -> number
    new Function("ABS", MakeNumberFunction(Math.Abs));
    new Function("ASIN", MakeNumberFunction(Math.Asin));
    new Function("ACOS", MakeNumberFunction(Math.Acos));
    new Function("ATAN", MakeNumberFunction(Math.Atan));
    new Function("COS", MakeNumberFunction(Math.Cos));
    new Function("EXP", MakeNumberFunction(Math.Exp));
    new Function("LN", MakeNumberFunction((Func<double, double>)Math.Log));
    new Function("LOG", MakeNumberFunction(Math.Log10));
    new Function("LOG10", MakeNumberFunction(Math.Log10));
    new Function("NEG", 9 /* print neatly */, MakeNumberFunction(x => -x));
    new Function("SIN", MakeNumberFunction(Math.Sin));
    new Function("SQRT", MakeNumberFunction(Math.Sqrt));
    new Function("TAN", MakeNumberFunction(Math.Tan));
    // <fun> : number * number -> number
    new Function("ATAN2", MakeNumberFunction(ExcelAtan2));
    new Function("CEILING", MakeNumberFunction(ExcelCeiling));
    new Function("FLOOR", MakeNumberFunction(ExcelFloor));
    new Function("MOD", MakeNumberFunction(ExcelMod));
    new Function("ROUND", MakeNumberFunction(ExcelRound));
    new Function("^", 8, MakeNumberFunction(ExcelPow));
    new Function("*", 7, MakeNumberFunction((x, y) => x * y));
    new Function("/", 7, MakeNumberFunction((x, y) => x / y));
    new Function("+", 6, MakeNumberFunction((x, y) => x + y ));
    new Function("-", 6, MakeNumberFunction((x, y) => x - y));
    new Function("&", 6, MakeFunction(ExcelConcat));
    new Function("<", 5, MakePredicate((x, y) => x < y));
    new Function("<=", 5, MakePredicate((x, y) => x <= y));
    new Function(">=", 5, MakePredicate((x, y) => x >= y));
    new Function(">", 5, MakePredicate((x, y) => x > y));
    new Function("=", 4, MakePredicate((x, y) => x == y));
    new Function("<>", 4, MakePredicate((x, y) => x != y));
    // AND : number* -> number,
    new Function("AND",   // Variadic, and non-strict in args 2, 3, ...
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          for (int i = 0; i < es.Length; i++) {
            Value vi = es[i].Eval(sheet, col, row);
            NumberValue ni = vi as NumberValue;
            if (ni != null) {
              if (ni.value == 0)
                return NumberValue.ZERO;
```

```
            } else if (vi is ErrorValue)
              return vi;
            else
              return ErrorValue.argTypeError;
          }
          return NumberValue.ONE;
        });
    // APPLY(fv,a1...an) applies closure fv to arguments a1...an, n>=0
    new Function("APPLY",  // Variadic
      MakeFunction(delegate(Value[] vs) {
      if (vs.Length < 1)
        return ErrorValue.argCountError;
      if (vs[0] is ErrorValue)
        return vs[0];
      FunctionValue fv = vs[0] as FunctionValue;
      if (fv == null)
        return ErrorValue.argTypeError;
      int argCount = vs.Length - 1;
      Value[] arguments = new Value[argCount];
      for (int i = 1; i < vs.Length; i++)
        arguments[i - 1] = vs[i];
      return fv.Apply(arguments);
    }));
    //  AVERAGE : (number | array)*  -> number
    new Function("AVERAGE", MakeNumberFunction(Average));
    // BENCHMARK(fv, n) evaluates fv to zero-arity FunctionValue,
    // then calls it n times and returns the number of nanoseconds per call
    new Function("BENCHMARK", MakeFunction(Benchmark));
    // CLOSURE(sdfname/fv,arguments...) creates a FunctionValue closure by partial
    // application of a sheet-defined function or an given FunctionValue
    new Function("CLOSURE", Closure);
    //  CONSTARRAY : value * number * number -> array
    new Function("CONSTARRAY", MakeFunction(ConstArray));
    // CHOOSE: number * any* -> any
    new Function("CHOOSE",  // Variadic, and non-strict in arg 2...
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length >= 1) {
            Value v0 = es[0].Eval(sheet, col, row);
            NumberValue n0 = v0 as NumberValue;
            if (n0 != null) {
              int index = (int)n0.value;
              if (1 <= index && index < es.Length)
                return es[index].Eval(sheet, col, row);
              else
                return ErrorValue.valueError;
            } else if (v0 is ErrorValue)
              return v0;
            else
              return ErrorValue.argTypeError;
          } else
            return ErrorValue.argCountError;
        });
    //  COLMAP : function * array -> array
    new Function("COLMAP", MakeFunction(ColMap));
    // COLS : array -> number
    new Function("COLUMNS", MakeNumberFunction(Columns));
    //  COUNTIF : area * fExpr -> number
    new Function("COUNTIF", MakeFunction(CountIf));
    // DEFINE(sdfname,outputcell,inputcells...) defines a sheet-defined function
    new Function("DEFINE",
    delegate(Sheet sheet, Expr[] es, int col, int row) {
      if (!sheet.IsFunctionSheet)
        return ErrorValue.Make("#FUNERR: Non-function sheet");
      if (es.Length < 2)
        return ErrorValue.argCountError;
      TextConst nameConst = es[0] as TextConst;
      if (nameConst == null)
        return ErrorValue.argTypeError;
      String name = nameConst.value.value.ToUpper();
      CellRef outputCell = es[1] as CellRef;
      if (outputCell == null)
```



```
        return ErrorValue.argTypeError;
      if (outputCell.sheet != null && outputCell.sheet != sheet)
        return ErrorValue.Make("#FUNERR: Output on another sheet");
      FullCellAddr output = outputCell.GetAbsoluteAddr(sheet, col, row);
      List<FullCellAddr> inputCells = new List<FullCellAddr>();
      for (int i = 2; i < es.Length; i++) {
        CellRef inputCell = es[i] as CellRef;
        if (inputCell == null)
          return ErrorValue.argTypeError;
        else if (inputCell.sheet != null && inputCell.sheet != sheet)
          return ErrorValue.Make("#FUNERR: Input on another sheet");
        else
          inputCells.Add(inputCell.GetAbsoluteAddr(sheet, col, row));
      }
      try {
        SdfManager.CreateFunction(name, output, inputCells);
      } catch (CyclicException) {
        return ErrorValue.Make("#FUNERR: Cyclic dependency in function");
      }
      return TextValue.MakeInterned(SdfManager.GetInfo(name).ToString());
    });
    // EQUAL(e1, e2) returns 1 if values e1 and e2 are non-errors and equal
    new Function("EQUAL", MakeNumberFunction(Equal));
    // ERR("message") produces an ErrorValue with the given constant message
    new Function("ERR",
    delegate(Sheet sheet, Expr[] es, int col, int row) {
      if (es.Length != 1)
        return ErrorValue.argCountError;
      TextConst messageConst = es[0] as TextConst;
      if (messageConst == null)
        return ErrorValue.argTypeError;
      return ErrorValue.Make("#ERR: " + messageConst.value.value);
    });
    // EXTERN("nameAndSignature", e1, ..., en) calls a .NET function "name"
    // having the given "signature", passing it converted values of e1...en
    new Function("EXTERN", CallExtern);
    // HARRAY: value* -> array
    new Function("HARRAY", MakeFunction(HArray));
    // HCAT: value* -> array
    new Function("HCAT", MakeFunction(HCat));
    // HSCAN: function * array * number -> array
    new Function("HSCAN", MakeFunction(HScan));
    // IF : number any any -> any,
    new Function("IF",                    // Note: non-strict in arg 2 and 3
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 3) {
            Value v0 = es[0].Eval(sheet, col, row);
            NumberValue n0 = v0 as NumberValue;
            if (n0 != null && !Double.IsInfinity(n0.value) && !Double.IsNaN(n0.value))
              if (n0.value != 0)
                return es[1].Eval(sheet, col, row);
              else
                return es[2].Eval(sheet, col, row);
            else if (v0 is ErrorValue)
              return v0;
            else
              return ErrorValue.argTypeError;
          } else
            return ErrorValue.argCountError;
        });
    // INDEX: any* * number * number  -> any     // (row, col) indexing, offset base 1
    new Function("INDEX", MakeFunction(Index));
    // ISARRAY : value -> number
    new Function("ISARRAY", MakeNumberFunction(IsArray));
    // ISERROR : value -> number -- NOT ErrorValue-strict
    new Function("ISERROR", MakeNumberFunction(IsError));
    // MAP: array * function -> array
    new Function("MAP", MakeFunction(Map));
    // MAX: value * -> number
    new Function("MAX", MakeNumberFunction(Max));
    // MIN: value * -> number
```

```
      new Function("MIN", MakeNumberFunction(Min));
      // NOT : number -> number,
      new Function("NOT",
        MakeNumberFunction(delegate(double n0) {
          return Double.IsNaN(n0) ? n0 : n0 == 0 ? 1.0 : 0.0;
        }));
      // OR : number number -> number,
      new Function("OR",  // Variadic, and non-strict in args 2, 3, ...
          delegate(Sheet sheet, Expr[] es, int col, int row) {
            for (int i = 0; i < es.Length; i++) {
              Value vi = es[i].Eval(sheet, col, row);
              NumberValue ni = vi as NumberValue;
              if (ni != null) {
                if (ni.value != 0)
                  return NumberValue.ONE;
              } else if (vi is ErrorValue)
                return vi;
              else
                return ErrorValue.argTypeError;
            }
            return NumberValue.ZERO;
          });
      // REDUCE: function * value * array -> value
      new Function("REDUCE", MakeFunction(Reduce));
      // ROWMAP : function * array -> array
      new Function("ROWMAP", MakeFunction(RowMap));
      // ROWS : array -> number
      new Function("ROWS", MakeNumberFunction(Rows));
      // SIGN : number -> number
      new Function("SIGN", MakeNumberFunction(Sign));
      // SLICE : array * number * number * number * number -> value
      new Function("SLICE", MakeFunction(Slice));
      // SPECIALIZE(fv) creates a specialized SDF from closure fv
      new Function("SPECIALIZE", MakeFunction(Specialize));
      // SUM : { number, array } * -> number
      new Function("SUM", MakeNumberFunction(SumNew));
      // SUMIF : function * array -> number
      new Function("SUMIF", MakeFunction(SumIf));
      // TABULATE : function * number * number -> array
      new Function("TABULATE", MakeFunction(Tabulate));
      // TRANSPOSE : array -> array
      new Function("TRANSPOSE", MakeFunction(Transpose));
      // VARRAY: value* -> array
      new Function("VARRAY", MakeFunction(VArray));
      // VCAT: value* -> array
      new Function("VCAT", MakeFunction(VCat));
      // VOLATILIZE(e1) is exactly as e1, but is volatile
      new Function("VOLATILIZE", MakeFunction(v => v), isVolatile: true);
      // VSCAN: function * array * number -> array
      new Function("VSCAN", MakeFunction(VScan));
    }

    private Function(String name, int fixity, Applier applier)
      : this(name, applier, fixity: fixity) { }

    internal Function(String name, Applier applier, int fixity = 0,
      bool placeHolder = false, bool isVolatile = false) {
      this.name = name;
      this.Applier = applier;
      this.fixity = fixity;
      this.IsPlaceHolder = placeHolder;
      this.isVolatile = isVolatile;
      table[name] = this;  // For Funsheet
    }

    public static Function MakeUnknown(String name) {
      return new Function(name.ToUpper(),
                          applier: delegate { return ErrorValue.nameError; },
                          placeHolder: true);
    }
```



```
    // Number-valued nullary function, eg RAND()
    private static Applier MakeNumberFunction(Func<double> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 0)
            return NumberValue.Make(dlg());
          else
            return ErrorValue.argCountError;
        };
    }

    // Number-valued constant nullary function, eg PI()
    private static Applier MakeConstant(Value value) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 0)
            return value;
          else
            return ErrorValue.argCountError;
        };
    }

    // Number-valued strict unary function, eg SIN(x)
    private static Applier MakeNumberFunction(Func<double, double> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 1) {
            Value v0 = es[0].Eval(sheet, col, row);
            return NumberValue.Make(dlg(Value.ToDoubleOrNan(v0)));
          } else
            return ErrorValue.argCountError;
        };
    }

    // Number-valued strict unary function, eg ISARRAY, ISERROR
    private static Applier MakeNumberFunction(Func<Value, double> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 1) {
            Value v0 = es[0].Eval(sheet, col, row);
            return NumberValue.Make(dlg(v0));
          } else
            return ErrorValue.argCountError;
        };
    }

    // Number-valued strict binary function, eg +
    private static Applier MakeNumberFunction(Func<double, double, double> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 2) {
            Value v0 = es[0].Eval(sheet, col, row),
                  v1 = es[1].Eval(sheet, col, row);
            return NumberValue.Make(dlg(Value.ToDoubleOrNan(v0), Value.ToDoubleOrNan(v1)));
          } else
            return ErrorValue.argCountError;
        };
    }

    // Number-valued strict binary function, eg EQUAL(e1,e2)
    private static Applier MakeNumberFunction(Func<Value, Value, double> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 2) {
            Value v0 = es[0].Eval(sheet, col, row),
                  v1 = es[1].Eval(sheet, col, row);
            return NumberValue.Make(dlg(v0, v1));
          } else
            return ErrorValue.argCountError;
        };
    }
```

```
    // Boolean-valued strict binary function, eg ==
    private static Applier MakePredicate(Func<double, double, bool> dlg) {
      return
        MakeNumberFunction(delegate(double x, double y) {
          return Double.IsNaN(x) ? x : Double.IsNaN(y) ? y : dlg(x, y) ? 1.0 : 0.0;
      });
    }

    // Number-valued strict variadic function, eg SUM, AVERAGE
    private static Applier MakeNumberFunction(Func<Value[], double> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          try {
            return NumberValue.Make(dlg(Eval(es, sheet, col, row)));
          } catch (ArgumentException) {
            return ErrorValue.argTypeError;
          }
        };
    }

    // Strict unary function, eg SPECIALIZE, TRANSPOSE
    private static Applier MakeFunction(Func<Value, Value> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 1)
            return dlg(es[0].Eval(sheet, col, row));
          else
            return ErrorValue.argCountError;
        };
    }

    // Strict binary function, eg &, COLMAP, SUMIF
    private static Applier MakeFunction(Func<Value, Value, Value> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 2) {
            Value v0 = es[0].Eval(sheet, col, row),
                  v1 = es[1].Eval(sheet, col, row);
            return dlg(v0, v1);
          } else
            return ErrorValue.argCountError;
        };
    }

    // Strict ternary function, eg REDUCE, TABULATE
    private static Applier MakeFunction(Func<Value, Value, Value, Value> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 3) {
            Value v0 = es[0].Eval(sheet, col, row),
                  v1 = es[1].Eval(sheet, col, row),
                  v2 = es[2].Eval(sheet, col, row);
            return dlg(v0, v1, v2);
          } else
            return ErrorValue.argCountError;
        };
    }

    // Strict ternary function, eg INDEX
    private static Applier MakeFunction(Func<Value, double, double, Value> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 3) {
            Value v0 = es[0].Eval(sheet, col, row),
                  v1 = es[1].Eval(sheet, col, row),
                  v2 = es[2].Eval(sheet, col, row);
            return dlg(v0, Value.ToDoubleOrNan(v1), Value.ToDoubleOrNan(v2));
          } else
            return ErrorValue.argCountError;
        };
```



```
    }

    // Strict quinternary function, eg SLICE
    private static Applier MakeFunction(Func<Value, double, double, double, double, Value>
dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          if (es.Length == 5) {
            Value v0 = es[0].Eval(sheet, col, row);
            double n1 = Value.ToDoubleOrNan(es[1].Eval(sheet, col, row)),
                   n2 = Value.ToDoubleOrNan(es[2].Eval(sheet, col, row)),
                   n3 = Value.ToDoubleOrNan(es[3].Eval(sheet, col, row)),
                   n4 = Value.ToDoubleOrNan(es[4].Eval(sheet, col, row));
            return dlg(v0, n1, n2, n3, n4);
          } else
            return ErrorValue.argCountError;
        };
    }

    // Strict variadic function, eg MAP
    private static Applier MakeFunction(Func<Value[], Value> dlg) {
      return
        delegate(Sheet sheet, Expr[] es, int col, int row) {
          Value[] vs = Eval(es, sheet, col, row);
          return dlg(vs);
        };
    }

    // Evaluate expression array
    private static Value[] Eval(Expr[] es, Sheet sheet, int col, int row) {
      Value[] vs = new Value[es.Length];
      for (int i = 0; i < es.Length; i++)
        vs[i] = es[i].Eval(sheet, col, row);
      return vs;
    }
  }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;
using System.Reflection;

namespace Corecalc {
  partial class AboutBox : Form {
    public AboutBox() {
      InitializeComponent();
      //  Initialize the AboutBox to display the product information from the assembly info
rmation.
      //  Change assembly information settings for your application through either:
      //  - Project->Properties->Application->Assembly Information
      //  - AssemblyInfo.cs
      this.Text = String.Format("About {0}", AssemblyTitle);
      this.labelProductName.Text = AssemblyProduct;
      this.labelVersion.Text = String.Format("Version {0}", AssemblyVersion);
      this.labelCopyright.Text = AssemblyCopyright;
      this.labelCompanyName.Text = AssemblyCompany;
      this.textBoxDescription.Text = AssemblyDescription;
    }

    #region Assembly Attribute Accessors

    public string AssemblyTitle {
      get {
        // Get all Title attributes on this assembly
        object[] attributes = Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(As
semblyTitleAttribute), false);
        // If there is at least one Title attribute
        if (attributes.Length > 0) {
          // Select the first one
          AssemblyTitleAttribute titleAttribute = (AssemblyTitleAttribute)attributes[0];
          // If it is not an empty string, return it
          if (titleAttribute.Title != "")
            return titleAttribute.Title;
        }
        // If there was no Title attribute, or if the Title attribute was the empty string,
 return the .exe name
        return System.IO.Path.GetFileNameWithoutExtension(Assembly.GetExecutingAssembly().C
odeBase);
      }
    }

    public string AssemblyVersion {
      get {
        return Assembly.GetExecutingAssembly().GetName().Version.ToString();
      }
    }

    public string AssemblyDescription {
      get {
        // Get all Description attributes on this assembly
        object[] attributes = Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(As
semblyDescriptionAttribute), false);
        // If there aren't any Description attributes, return an empty string
        if (attributes.Length == 0)
          return "";
        // If there is a Description attribute, return its value
        return ((AssemblyDescriptionAttribute)attributes[0]).Description;
      }
    }

    public string AssemblyProduct {
      get {
        // Get all Product attributes on this assembly
        object[] attributes = Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(As
semblyProductAttribute), false);
        // If there aren't any Product attributes, return an empty string
```



```
        if (attributes.Length == 0)
          return "";
        // If there is a Product attribute, return its value
        return ((AssemblyProductAttribute)attributes[0]).Product;
      }
    }

    public string AssemblyCopyright {
      get {
        // Get all Copyright attributes on this assembly
        object[] attributes = Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(As
semblyCopyrightAttribute), false);
        // If there aren't any Copyright attributes, return an empty string
        if (attributes.Length == 0)
          return "";
        // If there is a Copyright attribute, return its value
        return ((AssemblyCopyrightAttribute)attributes[0]).Copyright;
      }
    }

    public string AssemblyCompany {
      get {
        // Get all Company attributes on this assembly
        object[] attributes = Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(As
semblyCompanyAttribute), false);
        // If there aren't any Company attributes, return an empty string
        if (attributes.Length == 0)
          return "";
        // If there is a Company attribute, return its value
        return ((AssemblyCompanyAttribute)attributes[0]).Company;
      }
    }
    #endregion
  }
}
```

```
// Corecalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;                   // Size
using SDD2D = System.Drawing.Drawing2D; // Pen etc
using System.Windows.Forms;
using Corecalc.GUI;
using Corecalc.IO;
using Corecalc.Funcalc;   // SdfManager

namespace Corecalc {
  /// <summary>
  /// A WorkbookForm is the user interface of an open workbook.
  /// </summary>
  public partial class WorkbookForm : Form {
    public Workbook Workbook { get; private set; }
    private Benchmarks.Benchmarks test;
    public int precedentsDepth, dependentsDepth;

    public WorkbookForm(Workbook workbook, bool display) {
      SetWorkbook(workbook);
      InitializeComponent();
      StartPosition = FormStartPosition.CenterScreen;
      Size = new Size(900, 600);
      SetStatusLine(null);
      sheetHolder.Selected += sheetHolderSelected;
      DisplayWorkbook();
      if (display)
        ShowDialog();
    }

    private void sheetHolderSelected(Object sender, TabControlEventArgs e) {
      if (SelectedSheet != null) {
        precedentsDepth = dependentsDepth = 0;
        SelectedSheet.Reshow();
      }
    }

    private void SetWorkbook(Workbook newWorkbook) {
      if (this.Workbook != null) {
        this.Workbook.Clear();
        this.Workbook.OnFunctionsAltered -= workbook_OnFunctionsAltered;
      }
      this.Workbook = newWorkbook;
      this.Workbook.OnFunctionsAltered += workbook_OnFunctionsAltered;
    }
```



```
    void workbook_OnFunctionsAltered(String[] functions) {
      Console.WriteLine("Regenerating modified functions");
      SdfManager.Regenerate(functions);
    }

    private void exitToolStripMenuItem_Click(object sender, EventArgs e) {
      System.Environment.Exit(0);
    }

    private void aboutToolStripMenuItem1_Click(object sender, EventArgs e) {
      Form aboutBox = new AboutBox();
      aboutBox.ShowDialog();
    }

    private void a1ToolStripMenuItem_Click(object sender, EventArgs e) {
      Workbook.format.RefFmt = Formats.RefType.A1;
      Reshow(null);
    }

    private void c0R0ToolStripMenuItem_Click(object sender, EventArgs e) {
      Workbook.format.RefFmt = Formats.RefType.C0R0;
      Reshow(null);
    }

    private void r1C1ToolStripMenuItem_Click(object sender, EventArgs e) {
      Workbook.format.RefFmt = Formats.RefType.R1C1;
      Reshow(null);
    }

    // Add new sheet
    private void newWorkbookToolStripMenuItem_Click(object sender, EventArgs e) {
      InsertSheet(functionSheet: false);
    }

    public void InsertSheet(bool functionSheet) {
      String name = "Sheet" + (Workbook.SheetCount + 1);
      InsertSheet(new SheetTab(this, new Sheet(Workbook, name, functionSheet)));
    }

    private void InsertSheet(SheetTab sheetTab) {
      if (Workbook != null) {
        sheetHolder.TabPages.Add(sheetTab);
        sheetHolder.SelectTab(sheetTab);
      }
    }

    // Recalculate and reshow workbook

    private void Recalculate() {
      if (Workbook != null) {
        long elapsed = Workbook.Recalculate();
        Reshow(elapsed);
      }
    }

    private void RecalculateFull() {
      if (Workbook != null) {
        long elapsed = Workbook.RecalculateFull();
        Reshow(elapsed);
      }
    }

    private void RecalculateFullRebuild() {
      if (Workbook != null) {
        long elapsed = Workbook.RecalculateFullRebuild();
        Reshow(elapsed);
      }
    }

    private void recalculateMenuItem_Click(object sender, EventArgs e) {
```

```
    Recalculate();
  }

  private void recalculateFullMenuItem_Click(object sender, EventArgs e) {
    RecalculateFull();
  }

  private void recalculateFullRebuildMenuItem_Click(object sender, EventArgs e) {
    RecalculateFullRebuild();
  }

  private void Reshow(long? elapsed) {
    if (SelectedSheet != null)
      SelectedSheet.Reshow();
    SetStatusLine(elapsed);
  }

  public void SetStatusLine(long? elapsed) {
    double memory = System.GC.GetTotalMemory(false) / 1E6; // In MB
    statusLine.Text = String.Format("{0,-4} {1,8:F2} MB {2,8:D}",
      Workbook.format.RefFmt, memory, Workbook.RecalcCount);
    if (elapsed.HasValue)
      statusLine.Text += String.Format(" {0,8:D} ms", elapsed.Value);
    if (Workbook.Cyclic != null) {
      statusLine.Text += " " + Workbook.Cyclic.Message;
    }
  }

  // Set or remove (message=null) cell error mark
  public void SetCyclicError(String message) {
    if (Workbook.Cyclic != null) {
      FullCellAddr culprit = Workbook.Cyclic.culprit;
      sheetHolder.SelectTab(culprit.sheet.Name);
      SelectedSheet.SetCellErrorText(culprit.ca, message);
    }
  }

  // Copy cell
  private void copyToolStripMenuItem_Click(object sender, EventArgs e) {
    if (SelectedSheet != null)
      SelectedSheet.Copy();
  }

  // Delete cell
  private void deleteToolStripMenuItem_Click(object sender, EventArgs e) {
    if (SelectedSheet != null)
      SelectedSheet.Delete();
  }

  // Insert column
  private void columnToolStripMenuItem_Click(object sender, EventArgs e) {
    if (SelectedSheet != null)
      SelectedSheet.InsertColumns(1);
  }

  // Insert row
  private void rowToolStripMenuItem_Click(object sender, EventArgs e) {
    if (SelectedSheet != null)
      SelectedSheet.InsertRows(1);
  }

  // Paste copied or cut cells, or text
  private void pasteToolStripMenuItem_Click(object sender, EventArgs e) {
    if (SelectedSheet != null)
      SelectedSheet.Paste();
  }

  // Get selected sheet if any, else null
  private SheetTab SelectedSheet {
    get {
      return sheetHolder.TabCount > 0 ? sheetHolder.SelectedTab as SheetTab : null;
```



```
    }
  }

  public Benchmarks.Benchmarks Test {
    get {
      if (test == null) test = new Benchmarks.Benchmarks(true);
      return test;
    }
  }

  private void formulaBox_TextChanged(object sender, EventArgs e) {
    if (SelectedSheet != null)
      SelectedSheet.ChangeCurrentText(formulaBox.Text);
  }

  private void formulaBox_KeyPress(object sender, KeyPressEventArgs e) {
    if (SelectedSheet != null) {
      if (e.KeyChar == (char)Keys.Return) {
        SelectedSheet.SetCurrentCell(formulaBox.Text);
        e.Handled = true;
      } else if (e.KeyChar == (char)Keys.Escape) {
        SelectedSheet.Focus();
        e.Handled = true;
      }
    }
  }

  private void importSheetToolStripMenuItem_Click(object sender, EventArgs e) {
    OpenFileDialog ofd = new OpenFileDialog();
    WorkBookIO workbookio = new WorkBookIO();
    ofd.Filter = workbookio.SupportedFormatFilter();
    ofd.FilterIndex = workbookio.DefaultFormatIndex();
    if (ofd.ShowDialog() == DialogResult.OK) {
      Clear();
      Workbook wb = workbookio.Read(ofd.FileName);
      if (wb != null) {
        SetWorkbook(wb);
        DisplayWorkbook();
      }
    }
  }

  private void DisplayWorkbook() {
    if (Workbook != null) {
      foreach (Sheet sheet in Workbook)
        sheetHolder.TabPages.Add(new SheetTab(this, sheet));
      RecalculateFull();
      if (sheetHolder.TabCount > 0)
        sheetHolder.SelectTab(0);
    }
  }

  public void Clear() {
    sheetHolder.TabPages.Clear();
  }

  private void benchmarkStandardRecalculation_Click(object sender, EventArgs e) {
    int runs = 0;
    if (int.TryParse(numberOfRunsTextBox.Text, out runs))
      Test.BenchmarkRecalculation(this, runs);
  }

  private void fullRecalculationMenuItem_Click(object sender, EventArgs e) {
    int runs = 0;
    if (int.TryParse(numberOfRunsTextBox.Text, out runs))
      Test.BenchmarkRecalculationFull(this, runs);
  }

  private void recalculationFullRebuildMenuItem_Click(object sender, EventArgs e) {
    int runs = 0;
    if (int.TryParse(numberOfRunsTextBox.Text, out runs))
```

```
        Test.BenchmarkRecalculationFullRebuild(this, runs);
    }

    private void newFunctionSheetMenuItem_Click(object sender, EventArgs e) {
      this.InsertSheet(functionSheet: true);
    }

    private void sheetHolder_DrawItem(object sender, DrawItemEventArgs e) {
      int currentIndex = e.Index;
      Graphics g = e.Graphics;
      TabControl tc = (TabControl)sender;
      TabPage tp = tc.TabPages[currentIndex];

      Sheet currentSheet = Workbook[currentIndex];
      Brush theBrush = new SolidBrush(Color.Black);
      StringFormat sf = new StringFormat();
      sf.Alignment = StringAlignment.Center;
      Color col = currentSheet.IsFunctionSheet ? Color.LightPink : tp.BackColor;
      g.FillRectangle(new SolidBrush(col), e.Bounds);
      g.DrawString(currentSheet.Name, tc.Font, theBrush, e.Bounds, sf);
    }

    private void sdfMenuItem_Click(object sender, EventArgs e) {
      Form sdfForm = Application.OpenForms["sdf"];
      if (sdfForm == null) {
        sdfForm = new SdfForm(this, Workbook);
        sdfForm.Show();
      } else {
        ((SdfForm)sdfForm).PopulateFunctionListBox(true);
      }
    }

    public void ChangeFocus(FullCellAddr fca) {
      sheetHolder.SelectTab(fca.sheet.Name);
      SheetTab sheetTab = (SheetTab)sheetHolder.SelectedTab;
      sheetTab.ScrollTo(fca);
    }

    public void ChangeCellBackgroundColor(FullCellAddr fca, Color c) {
      sheetHolder.SelectTab(fca.sheet.Name);
      SelectedSheet.ChangeCellBackgroundColor(fca, c);
    }

    private void showFormulasMenuItem_Click(object sender, EventArgs e) {
      Workbook.format.ShowFormulas = showFormulasToolStripMenuItem.Checked;
      // Wrap formula text in cells
      //DataGridViewCellStyle dgvcs = new DataGridViewCellStyle();
      //dgvcs.WrapMode = showFormulasToolStripMenuItem.Checked ? DataGridViewTriState.True
: DataGridViewTriState.False;
      //foreach (SheetTab sheetTab in sheetHolder.TabPages)
      //  sheetTab.SetRowCellStyle(dgvcs);
      Reshow(null);
    }

    private void morePrecedentsToolStripMenuItem_Click(object sender, EventArgs e) {
      if (SelectedSheet != null) {
        precedentsDepth++;
        SelectedSheet.Refresh();
      }
    }

    private void fewerPrecedentsToolStripMenuItem_Click(object sender, EventArgs e) {
      if (SelectedSheet != null) {
        precedentsDepth = Math.Max(0, precedentsDepth - 1);
        SelectedSheet.Refresh();
      }
    }

    private void moreDependentsToolStripMenuItem_Click(object sender, EventArgs e) {
      if (SelectedSheet != null) {
        dependentsDepth++;
```



```
        SelectedSheet.Refresh();
      }
    }

    private void fewerDependentsToolStripMenuItem_Click(object sender, EventArgs e) {
      if (SelectedSheet != null) {
        dependentsDepth = Math.Max(0, dependentsDepth - 1);
        SelectedSheet.Refresh();
      }
    }

    private void eraseArrowsToolStripMenuItem_Click(object sender, EventArgs e) {
      precedentsDepth = dependentsDepth = 0;
      if (SelectedSheet != null)
        SelectedSheet.Refresh();
    }

    private void cutToolStripMenuItem_Click(object sender, EventArgs e) {
      // TODO, not implemented
    }
  }

  /// <summary>
  /// A SheetTab is a displayed sheet, shown as a tab page on the
  /// workbook's tab control.
  /// </summary>
  public class SheetTab : TabPage {
    public readonly Sheet sheet;
    private readonly DataGridView dgv;
    private readonly WorkbookForm gui;
    private readonly int[] colOffset, rowOffset;

    public SheetTab(WorkbookForm gui, Sheet sheet)
      : base(sheet.Name) {
      this.gui = gui;
      this.sheet = sheet;
      this.Name = sheet.Name;
      this.dgv = new DataGridView();
      dgv.ShowEditingIcon = false;
      dgv.Dock = DockStyle.Fill;
      Dock = DockStyle.Fill;
      // Display formula in the current cell and computed value in other cells
      dgv.CellFormatting +=
        delegate(Object sender, DataGridViewCellFormattingEventArgs e) {
          int col = e.ColumnIndex, row = e.RowIndex;
          if (col == dgv.CurrentCellAddress.X && row == dgv.CurrentCellAddress.Y) {
            Object obj = sheet.Show(col, row);
            if (obj != null) {
              e.Value = obj;
              e.FormattingApplied = true;
            }
          } else {
            Object obj = sheet.ShowValue(col, row);
            if (obj != null) {
              e.Value = obj;
              e.FormattingApplied = true;
            }
          }
        };
      // Show current cell's address, and show formula in formula box
      dgv.CellEnter +=
        delegate(Object sender, DataGridViewCellEventArgs arg) {
          int row = arg.RowIndex, col = arg.ColumnIndex;
          dgv.TopLeftHeaderCell.Value = new CellAddr(col, row).ToString();
          gui.formulaBox.Text = (String)dgv.CurrentCell.FormattedValue;
        };
      // Check that cell's contents is well-formed after edit
      dgv.CellValidating +=
        delegate(Object sender, DataGridViewCellValidatingEventArgs arg) {
          if (dgv.IsCurrentCellInEditMode) {   // Update only if cell was edited
            int row = arg.RowIndex, col = arg.ColumnIndex;
```

```
          Object value = arg.FormattedValue;
          if (value != null)
            SetCell(col, row, value.ToString(), arg);
        }
      };
    // Experiment with painting on the data grid view
    dgv.Paint += delegate(Object sender, PaintEventArgs arg) {
      base.OnPaint(arg);
      // Update column and row offset tables for drawing arrows between cells:
      int offset = dgv.RowHeadersWidth;
      for (int col = 0; col < sheet.Cols; col++) {
        colOffset[col] = offset;
        offset += dgv.Columns[col].Width;
      }
      colOffset[sheet.Cols] = offset;
      offset = dgv.ColumnHeadersHeight;
      for (int row = 0; row < sheet.Rows; row++) {
        rowOffset[row] = offset;
        offset += dgv.Rows[row].Height;
      }
      rowOffset[sheet.Rows] = offset;
      Pen pen = new Pen(Color.Blue, 1);
      pen.EndCap = SDD2D.LineCap.ArrowAnchor;
      pen.StartCap = SDD2D.LineCap.RoundAnchor;
      if (dgv.SelectedCells.Count > 0) {
        DataGridViewCell dgvc = dgv.SelectedCells[0];
        CellAddr ca = new CellAddr(dgvc.ColumnIndex, dgvc.RowIndex);
        Graphics g = arg.Graphics;
        g.SmoothingMode = SDD2D.SmoothingMode.AntiAlias;
        int x = dgv.RowHeadersWidth, y = dgv.ColumnHeadersHeight,
            w = dgv.DisplayRectangle.Width - x, h = dgv.DisplayRectangle.Height - y;
        // Clip headers *before* the scroll translation/transform
        g.Clip = new Region(new Rectangle(x, y, w, h));
        g.Transform = new SDD2D.Matrix(1, 0, 0, 1,
          -dgv.HorizontalScrollingOffset, -dgv.VerticalScrollingOffset);
        SupportArea.IdempotentForeach = false; // Draw all arrows into a cell
        DrawDependents(g, pen, gui.dependentsDepth, ca, new HashSet<CellAddr>());
        DrawPrecedents(g, pen, gui.precedentsDepth, ca, new HashSet<CellAddr>());
      }
    };
    dgv.SelectionChanged += delegate(Object sender, EventArgs arg) {
      if (gui.dependentsDepth != 0 || gui.precedentsDepth != 0) {
        gui.dependentsDepth = gui.precedentsDepth = 0;
        Refresh();
      }
    };
    // Strange: to hold sheet, we need an extra row, but not an extra column?
    dgv.ColumnCount = sheet.Cols;
    dgv.RowCount = sheet.Rows + 1;
    // Allocate offset tables to assist drawing arrows between cells
    colOffset = new int[sheet.Cols + 1];
    rowOffset = new int[sheet.Rows + 1];
    dgv.AllowUserToAddRows = false;
    // Put labels on columns and rows, disable (meaningless) row sorting:
    for (int col = 0; col < dgv.ColumnCount; col++) {
      dgv.Columns[col].Name = CellAddr.ColumnName(col);
      dgv.Columns[col].SortMode = DataGridViewColumnSortMode.NotSortable;
    }
    for (int row = 0; row < dgv.RowCount; row++)
      dgv.Rows[row].HeaderCell.Value = (row + 1).ToString();
    if (sheet.IsFunctionSheet) {
      DataGridViewCellStyle cellStyle = new DataGridViewCellStyle();
      cellStyle.BackColor = Color.LightPink;
      dgv.ColumnHeadersDefaultCellStyle = dgv.RowHeadersDefaultCellStyle = cellStyle;
    }
    // Somewhat arbitrary extension of the width -- using
    // Graphics.MeasureString("0000", dgv.Font) would be better
    dgv.RowHeadersWidth += 20;
    Controls.Add(dgv);
  }
```



```
  // Painting arrows to dependents and precedents of cell ca
  private void DrawDependents(Graphics g, Pen pen, int depth, CellAddr ca, HashSet<CellAd
dr> done) {
    if (depth > 0 && !done.Contains(ca)) {
      done.Add(ca);
      Cell cell = sheet[ca];
      if (cell != null) {
        cell.ForEachSupported(delegate(Sheet suppSheet, int suppCol, int suppRow) {
          CellAddr dependent = new CellAddr(suppCol, suppRow);
          if (suppSheet == sheet) {
            CellToCellArrow(g, pen, ca, dependent);
            DrawDependents(g, pen, depth - 1, dependent, done);
          }
        });
      }
    }
  }

  private void DrawPrecedents(Graphics g, Pen pen, int depth, CellAddr ca, HashSet<CellAd
dr> done) {
    if (depth > 0 && !done.Contains(ca)) {
      done.Add(ca);
      Cell cell = sheet[ca];
      if (cell != null) {
        cell.ForEachReferred(sheet, ca.col, ca.row, delegate(FullCellAddr precedent) {
          if (precedent.sheet == sheet) {
            CellToCellArrow(g, pen, precedent.ca, ca);
            DrawPrecedents(g, pen, depth-1, precedent.ca, done);
          }
        });
      }
    }
  }

  private void CellToCellArrow(Graphics g, Pen pen, CellAddr ca1, CellAddr ca2) {
    if (dgv[ca1.col, ca1.row].Displayed || dgv[ca2.col, ca2.row].Displayed) {
      // Sadly, dgv.GetCellDisplayRectangle returns Empty outside visible area
      int x1 = (colOffset[ca1.col] + colOffset[ca1.col + 1]) / 2,
          y1 = (rowOffset[ca1.row] + rowOffset[ca1.row + 1]) / 2,
          x2 = (colOffset[ca2.col] + colOffset[ca2.col + 1]) / 2,
          y2 = (rowOffset[ca2.row] + rowOffset[ca2.row + 1]) / 2;
      g.DrawLine(pen, x1, y1, x2, y2);
    }
  }

  // Attempt to parse s as cell contents, and set selected cell(s)

  private void SetCell(int col, int row, String text, DataGridViewCellValidatingEventArgs
 arg = null) {
    Cell cell = Cell.Parse(text, sheet.workbook, col, row);
    ArrayFormula oldArrayFormula = sheet[col, row] as ArrayFormula;
    gui.SetCyclicError(null);
    if (cell == null) {
      if (text.TrimStart(' ').StartsWith("=")) {    // Ill-formed formula, cancel edit
        if (arg != null) {
          ErrorMessage("Bad formula");
          arg.Cancel = true;
        }
        return;
      } else if (!String.IsNullOrWhiteSpace(text)) // Assume a quote expression
        cell = Cell.Parse("'" + text, sheet.workbook, col, row);
    }
    DataGridViewSelectedCellCollection dgvscc = dgv.SelectedCells;
    if (dgvscc.Count > 1 && cell is Formula) { // Array formula
      int ulCol = col, ulRow = row, lrCol = col, lrRow = row;
      foreach (DataGridViewCell dgvc in dgvscc) {
        ulCol = Math.Min(ulCol, dgvc.ColumnIndex);
        ulRow = Math.Min(ulRow, dgvc.RowIndex);
        lrCol = Math.Max(lrCol, dgvc.ColumnIndex);
        lrRow = Math.Max(lrRow, dgvc.RowIndex);
      }
```

```
        CellAddr ulCa = new CellAddr(ulCol, ulRow),
                 lrCa = new CellAddr(lrCol, lrRow);
        if (oldArrayFormula != null && arg != null
            && !(oldArrayFormula.caf.ulCa.Equals(ulCa)
                 && oldArrayFormula.caf.lrCa.Equals(lrCa)))
        {
          ErrorMessage("Cannot edit part of array formula");
          arg.Cancel = true;
          return;
        }
        sheet.SetArrayFormula(cell, col, row, ulCa, lrCa);
      } else { // One-cell formula, or constant, or null (parse error)
        if (oldArrayFormula != null && arg != null) {
          ErrorMessage("Cannot edit part of array formula");
          arg.Cancel = true;
          return;
        }
        sheet.SetCell(cell, col, row);
      }
      RecalculateAndShow();
      gui.SetCyclicError("Cyclic dependency");
    }

    private void ErrorMessage(String msg) {
      MessageBox.Show(msg, msg, MessageBoxButtons.OK, MessageBoxIcon.Error);
    }

    // Copy sheet's currently active cell to Clipboard, also in text format

    public void Copy() {
      CellAddr ca = new CellAddr(dgv.CurrentCellAddress);
      DataObject data = new DataObject();
      data.SetData(DataFormats.Text, sheet[ca].Show(ca.col, ca.row, sheet.workbook.format))
;
      data.SetData(ClipboardCell.COPIED_CELL, new ClipboardCell(sheet.Name, ca));
      Clipboard.Clear();
      Clipboard.SetDataObject(data, false);
    }

    // Paste from the Clipboard.  Need to distinguish:
    // 1. Paste from Corecalc formula Copy: preserve sharing
    // 2. Paste from Corecalc cell Cut: adjust referring cells and this, if formula
    // 3. Paste from text (e.g. from Excel), parse to new cell

    public void Paste() {
      if (Clipboard.ContainsData(ClipboardCell.COPIED_CELL)) {
        // Copy Corecalc cell
        ClipboardCell cc = (ClipboardCell)Clipboard.GetData(ClipboardCell.COPIED_CELL);
        Console.WriteLine("Pasting copied cell " + Clipboard.GetText());
        Cell cell = sheet.workbook[cc.FromSheet][cc.FromCellAddr];
        int col, row, cols, rows;
        GetSelectedColsRows(out col, out row, out cols, out rows);
        sheet.PasteCell(cell, col, row, cols, rows);
        RecalculateAndShow();
      } else if (Clipboard.ContainsData(ClipboardCell.CUT_CELL)) {
        // Move Corecalc cell
        Console.WriteLine("Pasting moved cell not implemented.");
      } else if (Clipboard.ContainsText()) {
        // Insert text (from external source)
        CellAddr ca = new CellAddr(dgv.CurrentCellAddress);
        String text = Clipboard.GetText();
        Console.WriteLine("Pasting text " + text);
        SetCell(ca.col, ca.row, text);
      }
    }

    private void GetSelectedColsRows(out int col, out int row, out int cols, out int rows)
{
      // Apparently SelectedColumns and SelectedRows only concern selection
      // of entire columns and rows, so we need to wade through all selected cells.
      int ulCol = int.MaxValue, ulRow = int.MaxValue,
```



```
          lrCol = int.MinValue, lrRow = int.MinValue;
      foreach (DataGridViewCell dgvc in dgv.SelectedCells) {
        ulCol = Math.Min(ulCol, dgvc.ColumnIndex);
        ulRow = Math.Min(ulRow, dgvc.RowIndex);
        lrCol = Math.Max(lrCol, dgvc.ColumnIndex);
        lrRow = Math.Max(lrRow, dgvc.RowIndex);
      }
      col = ulCol;
      cols = lrCol - ulCol + 1;
      row = ulRow;
      rows = lrRow - ulRow + 1;
    }

    public void Delete() {
      CellAddr ca = new CellAddr(dgv.CurrentCellAddress);
      sheet.SetCell(null, ca.col, ca.row);
      RecalculateAndShow();
    }

    public void InsertRows(int N) {
      sheet.InsertRowCols(dgv.CurrentCellAddress.Y, N, doRows: true);
      RecalculateAndShow();
    }

    public void InsertColumns(int N) {
      sheet.InsertRowCols(dgv.CurrentCellAddress.X, N, doRows: false);
      RecalculateAndShow();
    }

    public void RecalculateAndShow() {
      long elapsed = sheet.workbook.Recalculate();
      Reshow();
      gui.SetStatusLine(elapsed);
    }

    public void Reshow() {
      sheet.ShowAll(delegate(int c, int r, String value) { dgv[c, r].Value = value; });
      if (FunctionSheet) {
        foreach (FullCellAddr fca in SdfManager.cellToFunctionMapper.addressToFunctionList.
Keys)
          if (fca.sheet == sheet)
            ChangeCellBackgroundColor(fca, Color.LightCyan);
        foreach (FullCellAddr fca in SdfManager.cellToFunctionMapper.inputCellBag)
          if (fca.sheet == sheet)
            ChangeCellBackgroundColor(fca, Color.LightGreen);
        foreach (FullCellAddr fca in SdfManager.cellToFunctionMapper.outputCellBag)
          if (fca.sheet == sheet)
            ChangeCellBackgroundColor(fca, Color.LightSkyBlue);
      }
      gui.formulaBox.Text = (String)dgv.CurrentCell.FormattedValue;
      dgv.Focus();
    }

    public void ChangeCurrentText(String text) {
      dgv.CurrentCell.Value = text;
    }

    public void SetCurrentCell(String text) {
      CellAddr ca = new CellAddr(dgv.CurrentCellAddress);
      SetCell(ca.col, ca.row, text);
    }

    public void ChangeCellBackgroundColor(FullCellAddr fca, Color c) {
      dgv[fca.ca.col, fca.ca.row].Style.BackColor = c;
    }

    public void SetCellErrorText(CellAddr ca, String message) {
      dgv[ca.col, ca.row].ErrorText = message;
    }

    public void ScrollTo(FullCellAddr fca) {
```

```
      dgv.FirstDisplayedScrollingRowIndex = Math.Max(0, fca.ca.row - 1);
    }

    public string SheetName {
      get { return sheet.Name; }
    }

    public bool FunctionSheet {
      get { return sheet.IsFunctionSheet; }
      set { sheet.IsFunctionSheet = value; }
    }
  }

  /// <summary>
  /// A ClipboardCell is a copied cell and its cell address for holding
  /// in the MS Windows clipboard.
  /// </summary>
[Serializable]
  public class ClipboardCell {
    public const String COPIED_CELL = "CopiedCell";
    public const String CUT_CELL = "CutCell";
    public readonly String FromSheet;
    public readonly CellAddr FromCellAddr;

    public ClipboardCell(String fromSheet, CellAddr fromCellAddr) {
      this.FromSheet = fromSheet;
      this.FromCellAddr = fromCellAddr;
    }
  }
}
```



```
// Funcalc, spreadsheet with functions
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using Corecalc.Funcalc;   // SdfInfo, SdfManager

namespace Corecalc.GUI {
  public partial class SdfForm : Form {
    private WorkbookForm gui;

    public SdfForm(WorkbookForm gui, Workbook wb) {
      InitializeComponent();
      this.gui = gui;
      PopulateFunctionListBox(false);
    }

    public void PopulateFunctionListBox(bool rememberSelectedIndex) {
      int selectedIndex = functionsListBox.SelectedIndex;
      string selectedName = "";
      if (selectedIndex != -1)
        selectedName = ((SdfInfo)functionsListBox.Items[selectedIndex]).name;

      functionsListBox.Items.Clear();
      int i = 0;
      foreach (SdfInfo info in SdfManager.GetAllInfos()) {
        functionsListBox.Items.Add(info);
        if (selectedName == info.name)
          selectedIndex = i;
        i++;
      }
      if (rememberSelectedIndex && selectedIndex != -1)
        functionsListBox.SelectedIndex = selectedIndex;
    }

    public void PopulateFunctionListBox(string name) {
      name = name.ToUpper();
      for (int i = 0; i < functionsListBox.Items.Count; i++)
        if (((SdfInfo)functionsListBox.Items[i]).name.ToUpper() == name)
          functionsListBox.SelectedIndex = i;
    }

    private void functionsListbox_DoubleClick(object sender, EventArgs e) {
      RefreshInfo();
```

```
    }

    private void RefreshInfo() {
      if (functionsListBox.SelectedItem != null) {
        SdfInfo info = (SdfInfo)functionsListBox.SelectedItem;
        int minCol = info.outputCell.ca.col, minRow = info.outputCell.ca.row;
        foreach (FullCellAddr cell in info.inputCells) {
          minCol = Math.Min(minCol, cell.ca.col);
          minRow = Math.Min(minRow, cell.ca.row);
        }
        gui.ChangeFocus(new FullCellAddr(info.outputCell.sheet, minCol, minRow));
      }
    }

    private void ShowBytecode_Click(object sender, EventArgs e) {
      if (functionsListBox.SelectedItem != null) {
        SdfInfo info = (SdfInfo)functionsListBox.SelectedItem;
        SdfManager.ShowIL(info);
      }
    }
  }
}
```



```
// Funcalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006–2014 Thomas S. Iversen, Peter Sestoft

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.Diagnostics;            // Stopwatch
using System.Globalization;          // CultureInfo
using System.IO;                                        // MemoryStream, Stream
using System.Text;
using System.Xml;
using System.Windows.Forms;
using Corecalc;

namespace Corecalc.IO {
  /// <summary>
  /// An IOFormat determines how to read a given XML file containing a
  /// spreadsheet workbook.
  /// Currently, only Excel 2003 XMLSS format is supported.
  /// </summary>
  abstract class IOFormat {
    public abstract Workbook Read(String filename);
    private String fileExtension;
    private String description;

    public static Stream MakeStream(String s) {
      char[] cs = s.ToCharArray();
      byte[] bs = new byte[cs.Length];
      for (int i = 0; i < cs.Length; i++)
        bs[i] = (byte)(cs[i]);
      return new MemoryStream(bs);
    }

    public String GetFilter() {
      return description + "(*." + fileExtension + ")|*." + fileExtension;
    }

    public IOFormat(String fileextension, String description) {
      this.fileExtension = fileextension;
      this.description = description;
    }

    public Boolean ValidExtension(String ext) {
      return this.fileExtension == ext;
    }
  }

  /// <summary>
  /// Class XMLSSIOFormat can read XMLSS (Excel 2003 XML format) workbook files.
  /// </summary>
```

```
sealed class XMLSSIOFormat : IOFormat {
  const int MINROWS = 10, MINCOLS = 10;  // Must be positive

  readonly Formats fo = new Formats();

  public XMLSSIOFormat()
    : base("xml", "XMLSS") {
    fo.RefFmt = Formats.RefType.R1C1;
  }

  private int ParseRow(XmlReader rowreader, Workbook wb, Sheet sheet, int row,
                       IDictionary<String, Cell> cellParsingCache)
  {
    /* XMLSS has origo at (1,1) = (A,1) whereas Corecalc internal
     * representation has origo at (0,0) = (A,1).  Hence the col
     * and row indices are 1 higher in this method.
     */
    int cellCount = 0;
    int col = 0;
    XmlReader cellreader = rowreader.ReadSubtree();
    while (cellreader.ReadToFollowing("Cell")) {
      String colindexstr = cellreader.GetAttribute("ss:Index");
      String arrayrangestr = cellreader.GetAttribute("ss:ArrayRange");
      String formulastr = cellreader.GetAttribute("ss:Formula");
      String typestr = "";
      String dataval = "";

      if (colindexstr != null) {
        if (!int.TryParse(colindexstr, out col))
          col = 0; // Looks wrong, should be 1?
      } else
        col++;

      cellCount++;
      // If an array result occupies cells, do not overwrite
      // the formula with precomputed and cached data from
      // the XMLSS file. Instead skip the parsing and sheetupdate.
      if (sheet[col - 1, row - 1] != null)
        continue;

      using (XmlReader datareader = cellreader.ReadSubtree()) {
        if (datareader.ReadToFollowing("Data")) {
          typestr = datareader.GetAttribute("ss:Type");
          datareader.MoveToContent();
          dataval = datareader.ReadElementContentAsString();
        }
      }

      String cellString;
      if (formulastr != null)
        cellString = formulastr;
      else {
        // Anything else than formulas are values.
        // If XMLSS tells us it is a String we believe it
        if (typestr == "String")
          dataval = "'" + dataval;
        cellString = dataval;
      }

      // Skip blank cells
      if (cellString == "")
        continue;

      Cell cell;
      if (cellParsingCache.TryGetValue(cellString, out cell)) {
        // Copy the cell (both for mutable Formula cells and for cell-specific
        // metadata) and but share any sharable contents.
        cell = cell.CloneCell(col - 1, row - 1);
      } else {
        // Cell contents not seen before: scan, parse and cache
        cell = Cell.Parse(cellString, wb, col, row);
```



```
        if (cell == null)
          Console.WriteLine("BAD: Null cell from \"{0}\"", cellString);
        else
          cellParsingCache.Add(cellString, cell);
      }

      if (arrayrangestr != null && cell is Formula) { // Array formula
        string[] split = arrayrangestr.Split(":".ToCharArray());

        RARef raref1 = new RARef(split[0]);
        RARef raref2;
        if (split.Length == 1) {
          // FIXME: single cell result, but still array
          raref2 = new RARef(split[0]);
        } else {
          raref2 = new RARef(split[1]);
        }

        if (raref1 != null && raref2 != null) {
          CellAddr ulCa = raref1.Addr(col - 1, row - 1);
          CellAddr lrCa = raref2.Addr(col - 1, row - 1);
          // This also updates support sets, but that's useless, because
          // they will subsequently be reset by RebuildSupportGraph
          sheet.SetArrayFormula(cell, col - 1, row - 1, ulCa, lrCa);
        }
      } else {   // One-cell formula, or constant
        sheet[col - 1, row - 1] = cell;
      }
    }
    cellreader.Close();
    return cellCount;
  }

  private int ParseSheet(XmlReader sheetreader, Workbook wb, Sheet sheet,
                         IDictionary<String, Cell> cellParsingCache)
  {
    int cellCount = 0;
    if (sheetreader.ReadToFollowing("Table")) {
      int row = 0;
      using (XmlReader rowreader = sheetreader.ReadSubtree()) {
        while (rowreader.ReadToFollowing("Row")) {
          String rowindexstr = rowreader.GetAttribute("ss:Index");
          if (rowindexstr != null) {
            if (!int.TryParse(rowindexstr, out row))
              row = 0;
          } else
            row++;
          cellCount += ParseRow(rowreader, wb, sheet, row, cellParsingCache);
        }
      }
    }
    return cellCount;
  }

  private int ParseSheets(XmlTextReader reader, Workbook wb,
                          IDictionary<String, Cell> cellParsingCache)
  {
    int cellCount = 0;
    while (reader.ReadToFollowing("Worksheet")) {
      String sheetname = reader.GetAttribute("ss:Name");
      using (XmlReader sheetreader = reader.ReadSubtree()) {
        Sheet sheet = wb[sheetname];
        cellCount += ParseSheet(sheetreader, wb, sheet, cellParsingCache);
        if (sheet.IsFunctionSheet)
          ScanSheet(sheet, RegisterSdfs);
      }
    }
    return cellCount;
  }

  private void ScanSheet(Sheet sheet, Action<Sheet, int, int> f) {
```

```
    for (int row = 0; row < sheet.Rows; row++)
      for (int col = 0; col < sheet.Cols; col++)
        f(sheet, col, row);
  }

  // Register SDFs (and maybe later: convert DELAY calls to DelayCell).
  private void RegisterSdfs(Sheet sheet, int col, int row) {
    Cell cell = sheet[col, row];
    if (cell == null || !(cell is Formula))
      return;
    Expr e = (cell as Formula).Expr;
    if (!(e is FunCall))
      return;
    FunCall funCall = e as FunCall;
    Expr[] es = funCall.es;
    switch (funCall.function.name) {
      case "DEFINE":
        if (es.Length >= 2 && es[0] is TextConst && es[1] is CellRef) {
          String sdfName = (es[0] as TextConst).value.value;
          FullCellAddr outputCell = (es[1] as CellRef).GetAbsoluteAddr(sheet, col, row);
          FullCellAddr[] inputCells = new FullCellAddr[es.Length - 2];
          bool ok = true;
          for (int i = 2; ok && i < es.Length; i++) {
            CellRef inputCellRef = es[i] as CellRef;
            ok = inputCellRef != null;
            if (ok)
              inputCells[i - 2] = inputCellRef.GetAbsoluteAddr(sheet, col, row);
          }
          if (ok)
            Funcalc.SdfManager.Register(outputCell, inputCells, sdfName);
        }
        break;
      case "DELAY":
        break;
      default:
        /* do nothing */
        break;
    }
  }

  // Create SDFs
  private void CreateSdfs(Sheet sheet, int col, int row) {
    Cell cell = sheet[col, row];
    if (cell == null || !(cell is Formula))
      return;
    Expr e = (cell as Formula).Expr;
    if (!(e is FunCall))
      return;
    FunCall funCall = e as FunCall;
    Expr[] es = funCall.es;
    switch (funCall.function.name) {
      case "DEFINE":
        if (es.Length >= 2 && es[0] is TextConst && es[1] is CellRef) {
          String sdfName = (es[0] as TextConst).value.value;
          FullCellAddr outputCell = (es[1] as CellRef).GetAbsoluteAddr(sheet, col, row);
          FullCellAddr[] inputCells = new FullCellAddr[es.Length - 2];
          bool ok = true;
          for (int i = 2; ok && i < es.Length; i++) {
            CellRef inputCellRef = es[i] as CellRef;
            ok = inputCellRef != null;
            if (ok)
              inputCells[i - 2] = inputCellRef.GetAbsoluteAddr(sheet, col, row);
          }
          if (ok)
            Funcalc.SdfManager.CreateFunction(sdfName, outputCell, inputCells);
        }
        break;
      default:
        /* do nothing */
        break;
    }
```



```
  }

  private Boolean isXMLSS(XmlTextReader reader) {
    return reader.NodeType == XmlNodeType.Element &&
           reader.Name == "Workbook" &&
           reader.GetAttribute("xmlns") != null;
  }

  private void ParseWorkBook(XmlTextReader reader, Workbook wb) {
    int cellCount = 0;
    // To help find and share cells with identical R1C1 representation:
    IDictionary<String, Cell> cellParsingCache = new Dictionary<String, Cell>();
    while (reader.Read())
      if (isXMLSS(reader))
        cellCount += ParseSheets(reader, wb, cellParsingCache);
    foreach (Sheet sheet in wb)
      if (sheet.IsFunctionSheet)
        ScanSheet(sheet, CreateSdfs);
    wb.RebuildSupportGraph();
    wb.ResetVolatileSet();
    Console.Write("Read XMLSS, {0} cells of which {1} unique ", cellCount, cellParsingCache.Count);
  }

  private Workbook MakeEmptySheets(XmlTextReader reader) {
    Workbook wb = null;
    while (reader.Read()) {
      if (isXMLSS(reader)) {
        wb = new Workbook();
        while (reader.ReadToFollowing("Worksheet")) {
          String sheetname = reader.GetAttribute("ss:Name");
          bool functionSheet = sheetname.StartsWith("@");
          using (XmlReader sheetreader = reader.ReadSubtree()) {
            int cols = MINCOLS, rows = MINROWS; // In the sheet
            if (sheetreader.ReadToFollowing("Table")) {
              // Try to use Expanded{Row,Column}Count although not required
              String tmpstr = sheetreader.GetAttribute("ss:ExpandedRowCount");
              if (int.TryParse(tmpstr, out rows))
                rows = Math.Max(rows, MINROWS);
              tmpstr = sheetreader.GetAttribute("ss:ExpandedColumnCount");
              if (int.TryParse(tmpstr, out cols))
                cols = Math.Max(cols, MINCOLS);
            }
            new Sheet(wb, sheetname, cols, rows, functionSheet);
          }
        }
      }
    }
    return wb;
  }

  public override Workbook Read(String filename) {
    XmlTextReader reader = null;
    Workbook wb = null;
    Stopwatch watch = new Stopwatch();
    watch.Reset(); watch.Start();
    try {
      reader = new XmlTextReader(filename);
      reader.WhitespaceHandling = WhitespaceHandling.None;
      wb = MakeEmptySheets(reader);
      if (reader != null)
        reader.Close();
      if (wb != null) {
        reader = new XmlTextReader(filename);
        reader.WhitespaceHandling = WhitespaceHandling.None;
        ParseWorkBook(reader, wb);
      }
    } catch (IOException exn) {
      String msg = "Cannot read workbook";
      MessageBox.Show(msg + "\n" + exn, msg, MessageBoxButtons.OK, MessageBoxIcon.Error);
    } finally {
      if (reader != null)
```

```
        reader.Close();
      }
      watch.Stop();
      Console.WriteLine("in {0} ms", watch.ElapsedMilliseconds);
      return wb;
    }
  }

  /// <summary>
  /// A WorkBookIO can read certain XML workbook files.
  /// </summary>
  public class WorkBookIO {
    private List<IOFormat> formats;
    private IOFormat defaultformat;
    public WorkBookIO() {
      formats = new List<IOFormat>();
      AddFormat(new XMLSSIOFormat(), def: true);
      // AddFormat(new GnumericIOFormat(), def: false); // Removed
    }

    private void AddFormat(IOFormat format, bool def) {
      formats.Add(format);
      if (def)
        defaultformat = format;
    }

    private IOFormat FindFormat(String filename) {
      String[] fields = filename.Split(('.').ToCharArray());
      String ext = fields[fields.Length - 1];
      foreach (IOFormat format in formats) {
        if (format.ValidExtension(ext))
          return format;
      }
      return null;
    }

    // Attempt to read workbook from file using a supported
    // format; may return null.

    public Workbook Read(String filename) {
      IOFormat format = FindFormat(filename);
      // If we found a format, try it
      if (format != null)
        return format.Read(filename);
      else
        return null;
    }

    public String SupportedFormatFilter() {
      StringBuilder sb = new StringBuilder();
      foreach (IOFormat ioformat in formats) {
        sb.Append(ioformat.GetFilter());
        sb.Append("|");
      }

      sb.Append("All files (*.*)|*.*");
      return sb.ToString();
    }

    public int DefaultFormatIndex() {
      return formats.IndexOf(defaultformat) + 1;
    }
  }
}
```



```
// Corecalc, a spreadsheet core implementation

// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Collections.Generic;
using System.IO;
using Corecalc.IO;

namespace Corecalc {
  /// <summary>
  /// Class Program contains the Main function to start the GUI version of
  /// Funcalc.
  /// </summary>
  class Program {
  [STAThread]
    public static void Main(String[] args) {
      if (args.Length == 0) {
        // GUI, with new empty workbook
        new WorkbookForm(new Workbook(), display: true);
        return;
      } else if (args.Length == 1) {
        FileInfo fi = new FileInfo(args[0]);
        Console.WriteLine(fi);
        switch (fi.Extension) {
          case ".xml": // Attempt to open existing workbook in GUI
            Workbook wb = new WorkBookIO().Read(fi.FullName);
            if (wb != null)
              new WorkbookForm(wb, display: true);
            return;
          default:
            break;
        }
      }
      Console.WriteLine("Usage: Funcalc [workbook.xml]\n");
    }
  }
}
```

```
ï»¿using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("Funcalc 2014")]
[assembly: AssemblyDescription(@"A core spreadsheet implementation with runtime code generation for sheet-defined functions.
For background, see Peter Sestoft: Spreadsheet Implementation Technology. Basics and Extensions.  MIT Press 2014, and http://www.itu.dk/people/sestoft/corecalc/")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("IT University of Copenhagen, Denmark")]
[assembly: AssemblyProduct("Funcalc")]
[assembly: AssemblyCopyright("Copyright Â© Peter Sestoft and others 2006-2014")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components.  If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("574a275c-4946-460c-b160-b12e7072aef0")]

// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
[assembly: AssemblyVersion("0.14.0.0")]
[assembly: AssemblyFileVersion("0.14.0.0")]
[assembly: InternalsVisibleTo("ExcelCalc")]
```



```
// Funcalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Diagnostics;
using SC = System.Collections;
using System.Collections.Generic;
using System.Text;

namespace Corecalc {
  /// <summary>
  /// A Sheet is a rectangular [col,row]-indexable collection of Cells,
  /// represented by a SheetRep object.
  /// A Sheet belongs to a single Workbook, and only once.
  /// </summary>
  public sealed class Sheet : IEnumerable<Cell> {
    public const int cols = 20, rows = 1000;   // Default sheet size
    private String name;
    public readonly Workbook workbook;         // Non-null
    private readonly SheetRep cells;
    public int Cols { get; private set; }
    public int Rows { get; private set; }

    private bool isFunctionSheet;

    public Sheet(Workbook workbook, String name, bool functionSheet)
      : this(workbook, name, cols, rows, functionSheet) {
    }

    public Sheet(Workbook workbook, String name, int cols, int rows, bool functionSheet) {
      this.workbook = workbook;
      this.name = name;
      this.cells = new SheetRep();
      Cols = cols;
      Rows = rows;
      this.isFunctionSheet = functionSheet;
      workbook.AddSheet(this);
    }

    // Recalculate all cells by evaluating their contents
    public void RecalculateFull() {
      cells.Forall((col, row, cell) => cell.Eval(this, col, row));
    }

    // Show the contents of all non-null cells
    public void ShowAll(Action<int,int,String> show) {
      // Should use cells.Forall(show) but that may not clean up newly empty cells?
      for (int col = 0; col < Cols; col++)
        for (int row = 0; row < Rows; row++) {
```

```
          Cell cell = cells[col, row];
          show(col, row, cell != null ? ShowValue(col, row) : null);
        }
    }

    // Reset recomputation flags after a circularity has been found
    public void ResetCellState() {
      foreach (Cell cell in cells)
        if (cell != null)
          cell.ResetCellState();
    }

    // From parsed constant or formula or null but not array formula, at sheet[col, row]
    public void SetCell(Cell cell, int col, int row) {
      Debug.Assert(!(cell is ArrayFormula));
      this[col, row] = cell;
      if (cell != null)
        cell.AddToSupportSets(this, col, row, 1, 1);
    }

    // Insert cell, which must be Formula, as array formula
    // in area ((ulCol,ulRow), (lrCol, lrRow))
    public void SetArrayFormula(Cell cell, int col, int row, CellAddr ulCa, CellAddr lrCa)
    {
      Formula formula = cell as Formula;
      if (cell == null)
        throw new Exception("Invalid array formula");
      else {
        CachedArrayFormula caf = new CachedArrayFormula(formula, this, col, row, ulCa, lrCa
);
        // Increase support sets of cells referred by formula
        formula.AddToSupportSets(this, col, row, 1, 1);
        Interval displayCols = new Interval(ulCa.col, lrCa.col),
                 displayRows = new Interval(ulCa.row, lrCa.row);
        // The underlying formula supports (only) the ArrayFormula cells in display range
        formula.ResetSupportSet();
        formula.AddSupport(this, col, row, this, displayCols, displayRows);
        int cols = lrCa.col - ulCa.col + 1, rows = lrCa.row - ulCa.row + 1;
        for (int c = 0; c < cols; c++)
          for (int r = 0; r < rows; r++)
            this[ulCa.col + c, ulCa.row + r] = new ArrayFormula(caf, c, r);
      }
    }

    // Copy cell to area ((col,row), (col+cols-1,row+rows-1))
    // Probably makes no sense when the cell is an array formula
    public void PasteCell(Cell cell, int col, int row, int cols, int rows) {
      for (int c = 0; c < cols; c++)
        for (int r = 0; r < rows; r++)
          // TODO: When cell contains CellRef or CellArea, should check that
          // references are not invalid, as in A0 or A-1:B2
          this[col + c, row + r] = cell.CloneCell(col, row);
      cell.AddToSupportSets(this, col, row, cols, rows);
    }

    // Move cell (fromCol, fromRow) to cell (col, row)
    public void MoveCell(int fromCol, int fromRow, int col, int row) {
      Cell cell = cells[fromCol, fromRow];
      this[col, row] = cell.MoveContents(col - fromCol, row - fromRow);
    }

    // Insert N new rows just before row R >= 0
    // TODO: Consider replacing all assignments to cells[,] with assignments to this[,],
    // for proper maintenance of support sets and volatile status
    public void InsertRowCols(int R, int N, bool doRows) {
      // Check that this will not split a array formula
      if (R >= 1)
        if (doRows) {
          for (int col = 0; col < Cols; col++) {
            Cell cell = cells[col, R - 1];
            ArrayFormula mf = cell as ArrayFormula;
```



```
            if (mf != null && mf.Contains(col, R))
              throw new Exception("Row insert would split array formula");
          }
        }
        else {
          for (int row = 0; row < Rows; row++) {
            Cell cell = cells[R - 1, row];
            ArrayFormula mf = cell as ArrayFormula;
            if (mf != null && mf.Contains(R, row))
              throw new Exception("Column insert would split array formula");
          }
        }
      // Adjust formulas in all sheets.  The dictionary records adjusted
      // expressions to preserve sharing of expressions where possible.
      Dictionary<Expr, Adjusted<Expr>> adjusted
        = new Dictionary<Expr, Adjusted<Expr>>();
      foreach (Sheet sheet in workbook) {
        for (int r = 0; r < sheet.Rows; r++)
          for (int c = 0; c < sheet.Cols; c++) {
            Cell cell = sheet.cells[c, r];
            if (cell != null)
              cell.InsertRowCols(adjusted, this, sheet == this, R, N,
                doRows ? r : c, doRows);
          }
      }
      if (doRows) {
        // Move the rows R, R+1, ... later by N rows in current sheet
        for (int r = Rows - 1; r >= R + N; r--)
          for (int c = 0; c < Cols; c++)
            cells[c, r] = cells[c, r - N];
        // Finally, null out the fresh rows
        for (int r = 0; r < N; r++)
          for (int c = 0; c < Cols; c++)
            cells[c, r + R] = null;
      }
      else {
        // Move the columns R, R+1, ... later by N columns in current sheet
        for (int c = Cols - 1; c >= R + N; c--)
          for (int r = 0; r < Rows; r++)
            cells[c, r] = cells[c - N, r];
        // Finally, null out the fresh columns
        for (int c = 0; c < N; c++)
          for (int r = 0; r < Rows; r++)
            cells[c + R, r] = null;
      }
    }

    // Show contents, if any, of cell at (col, row)
    public String Show(int col, int row) {
      if (0 <= col && col < Cols && 0 <= row && row < Rows) {
        Cell cell = cells[col, row];
        if (cell != null)
          return cell.Show(col, row, workbook.format);
      }
      return null;
    }

    // Show value (or contents), if any, of cell (col, row)
    public String ShowValue(int col, int row) {
      if (0 <= col && col < Cols && 0 <= row && row < Rows) {
        Cell cell = cells[col, row];
        if (cell != null)
          if (workbook.format.ShowFormulas)
            return cell.Show(col, row, workbook.format);
          else
            return cell.ShowValue(this, col, row);
      }
      return null;
    }

    // Get and set cell contents, maintaining support and volatility information
```

```
    public Cell this[int col, int row] {
      get {
        return col < Cols && row < Rows ? cells[col, row] : null;
      }
      set {
        if (col < Cols && row < Rows) {
          Cell oldCell = cells[col, row];
          if (oldCell != value) {
            if (oldCell != null) {
              oldCell.TransferSupportTo(ref value);  // Assumes oldCell used nowhere else
              workbook.DecreaseVolatileSet(oldCell, this, col, row);
              oldCell.RemoveFromSupportSets(this, col, row);
            }
            workbook.IncreaseVolatileSet(value, this, col, row);
            // We do not add to support sets here; that would fragment the support sets
            cells[col, row] = value;
            workbook.RecordCellChange(col, row, this);
          }
        }
      }
    }

    public Cell this[CellAddr ca] {
      get {
        return this[ca.col, ca.row];
      }
      private set {
        this[ca.col, ca.row] = value;
      }
    }

    public String Name {
      get { return name; }
      set { name = value; }
    }

    public bool IsFunctionSheet {
      get { return isFunctionSheet; }
      set { isFunctionSheet = value; }
    }

    public IEnumerator<Cell> GetEnumerator() {
      return cells.GetEnumerator();
    }

    SC.IEnumerator SC.IEnumerable.GetEnumerator() {
      return GetEnumerator();
    }

    // Detect blocks of formula copies, for finding compact support sets
    public void AddToSupportSets() {
      int sheetCols = Cols, sheetRows = Rows;
      cells.Forall((int col, int row, Cell cell) =>
        {
          if (cell is ArrayFormula) {
            // Do not try to detect copies of array formulas.
            // CHECK THIS!
            ArrayFormula af = cell as ArrayFormula;
            af.AddToSupportSets(this, col, row, 1, 1);
          }
          else if (cell is Formula) {
            Formula f = cell as Formula;
            if (!f.Visited) {
              Expr expr = f.Expr;
              // (1) Find a large rectangle containing only formula f
              // (1a) Find largest square of copies with upper left corner = (col, row)
              int size = 1;
              while (col + size < sheetCols && row + size < sheetRows
                     && CheckCol(col + size, row, expr, size)
                     && CheckRow(col, row + size, expr, size)) {
                size++;
```



```
              }
              // All cells in sheet[col..col+size-1, row..row+size-1] contain expression ex
pr
              // (1b) Try extending square with more rows below
              int rows = size;
              while (row + rows < sheetRows && CheckRow(col, row + rows, expr, size - 1))
                rows++;
              // sheet[col..col+size-1, row..row+rows-1] contains expr
              // (1c) Try extending square with more columns to the right
              int cols = size;
              while (col + cols < sheetCols && CheckCol(col + cols, row, expr, size - 1))
                cols++;
              // sheet[col..col+cols-1, row..row+size-1] contains expr
              if (rows > cols)
                cols = size;
              else
                rows = size;
              // All cells in sheet[col..col+cols-1, row..row+rows-1] contain expression ex
pr
              // (2) Mark all cells in the rectangle visited
              for (int deltaCol = 0; deltaCol < cols; deltaCol++)
                for (int deltaRow = 0; deltaRow < rows; deltaRow++)
                  (this[col + deltaCol, row + deltaRow] as Formula).Visited = true;
              // (3) Update the support sets of cells referred to from expr
              expr.AddToSupportSets(this, col, row, cols, rows);
            }
          }
        });
      this.ResetCellState(); // Undo changes made to the sheet's cells' states
    }

    // Check the row sheet[col..col+size, row] for formulas identical to expr
    private bool CheckRow(int col, int row, Expr expr, int size) {
      for (int deltaCol=0; deltaCol<=size; deltaCol++) {
        Formula fcr = this[col+deltaCol, row] as Formula;
        if (fcr == null || fcr.Visited || fcr.Expr != expr)
          return false;
      }
      return true;
    }

    // Check the column sheet[col, row..row+size] for formulas identical to expr
    private bool CheckCol(int col, int row, Expr expr, int size) {
      for (int deltaRow=0; deltaRow<=size; deltaRow++) {
        Formula fcr = this[col, row+deltaRow] as Formula;
        if (fcr == null || fcr.Visited || fcr.Expr != expr)
          return false;
      }
      return true;
    }

    // Add supportedSheet[supportedColumns, supportedRows] to support set of this[col,row]
    public void AddSupport(int col, int row, Sheet supportedSheet,
                           Interval supportedCols, Interval supportedRows)
    {
      if (this[col, row] == null)
        this[col, row] = new BlankCell();
      if (this[col, row] != null) // May still be null because outside sheet
        this[col, row].AddSupport(this, col, row, supportedSheet, supportedCols, supportedR
ows);
    }

    public void IncreaseVolatileSet() {
      cells.Forall((col, row, cell) => workbook.IncreaseVolatileSet(cell, this, col, row));
    }

    public override string ToString() {
      return name;
    }
  }
```

```
  /// <summary>
  /// A SheetRep represents a sheet's cell array sparsely, quadtree style, with four
  /// levels of tiles, each conceptually a 16-column by 32-row 2D array,
  /// for up to SIZEW = 2^16 = 64K columns and SIZEH = 2^20 = 1M rows.

  /// </summary>
  class SheetRep : IEnumerable<Cell> {
    // Sizes are chosen so that addresses can be calculated by bit-operations,
    // and the 2D tiles are represented by 1D arrays for speed.  The mask
    // MW equals ...01111 with LOGW 1's, so (c&MW) equals (c%M); MH ditto.

    private const int // Could be uint, but then indexes must be cast to int
      LOGW = 4, W = 1 << LOGW, MW = W - 1, SIZEW = 1 << (4 * LOGW), // cols
      LOGH = 5, H = 1 << LOGH, MH = H - 1, SIZEH = 1 << (4 * LOGH); // rows
    private readonly Cell[][][][][] tile0 = new Cell[W * H][][][];

    public Cell this[int c, int r] {
      get {
        if (c < 0 || SIZEW <= c || r < 0 || SIZEH <= r)
          return null;
        Cell[][][][] tile1 = tile0[((c >> (3 * LOGW)) & MW) << LOGH) | ((r >> (3 * LOGH)) &
MH)];
        if (tile1 == null)
          return null;
        Cell[][][] tile2 = tile1[((c >> (2 * LOGW)) & MW) << LOGH) | ((r >> (2 * LOGH)) & MH
)];
        if (tile2 == null)
          return null;
        Cell[] tile3 = tile2[((c >> (1 * LOGW)) & MW) << LOGH) | ((r >> (1 * LOGH)) & MH)]
;
        if (tile3 == null)
          return null;
        return tile3[((c & MW) << LOGH) | (r & MH)];
      }
      set {
        if (c < 0 || SIZEW <= c || r < 0 || SIZEH <= r)
          return;
        int index0 = (((c >> (3 * LOGW)) & MW) << LOGH) | ((r >> (3 * LOGH)) & MH);
        Cell[][][][] tile1 = tile0[index0];
        if (tile1 == null)
          if (value == null)
            return;
          else
            tile1 = tile0[index0] = new Cell[W * H][][];
        int index1 = (((c >> (2 * LOGW)) & MW) << LOGH) | ((r >> (2 * LOGH)) & MH);
        Cell[][][] tile2 = tile1[index1];
        if (tile2 == null)
          if (value == null)
            return;
          else
            tile2 = tile1[index1] = new Cell[W * H][];
        int index2 = (((c >> (1 * LOGW)) & MW) << LOGH) | ((r >> (1 * LOGH)) & MH);
        Cell[] tile3 = tile2[index2];
        if (tile3 == null)
          if (value == null)
            return;
          else
            tile3 = tile2[index2] = new Cell[W * H];
        int index3 = ((c & MW) << LOGH) | (r & MH);
        tile3[index3] = value;
      }
    }

    // Yield all the sheet's non-null cells
    public IEnumerator<Cell> GetEnumerator() {
      foreach (Cell[][][][] tile1 in tile0)
        if (tile1 != null)
          foreach (Cell[][][] tile2 in tile1)
            if (tile2 != null)
              foreach (Cell[][] tile3 in tile2)
                if (tile3 != null)
```



```
                  foreach (Cell cell in tile3)
                    if (cell != null)
                      yield return cell;
    }

    SC.IEnumerator SC.IEnumerable.GetEnumerator() {
      return GetEnumerator();
    }

    // Sparse iteration, over non-null cells only
    public void Forall(Action<int, int, Cell> act) {
      int i0 = 0;
      foreach (Cell[][][] tile1 in tile0) {
        int i1 = 0, c0 = (i0 >> LOGH) << (3 * LOGW), r0 = (i0 & MH) << (3 * LOGH);
        if (tile1 != null)
          foreach (Cell[][] tile2 in tile1) {
            int i2 = 0, c1 = (i1 >> LOGH) << (2 * LOGW), r1 = (i1 & MH) << (2 * LOGH);
            if (tile2 != null)
              foreach (Cell[] tile3 in tile2) {
                int i3 = 0, c2 = (i2 >> LOGH) << (1 * LOGW), r2 = (i2 & MH) << (1 * LOGH);
                if (tile3 != null)
                  foreach (Cell cell in tile3) {
                    if (cell != null)
                      act(c0 | c1 | c2 | i3 >> LOGH, r0 | r1 | r2 | i3 & MH, cell);
                    i3++;
                  }
                i2++;
              }
            i1++;
          }
        i0++;
      }
    }
  }
}
```

```
// Funcalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Text;
using System.Collections.Generic;
using SC = System.Collections;

// Delegate types, exception classes, formula formatting options,
// and specialized collection classes

namespace Corecalc {

  /// <summary>
  /// An IDepend is an object such as Cell, Expr, CGExpr, ComputeCell
  /// that can tell what full cell addresses it depends on.
  /// </summary>
  public interface IDepend {
    void DependsOn(FullCellAddr here, Action<FullCellAddr> dependsOn);
  }

  /// <summary>
  /// Applier is the delegate type used to represent implementations of
  /// built-in functions and sheet-defined functions in the interpretive
  /// implementation.
  /// </summary>
  /// <param name="sheet">The sheet containing the cell in which the function is called.</p
aram>
  /// <param name="es">The function call's argument expressions.</param>
  /// <param name="col">The column containing the cell in which the function is called.</pa
ram>
  /// <param name="row">The row containing the cell in which the function is called.</param
>
  /// <returns></returns>
  public delegate Value Applier(Sheet sheet, Expr[] es, int col, int row);

  /// <summary>
  /// A CyclicException signals that a cyclic dependency is discovered
  /// during evaluation.
  /// </summary>
  public class CyclicException : Exception {
    public readonly FullCellAddr culprit;

    public CyclicException(String msg, FullCellAddr culprit) : base(msg) {
      this.culprit = culprit;
    }
  }

  /// <summary>
```



```
  /// An ImpossibleException signals a violation of internal consistency
  /// assumptions in the spreadsheet implementation.
  /// </summary>
  class ImpossibleException : Exception {
    public ImpossibleException(String msg) : base(msg) { }
  }

  /// <summary>
  /// A NotImplementedException signals that something could have
  /// been implemented but was not.
  /// </summary>
  class NotImplementedException : Exception {
    public NotImplementedException(String msg) : base(msg) { }
  }

  // ------------------------------------------------------------------------
  // Formula formatting options

  public class Formats {
    public enum RefType { A1, C0R0, R1C1 }
    private bool showFormulas = false;
    private RefType refFmt = RefType.A1;
    private char argDelim = ',';
    private char rangeDelim = ':';

    public RefType RefFmt {
      get { return refFmt; }
      set { refFmt = value; }
    }

    public char RangeDelim {
      get { return rangeDelim; }
      set { rangeDelim = value; }
    }

    public char ArgDelim {
      get { return argDelim; }
      set { argDelim = value; }
    }

    public bool ShowFormulas
    {
      get { return showFormulas; }
      set { showFormulas = value; }
    }
  }

  // ------------------------------------------------------------------------
  // A hash bag, a replacement for C5.HashBag<T>

  public class HashBag<T> : IEnumerable<T> {
    // Invariant: foreach (k,v) in multiplicity, v>0
    private readonly IDictionary<T, int> multiplicity = new Dictionary<T, int>();

    public bool Add(T item) {
      int count;
      if (multiplicity.TryGetValue(item, out count))
        multiplicity[item] = count + 1;
      else
        multiplicity[item] = 1;
      return true;
    }

    public bool Remove(T item) {
      int count;
      if (multiplicity.TryGetValue(item, out count)) {
        count--;
        if (count == 0)
          multiplicity.Remove(item);
        else
          multiplicity[item] = count;
```

```
      return true;
    } else
      return false;
  }

  public void AddAll(IEnumerable<T> xs) {
    foreach (T x in xs)
      Add(x);
  }

  public void RemoveAll(IEnumerable<T> xs) {
    foreach (T x in xs)
      Remove(x);
  }

  public IEnumerable<KeyValuePair<T,int>> ItemMultiplicities() {
    foreach (KeyValuePair<T, int> entry in multiplicity)
      yield return entry;
  }

  public void Clear() {
    multiplicity.Clear();
  }

  public IEnumerator<T> GetEnumerator() {
    foreach (KeyValuePair<T, int> entry in multiplicity)
      for (int i=0; i<entry.Value; i++)
        yield return entry.Key;
  }

  SC.IEnumerator SC.IEnumerable.GetEnumerator() {
    return GetEnumerator();
  }
}

// ----------------------------------------------------------------
// An data structure that preserves insertion order of unique elements,
// and fast Contains, Add, AddAll, Intersection, Difference, and UnsequencedEquals

public class HashList<T> : IEnumerable<T> where T : IEquatable<T> {
  // Invariants: No duplicates in seq; seq and set have the same
  // sets of items and the same number of items.
  private readonly List<T> seq = new List<T>();
  private readonly HashSet<T> set = new HashSet<T>();

  public bool Contains(T item) {
    return set.Contains(item);
  }

  public int Count { get { return seq.Count; }}

  public bool Add(T item) {
    if (set.Contains(item))
      return false;
    else {
      seq.Add(item);
      set.Add(item);
      return true;
    }
  }

  public void AddAll(IEnumerable<T> xs) {
    foreach (T x in xs)
      Add(x);
  }

  public static HashList<T> Union(HashList<T> ha1, HashList<T> ha2) {
    HashList<T> result = new HashList<T>();
    result.AddAll(ha1);
    result.AddAll(ha2);
    return result;
```



```
  }

  public static HashList<T> Intersection(HashList<T> ha1, HashList<T> ha2) {
    HashList<T> result = new HashList<T>();
    foreach (T x in ha1)
      if (ha2.Contains(x))
        result.Add(x);
    return result;
  }

  public static HashList<T> Difference(HashList<T> ha1, HashList<T> ha2) {
    HashList<T> result = new HashList<T>();
    foreach (T x in ha1)
      if (!ha2.Contains(x))
        result.Add(x);
    return result;
  }

  public bool UnsequencedEquals(HashList<T> that) {
    if (this.Count != that.Count)
      return false;
    foreach (T x in this.seq)
      if (!that.set.Contains(x))
        return false;
    return true;
  }

  public T[] ToArray() {
    return seq.ToArray();
  }

  public IEnumerator<T> GetEnumerator() {
    return seq.GetEnumerator();
  }

  SC.IEnumerator SC.IEnumerable.GetEnumerator() {
    return GetEnumerator();
  }
}

/// <summary>
/// Machinery to cache the creation of objects of type U, when created
/// from objects of type T, and for later access via an integer index.
/// </summary>
/// <typeparam name="T">The type of key, typically String</typeparam>
/// <typeparam name="U">The type of resulting cached item</typeparam>

sealed class ValueCache<T, U> where T : IEquatable<T> {
  // Assumes: function make is monogenic
  // Invariant: array[dict[x]].Equals(make(x))
  private readonly IDictionary<T, int> dict = new Dictionary<T, int>();
  private readonly IList<U> array = new List<U>();
  private readonly Func<int, T, U> make;

  public ValueCache(Func<int, T, U> make) {
    this.make = make;
  }

  public int GetIndex(T x) {
    int index;
    if (!dict.TryGetValue(x, out index)) {
      index = array.Count;
      dict.Add(x, index);
      array.Add(make(index, x));
    }
    return index;
  }

  public U this[int index] {
    get { return array[index]; }
  }
```

```
  }

  /// <summary>
  /// Machinery to store objects of type T for later access via an integer index.
  /// </summary>
  /// <typeparam name="T">The type of item stored in the array</typeparam>

  sealed class ValueTable<T> where T : IEquatable<T> {
    private readonly IDictionary<T, int> dict = new Dictionary<T, int>();
    private readonly IList<T> array = new List<T>();

    public int GetIndex(T x) {
      int index;
      if (!dict.TryGetValue(x, out index)) {
        index = array.Count;
        array.Add(x);
        dict.Add(x, index);
      }
      return index;
    }

    public T this[int index] {
      get { return array[index]; }
    }
  }
}
```



```
// Corecalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006–2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Reflection;
using System.Reflection.Emit;             // LocalBuilder etc
using System.Text;
using System.Diagnostics;
using System.Collections.Generic;
using Corecalc.Funcalc;                   // For SdfInfo

namespace Corecalc {
  /// <summary>
  /// A Value is the result of evaluating an expression in the
  /// interpretive spreadsheet implementation.
  /// </summary>
  public abstract class Value : IEquatable<Value> {
    public static readonly Type type = typeof(Value);
    public static readonly MethodInfo toDoubleOrNanMethod
      = type.GetMethod("ToDoubleOrNan", new Type[] { type });

    public virtual void Apply(Action<Value> act) {
      act(this);
    }

    public abstract bool Equals(Value v);

    // Called from interpreted EXTERN and from generated bytecode
    public static Object ToObject(Value v) {
      return v.ToObject();
    }

    public static double ToDoubleOrNan(Value v) {
      if (v is NumberValue)
        return (v as NumberValue).value;
      else if (v is ErrorValue)
        return (v as ErrorValue).ErrorNan;
      else
        return ErrorValue.argTypeError.ErrorNan;
    }

    // For external methods that do not return anything -- or make it null?
    public static Value MakeVoid(Object o /* ignored */) {
      return TextValue.VOID;
    }

    public abstract Object ToObject();
  }
```

```
/// <summary>
/// A NumberValue holds a floating-point number resulting from evaluation.
/// </summary>
public class NumberValue : Value {
  public readonly double value;

  public static readonly NumberValue
    ZERO = new NumberValue(0.0),
    ONE = new NumberValue(1.0),
    PI = new NumberValue(Math.PI);

  public new static readonly Type type = typeof(NumberValue);
  public static readonly FieldInfo
    zeroField = type.GetField("ZERO"),
    oneField = type.GetField("ONE"),
    piField = type.GetField("PI");
  public static readonly MethodInfo
    makeMethod = type.GetMethod("Make", new Type[] { typeof(double) });

  private NumberValue(double value) {
    Debug.Assert(!Double.IsInfinity(value) && !Double.IsNaN(value));
    this.value = value;
  }

  public static Value Make(double d) {
    if (double.IsInfinity(d))
      return ErrorValue.numError;
    else if (double.IsNaN(d))
      return ErrorValue.FromNan(d);
    else if (d == 0)
      return ZERO;
    else if (d == 1)
      return ONE;
    else
      return new NumberValue(d);
  }

  public override bool Equals(Value v) {
    return v is NumberValue && (v as NumberValue).value == value;
  }

  public override int GetHashCode() {
    return value.GetHashCode();
  }

  public override Object ToObject() {
    return (Object)value;
  }

  public static Object ToDouble(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)nv.value : null;      // Causes boxing
  }

  public static Value FromDouble(Object o) {
    if (o is double)
      return Make((double)o);
    else
      return ErrorValue.numError;
  }

  public static Object ToSingle(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)(float)nv.value : null;       // Causes boxing
  }

  public static Value FromSingle(Object o) {
    if (o is float)
      return Make((float)o);
    else
      return ErrorValue.numError;
```



```
  }

  public static Object ToInt64(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)(long)nv.value : null; // Causes boxing
  }

  public static Value FromInt64(Object o) {
    if (o is long)
      return Make((long)o);
    else
      return ErrorValue.numError;
  }

  public static Object ToInt32(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)(int)nv.value : null; // Causes boxing
  }

  public static Value FromInt32(Object o) {
    if (o is int)
      return Make((int)o);
    else
      return ErrorValue.numError;
  }

  public static Object ToInt16(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)(short)nv.value : null; // Causes boxing
  }

  public static Value FromInt16(Object o) {
    if (o is short)
      return Make((short)o);
    else
      return ErrorValue.numError;
  }

  public static Object ToSByte(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)(sbyte)nv.value : null; // Causes boxing
  }

  public static Value FromSByte(Object o) {
    if (o is sbyte)
      return Make((sbyte)o);
    else
      return ErrorValue.numError;
  }

  public static Object ToUInt64(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)(ulong)nv.value : null; // Causes boxing
  }

  public static Value FromUInt64(Object o) {
    if (o is ulong)
      return Make((ulong)o);
    else
      return ErrorValue.numError;
  }

  public static Object ToUInt32(Value v) {
    NumberValue nv = v as NumberValue;
    return nv != null ? (Object)(uint)nv.value : null; // Causes boxing
  }

  public static Value FromUInt32(Object o) {
    if (o is uint)
      return Make((uint)o);
    else
```

```
        return ErrorValue.numError;
    }

    public static Object ToUInt16(Value v) {
      NumberValue nv = v as NumberValue;
      return nv != null ? (Object)(ushort)nv.value : null; // Causes boxing
    }

    public static Value FromUInt16(Object o) {
      if (o is ushort)
        return Make((ushort)o);
      else
        return ErrorValue.numError;
    }

    public static Object ToByte(Value v) {
      NumberValue nv = v as NumberValue;
      return nv != null ? (Object)(byte)nv.value : null; // Causes boxing
    }

    public static Value FromByte(Object o) {
      if (o is byte)
        return Make((byte)o);
      else
        return ErrorValue.numError;
    }

    public static Object ToBoolean(Value v) {
      NumberValue nv = v as NumberValue;
      return nv != null ? (Object)(nv.value != 0) : null;       // Causes boxing
    }

    public static Value FromBoolean(Object o) {
      if (o is bool)
        return (bool)o ? ONE : ZERO;
      else
        return ErrorValue.numError;
    }

    // Conversion between System.DateTime ticks and Excel-style date numbers
    private static readonly long basedate = new DateTime(1899, 12, 30).Ticks;
    private static readonly double daysPerTick = 100E-9 / 60 / 60 / 24;

    public static double DoubleFromDateTimeTicks(long ticks) {
      return (ticks - basedate) * daysPerTick;
    }

    public override String ToString() {
      return value.ToString();
    }
  }

  /// <summary>
  /// A TextValue holds a string resulting from evaluation.
  /// </summary>
  public class TextValue : Value {
    public readonly String value;  // Non-null

    public new static readonly Type type = typeof(TextValue);
    public static readonly FieldInfo
      valueField = type.GetField("value"),
      emptyField = type.GetField("EMPTY"),
      voidField = type.GetField("VOID");
    public static readonly MethodInfo
      fromIndexMethod = type.GetMethod("FromIndex"),
      fromNakedCharMethod = type.GetMethod("FromNakedChar"),
      toNakedCharMethod = type.GetMethod("ToNakedChar");

    // Caching TextValues by string contents, and for access by integer index
    private static ValueCache<String, TextValue> textValueCache
      = new ValueCache<String, TextValue>((index, s) => new TextValue(s));
```



```
    public static readonly TextValue
      EMPTY = MakeInterned(String.Empty),
      VOID = MakeInterned("<void>");

    private TextValue(String value) {
      this.value = value;
    }

    public static int GetIndex(String s) {
      return textValueCache.GetIndex(s);
    }

    public static TextValue MakeInterned(String s) {
      return textValueCache[textValueCache.GetIndex(s)];
    }

    public static TextValue Make(string s) {
      Debug.Assert(s != null);  // Else use the defensive FromString
      if (s == "")
        return TextValue.EMPTY;
      else
        return new TextValue(s);  // On purpose NOT interned!
    }

    // These five are called also from generated code:

    public static TextValue FromIndex(int index) {
      return textValueCache[index];
    }

    public static Value FromString(Object o) {
      if (o is String)
        return Make(o as String);
      else
        return ErrorValue.argTypeError;
    }

    public static String ToString(Value v) {
      TextValue tv = v as TextValue;
      return tv != null ? tv.value : null;
    }

    public static Value FromNakedChar(char c) {
      return Make(c.ToString());
    }

    public static char ToNakedChar(TextValue v) {
      return v.value.Length >= 1 ? v.value[0] : '\0';
    }

    public static Value FromChar(Object o) {
      if (o is char)
        return Make(((char)o).ToString());
      else
        return ErrorValue.argTypeError;
    }

    public static Object ToChar(Value v) {
      TextValue tv = v as TextValue;
      return tv != null && tv.value.Length >= 1 ? (Object)tv.value[0] : null; // causes box
ing
    }

    public override bool Equals(Value v) {
      return v is TextValue && (v as TextValue).value.Equals(value);
    }

    public override int GetHashCode() {
      return value.GetHashCode();
    }
```

```
    public override Object ToObject() {
      return (Object)value;
    }

    public override String ToString() {
      return value;
    }
  }

  /// <summary>
  /// An ObjectValue holds a .NET object, typically resulting from calling
  /// an external function; may be null.
  /// </summary>
  public class ObjectValue : Value {
    public readonly Object value;
    public static readonly ObjectValue nullObjectValue = new ObjectValue(null);

    public ObjectValue(Object value) {
      this.value = value;
    }

    // Used from EXTERN and from generated bytecode
    public static Value Make(Object o) {
      if (o == null)
        return nullObjectValue;
      else
        return new ObjectValue(o);
    }

    public override bool Equals(Value v) {
      return v is ObjectValue && (v as ObjectValue).value.Equals(value);
    }

    public override Object ToObject() {
      return value;
    }

    public override String ToString() {
      return value == null ? "null" : value.ToString();
    }
  }

  /// <summary>
  /// An ErrorValue is a value indicating failure of evaluation.
  /// </summary>
  public class ErrorValue : Value {
    public readonly String message;
    public readonly int index;

    // Standard ErrorValue objects and static fields, shared between all functions:
    public new static readonly Type type = typeof(ErrorValue);

    public static readonly MethodInfo
      fromNanMethod = type.GetMethod("FromNan"),
      fromIndexMethod = type.GetMethod("FromIndex");

    // Caching ErrorValues by message string, and for access by integer index
    private static readonly ValueCache<String, ErrorValue> errorTable
      = new ValueCache<string, ErrorValue>((index, message) => new ErrorValue(message, inde
x));

    public static readonly ErrorValue
      // The numError is first so it gets indedunx zero; necessary because
      // System.Math functions produce NaN with error code zero:
      numError =          Make("#NUM!"),
      argCountError =     Make("#ERR: ArgCount"),
      argTypeError =      Make("#ERR: ArgType"),
      nameError =         Make("#NAME?"),
      refError =          Make("#REF!"),
      valueError =        Make("#VALUE!"),
```



```
      naError =           Make("#NA"),
      tooManyArgsError = Make("#ERR: Too many arguments");

    private ErrorValue(String message, int errorIndex) {
      this.message = message;
      this.index = errorIndex;
    }

    public static int GetIndex(String message) {
      return errorTable.GetIndex(message);
    }

    public static ErrorValue Make(String message) {
      return errorTable[errorTable.GetIndex(message)];
    }

    public double ErrorNan {
      get { return MakeNan(index); }
    }

    // These two are also called from compiled code, through reflection:

    public static ErrorValue FromNan(double d) {
      return errorTable[ErrorCode(d)];
    }

    public static ErrorValue FromIndex(int errorIndex) {
      return errorTable[errorIndex];
    }

    public override bool Equals(Value v) {
      return v is ErrorValue && (v as ErrorValue).index == index;
    }

    public override int GetHashCode() {
      return index;
    }

    public override Object ToObject() {
      return (Object)this;
    }

    public override String ToString() {
      return message;
    }

    // From error code index (int) to NaN (double) and back

    public static double MakeNan(int errorIndex) {
      long nanbits = System.BitConverter.DoubleToInt64Bits(Double.NaN);
      return System.BitConverter.Int64BitsToDouble(nanbits | (uint)errorIndex);
    }

    public static int ErrorCode(double d) {
      return (int)System.BitConverter.DoubleToInt64Bits(d);
    }
  }

  /// <summary>
  /// An ArrayValue holds an array (rectangle) of Values resulting from evaluation.
  /// </summary>
  public abstract class ArrayValue : Value {
    public new static readonly Type type = typeof(ArrayValue);

    public abstract int Cols { get; }

    public abstract int Rows { get; }

    // Get cell value from array value
    public virtual Value this[CellAddr ca] {
      get { return this[ca.col, ca.row]; }
```

```
    }

    public abstract Value this[int col, int row] { get; }

    public override Object ToObject() {
      return (Object)this;
    }

    // Used to implement INDEX(area,row,col), truncated 1-based indexing
    public Value Index(double deltaRow, double deltaCol) {
      int col = (int)deltaCol - 1, row = (int)deltaRow - 1;
      if (0 <= col && col < Cols && 0 <= row && row < Rows)
        return this[col, row] ?? NumberValue.ZERO;
      else
        return ErrorValue.refError;
    }

    public abstract Value View(CellAddr ulCa, CellAddr lrCa);

    public abstract Value Slice(CellAddr ulCa, CellAddr lrCa);

    // Used to implement SLICE(arr, r1, c1, r2, c2)
    // A slice may be empty (when r1=r2+1 or c1=c2+1) whereas a View usually is not
    public Value Slice(double r1, double c1, double r2, double c2) {
      int ir1 = (int)r1 - 1, ic1 = (int)c1 - 1, ir2 = (int)r2 - 1, ic2 = (int)c2 - 1;
      if (0 <= ir1 && ir1 <= ir2 + 1 && ir2 < Rows && 0 <= ic1 && ic1 <= ic2 + 1 && ic2 < C
ols)
        return Slice(new CellAddr(ic1, ir1), new CellAddr(ic2, ir2));
      else
        return ErrorValue.refError;
    }

    // These are called from interpreted EXTERN and from generated bytecode
    public static double[] ToDoubleArray1D(Value v) {
      ArrayValue arr = v as ArrayValue;
      if (arr != null && arr.Rows == 1) {
        double[] res = new double[arr.Cols];
        for (int c = 0; c < arr.Cols; c++)
            if (arr[c, 0] is NumberValue)
              res[c] = (arr[c, 0] as NumberValue).value;
            else
              return null;
        return res;
      } else
        return null;
    }

    public static double[,] ToDoubleArray2D(Value v) {
      ArrayValue arr = v as ArrayValue;
      if (arr != null)
        return arr.ToDoubleArray2DFast();
      else
        return null;
    }

    public static String[] ToStringArray1D(Value v) {
      ArrayValue arr = v as ArrayValue;
      if (arr != null && arr.Rows == 1) {
        String[] res = new String[arr.Cols];
        for (int c = 0; c < arr.Cols; c++)
            if (arr[c, 0] is TextValue)
              res[c] = (arr[c, 0] as TextValue).value;
            else
              return null;
        return res;
      } else
        return null;
    }

    public static Value FromStringArray1D(Object o) {
      String[] ss = o as String[];
```



```
      if (ss != null) {
        Value[,] vs = new Value[ss.Length, 1];
        for (int c = 0; c < ss.Length; c++)
          vs[c, 0] = TextValue.FromString(ss[c]);
        return new ArrayExplicit(vs);
      } else
        return ErrorValue.argTypeError;
    }

    public static Value FromDoubleArray1D(Object o) {
      double[] xs = o as double[];
      if (xs != null) {
        Value[,] vs = new Value[xs.Length, 1];
        for (int c = 0; c < xs.Length; c++)
          vs[c, 0] = NumberValue.Make(xs[c]);
        return new ArrayExplicit(vs);
      } else
        return ErrorValue.argTypeError;
    }

    public static Value FromDoubleArray2D(Object o) {
      double[,] xs = o as double[,];
      if (xs != null)
        return new ArrayDouble(xs);
      else
        return ErrorValue.argTypeError;
    }

    // Override in ArrayDouble for efficiency
    public virtual double[,] ToDoubleArray2DFast() {
      double[,] res = new double[Rows, Cols];
      for (int c = 0; c < Cols; c++)
        for (int r = 0; r < Rows; r++)
          if (this[c, r] is NumberValue)
            res[r, c] = (this[c, r] as NumberValue).value;
          else
            return null;
      return res;
    }

    public void Apply(Action<double> act) {
      for (int c = 0; c < Cols; c++) {
        for (int r = 0; r < Rows; r++) {
          Value v = this[c, r];
          if (v != null) // Only non-blank cells contribute
            if (v is ArrayValue)
              (v as ArrayValue).Apply(act);
            else
              act(Value.ToDoubleOrNan(v));
        }
      }
    }

    public override void Apply(Action<Value> act) {
      for (int c = 0; c < Cols; c++) {
        for (int r = 0; r < Rows; r++) {
          Value v = this[c, r];
          if (v != null) // Only non-blank cells contribute
            if (v is ArrayValue)
              (v as ArrayValue).Apply(act);
            else
              act(v);
        }
      }
    }

    public static bool EqualElements(ArrayValue arr1, ArrayValue arr2) {
      if (arr1 == arr2)
        return true;
      if (arr1 == null || arr2 == null)
        return false;
```

```
      if (arr1.Rows != arr2.Rows || arr1.Cols != arr2.Cols)
        return false;
      for (int c = 0; c < arr1.Cols; c++)
        for (int r = 0; r < arr1.Rows; r++) {
          Value v1 = arr1[c, r], v2 = arr2[c, r];
          if (v1 != v2)
            if (v1 == null || v2 == null)
              return false;
            else if (!(v1.Equals(v2)))
              return false;
        }
      return true;
    }

    public override int GetHashCode() {
      int result = Rows * 37 + Cols;
      for (int i = 0; i < Rows && i < Cols; i++)
        result = result * 37 + this[i, i].GetHashCode();
      return result;
    }

    public override String ToString() {
      StringBuilder sb = new StringBuilder();
      for (int r = 0; r < Rows; r++) {
        for (int c = 0; c < Cols; c++) {
          Value v = this[c, r];
          sb.Append(v == null ? "[none]" : v.ToString());
          if (c < Cols - 1)
            sb.Append("\t");
        }
        if (r < Rows - 1)
          sb.Append("\n");
      }
      return sb.ToString();
    }
  }

  /// <summary>
  /// An ArrayView is a rectangular view of a sheet, the
  /// result of evaluating a CellArea expression.  Accessing an
  /// element of an ArrayView may cause a cell to be evaluated.
  /// </summary>
  public class ArrayView : ArrayValue, IEquatable<ArrayView> {
    public readonly CellAddr ulCa, lrCa;  // ulCa to the left and above lrCa
    public readonly Sheet sheet;          // non-null
    private readonly int cols, rows;

    private ArrayView(CellAddr ulCa, CellAddr lrCa, Sheet sheet) {
      this.sheet = sheet;
      this.ulCa = ulCa;
      this.lrCa = lrCa;
      this.cols = lrCa.col - ulCa.col + 1;
      this.rows = lrCa.row - ulCa.row + 1;
    }

    public static ArrayView Make(CellAddr ulCa, CellAddr lrCa, Sheet sheet) {
      CellAddr.NormalizeArea(ulCa, lrCa, out ulCa, out lrCa);
      return new ArrayView(ulCa, lrCa, sheet);
    }

    public override int Cols { get { return cols; } }

    public override int Rows { get { return rows; } }

    // Evaluate and get value at offset [col, row], 0-based
    public override Value this[int col, int row] {
      get {
        if (0 <= col && col < Cols && 0 <= row && row < Rows) {
          int c = ulCa.col + col, r = ulCa.row + row;
          Cell cell = sheet[c, r];
          if (cell != null)
```



```
            return cell.Eval(sheet, c, r);
          else
            return null;
        } else
          return ErrorValue.naError;
      }
    }

    public override Value View(CellAddr ulCa, CellAddr lrCa) {
      return ArrayView.Make(ulCa.Offset(this.ulCa), lrCa.Offset(this.ulCa), sheet);
    }

    public override Value Slice(CellAddr ulCa, CellAddr lrCa) {
      return new ArrayView(ulCa.Offset(this.ulCa), lrCa.Offset(this.ulCa), sheet);
    }

    public void Apply(Action<FullCellAddr> act) {
      int col0 = ulCa.col, row0 = ulCa.row;
      for (int c = 0; c < cols; c++)
        for (int r = 0; r < rows; r++)
          act(new FullCellAddr(sheet, col0 + c, row0 + r));
    }

    public override bool Equals(Value v) {
      return v is ArrayView && Equals(v as ArrayView)
             || EqualElements(this, v as ArrayValue);
    }

    // Used at codegen time for numbering and caching CGNormalCellArea expressions
    // in sheet-defined functions.  NB: Must not compare element values.
    public bool Equals(ArrayView other) {
      return sheet == other.sheet && ulCa.Equals(other.ulCa) && lrCa.Equals(other.lrCa);
    }

    public override bool Equals(Object o) {
      return o is ArrayView && Equals((ArrayView)o);
    }

    public override int GetHashCode() {
      return (ulCa.GetHashCode() * 29 + lrCa.GetHashCode()) * 37 + sheet.GetHashCode();
    }
  }

  /// <summary>
  /// An ArrayExplicit is a view of a materialized array of Values, typically
  /// resulting from evaluating TRANSPOSE or other array-valued functions.
  /// </summary>
  public class ArrayExplicit : ArrayValue {
    public readonly CellAddr ulCa, lrCa;      // ulCa to the left and above lrCa
    public readonly Value[,] values;          // non-null
    private readonly int cols, rows;

    public ArrayExplicit(Value[,] values)
      : this(new CellAddr(0, 0), new CellAddr(values.GetLength(0) - 1, values.GetLength(1)
- 1), values) { }

    public ArrayExplicit(CellAddr ulCa, CellAddr lrCa, Value[,] values) {
      this.ulCa = ulCa;
      this.lrCa = lrCa;
      this.values = values;
      this.cols = lrCa.col - ulCa.col + 1;
      this.rows = lrCa.row - ulCa.row + 1;
    }

    public override Value View(CellAddr ulCa, CellAddr lrCa) {
      return new ArrayExplicit(ulCa.Offset(this.ulCa), lrCa.Offset(this.ulCa), values);
    }

    public override Value Slice(CellAddr ulCa, CellAddr lrCa) {
      return View(ulCa, lrCa);
    }
```

```
    public override int Cols { get { return cols; } }

    public override int Rows { get { return rows; } }

    // Evaluate and get value at offset [col, row], 0-based
    public override Value this[int col, int row] {
      get {
        if (0 <= col && col < Cols && 0 <= row && row < Rows) {
          int c = ulCa.col + col, r = ulCa.row + row;
          return values[c, r];
        } else
          return ErrorValue.naError;
      }
    }

    public override bool Equals(Value v) {
      return EqualElements(this, v as ArrayValue);
    }
  }

  /// <summary>
  /// An ArrayMatrix is a rectangular array of non-wrapped doubles,
  /// typically the result of evaluating a linear algebra operation,
  /// or of unboxing all elements of an ArrayValue.
  /// </summary>
  public class ArrayDouble : ArrayValue {
    // Note: rows and columns are swapped relative to ArrayExplicit,
    // so indexing should go matrix[row, col] -- as in .NET 2D arrays
    public readonly double[,] matrix;

    public new static readonly Type type = typeof(ArrayDouble);
    public static readonly MethodInfo
      makeMethod = type.GetMethod("Make");

    public ArrayDouble(int cols, int rows) {
      this.matrix = new double[rows, cols];
    }

    public ArrayDouble(double[,] matrix) {
      this.matrix = matrix;
    }

    public override int Cols {
      get { return matrix.GetLength(1); }
    }

    public override int Rows {
      get { return matrix.GetLength(0); }
    }

    public override Value this[int col, int row] {
      get { return NumberValue.Make(matrix[row, col]); }
    }

    public override Value View(CellAddr ulCa, CellAddr lrCa) {
      int cols = Cols, rows = Rows, col0 = ulCa.col, row0 = ulCa.row;
      Value[,] vals = new Value[cols, rows];
      for (int c = 0; c < cols; c++)
        for (int r = 0; r < rows; r++)
          vals[c, r] = NumberValue.Make(matrix[row0 + r, col0 + c]);
      return new ArrayExplicit(vals);
    }

    public override Value Slice(CellAddr ulCa, CellAddr lrCa) {
      return View(ulCa, lrCa);
    }

    // Fast, but external array modification could undermine semantics
    public override double[,] ToDoubleArray2DFast() {
      return matrix;
```



```
    }

    public static Value Make(Value v) {
      if (v is ArrayDouble)
        return v;
      else if (v is ArrayValue) {
        ArrayValue arr = v as ArrayValue;
        int cols = arr.Cols, rows = arr.Rows;
        ArrayDouble result = new ArrayDouble(cols, rows);
        for (int r = 0; r < rows; r++)
          for (int c = 0; c < cols; c++)
            result.matrix[r, c] = Value.ToDoubleOrNan(arr[c, r]);
        return result;
      } else
        return ErrorValue.argTypeError;
    }

    public override bool Equals(Value v) {
      return EqualElements(this, v as ArrayValue);
    }
  }

  /// <summary>
  /// A FunctionValue is a partially applied sheet-defined function, ie. a closure.
  /// </summary>
  public class FunctionValue : Value {
    // Depends on Funcalc.SdfInfo and so probably should be elsewhere.
    public readonly SdfInfo sdfInfo;
    public readonly Value[] args;  // Invariant: args.Length==sdfInfo.Arity
    private readonly int arity;    // Invariant: arity==number of #NA in args
    private readonly Delegate mergeAndCall;

    // Used by code generation for CGApply and CGClosure
    public new static readonly Type type = typeof(FunctionValue);
    public static readonly MethodInfo
      makeMethod = type.GetMethod("Make"),
      furtherApplyMethod = type.GetMethod("FurtherApply");
    private static readonly FieldInfo
      sdfInfoField = type.GetField("sdfInfo"),
      argsField = type.GetField("args");
    private static readonly Type[] mergeAndCallDelegateType
      = { typeof(Func<FunctionValue,Value>),
          typeof(Func<FunctionValue,Value,Value>),
          typeof(Func<FunctionValue,Value,Value,Value>),
          typeof(Func<FunctionValue,Value,Value,Value,Value>),
          typeof(Func<FunctionValue,Value,Value,Value,Value,Value>),
          typeof(Func<FunctionValue,Value,Value,Value,Value,Value,Value>),
          typeof(Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value>),
          typeof(Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value,Value>),
          typeof(Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value,Value,Value>)
,
          typeof(Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value,Value,Value,V
alue>),
          typeof(Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value,Value,Value,V
alue,Value>) };
    private static readonly IDictionary<int, Delegate> mergeDelegateCache
      = new Dictionary<int, Delegate>();
    private static readonly Type[][] mergerArgTypes;

    public static readonly MethodInfo[] callMethods
      = { type.GetMethod("Call0"),
          type.GetMethod("Call1"),
          type.GetMethod("Call2"),
          type.GetMethod("Call3"),
          type.GetMethod("Call4"),
          type.GetMethod("Call5"),
          type.GetMethod("Call6"),
          type.GetMethod("Call7"),
          type.GetMethod("Call8"),
          type.GetMethod("Call9") };
```

```
  static FunctionValue() {
    mergerArgTypes = new Type[mergeAndCallDelegateType.Length][];
    for (int a=0; a<mergeAndCallDelegateType.Length; a++) {
      Type[] argTypes = mergerArgTypes[a] = new Type[a+1];
      argTypes[0] = FunctionValue.type;
      for (int i = 0; i < a; i++)
        argTypes[i + 1] = Value.type;
    }
  }

  public FunctionValue(SdfInfo sdfInfo, Value[] args) {
    this.sdfInfo = sdfInfo;
    // A null or empty args array is equivalent to an array of all NA
    if (args == null || args.Length == 0) {
      args = new Value[sdfInfo.arity];
      for (int i = 0; i < args.Length; i++)
        args[i] = ErrorValue.naError;
    }
    // Requirement: There will be no further writes to the args array
    this.args = args;
    int k = 0;
    for (int i = 0; i < args.Length; i++)
      if (args[i] == ErrorValue.naError)
        k++;
    this.arity = k;
    this.mergeAndCall = MakeMergeAndCallMethod();
  }

  public override bool Equals(Object obj) {
    return Equals(obj as FunctionValue);
  }

  public override bool Equals(Value v) {
    return Equals(v as FunctionValue);
  }

  public bool Equals(FunctionValue that) {
    if (that == null
        || this.sdfInfo.index != that.sdfInfo.index
        || this.args.Length != that.args.Length)
      return false;
    for (int i = 0; i < args.Length; i++)
      if (!this.args[i].Equals(that.args[i]))
        return false;
    return true;
  }

  public override int GetHashCode() {
    int result = sdfInfo.index * 37 + args.Length;
    for (int i = 0; i < args.Length; i++)
      result = result * 37 + args[i].GetHashCode();
    return result;
  }

  public override Object ToObject() {
    return (Object)this;
  }

  // This array statically allocated and reusedto avoid allocation
  private static readonly Value[] allArgs = new Value[10];

  public Value Apply(Value[] vs) {
    if (arity != vs.Length)
      return ErrorValue.argCountError;
    else {
      MergeArgs(vs, allArgs);
      return sdfInfo.Apply(allArgs);
    }
  }

  // Replace #NA from args with late arguments in output.
```



```
  // There is a special version of this loop in CGSdfCall.PEval too
  private void MergeArgs(Value[] late, Value[] output) {
    int j = 0;
    for (int i = 0; i < args.Length; i++)
      if (args[i] != ErrorValue.naError)
        output[i] = args[i];
      else
        output[i] = late[j++];
  }

  // These methods are called both directly and by generated code/reflection:

  public static FunctionValue Make(int sdfIndex, Value[] vs) {
    return new FunctionValue(SdfManager.GetInfo(sdfIndex), vs);
  }

  public Value FurtherApply(Value[] vs) {
    if (vs.Length == 0)
      return this;
    else if (vs.Length != arity)
      return ErrorValue.argCountError;
    else {
      Value[] newArgs = new Value[args.Length];
      MergeArgs(vs, newArgs);
      return new FunctionValue(sdfInfo, newArgs);
    }
  }

  public Value Call0() {
    return sdfInfo.Apply(args); // Shortcut when no late arguments
  }

  public Value Call1(Value v1) {
    if (arity != 1)
      return ErrorValue.argCountError;
    else {
      var caller = (Func<FunctionValue,Value,Value>)mergeAndCall;
      return caller(this, v1);
    }
  }

  public Value Call2(Value v1, Value v2) {
    if (arity != 2)
      return ErrorValue.argCountError;
    else {
      var caller = (Func<FunctionValue,Value,Value,Value>)mergeAndCall;
      return caller(this, v1, v2);
    }
  }

  public Value Call3(Value v1, Value v2, Value v3) {
    if (arity != 3)
      return ErrorValue.argCountError;
    else {
      var caller = (Func<FunctionValue,Value,Value,Value,Value>)mergeAndCall;
      return caller(this, v1, v2, v3);
    }
  }

  public Value Call4(Value v1, Value v2, Value v3, Value v4) {
    if (arity != 4)
      return ErrorValue.argCountError;
    else {
      var caller = (Func<FunctionValue,Value,Value,Value,Value,Value>)mergeAndCall;
      return caller(this, v1, v2, v3, v4);
    }
  }

  public Value Call5(Value v1, Value v2, Value v3, Value v4, Value v5) {
    if (arity != 5)
      return ErrorValue.argCountError;
```

```
      else {
        var caller = (Func<FunctionValue,Value,Value,Value,Value,Value,Value>)mergeAndCall;
        return caller(this, v1, v2, v3, v4, v5);
      }
    }

    public Value Call6(Value v1, Value v2, Value v3, Value v4, Value v5, Value v6) {
      if (arity != 6)
        return ErrorValue.argCountError;
      else {
        var caller = (Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value>)mergeAn
dCall;
        return caller(this, v1, v2, v3, v4, v5, v6);
      }
    }

    public Value Call7(Value v1, Value v2, Value v3, Value v4, Value v5, Value v6, Value v7
) {
      if (arity != 7)
        return ErrorValue.argCountError;
      else {
        var caller = (Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value,Value>)m
ergeAndCall;
        return caller(this, v1, v2, v3, v4, v5, v6, v7);
      }
    }

    public Value Call8(Value v1, Value v2, Value v3, Value v4, Value v5, Value v6, Value v7
, Value v8) {
      if (arity != 8)
        return ErrorValue.argCountError;
      else {
        var caller = (Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value,Value,Va
lue>)mergeAndCall;
        return caller(this, v1, v2, v3, v4, v5, v6, v7, v8);
      }
    }

    public Value Call9(Value v1, Value v2, Value v3, Value v4, Value v5, Value v6, Value v7
, Value v8, Value v9) {
      if (arity != 9)
        return ErrorValue.argCountError;
      else {
        var caller = (Func<FunctionValue,Value,Value,Value,Value,Value,Value,Value,Value,Va
lue,Value>)mergeAndCall;
        return caller(this, v1, v2, v3, v4, v5, v6, v7, v8, v9);
      }
    }

    public int Arity { get { return arity; } }

    // Generate a method EntryN(fv,v1,...,vN) to merge early arguments from
    // fv.args with late arguments v1...vN and call fv.sdfInfo delegate.
    // Assumptions: The call-time fv.args array has the same length as the
    // generation-time args array, and call-time fv.arity equals generation-time arity.

    private Delegate MakeMergeAndCallMethod() {
      Delegate result;
      int pattern = NaPattern(args);
      if (!mergeDelegateCache.TryGetValue(pattern, out result)) {
        // Console.WriteLine("Created function value merger pattern {0}", pattern);
        DynamicMethod method = new DynamicMethod("EntryN", Value.type, mergerArgTypes[Arity]
, true);
        ILGenerator ilg = method.GetILGenerator();
        // Load and cast the SDF delegate to call
        ilg.Emit(OpCodes.Ldsfld, SdfManager.sdfDelegatesField);  // sdfDelegates
        ilg.Emit(OpCodes.Ldarg_0);                               // sdfDelegates, fv
        ilg.Emit(OpCodes.Ldfld, sdfInfoField);                   // sdfDelegates, fv.sdfInf
o
        ilg.Emit(OpCodes.Ldfld, SdfInfo.indexField);             // sdfDelegates, fv.sdfInf
o.index
```



```
        ilg.Emit(OpCodes.Ldelem_Ref);                            // sdfDelegates[fv.sdfInfo
.index]
        ilg.Emit(OpCodes.Castclass, sdfInfo.MyType);             // sdf delegate of appropr
iate type
        // Save the early argument array to a local variable
        LocalBuilder argsArray = ilg.DeclareLocal(typeof(Value[]));
        ilg.Emit(OpCodes.Ldarg_0);                               // sdf, fv
        ilg.Emit(OpCodes.Ldfld, argsField);                      // sdf, fv.args
        ilg.Emit(OpCodes.Stloc, argsArray);                      // sdf
        // Do like MergeArgs(new Value[] { v1...vn }, ...) but without array overhead:
        int j = 1; // The first late argument is arg1
        for (int i = 0; i < args.Length; i++) {
          if (args[i] != ErrorValue.naError) { // Load early arguments from fv.args
            ilg.Emit(OpCodes.Ldloc, argsArray);
            ilg.Emit(OpCodes.Ldc_I4, i);
            ilg.Emit(OpCodes.Ldelem_Ref);
          } else // Load late arguments from the EntryN method's parameters
            ilg.Emit(OpCodes.Ldarg, j++);
        }
        ilg.Emit(OpCodes.Call, sdfInfo.MyInvoke);
        ilg.Emit(OpCodes.Ret);
        result = method.CreateDelegate(mergeAndCallDelegateType[arity]);
        mergeDelegateCache[pattern] = result;
      }
      return result;
    }

    // Convert a #NA-pattern to an index for caching purposes; think
    // of the #NA-pattern as a dyadic number with #NA=1 and non-#NA=2
    private static int NaPattern(Value[] args) {
      int index = 0;
      for (int i = 0; i < args.Length; i++)
        index = 2 * index + (args[i] == ErrorValue.naError ? 1 : 2);
      return index;
    }

    public static String FormatAsCall(String name, params Object[] args) {
      StringBuilder sb = new StringBuilder();
      sb.Append(name).Append("(");
      if (args.Length > 0)
        sb.Append(args[0]);
      for (int i = 1; i < args.Length; i++)
        sb.Append(",").Append(args[i]);
      return sb.Append(")").ToString();
    }

    public override String ToString() { // Don't show args if all are #NA
      return Arity < args.Length ? FormatAsCall(sdfInfo.name, args) : sdfInfo.name;
    }
  }
}
```

```
// Funcalc, a spreadsheet core implementation
// ------------------------------------------------------------------------
// Copyright (c) 2006-2014 Peter Sestoft and others

// Permission is hereby granted, free of charge, to any person
// obtaining a copy of this software and associated documentation
// files (the "Software"), to deal in the Software without
// restriction, including without limitation the rights to use, copy,
// modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:

//  * The above copyright notice and this permission notice shall be
//    included in all copies or substantial portions of the Software.

//  * The software is provided "as is", without warranty of any kind,
//    express or implied, including but not limited to the warranties of
//    merchantability, fitness for a particular purpose and
//    noninfringement.  In no event shall the authors or copyright
//    holders be liable for any claim, damages or other liability,
//    whether in an action of contract, tort or otherwise, arising from,
//    out of or in connection with the software or the use or other
//    dealings in the software.
// ------------------------------------------------------------------------

using System;
using System.Diagnostics;
using SC = System.Collections;
using System.Collections.Generic;
using Corecalc.Funcalc;

namespace Corecalc {
  /// <summary>
  /// A Workbook is a collection of distinct named Sheets.
  /// </summary>
  public sealed class Workbook : IEnumerable<Sheet> {
    public event Action<String[]> OnFunctionsAltered;

    private readonly List<Sheet> sheets           // All non-null and distinct
      = new List<Sheet>();
    public readonly Formats format                // Formula formatting options
      = new Formats();

    // For managing recalculation of the workbook
    public CyclicException Cyclic { get; private set; }  // Non-null if workbook has cycle
    public uint RecalcCount { get; private set; }        // Number of recalculations done
    public bool UseSupportSets { get; private set; }
    private readonly List<FullCellAddr> editedCells
      = new List<FullCellAddr>();
    private readonly HashSet<FullCellAddr> volatileCells
      = new HashSet<FullCellAddr>();
    private readonly Queue<FullCellAddr> awaitsEvaluation
      = new Queue<FullCellAddr>();

    public Workbook() {
      RecalcCount = 0;
      UseSupportSets = false;
      SdfManager.ResetTables();
    }

    public void AddSheet(Sheet sheet) {
      sheets.Add(sheet);
    }

    public void RecordCellChange(int col, int row, Sheet sheet) {
      editedCells.Add(new FullCellAddr(sheet, col, row));
    }

    public Sheet this[String name] {
      get {
        name = name.ToUpper();
```



```
        foreach (Sheet sheet in sheets)
          if (sheet.Name.ToUpper() == name)
            return sheet;
        return null;
      }
    }

    public Sheet this[int i] {
      get { return sheets[i]; }
    }

    // Recalculate from recalculation roots only, using their supported sets
    public long Recalculate() {
      // Now Cyclic != null or for all formulas f, f.state==Uptodate
      if (Cyclic != null || CheckForModifiedSdf())
        return RecalculateFullAfterSdfCheck();
      else
        return TimeRecalculation(delegate {
          UseSupportSets = true;
          // Requires for all formulas f, f.state==Uptodate
          // Stage (1): Mark formulas reachable from roots, f.state=Dirty
          SupportArea.IdempotentForeach = true;
          foreach (FullCellAddr fca in volatileCells)
            Cell.MarkCellDirty(fca.sheet, fca.ca.col, fca.ca.row);
          foreach (FullCellAddr fca in editedCells)
            Cell.MarkCellDirty(fca.sheet, fca.ca.col, fca.ca.row);
          // Stage (2): Evaluate Dirty formulas (and Dirty cells they depend on)
          awaitsEvaluation.Clear();
          SupportArea.IdempotentForeach = true;
          foreach (FullCellAddr fca in editedCells)
            Cell.EnqueueCellForEvaluation(fca.sheet, fca.ca.col, fca.ca.row);
          foreach (FullCellAddr fca in volatileCells)
            Cell.EnqueueCellForEvaluation(fca.sheet, fca.ca.col, fca.ca.row);
          while (awaitsEvaluation.Count > 0)
            awaitsEvaluation.Dequeue().Eval();
        });
    }

    public void AddToQueue(Sheet sheet, int col, int row) {
      awaitsEvaluation.Enqueue(new FullCellAddr(sheet, col, row));
    }

    public long RecalculateFull() {
      CheckForModifiedSdf();
      return RecalculateFullAfterSdfCheck();
    }

    // Unconditionally recalculate all cells a la Ctrl+Alt+F9
    public long RecalculateFullAfterSdfCheck() {
      return TimeRecalculation(delegate {
        UseSupportSets = false;
        ResetCellState();
        // For all formulas f, f.state==Dirty
        foreach (Sheet sheet in sheets)
          sheet.RecalculateFull();
      });
    }

    public long RecalculateFullRebuild() {
      return TimeRecalculation(delegate {
        UseSupportSets = false;
        RebuildSupportGraph();  // Leaves all cells Dirty
        foreach (Sheet sheet in sheets)
          sheet.RecalculateFull();
      });
    }

    // Timing, and handling of cyclic dependencies
    private long TimeRecalculation(Action act) {
      Cyclic = null;
      RecalcCount++;
```

```
    Stopwatch sw = new Stopwatch();
    sw.Start();
    try {
      act();
    } catch (Exception exn) {
      ResetCellState();       // Mark all cells Dirty
      if (exn is CyclicException)
        Cyclic = exn as CyclicException;
      else
        Console.WriteLine("BAD: {0}", exn);
    }
    sw.Stop();
    editedCells.Clear();
    return sw.ElapsedMilliseconds;
  }

  private bool CheckForModifiedSdf() {
    if (RecalcCount != 0) {
      String[] modifiedFunctions = SdfManager.CheckForModifications(editedCells);
      if (modifiedFunctions.Length != 0 && OnFunctionsAltered != null) {
        OnFunctionsAltered(modifiedFunctions);
        return true;
      }
    }
    return false;
  }

  private void ResetCellState() {
    foreach (Sheet sheet in sheets)
      sheet.ResetCellState();
  }

  public void RebuildSupportGraph() {
    Console.WriteLine("Rebuilding support graph");
    foreach (Sheet sheet in this)
      foreach (Cell cell in sheet)
        cell.ResetSupportSet();
    ResetCellState(); // Mark all cells Dirty ie. not Visited
    foreach (Sheet sheet in this)
      sheet.AddToSupportSets();
    // Leaves all cells Dirty
  }

  public void ResetVolatileSet() {
    volatileCells.Clear();
    foreach (Sheet sheet in this)
      sheet.IncreaseVolatileSet();
  }

  public void IncreaseVolatileSet(Cell cell, Sheet sheet, int col, int row) {
    if (cell != null && cell.IsVolatile)
      volatileCells.Add(new FullCellAddr(sheet, col, row));
  }

  public void DecreaseVolatileSet(Cell cell, Sheet sheet, int col, int row) {
    Formula f = cell as Formula;
    if (f != null)
      volatileCells.Remove(new FullCellAddr(sheet, col, row));
  }

  public int SheetCount {
    get { return sheets.Count; }
  }

  IEnumerator<Sheet> IEnumerable<Sheet>.GetEnumerator() {
    foreach (Sheet sheet in sheets)
      yield return sheet;
  }

  SC.IEnumerator SC.IEnumerable.GetEnumerator() {
    foreach (Sheet sheet in sheets)
```



```
      yield return sheet;
  }

  public void Clear() {
    sheets.Clear();
    editedCells.Clear();
    volatileCells.Clear();
    awaitsEvaluation.Clear();
  }
 }
}
```