User-defined functions in spreadsheets

Daniel S. Cortes

Morten W. Hansen

Supervisor: Peter Sestoft

Master's thesis in Software Development
IT University of Copenhagen, September 2006

Foreword

This Master's thesis is written by Daniel S. Cortes and Morten W. Hansen in the period from February to August 2006. With this thesis we finish our Master degree in software development on the IT University of Copenhagen.

Both of us have a Master degree in actuarial science from Copenhagen University and have been employed in the Danish pension and insurance business for several years. With this background we wanted to combine our knowledge from 'the real world' with our knowledge on software development. And what would be more obvious than to focus on the most used software application in the financial sector - the spreadsheet application.

We wish to thank our supervisor Peter Sestoft for giving us the idea for this thesis and for the very committed supervising.

1st September 2006

Daniel S. Cortes & Morten W. Hansen

Resume

I dette afsluttende projekt på Masteruddannelsen i softwareudvikling implementerer vi udvidelser til regneark, som integrerer brugerdefinerede funktioner i regnearkets gitter. Målet er at validere, om de foreslåede ideer i 'A User-Centred Approach to Functions in Excel' af Blackwell, Burnett and Peyton Jones [JBB03] kan implementeres, og i givet fald i hvilken udstrækning og hvem der kan drage fordel heraf. Herunder diskuteres artiklen af Blackwell, Burnett and Peyton Jones med fokus på en egentlig implementering. Denne diskussion bruges som vores teoretiske grundlag.

Vi implementerer de brugerdefinerede funktioner over fire prototyper ved hjælp af en evolutionær udviklingsmetode med iterative udvidelser. De fire prototype er:

1. Simpel funktion
2. Avanceret funktion
3. Matrix funktion
4. Rekursiv og højere-ordens funktion

I hver prototype præsenterer vi en case, hvor vi anvender regneark, der bruges i pensions- og forsikringsbranchen i Danmark. Disse fire cases udgør vores praktiske grundlag.

Efter fire succesfulde implementeringer af de brugerdefinerede funktioner konkluderer vi, at de foreslåede ideer - med visse modifikationer og ændringer - kan implementeres. Derudover har vi, ved at udvide funktionaliteten til at inkludere rekursive og højere-ordens funktioner, forøget den oprindelige målgruppe. Dette er lykkes uden væsentligt at påvirke brugerens oplevelse.

Med vores modifikationer, ændringer og udvidelser tror vi på, at mange slut-brugere vil få betydelig fordel af disse brugerdefinerede funktioner, og at potentialet for en sådan udvidelse af regneark er stort.

Summary

In this Master's thesis we implement extensions to spreadsheets that integrate user-defined functions into the spreadsheet grid. The purpose is to validate whether the proposed ideas described in 'A User-Centred Approach to Functions in Excel' by Blackwell, Burnett and Peyton Jones [JBB03] can be implemented or not, and to which extent and benefit it can be used in practice. In doing so we discuss the paper by Blackwell, Burnett and Peyton Jones with focus on implementation and technical design and use this as our theoretical basis.

Using an evolutionary development via an iterative enhancement method we implement the user-defined functions over four prototypes:

1. Simple function
2. Advanced function
3. Matrix function
4. Recursive and higher-order function

In each prototype we present a case where we use spreadsheets that are currently used in the pension and insurance business in Denmark. These four cases represent our practical basis.

After having successfully implemented the user-defined functions in all four prototypes we conclude that - with some modifications and changes - the proposed ideas presented by Blackwell, Burnett and Peyton Jones can be implemented. Furthermore by extending the functionality to include recursive and higher-order functions we broaden the initially proposed target audience without significantly damaging the usability of the solution.

With our modifications, changes and extensions we strongly believe that many end-users will benefit from the user-defined function extension and that the potential of this is significant.

Contents

1 Introduction

One of the most successful software applications ever invented is the spreadsheet. The number of spreadsheet users is several million worldwide and still increasing. The success is based on the fact that spreadsheets have high usability and are easy to learn but can still be used for a very broad range of tasks including very complex ones. But even though spreadsheets can be used for complex tasks it lacks the ability to define re-usable abstractions which is one of the most fundamental mechanisms for handling complex and elaborate models. The consequence of this is well-known, spreadsheets with thousands of cells and formulas.

As an attempt to solve this problem many spreadsheet applications have added the possibility for the user to access an attached programming language (e.g. Visual Basic for Applications (VBA) in Excel). Here the user can define re-usable abstractions such as functions and macros. But this has been without great success since only a fraction of the spreadsheet users are able to understand and use these languages.

1.1 Purpose and problem formulation

In the paper 'A User-Centred Approach to Functions in Excel' by Blackwell, Burnett, Peyton Jones (2003) [JBB03] a theoretical basis is described that proposes changes to the Excel spreadsheet application which integrates user-defined functions into the spreadsheet grid. The theory is not concretized into an actual implementation though. This paper will therefore try to answer the following question:

With the ideas described in [JBB03] as a theoretical basis and with a practical basis from used spreadsheets is it then possible to implement and benefit from the user-defined functions in spreadsheets?

We are both actuaries currently employed in, respectively, a life insurance company and a software company making solutions for insurance companies. We use this in answering the question and take real commercially used spreadsheets and try to redefine them using the user-defined functions.

We use an iterative approach where we through four prototypes will implement the user-defined functions. For every prototype we will discuss the design (technical and user-orientated), the usability and the solution compared to the theoretical basis.

1.2 Practical details and limitations

Since we naturally do not have access to Excel source code we have to make our own spreadsheet application and implement the user-defined functions there. Alternatively we could have used Gnumerics or OpenOffice Calc which are open source spreadsheets, but these are major software packages and often not very well documented when it comes to their inner workings. Understanding these packages would have been a major part of the project. Instead we extend the basic spreadsheet application developed by Peter Sestoft (see [Ses05], a forthcoming ITU tech report). The development environment is Microsoft Visual Studio C#.

In this paper we are focusing on answering the question above and the goal is therefore not to deliver an end-user ready spreadsheet application but to discuss the problems/issues concerning the implementation. We will therefore not implement many of the 'standard' features found in spreadsheet applications such as menus, formats, printing options, etc.

Likewise we will not implement many of the user-orientated features presented in [JBB03]. This includes:

- The 'function-entry wizard'.
- The automatic creation of user-defined functions via formulas.
- Pop-up box asking whether or not to modify an instance of a function or all instances of the function.

In general we only implement the parts of the GUI that is necessary to show and illustrate the functionality.

1.3 This paper

In the follow section we define what a spreadsheet is and the terms we use throughout the paper. In section 3 we outline and analyze the theory with focus on real implementation while Section 4 briefly explains the methodological approach. The next four sections contain the description of the four prototypes:

- Section 5 : Simple functions
- Section 6 : Advanced functions
- Section 7 : Matrix functions
- Section 8 : Higher-order functions

Last, but not least, we conclude by answering the question above in section 9.

1.4 Related work

Like us, others have worked on the core spreadsheet application set out by Peter Sestoft. Especially we would like to mention Thomas Iversen's work with runtime compilation of spreadsheet (see [Ive06]). An interesting study would be to combine his works with the works in this paper.

The authors of the paper 'A User-Centred Approach to Functions in Excel' have a patent pending for the user-defined spreadsheet functions (application number US2004103366).

2 About spreadsheets

Almost everyone knows more or less what a spreadsheet is but to get a common baseline of understanding and to define the concepts and notation used in this paper, we will shortly explain how we define a spreadsheet and the associated concepts.

2.1 What is a spreadsheet?

A spreadsheet is a type of computer program that displays a group of cells in a 2D graph pattern and allows for easy mathematical operations and relationships among the cells. Today's most popular spreadsheet is Microsoft Excel.

A spreadsheet consists of one or more worksheets (for simplicity sometimes called sheets in the rest of this work), each of which contains many rows and columns of cells. Each cell can contain data (a number or a string) or a formula. Cells can be grouped into ranges containing one or more cells, and ranges can be named and referred to by name. In that way cells can be compared to variables in a sequential programming model.

A formula in a cell can use the contents of other cells, external variables such as the current date and time or built-in functions. A spreadsheet can therefore be seen as a mathematical graph where the nodes are spreadsheet cells and the edges are references to other cells specified in formulas (dependency graph).

The cells in a spreadsheet will automatically update (recalculate) when the cells on which they depend have been changed.

Even though many users do not experience it that way, a spreadsheet can be seen as a functional programming language where the programmer writes simple functions to create the desired model. Considered as a functional programming language a spreadsheet is quite limited, e.g.:

- One cannot create new functions but only use the built-in ones.
- Recursive functions do not exist.
- Higher-order functions do not exist.

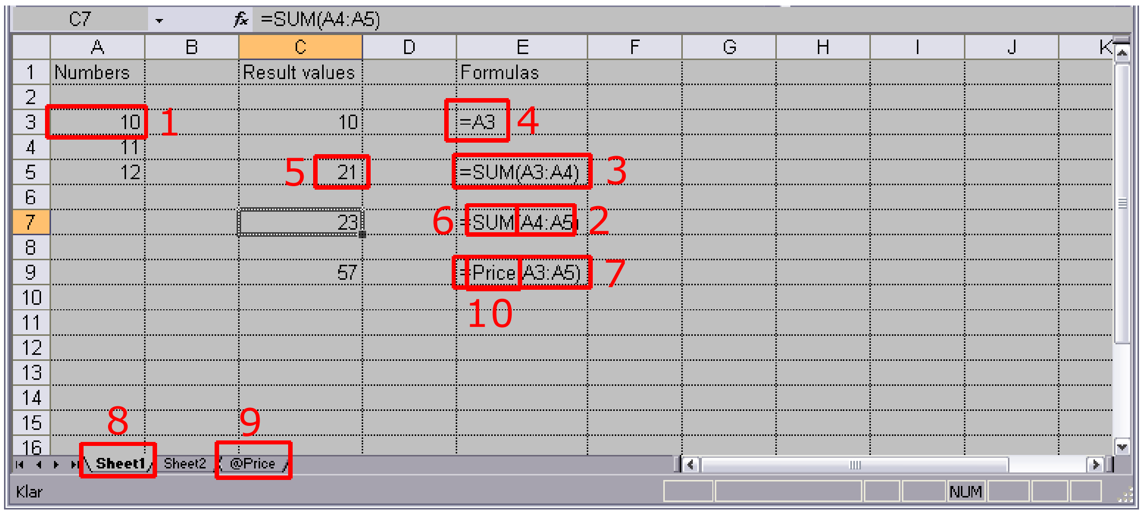
The above mentioned functionalities can usually be accomplished by using an underlying programming language (e.g. VBA in Excel), i.e. via the use of another programming language.

2.2 Definition of terms

This subsection shortly defines, in alphabetical order, the most used terms in this paper. A reference to an example in figure 1 is presented after some of the descriptions.

Arguments (to a function)	The input parameters to a function. This can be constants (text strings or floating-point numbers), formulas, matrix formulas or user-defined functions. (2)
Built-in function	A predefined function in the spreadsheet application that cannot be altered. (6)
Cell	A small box in a sheet, where the data (a constant (text string or floating-point number), a formula or a matrix formula) is stored. (1)
Cell reference	A reference to a specific cell in the same sheet or in another sheet in the spreadsheet. In Excel a cell reference to a cell in the same sheet is represented as the intersection of a column letter and a row number. (4)
Error message	A dialog box displayed to inform the user that the particular action is not allowed or can not be done.
Formula	A set of instructions which is used to compute the value of a cell. Can be e.g. a cell reference or function call. (3)
Function call	A initiation of a function by applying the required arguments. (7)
Function sheet	A sheet that contains a user-defined function. It works just as a normal worksheet but has the one purpose to hold a user-defined function. (9)
Function signature	The definition of which arguments the function takes and how the result is returned.
Matrix formula	A formula that contains matrix values.
Result (of a function)	The output of a function. This can be a constant (text string or floating-point number), a formula, a matrix formula or a user-defined function. (5)
Sheet	A sheet consists of a grid of cells and can either be a worksheet or a function sheet. (8)
Spreadsheet	A spreadsheet consists of one or more sheets.
User (of a spreadsheet)	An end-user of the spreadsheet application.

User-defined function	A function that is created by the user using the features presented in this paper. Notice that this term does <u>not</u> include user-defined functions that is created in a underlying programming language (e.g. VBA) - when we refer to these types of user-defined functions we explicitly state it. (10)
Worksheet	A normal sheet. (8)



C7 =SUM(A4:A5)

	A	B	C	D	E
1	Numbers		Result values		Formulas
2					
3	10		10		=A3
4	11				
5	12		21		=SUM(A3:A4)
6					
7			23		=SUM(A4:A5)
8					
9			57		=Price(A3:A5)

Figure 1: Example of a spreadsheet.

3 Theory and analysis of user-defined functions

In [JBB03] Blackwell, Burnett and Peyton Jones proposes a change to the Excel spreadsheet application which integrates user-defined functions into the spreadsheet grid. To design these functions they use a user-centred approach with nine design concepts from the HCI world as their basis.

In section 3.1 we will outline the theory and main ideas presented in [JBB03].

In section 3.2 we will analyze the theory with focus on real implementation. This requires:

- Clarification of the theory where needed.
- Discussion of the ideas.
- Review of not answered questions.

3.1 Theoretical basis

As explained above this section describes the key points and thoughts in [JBB03]. We will later (in section 3.2) take a critical view of these points on the basis of the real implementation.

One can divide the article into two parts, a part that describes why there is a need for user-defined functions in spreadsheets, and a part that describes the main design ideas of the user-defined functions.

3.1.1 The need for user-defined functions

To analyze the need for user-defined functions [JBB03] states the main advantages by having user-defined functions in a spreadsheet seen from the users point of view:

Avoid repetition: By using user-defined functions one can avoid repeating formulas every time a new spreadsheet needs the same formula. In other words one has the opportunity to identify, name and reuse code from a previous spreadsheet.

Reduce errors during maintenance: Excel encourages the use of 'copy-paste' of complex formulas. This might create problems later when the copied formulas must be changed since one has to not only change the copied formulas but also the places where the formulas were copied from.

With user-defined functions the formula is encapsulated and thereby the risk of these kinds of errors is reduced. In addition one only has to change one place to change all the formulas in the spreadsheet.

Real estate management: To call a user-defined function one only needs one cell even though the definition of the function in terms of space fills up several cells. This is particularly nice if the calculation requires many intermediate results. In this way we save a lot of space in the sheet that calls the user-defined function.

Encapsulate and re-use domain-specific expertise: User-defined functions support reuse. Users that are domain experts can create libraries of functions that are dedicated to special use or application that can be used by non-expert users.

Intellectual property protection: User-defined functions are easy to encapsulate and thereby protect your intellectual property rights. Hiding the function implementation from the function user is a standard way of intellection property protection used by other branches of software development. The encapsulation increases security on distribution of spreadsheets.

Performance: A function that are represented wisely can be compiled to byte code, JIT's (JIT = Just-In-Time compilation) to machine code or whatever, with performance benefits.

These general advantages of having user-defined functions are well known and it is not a brand new idea that users can define their own functions, e.g. in Excel users can use VBA (Visual Basic for Application). The problem here is that the ordinary user of Excel does not know VBA and the costs that are related to educating these users to a level where they can use VBA are very high. This is mainly because the difference between Excel and VBA is too big:

- The programming paradigm is different. VBA is a imperative language while Excel is a declarative language.
- The notation is different. In VBA the notation consists of blocks of text while Excel has a grid of cells.
- The programming environment is different. VBA uses Visual Studio while Excel uses a spreadsheet grid.
- The debugging model is different. VBA uses the Visual Studio debugger while Excel uses a manual cell-to-cell debugger.

Target audience

Given the above mentioned problems with the functionality that is available today the target audience for the user-defined functions is the large group of ordinary users that is familiar to spreadsheets and has a need for non-trivial formulas but can not use e.g. VBA. More specifically the assumption is that to be able to use the design of the user-defined functions one has to be able to:

1. Use several different built-in functions and not only infix operators.
2. Use 'copy-paste' or replicate operations so a formula systematically is changed to operate on the new location.
3. Use more than one worksheet in a spreadsheet.

In other words the target audience consists of users that need to handle more ambitious and complicated applications than possible today. More advanced users (e.g. users with good VBA knowledge) will also benefit from the user-defined functions.

The goal is therefore to give the users some tools for handling complex problems that previously required a programmer.

Concepts for measuring good design

For measuring the quality of the design of the user-defined functions [JBB03] uses some well known design concepts from the HCI (Human-Computer Interaction) world:

Abstraction gradient	What are the levels of abstraction? Can fragments of the design be encapsulated?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Error-proneness	Does the design of the notation induce careless mistakes?
Hidden dependencies	Is every dependency overtly indicated? And are the dependencies perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partly completed program be executed to obtain relevant feedback?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?
Viscosity	How much effort is required to perform a single change?
Visibility and juxtaposability	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Together with the above concepts and a simple cost-benefit analysis this gives some indicators for measuring the proposed design.

3.1.2 User-defined functions in Excel

With the stated main goal for the user-defined functions and with the definition of the target audience [JBB03] sets up the following ground rule:

The implementation of a function must be defined by a spreadsheet, because that is the only computational paradigm understood by our target audience.

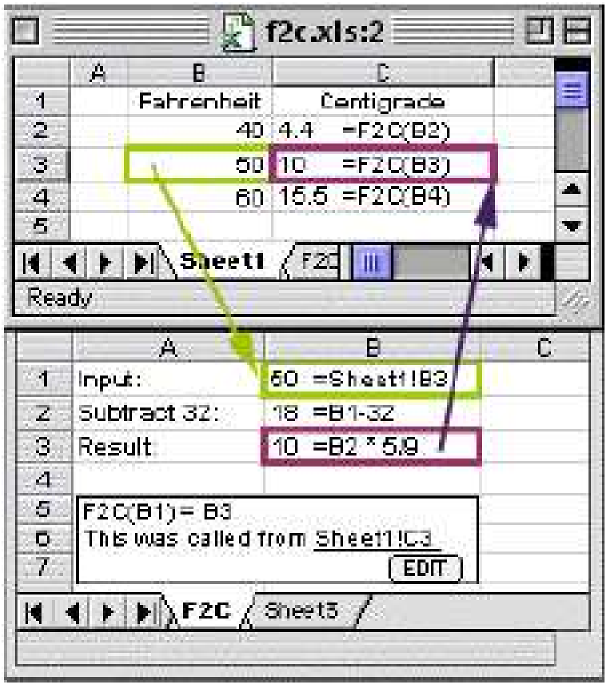
With this rule as basis a 'function instance sheet' (called 'function sheet' in the rest of this paper) is defined that contains the actual implementation of a user-defined function. The idea is that such a function sheet works exactly as a normal worksheet.

The function sheet is integrated in the spreadsheet by containing the user-defined function with formulas that has references to cells in other sheets in the spreadsheet. The formulas in the function sheet will thereby have direct references to 'live' data in other sheets and should one follow the concept another call to the user-defined function would create another function sheet equal to the first one but with different data references, i.e. a new instance of the function. The signature of the user-defined function is fixed predefined cells in the function sheet, i.e. other instances of the function will use the same cells for input and output as the first one.

With the idea above a user-defined function will be defined in a sheet that looks like every other sheet in the spreadsheet. It will also be evaluated the same way, i.e. all values will be continuously calculated and shown. Does the calling sheet change then so will the function sheets and vice versa.

Likewise will other spreadsheet functionalities such as formats, column and row widths/heights, frames, etc. be available exactly as they are in normal sheets.

See figure 2 for an example of a user-defined function as illustrated in [JBB03] (figure 2).



	A	B	C
1		Fahrenheit	Centigrade
2		40	4.4 =F2C(B2)
3		50	10 =F2C(B3)
4		60	15.5 =F2C(B4)
5			

	A	B	C
1	Input:	50 =Sheet1!B3	
2	Subtract 32:	18 =B1-32	
3	Result	10 =B2 * 5/9	
4			

F2C(B1)= B3
This was called from Sheet1!C3

Figure 2: A user-defined function as illustrated in [JBB03] (figure 2).

Good design?

The main idea obviously supports the ground rule but what about the outlined design concepts?

The *consistency* is good. A user from the target audience will with a short introduction be able to use the design since the available functionalities should not be foreign.

The *role-expressiveness* is good. Excel already does a good job here by indicating related (top-level) cells with colored frames.

Hidden dependencies are a general problem in Excel if one considers dependencies across different sheets. Here it is solved by extending Excel's dataflow arrows to also work across sheets.

Visibility and juxtaposability is improved in comparison to Excel since the design of a formula shows both the formula and the result value in a cell.

Premature commitment is solved by given the user the possibility to change his or hers formulas into user-defined functions. When the user realizes that a user-defined function might be a good idea he/she can get the user-defined function created automatically via the cells property window. In this way the user will avoid making a decision whether or not the formula should be a user-defined function or not before knowing how the formula will look like.

The *abstraction gradient* is low since a function sheet is an instance of a function and does not have a 'definition sheet' attached to it. In this way the user will never have to consider the abstract definition of the function.

Apparent problems

Especially regarding the discussion of the abstraction gradient the above produces a couple of questions that must be answered:

- If there are several thousand calls to a user-defined function, how can one avoid several thousand function instances?

  To avoid that the user has to face a large number of function instances that will be impossible to handle, an 'instance tree' is made for every instance of a user-defined function. With help from such a tree the user can navigate up and down in the function calls that are present in the current cell. Only the relevant instances of the function will be shown to the user.

- If it is only an instance of the function that is shown, how can one then change the definition of the function?

  This problem is solved by giving the user the possibility to change either all instances of the function (and thereby indirectly changing the definition of the function) or only the current one. This possibility is offered to the user in case of changes in a function sheet.

- What is the possibility for debugging of the user-defined functions?

  The possibilities are the same as Excel offers where changes in the source immediately are reflected in changes in the result. I.e. debugging and programming is so integrated that one can see it as one process.

Matrices

One cannot imagine a good spreadsheet representation without the ability to handle matrices. In the built-in functions in Excel matrices are supported such that matrices can be used as arguments, e.g. SUM(A1:D10), and the user-defined functions should therefore also support matrices.

This decision gives rise to problems defining the input in the function sheets since we will neither specify the size of the input matrix nor handle multiple input matrices (potentially of different size) in the same function sheet.

The above is solved by letting the input matrix live in a single cell in the function sheet. This requires that there is added an extra 'matrix' run-time type.

Additionally the following basic design choices are made:

- Every formula can have a matrix as result.
- Matrices are two-dimensional.
- A vector is a special case of a matrix (1×n or n×1).
- A scalar is transformed into a 1×1 matrix but only in connection to matrix applications.

3.2 Analysis and use of the theory

With the ideas from [JBB03] as our basis, our goal is to implement the described user-defined functions. Before we do this it is necessary to go through the paper of [JBB03] with real implementation in mind.

The design in the paper is described in regard to the user's experiences and demands but the paper lacks discussions regarding the more design technical problems which always pops up when preparing an implementation.

In this section we will therefore analyze the ideas and the basic design of the user-defined functions with a real implementation in mind.

3.2.1 The need and the target audience

In the day-to-day work with spreadsheets, problems are solved by the available and well known tools. The majority of the spreadsheet users know the built-in functions but there are only a small fraction of these who uses the more advanced ones. Partly because they are too difficult to learn and partly because the users do not think that they will solve their problem. This group of users does not know that there exist possibilities to define their own functions (e.g. via VBA in Excel) and can therefore not use this to their advantage. This often results in very large and slow spreadsheets containing several worksheets that are almost impossible to keep well-maintained. A lot of these spreadsheet solutions could be improved by the use of the user-defined functions from [JBB03].

We think that this group of users without doubt will benefit from easily accessible user-defined functions that can help to solve their problems. Since these functions will be quick and easy to use we also think that advanced users - users that make use of VBA - will benefit from these functions.

If the user-defined functions operates well both technically and visually they could most likely replace some of the built-in and VBA-defined functions. This might especially be the case in companies where local libraries are created for the companies' specific functions. Such functions could be loaded in while starting-up a spreadsheet (e.g. as template sheets) and thereby be ready for use exactly like the built-in functions today.

In our experience from the practical use of spreadsheets (primarily Excel) there is also a significant need for easily accessible, easy-to-learn and visually well-defined user-defined functions. The target audience is - as described in [JBB03] - primarily the large group of ordinary users but as mentioned earlier we think that also more advanced user will benefit from these functions.

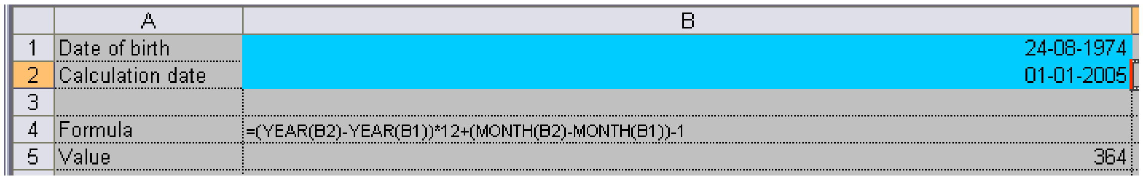
3.2.2 User-orientated design versus technical design

With the ideas and the design from [JBB03] one can easily put up a concrete example from 'the real world':

Example: Age calculation

In the life insurance business one has to define exactly how to calculate a person's age from his or hers social security number (SSN) or date of birth. This may sound easy but almost all IT systems used in this business run (calculation wise) in monthly intervals. This means that the company has to exactly define how to calculate the age.

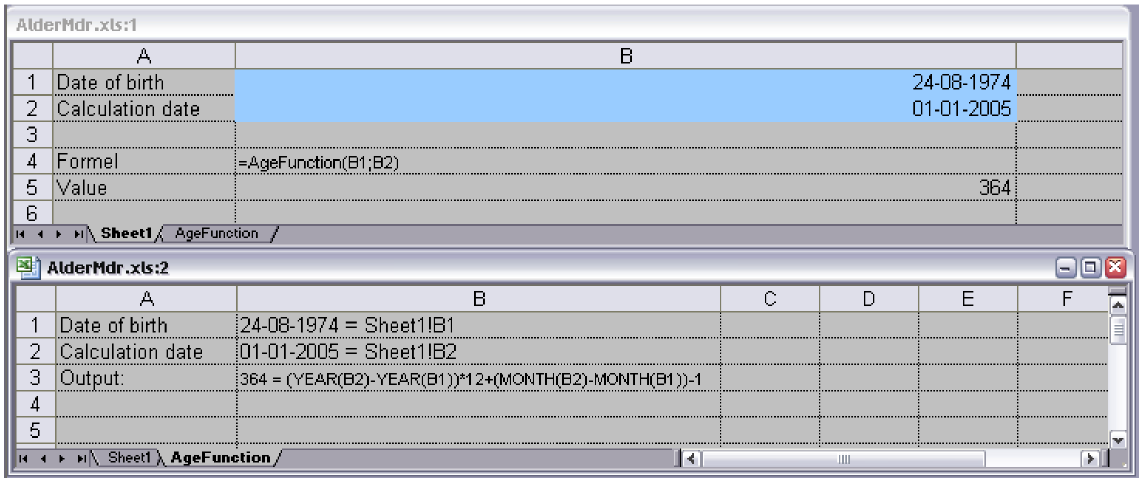
Figure 3 shows an 'ordinary' Excel where the age of a person is calculated in months on a given date from the person's date of birth. In the version of this spreadsheet that is used in 'the real world' several ages on several persons on several given dates are calculated but for simplicity we have modified the spreadsheet to only handling one at a time. As a curiosity one notice the subtraction of 1, this is due to the fact that insurance companies normally calculates your age as if your birthday is the first of the month after your actual birthday. In the figure this means that the birthday used is the first of September 1974.



	A	B
1	Date of birth	24-08-1974
2	Calculation date	01-01-2005
3		
4	Formula	=(YEAR(B2)-YEAR(B1))*12+(MONTH(B2)-MONTH(B1))-1
5	Value	364

Figure 3: 'Ordinary' Excel sheet for age calculation

Figure 4 shows how the user-defined functions could be thought implemented as described in [JBB03]. Notice, that the 'Edit' window described in the paper is not on the figure.



AlderMdr.xls:1

	A	B
1	Date of birth	24-08-1974
2	Calculation date	01-01-2005
3		
4	Formel	=AgeFunction(B1;B2)
5	Value	364
6		

AlderMdr.xls:2

	A	B	C	D	E	F
1	Date of birth	24-08-1974 = Sheet1!B1				
2	Calculation date	01-01-2005 = Sheet1!B2				
3	Output:	364 = (YEAR(B2)-YEAR(B1))*12+(MONTH(B2)-MONTH(B1))-1				
4						
5						

Figure 4: Excel sheet for age calculation by usage of user-defined functions

Even though the above example is very simple it is easy to see that many of the outlined benefits of user-defined functions are in play. To visualize good design is one thing but to actually make it is something else.

In [JBB03] focus is mainly on the user's experience of the user-defined functions (the design is actually realized from the user's demands and needs) but the paper leaves behind some unanswered questions that must be answered while doing the implementation (the technical design).

In the following sections we will outline the most obvious ones. A detailed discussion of the design and the problems/issues created hereby will be taken as they occur through our iterative approach.

The nine concepts for measuring good design

With the nine concepts for measuring good design (see section 3.1.1) as basis the next step in the article [JBB03] is to develop their design of the user-defined functions. In this section we will discuss these concepts and hereby review their importance and influence on the technical design.

Consistency: With the defined target audience the consistency in the design has to be good. If not, we will end up in a situation like the one we have today with VBA versus Excel. It is therefore a critical area both in the overall design but also in every detail. The overall user-orientated design presented in the paper provides us with a good design and it is therefore important that the technical design does not spoil it.

Role-expressiveness and hidden dependencies: Like with the consistency it is important that the user understands how the available components relates to each other and that these relations are visualized to the user. To visualize the dependencies and relations [JBB03] proposes to use Excel's current visual properties, i.e. colored frames and arrow indicating the dependencies (extended to work across sheets). This solution might work when we are dealing with few and simple function sheets but one can easily think of situations with several non-simple function sheets where this results in a visual chaos of arrows and colored frames.

Visibility and juxtaposability: This area is proposed solved by letting the user see both the formula and the result in each cell. This is supposed to include all visible cells and not only the active ones. At first sight this may seem okay but in large sheets with several non-trivial formulas it will be very unhandy and difficult to distinguish between results and formulas.

Premature commitment: To give the user the possibility to automatically get a user-defined function defined from a formula seems like a good and user friendly idea. An implementation of this is not in scope for this paper though (see section 1.2).

Abstraction gradient: The level of abstraction is kept down by avoiding the user to make decisions regarding the definition of the user-defined function instead the user only sees and uses instances of the function. On changes in a function sheet the user will be prompted whether the change should include all other instances of the function or only this specific instance. These considerations seem reasonable. As we shall see later on it is not as easy to implement as it seems.

The next three concepts are not discussed in [JBB03]. We will briefly discuss them here though.

Viscosity: Since changing a cell in a function sheet is no different than changing a cell in a normal worksheet the viscosity in the design is not different than normal Excel.

Progressive evaluation: This area is improved in comparison to normal Excel since the users can encapsulate complex formulas and thereby easier being able to handling the formulas.

Error-proneness: At first sight the tendency to errors is no different than normal Excel but because of the problems regarding the role-expressiveness and hidden dependencies there might be areas around situations with several non-simple function sheets where the user potentially has to handle a large number of instances, function sheets and the dependencies across these.

A function sheet is an instance

In the proposed design a function sheet is only an instance of the user-defined function. The user never defines the function but only instances of it. Nevertheless it should be possible to change the definition of a user-defined function, not only an instance of it.

A good implementation of this is not obvious. One solution could be to define a kind of default instance in the technical design. All function calls should then have some sort of copy of this default instance. Thereby a change in the default instance will change all the functions instances. The problem is not 100% solved because the function is not completely defined by the default instance: The user must still define the function signature. This will affect one or more of the outlined design concepts. E.g. what is the affect of letting the user state the function signature? Will it change the user-oriented design severely or is it only a negligible detail?

The function signature

An important part of the users experience with a spreadsheet is how intuitive it is to define the function signature. Naturally we want the signature to be easy to handle and edit both when working with function sheets, when editing the signature and when using the function. But it is not straight forward to find a good way to fulfill this wish.

The proposed solution in [JBB03] is to let every function sheet contain an edit button which shows the function signature in a dialog window. Then it is possible to change, delete and add arguments, names, documentation, etc. It is here indirectly assumed that e.g. the arguments to the function are fixed to a specific area in the sheet (e.g. the first column). This solution is easy to implement and easy for the user to understand, but it is not plausible for several reasons:

- Allocating a part of the sheet that the user cannot control is generally a bad idea. Users like to be in control and different users will have different preferences for their setup. I.e. it limits the user's real-estate management.
- User input is not removed completely because we are still required to define how the result is represented in the function sheet. Allocation a part of the spreadsheet

as the result area will limit the possible result set of the function by restricting or eliminating matrix results (see section 7).

- Most built-in functions in Excel are functions with a fixed number of arguments. Limiting the number of arguments in a predefined column will require user input and some kind of visual indication that the argument cells are restricted to a certain area. Since the restriction is most likely to be the default setting this visualization will be used most of the time.

Every function call is an instance of the function

One of the main ideas in the design is that every function call to a user-defined function is an instance of the function. But what happens when there is several thousand calls to a user-defined function and thereby several thousand function instances?

In [JBB03] this question is also asked but only answered/solved seen from the user's side (see section 3.1.2) by creating an instance tree. But how about the technical side of the problem? Consider how several thousand function instances will affect performance, e.g. recomputation of the spreadsheet, memory usage, etc.

Recursion

In [JBB03] recursive user-defined functions are proposed not to be supported. Three reasons are stated:

1. Recursive functions are less useful than in mainstream functional languages.
2. Recursion threatens consistency because of the proposed linked-worksheet model (dataflow arrows across sheets).
3. Recursion leads to deep invocation stacks.

As argued in section 3.2.1 we think that the user-defined functions can be beneficial also to advanced users. This group of users is used to recursive functions and will assume that a user-defined function application supports recursion. Further, we see problems concerning the use of dataflow arrows across sheets (see section 3.2.2) and will not recommend an implementation of it. Lastly, deep invocation stacks may occur in connection with recursion but as we shall see one can find solutions to handle these situations.

Matrices and higher-order functions

As we shall see later on the possible use of matrices in user-defined functions are limited if we do not support higher-order functions. Higher-order functions are not discussed in [JBB03] but as we also argued in the previous section, more advanced users will assume that the application supports higher-order functions.

4 Methodological approach

Before developing a spreadsheet with user-defined functions we need to define our methodological approach. To define which approach to use we need to consider the following:

1. We do not initially have a 100% clear picture of the detailed design requirements.
2. We prefer a progressive and continuous development since we initially do not know whether or not the theory works in practice and to be able to have control over the progress.
3. We wish to uncover essential design aspects and identification of potential risks early in our development.

To support these considerations we have chosen an evolutionary development via an iterative enhancement method.

By evolutionary development we mean development that requires a sequence of cycles of design, implementation and evaluation without any attempt to capture a complete set of requirements in advance. A prototype of partially known requirements is implemented first. When more understanding of the requirements is gained, new requirements are then implemented. Each successive prototype explores new needs and refines functionality and design that has already been implemented. See [Flo84].

To accomplish this we use the iterative enhancement method which [BT75] defines as *'a practical means of using a top-down, step-wise refinement approach to software development. A practical approach to the problem is to start with a simple initial subset of the problem and iteratively enhance existing versions until a full system is implemented. At each step of the process, not only extensions but also design modifications can be made. In fact, each step can make use of stepwise refinement in a more effective way as the system becomes better understood through the iterative process.'*. The iterative enhancement method assumes that there is an initial 'project list' which is achieved in stages beginning with the implementation of a simple subset and undergoing successive refinements to the software as the development goes on until a final implementation is developed. While the software design can undergo modification one should note that the project specifications themselves do not change 'stepwise'.

The four prototypes

Due to the above considerations and the described method we divide our development into four prototypes:

Prototype 1 Simple function: Includes

- Setting up a spreadsheet.
- Import from other spreadsheets.
- Simple function where we have N known arguments, the parameters are simple and the result is simple.

Prototype 2 Advanced function: Includes

- Simple function where we have a variable number of arguments and the number of arguments are known when the function is called.
- Referring function. Function which refer to cells outside the function sheet.

Prototype 3 Matrix function: Includes

- Matrix function where both the argument and the result is a matrix.

Prototype 4 Recursive and higher-order function: Includes

- Recursive function. Function that calls itself.
- Higher-order function. Function that has another function as argument.

The next four sections describe the four prototypes one by one.

5 Prototype 1: Simple functions

In the first prototype we set up a spreadsheet design and make it work with a simple user-defined function.

Our primary objectives are:

1. Setting-up a spreadsheet environment and design.
2. Making import from other spreadsheets possible.
3. Implementation of a simple user-defined function.

To be able to implement any user-defined functions in a spreadsheet we will of course first have to define and set-up a spreadsheet. This involves exactly defining how each element of the spreadsheet must be implemented (see section 2.2 for a definition of the terms used).

In order for us to make easy use of the cases from Excel spreadsheets and to avoid having to reenter test data we must also have an import functionality that can load these spreadsheets into our own.

Last but not least we will implement the case spreadsheet described in section 3.2.2.

5.1 Setting-up and designing a spreadsheet

In the following we describe the spreadsheet representation. This will include definition and description of simple data and objects but also of areas more complex.

Most of the terms in the following tables are based on [Ses05].

Term	**Description**
Spreadsheet	A collection of sheets.
Sheet	A rectangular array whose elements may contain null or a cell.
Non-null cell	May be • a constant floating-point number or • a constant text string or • a formula or • a matrix formula.
Formula	Consists of • a non-null expression to produce the cell's value and • a cached value and • a spreadsheet reference and • an up-to-date field and a visited field.
Matrix formula	Contains a non-null cached matrix formula shared among a number of cells. The cached matrix formula produces a matrix value giving the values of all those cells, so a matrix formula also contains the cell address in the matrix value of that cell's value.
Cached matrix formula	Consists of • a formula and • the address at which that formula was entered and • the corners of the rectangle sharing the formula.

Term	**Description**
Expression	May be • a constant floating-point number or • a constant text string or • a cell reference or • an area reference (two relative/absolute references) or • a call of an operator or function.
Value	Produced by evaluation of an expression and may be • a floating-point number or • a text string or • an error value or • a matrix value.
Atomic value	May be • a floating-point number or • a text string.
Matrix value	A rectangular array of values, some of which may be null. A matrix of size 1x1 is distinct from an atomic value.
Cell address	The absolute, zero-based location (col, row) of a cell in a sheet.
RARef	A relative/absolute reference (colAbs, col, rowAbs, row) used to represent cell references A1, \$A\$1, \$A1, A\$1 and area references A1:\$B2 and so on. If the colAbs field is true, then the column reference col is absolute (\$), otherwise relative (similar for the row references).
Function	Represents a built-in function (e.g. SUM, SIN, etc.) or operator (e.g. +, -, etc.).

5.1.1 Representation of cell reference

Absolute references are stored as absolute zero-based cell addresses and relative references as offsets relative to the address of the containing cell. This is represented by the class RARef as shown in listing 1.

```
class RARef
{
  public bool colAbs, rowAbs; // true if reference is absolute
  public int colRef, rowRef;
  ...
}
```

Listing 1: Representation of cell reference

5.1.2 Absolute and relative sheet references

In an expression a reference to a cell is represented by the class CellRef. To enable references to cells in other sheets (the Sheet1!A1 notation in Excel) a Sheet field is added to CellRef class as shown in listing 2. If the sheet field is non-null then the reference is sheet-absolute and refers to a cell in that sheet. If the sheet field is null then the reference is sheet-relative and refers to a cell in the current sheet.

```
class CellRef : Expr {
  private readonly RARef raref;
  private readonly Sheet sheet; // null if sheet-relative, else sheet-absolute
  ...
}
```

Listing 2: Determining absolute and relative sheet references

5.1.3 Recomputation

A spreadsheet is recomputed by recomputing every sheet. A sheet is recomputed by recomputing every cell. A formula cell caches its value and a matrix formula caches the value of the underlying matrix-valued expression, which is shared between all the cells participating in the same matrix formula.

Cached expressions has two flags: *Visited* and *uptodate*. At the beginning of a recomputation both are set to false.

A cached expression is evaluated as follows:

1. If *uptodate* is true, return cached value.

2. Else, if *visited* is true, the cell depends on itself, stop and report a cyclic dependency.
3. Else, set *visited* to true and evaluate the cell's expression.
4. If the evaluation succeeds, set *uptodate* to true, cache the result value and return it.

We introduce a spreadsheet-global field '*set*' and maintains the following invariant:

Between recomputations, the visited and uptodate fields of every cached expression equals the set field of the spreadsheet.

Resetting the *visited* and *uptodate* fields of all cached expressions to false is a simple matter of inverting the global *set* field.

5.1.4 Cyclic references

The value of a cell may depend on itself. The purpose of a cached expression's *visited* field is to discover such dependencies. After discovery of a cycle, all cached expressions have their *visited* and *uptodate* fields reset to the value of the global *set* field.

5.1.5 Functions with multiple arguments

Functions such as SUM and AVG take multiple arguments. These are evaluated by applying a suitable action to all arguments; recursively applying it to the elements of every matrix-valued argument.

5.1.6 Functions with matrix-valued results

Some functions produce a matrix value as result. The matrix value result is simply a matrix of values.

5.1.7 Parsing of spreadsheet formulas

Formulas are parsed by a simple parser generated using CoCo/R (see [MWL]).

5.2 Importing from other spreadsheets

During development we want be able to easily use the cases from Excel spreadsheets and to avoid having to reenter test data each time the program is modified. We could define our own format, but choosing to import data from Excel gives us several advantages. Excel gives us much richer environment for editing test data and relieves us from the task of defining and implementing a file format.

Recent versions of Microsoft Office supports an XML file format (see [Mic]), for which Microsoft publishes the schemas. These schemas make importing from Excel a question of parsing XML and building the spreadsheet accordingly.

The only major obstacle is the different representations of cell references. As opposed to our A1 representation (the one used by default in Excel user input), the Excel XML format uses a R1C1 format (row number and column number). The transition to our A1 representation is handled by a regular expression before parsing formulas. This is slightly odd since the internal representation of our program is R1C1, but enables us to use the formula parsing described above without futher modifications.

Since function sheets are not supported in Excel we have defined a naming scheme for importing functions:

1. Prefixing the name of the sheet with '@' implies that the sheet will be imported as a function sheet. The name of the function will not include the '@'.
2. Function signatures are represented in a special format inside the sheet. The format depends on the prototype in question, but is generally a textual representation of references to the cells where arguments and results can be found.
3. The signature is a range of cells which is located using a named range in Excel. The name of the range is the name of the function postfixed with the string '_signature'.

Calling a user-defined function from a cell in Excel will return #NAME but will be evaluated correctly once imported in our prototype, given the signature is defined as described.

5.3 Implementation of a simple user-defined function

Now that we have a spreadsheet and import functionality defined we focus on the implementation of a simple user-defined function.

During this process we have to make some design considerations and decisions. In the following we have described the most important of these.

5.3.1 Defining function signatures

To define a function we need to define the signature, i.e. define which arguments it takes and how the result is returned. In the context of spreadsheets we furthermore need to define how the arguments and the result are represented in the function sheet. The ease of defining a function is a very important part of the user experience with function sheets. If this part is too complex, then it will never be used in practice.

We will discuss three solutions which are familiar to spreadsheet users and choose one for our implementation.

Predefined cells: To accommodate the mindset of the spreadsheet users we would like to represent the arguments in the function sheet as values in cells. The simplest solution involves allocating an area of the sheet for the arguments, say the first column of the function sheet. But this solution has several drawbacks as stated earlier in section 3.2.2.

On the other hand this solution has the benefit that representing the arguments in cells allows for testing of the function by inserting values directly into the function sheet and checking the result cells as one would do with any other sheet. This is desirable also in a more complex solution.

User-defined cells: One of the problems with the solution above is that it limits the user's possibilities. To solve this issue one can let the user define which cells hold the arguments of the function. This solution requires only a simple extension to the user input and keeps the benefits from the predefined cells solution.

The function signature will now be on a form as in the example below.

Price(A3;A5;B8)=C4

Note that the result value also can be a range of cells (e.g. C1:C8).

The number of arguments will automatically be limited to the number of arguments given by the user (three in the example above). If the user wants a variable number of arguments he or she will have to select a column for the arguments. Excel already supports a functionality we can reuse: The formula =SUM(F:F) is valid for the built-in function SUM that in this case returns the sum of all values in column F.

In this way a function with two mandatory and some optional arguments has the form as the example below.

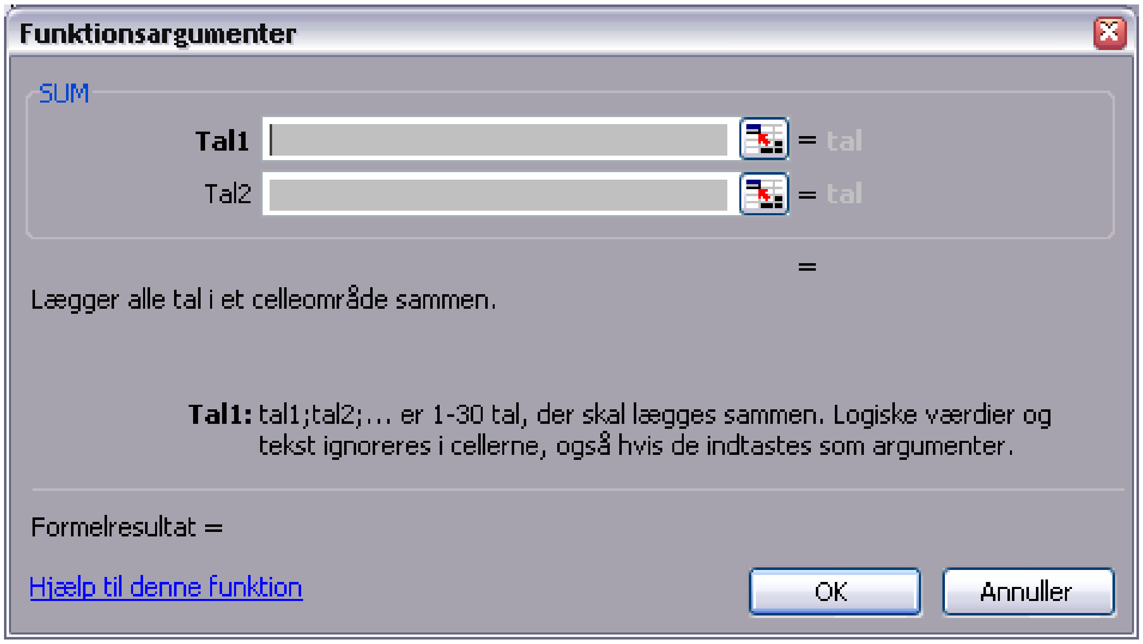
Price(A3;A5;B:B)=C4

Note that in most of Excel's built-in functions with variable number of arguments the number of arguments are limited to e.g. 30. Also note that there can be only one variable range as argument to a function and that this range has to be the last argument of the function.

When selecting the location of the function arguments the user can choose to name the arguments. This can be very useful when the function is used by other sheets since it gives information about the arguments. In that way it will become quicker and easier to use the function.

Named arguments: In Excel the user can name a cell or a range of cells and use the name in formulas instead of using the cell reference. The names are shared between all sheets which makes it easier to refer to cells in other sheets.

If we introduce local names for each function sheet this feature can be used to handle the arguments. The user will then define the arguments in a dialogue which will be familiar to the one used in Excel (see figure 5) just with the possibility to name the arguments.



Lægger alle tal i et celleområde sammen.

Tal1: tal1;tal2;... er 1-30 tal, der skal lægges sammen. Logiske værdier og tekst ignoreres i cellerne, også hvis de indtastes som argumenter.

Figure 5: Screenshot from Excel (Danish)

Clever naming of arguments will ease the use of the user-defined functions in formulas but there are some drawbacks connected to this solution. There is an inherited conflict with the global naming system found in Excel, since defined names are global. This will require that our target audience can distinguish between local and global names. One could use a prefix on the function arguments (e.g. %argument1,

%argument2, etc.) but to not confuse the user the prefix would have to appear on the signature. One can not expect the user to key in the names beginning with e.g. % though. So it is doubtful whether the confusion can be avoided.

To summarize, we have three possible designs for a function signature, predefined cells, user-defined cells named arguments. We choose the user-defined cells design for three reasons:

1. Even though the predefined cells design is the design that best satisfies the nine concepts for good design (see section 3.1.1) it conflicts with our ambition to implement matrix functions in a later prototype.
2. With the user-defined cells design we are able to implement matrix functions later on and at the same time satisfy the nine concepts. The cost is that the user will have to define the location of the function arguments but this is considered to be a rather small addition.
3. The named arguments design is good idea seen from an advanced spreadsheet user's point of view but our target audience will definitely run into problems understanding the design. In other words the abstraction gradient concept would fail with this solution (Note that even the naming system found in Excel is rarely used by people that belong to our target audience).

Defining the result value

When a new function sheet is first added to the spreadsheet the user can treat it like any other normal sheet. In order not to violate the premature commitment concept we do not want to force the user to make a decision about the function signature until he/she is ready to do so. Therefore one has to look at how the function behaves before the user has given any input to the function. An empty function sheet will be a function without arguments and without a result value. Since a function can exist without arguments there is no problem in missing arguments, but if there is no result value defined we have a problem.

We have two options how to define the result value:

1. Define that the result value is always the value in a single predefined cell, e.g. A1. The predefined cell will be changeable by the user and the function will always have a result value. This simplifies the definition of the function but claims an area of the screen at first use, which forces the user to change the signature in order to define the sheet the way he/she wants it.

2. Return an error message or undefined value.
   Our target audience will most likely recognize the value #UNDEFINED since it is used in Excel. The user is this way forced to define the result value for the function, a step he or she probably would take anyway. Most importantly we delay the decision to a time when the user is ready to make the decision.

If one chooses the first solution, the default signature will be on the form as the example below.

Price()=A1

otherwise the form will be, e.g.

Price()=#UNDEFINED

We have chosen the second option since it supports the premature commitment concept and because our target audience is used to handle error messages from Excel. Moreover choosing the first solution will conflict with the consistency concept since the predefined result value conflicts with the thoughts of defining the arguments by user-defined cells (although the predefined result value is changeable).

5.3.2 Representing the function instance

When a function sheet is invoked we want to represent the instance resulting from the invocation, preferably in an optimized efficient way. The first representation is rather naive: When a function sheet is invoked a full copy of the function sheet is made, the copy is then modified to represent the arguments and then recomputed. The result of the function can be read from the copy which also contains all intermediate results from the recomputation.

Copying the function sheet is a safe way to represent the function instance because the copy is totally separated from the original, but it is hardly efficient. We can improve on this deficiency by looking at the stability of the data in our function sheet. In a normal sheet each cell holds one value, represented either as a constant value or as the result of a computation. Constant cells can be copied by-reference, whereas cells that change need a deep copy.

We have the following rules for copying a function sheet:

1. A new instance of the function sheet is created to represent the copy.

2. Cells are copied by-reference or by-value depending on the cell type.
    (a) By definition, constant cells do not change and are thus copied by-reference.
    (b) Formula cells change depending on the referenced cells or if the function is volatile (like Random). One should be able to determine whether a function is constant or not. Non-constant functions are copied by-reference. Internally the formula keeps a reference to the parsed expression; this reference does not change and is reused in the copy.
    (c) Cells containing arguments can be treated as constant values in the copy. The original cell is not copied; instead a new is value is inserted.

This is a more efficient solution than the full deep copy, but it can be further optimized. However, this solution satisfies our goals for this prototype. We will revisit the instance copy in later prototypes.

5.3.3 Invocation list

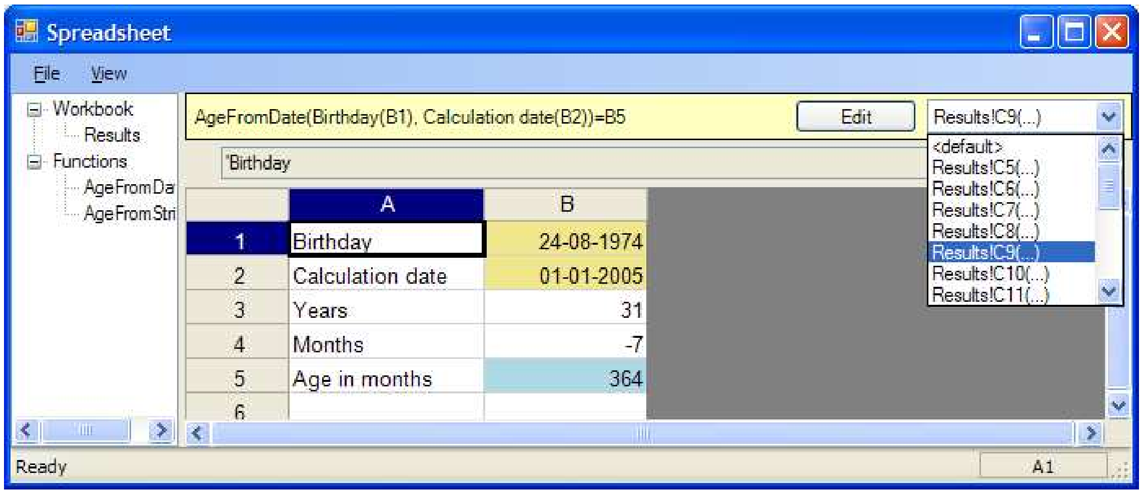
In [JBB03] the possibility for the user to navigate through a so-called "invocation tree" is mentioned. This requires that the spreadsheet for each user-defined function stores an invocation list. This list gives the user the opportunity to examine the function sheet as it looks at every call to the function. Seen from the user it is possible to navigate through several sheets via a tree structure but seen from the function sheet's point of view it is only a list of calls to functions. In connection to the design of these lists there are a number of decisions that have to be made.

When a function is called, a new item on the invocation list is created. This item contains a reference to the calling cell, arguments and a representation of the result of the call. In the following we discuss how these items look like and for how long they must be "kept alive".

Cell references: A reference to the calling cell is given by a cell reference with information about the sheet containing the calling cell. This is not necessarily a unique reference though. One could choose to set a postfix on the name, e.g. F#1, F#2, etc., if there are multiple calls to F from the same cell. Since it is not clear that it is the call far to the left in a formula that is considered to be the first call (evaluated first), one has to make sure that the numbering feels natural for the user. This is especially important in respect to non-strict functions (e.g. IF) where an argument may not be evaluated at all. A possible solution is to number the calls in connection to the parsing of the formula.

Arguments: The arguments to a function can be represented as expressions or as values. In [JBB03] it is proposed to use expressions but if these exceed a certain size they will not give the user any valuable information, which would be in conflict with the visibility concept. If on the other hand we represent the arguments as values, the invocation list will give an overview of how each function evaluates the argument expressions. With this solution it can still be an advantage to postfix the calling cell as described above, even though we can distinguish between the arguments, since it might be difficult for the user to track the call back to the right place in the formula.

Representation of the result: The representation of the result is at first glance a copy of the function sheet (see section 5.3.2). When the user chooses an item (see figure 6) on the list the related copy of the sheet is made visible.



AgeFromDate(Birthday(B1), Calculation date(B2))=B5

'Birthday

	A	B
1	Birthday	24-08-1974
2	Calculation date	01-01-2005
3	Years	31
4	Months	-7
5	Age in months	364
6		

Figure 6: Selecting an invocation from the list.

But it can be difficult for the user to see what has been changed when editing the function sheet. This is not important unless the user changes the cells where the arguments are placed. To solve this, one could protect these cells or always change sheet to the function sheet when a edition is in process. This solution is connected to the life time of the items in the list.

Life time for an item: In [JBB03] there is spend some effort discussing what to do with the items on the invocation list if there is a change in the function signature. This must be based on the assumption that the life time of these items are relatively long or perhaps that the list is only cleared when the user tells it to.

To some extent we agree to this, i.e. we think that the functionality will become too weak if the invocation list only contains the latest recomputations of the sheet. But we also realize that it will require a new flag on the items if the user must be able to distinguish between new and old function calls. In this way the list will be updated with new items if the function signature is changed.

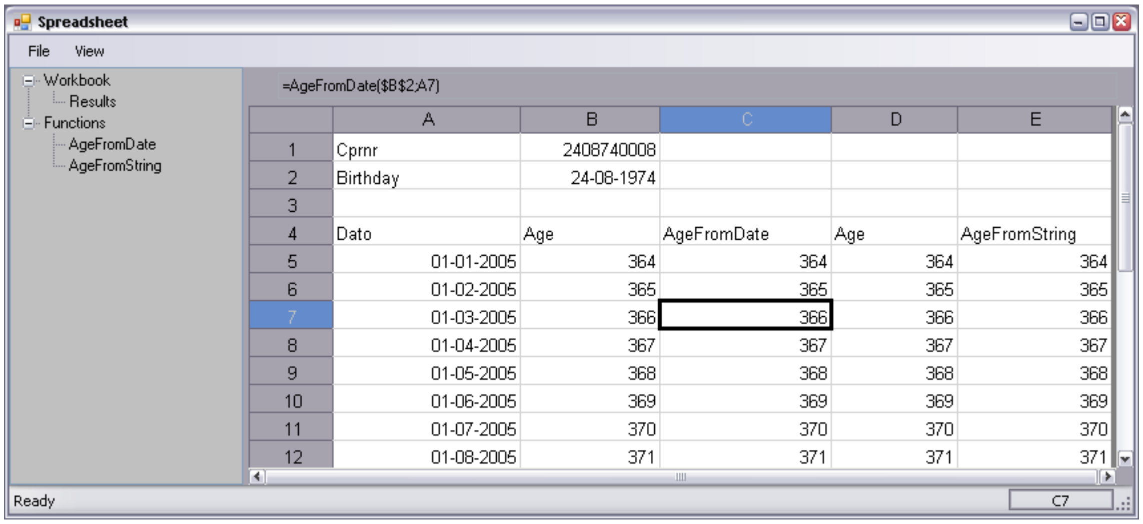
In Excel one can decide that the spreadsheet must only be recomputed at the user's command (via the "F9" button). This feature makes our items useless until a recomputation is carried out. To control this we have to require that the necessary cells are recomputed before one can navigate through the invocation list. For the time being our implementation is designed to recompute the entire spreadsheet when a function is changed. This means that it is relatively simple to get an updated invocation list. There are several drawbacks to this solution though as we will see in the later prototypes.

To represent the invocation list to the user we have added a drop-down list in the top right corner on the function sheets, as shown in figure 6.

5.4 Case: Calculation Age

In our field of work, one simple but very common calculation is a person's age in months on a given date. The input can be the birthday or a social security number. The Danish social security number ('CPR number') was introduced in 1968, long before anyone thought about the Y2K problem and thus gives the birth year with only 2 digits. To compensate, the century is encoded in the serial number, e.g. if the year is 00 and the serial number starts with 0,1,2, or 3 the year of birth is 1900, otherwise it is 2000. This adds complexity to age calculation formula which can easily be converted into a user-defined function.

Our example calculates the age from a date or from a string assuming that the birth year is in the 20th century. After importing the original Excel spreadsheet to our own spreadsheet we can use a user-defined function to do the desired calculation. Figure 7 shows the function in action.



=AgeFromDate($B$2;A7)

	A	B	C	D	E
1	Cprnr	2408740008			
2	Birthday	24-08-1974			
3					
4	Dato	Age	AgeFromDate	Age	AgeFromString
5	01-01-2005	364	364	364	364
6	01-02-2005	365	365	365	365
7	01-03-2005	366	366	366	366
8	01-04-2005	367	367	367	367
9	01-05-2005	368	368	368	368
10	01-06-2005	369	369	369	369
11	01-07-2005	370	370	370	370
12	01-08-2005	371	371	371	371

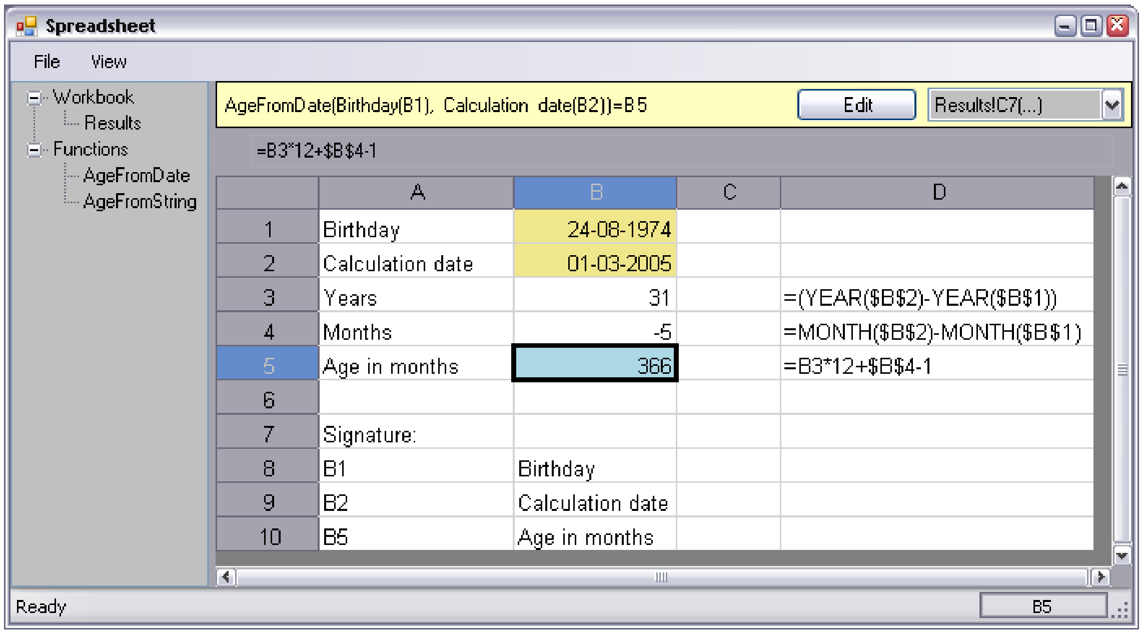
Figure 7: Screenshot from the age calculation spreadsheet.

Figure 8 shows the function sheet for the age calculation in months from a date value. The yellow ribbon in the top of the screen contains the function signature

AgeFromDate(Birthday(B1),Calculation date(B2))=B5.

The signature is imported from Excel by the cells A8:B10. This example contains intermediate results, something which is not possible with regular functions. Also notice that the drop-down in the ribbon shows that the current instance was invoked from *Results!C7*, giving a calculation date of 1-3-2005. The screen shows exactly the result (including intermediates) from this invocation.

Notice that we have showed the formulas used in column B in the respective row in column D. We will use this representation throughout the cases in the paper. Also notice that the Danish date format is dd-mm-yyyy.



AgeFromDate(Birthday(B1), Calculation date(B2))=B5

=B3*12+$B$4-1

	A	B	C	D
1	Birthday	24-08-1974		
2	Calculation date	01-03-2005		
3	Years	31		=(YEAR($B$2)-YEAR($B$1))
4	Months	-5		=MONTH($B$2)-MONTH($B$1)
5	Age in months	366		=B3*12+$B$4-1
6				
7	Signature:			
8	B1	Birthday		
9	B2	Calculation date		
10	B5	Age in months		

Figure 8: Screenshot from the function sheet for the age calculation.

Apart from the discussed solutions made on the technical design in section 5.3 the visual interface to the user is slightly different than the one proposed in [JBB03]. This is primarily caused by two facts:

- The visual representation of multiple sheets in Excel is replaced by a ”sheet tree” in a separate window to the far left of the screen.
- The visual representation of the invocation list is shown in [JBB03] as colored arrows (expanding Excel's existing feature) whereas we visualize the list as a drop-down list.

For both of the above bullets the difference is caused by our limitations in this work (see section 1.2) but as we shall see later the colored arrows representation of the invocation list might turn out to be unfortunate.

The differences impact the hidden dependencies and role-expressiveness concepts but it is difficult to say whether it is a positive or negative impact (depends on the user). Therefore we regard these impacts to be negligible. Alternatively it should be fairly easy to make the visual interface as desired but this is beyond the scope of our work.

6 Prototype 2: Advanced functions

In this prototype we expand our first prototype to handle more advanced user-defined functions.

Our primary objectives are:

1. Implementation of a simple function with variable number of arguments. The number of arguments is known when the function is invoked.
2. Implementation of a referring function, i.e. a function which refers to cells outside the function sheet.
3. Implementation of a non-strict, lazy evaluation of arguments. Arguments are evaluated when used, thus unused arguments are never evaluated.

We will have to modify central parts of our design from prototype 1 in order to incorporate solutions for the above objectives. But the first major change, which will also facilitate solutions to future objectives, is due to a drawback in the design of prototype 1. We will start with a discussion of this drawback.

6.1 Instance copy

Until now we have made a full copy of the function sheet when we had a new call to the function (see section 5.3.2). A consequence of this is that we will evaluate all data in the sheet including auxiliary data (i.e. data not used). This is a very inefficient solution since we only need to evaluate the required values to show the result.

To optimize the instance copy we need to revise the invocation list design in prototype 1 (see section 5.3.3).

6.2 Invocation list revised

When a function call returns, the instance copy of the function sheet is no longer interesting since the result is already extracted. The copy is only used to show intermediate results and can be recreated when the user chooses an item on the invocation list. This leads us to focus on a solution where the representation of the invocations on the invocation list only contains the calling cell and the evaluated arguments.

Notice that this solution will also be helpful if one wish to implement code generation since one does not need to get the intermediate results out of the code generated. If one wishes to inspect the calculated sheet, it can easily be recreated as done previously. If, on the other hand, one wishes to examine the code generated, it can be inspected using an IL debugger.

Recall that only formula values and arguments change between invocations and these are the values we need to keep track of. Following the thoughts above we introduce a CalculationContext which contains a collection of (CellAddr, Value) pairs, where results are stored when formulas are evaluated in the function call. This is, if e.g. cell A7 contains a formula which have been evaluated to 42 in this function call, there would exist an entry (A7, 42) in the context. Each time a function sheet is invoked a new CalculationContext is created and added to the function's invocation list. After the sheet has been calculated the CalculationContext is only used for viewing purposes. If the sheet has not been evaluated the CalculationContext will be empty. All sheets (both function sheets and worksheets) have a default context which stores the result of the normal calculation. If the user chooses to edit the function sheet he or she will edit the default context.

```
public class CalculationContext
{
  private Expr[] arguments;
  private Dictionary<CellAddr, Value> values;

  ...

  public Value this[CellAddr ca]
  {
    get { ... }
    set { ... }
  }
}
```

Listing 3: Partial class definition of CalculationContext

Listing 3 shows that CalculationContext contains a list of arguments (this is null for the default context in Worksheet) and a Dictionary with the (CellAddr, Value) values we have calculated. We change the implementation to pass along the CalculationContext instead of the instance copy when we recompute sheets. An indexer is introduced to access the dictionary and handle exception rising from missing values.

6.2.1 Function arguments

The function arguments (together with the function name) are what identifies the function call and must therefore be part of the CalculationContext. The cells where the arguments reside are initially (when the function sheet is created or imported) of any type the user decides, most likely a ConstCell with some default value.

In prototype 1 the arguments were inserted in the chosen cells in the function sheet. We can no longer replace the cells with the value of the argument because we only have one copy of the sheet. Now we need to support the arguments in the CalculationContext. When a CalculationContext is initialized the arguments are placed on the internal list of values. To read the arguments back from the context we introduce a new kind of cell called ArgumentCell. See subsection 6.2.3 below for more details on ArgumentCell.

6.2.2 Recomputation

The method for recomputation of the spreadsheet described in section 5.1.3 now needs to operate on CalculationContext. This means that:

- A formula cell reads the result from its CalculationContext, other cells are unaffected.
- To start the recomputation we empty the CalculationContext for results using a reset method. The reset methods call the method Clear on the internal dictionary , so after reset no (CellAddr, Value)-pair exits for the context.
- A formula cell is set to 'visited' by placing a special VisitValue (special class inheriting from Value acting as a marker and found as a static property on the Value class) in the context. This value is only used to detect cycles.
- A formula cell is up-to-date if there exist a Value different from VisitValue for the given cell. We have a cycle if we encounter the VisitValue during calculations.

For a source code representation of the last two points, refer to listing 4. Notice that a CyclicException is thrown if we are about to return the special VisitValue marker.

```
public override Value Eval(CalculationContext context, int col, int row)
{
  CellAddr ca = new CellAddr(col, row);
  Value v = context[ca];
  if (v == null)
  {
    context[ca] = Value.VisitValue; // indicates that the result is pending
    v = expression.Eval(context, col, row);
    context[ca] = v;
  }
  if (v is VisitValue)
    throw new CyclicException();
  return v;
}
```

Listing 4: Evaluation of formulas using CalculationContext

By using CalculationContext this way we have simplified the visited/up-to-date pattern from section 5.1.3 since we do not need to call every cell to reset the sheet. Now resetting the sheet is simply a matter of clearing the values in the context.

6.2.3 Non-strict arguments

How do we handle the situation where an argument evaluates to an error but may be unused?

If we do nothing about it we might get an error value in some cell which we will 'carry around'. If the error does not affect the result it does not matter. One could think of a function like e.g. 'IF X > 0 THEN RETURN 1/X ELSE RETURN -1' which is represented in a spreadsheet as IF(A1>0, 1/A1, -1). The function will evaluate and give a result even though X is 0 since the calculation continues with error values. Obviously evaluating all arguments works, but since dividing with zero is impossiple we need to change this. Also, this of the clock cycles wasted by calling such a function several times or having cyclic references involved in such formulas.

To facilitate non-strict arguments we make two changes:

1. When the function must be computed we only evaluate the result cell. It will then run through its dependencies to evaluate these. Unused cells will not be evaluated, for instance auxiliary cells the user created for testing the function inside the function sheet.
2. To avoid evaluating arguments, the ArgumentCell must hold a reference to the argument expression (including the sheet from where it originates) and only evaluate the argument when needed, e.g. the expression 1/A1 is only evaluated when A1 is greater than zero in the formula above. When evaluating an argument the value is placed on the CalculationContext like the results of a formula cell.

But we must be aware that we might be forced to evaluate the arguments anyway since:

1. Upon displaying a sheet all cells are evaluated. So cells that are not evaluated in connection with the function call will be evaluated for display.
2. If the arguments are vital for the visual representation of the function call they must of course be evaluated upon display.

6.3 Supporting variable number of arguments

To support a variable number of arguments we need to change the function signature of the user-defined functions used in prototype 1. But before we can do that we have to make sure that we represent a function with a variable number of arguments in a sensible way. Such a representation can be done in many ways:

1. The simplest way is to represent it as a function where one can choose to either set a fixed number of arguments or set all arguments to optional. The great disadvantage here is that we limit the user's opportunities to define flexible and complex functions.

2. Another possibility is to represent a function is one which takes a fixed number of arguments and a number of optional arguments. This representation allows for more flexible and complex functions but we are still forcing the user to a certain number of arguments (both fixed and optional).

3. The third possibility is to represent it as above just with an unbounded number of optional arguments. This is similar to the ellipsis notation used in C++.

In this prototype we choose to represent the optional arguments as a cell area, thus limiting the number of optional arguments. The fact that the cell area is limited does not violate or debase any of the design concepts since standard Excel functions have a limited number of arguments (a high limit of $\sim$30 though). Alternatively the user can use an entire column for the optional arguments (this is also a cell area).

To implement this, the signature will now consist of the list of cell addresses (with individual names) found in prototype 1 followed by an optional cell area (with one name) defining the location of the function arguments. The optional argument name will be postfixed with a number to indicate the number of optional arguments.

When a function is invoked the number of arguments is known and we apply these to the CalculationContext. Unused arguments will have null values, which will be converted to zero or empty strings when used in formulas. This is standard behavior of spreadsheets when no value has been entered into a cell.

Applying the function sheet to the arguments is straightforward. However, when filling the arguments into the cell area one must either go by rows first or by columns first. This is a design decision since neither way is favored by the implementation nor has any effect on any of the design concepts. One could let the decision be up to the user but this will introduce additional complexity and affect the abstraction gradient concept. After these considerations we merely took the decision to go by rows first (rows from left to right and columns from top to bottom) like reading a book (in a western language, that is).

6.4 Referring outside the function sheet

Referring to other sheets from inside a function call should be straightforward and the user would expect it to work just like any other reference to other sheets. But what about referring to another function sheet without a function call? To get predictable results we need some kind of encapsulation.

A reasonable restriction will be to disallow other sheets to refer inside (another) function invocation, so reference to function sheets are only made through function calls. This would allow us to treat cell values inside the function sheet as local variables. Referring to the function's public variables (the default context) could be allowed - this would also treat Worksheet and FunctionSheet alike (no special implementation needed).

The CalculationContext implementation naturally supports referring outside the function's sheet under the above assumptions. No additional changes to the implementation are required to support this.

Notice that our naming scheme for importing from Excel makes it naturally hard to refer to cells inside a function sheet because the prefix '@' is removed from the name during import.

6.5 Case: Risk premium calculation on a waiver of premium

In insurance companies and pension funds the insured usually buys a product called waiver of premium in combination with other risk products (e.g. death insurance). A waiver of premium is a product where the insurance company pays the premium for the insured if the insured gets disabled.

To calculate the price the insured has to pay for this product (called risk premium) we need several pieces of information:

- The age of the insured, x.
- The insured's age at retirement, n.
- The annual premium paid by the insured, π.
- The insured's occupation, e.
- The insured's deferred period, k.

The monthly risk premium of a waiver of premium is calculated by this formula:

$$RP_x = \frac{1}{12} \cdot (f \cdot \mu_x \cdot S_x),$$

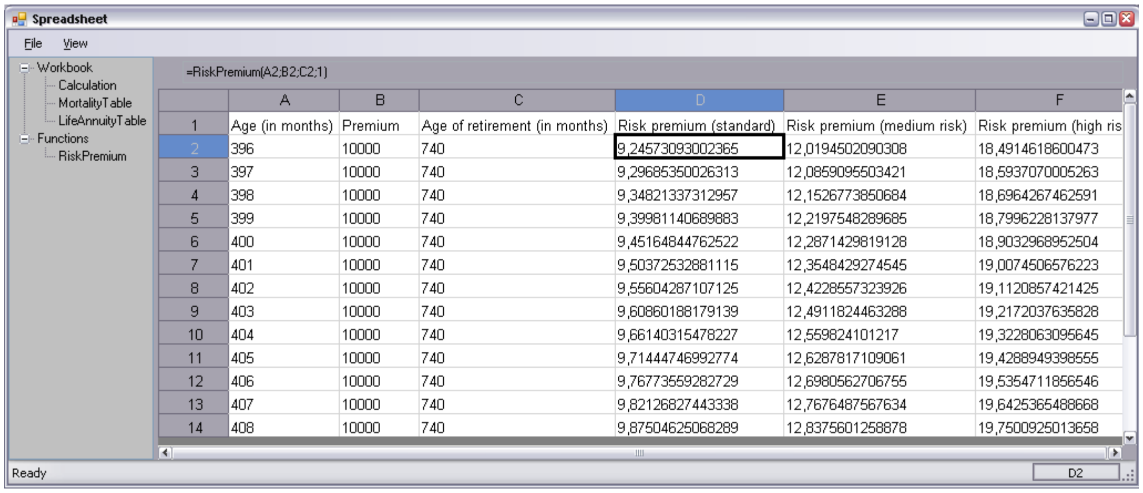
where

- μ_x is the disability intensity at age x (the probability that the insured gets disabled at age $x + dt$ given that his/hers age is x),

- S_x is the sum at risk at age x and is calculated as $S_x = \pi \cdot \bar{a}_{x:\bar{n}|}$ where $\bar{a}_{x:\bar{n}|}$ is the life-dependent premium annuity found by a table lookup, and
- f is a factor measuring the risk regarding occupation and/or deferred period and is calculated as $f = p + e + k$, where p is a parameter covering different variants of the product (for simplicity we set $p = 1$ corresponding to a normal product), e is a parameter covering the risk of occupation and k is a parameter covering the risk of the length of the deferred period.

Some insurance companies define three groups of insured, a standard, a medium and a high risk group. The only difference (ignoring health rating for simplicity) is the factor f. For an insured in the standard risk group, e and k is not used; for an insured in the medium risk group, either e or k are used; and for an insured in the high risk group, both e and k are used. How to calculate the e's and k's are complex and are therefore omitted in this example. In other words we have two optional arguments, e and k. Let us try to make a user-defined function that calculates the risk premium and handles the optional arguments.

To begin with we would like a spreadsheet with a worksheet called 'Calculation' and two worksheets used for table lookups, 'MortalityTable' and 'LifeAnnuityTable'. Furthermore we of course want a function sheet with our user-defined function. See figure 9.



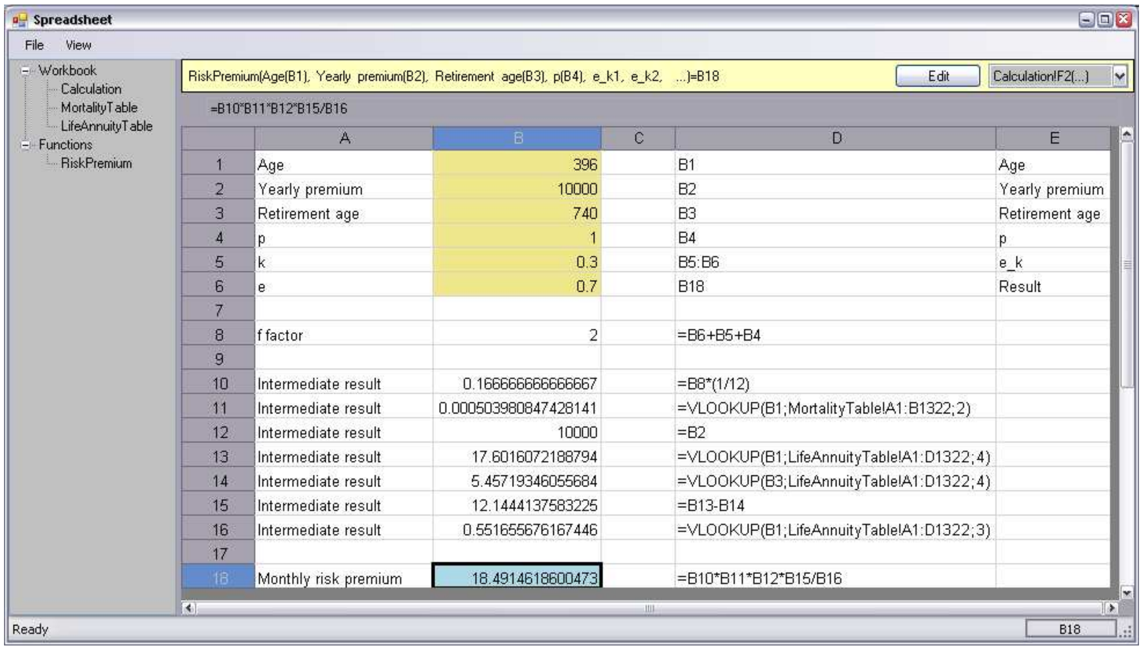
=RiskPremium(A2;B2;C2;1)

	A	B	C	D	E	F
1	Age (in months)	Premium	Age of retirement (in months)	Risk premium (standard)	Risk premium (medium risk)	Risk premium (high ris
2	396	10000	740	9,24573093002365	12,0194502090308	18,4914618600473
3	397	10000	740	9,29685350026313	12,0859095503421	18,5937070005263
4	398	10000	740	9,34821337312957	12,1526773850684	18,6964267462591
5	399	10000	740	9,39981140689883	12,2197548289685	18,7996228137977
6	400	10000	740	9,45164844762522	12,2871429819128	18,9032968952504
7	401	10000	740	9,50372532881115	12,3548429274545	19,0074506576223
8	402	10000	740	9,55604287107125	12,4228557323926	19,1120857421425
9	403	10000	740	9,60860188179139	12,4911824463288	19,2172037635828
10	404	10000	740	9,66140315478227	12,559824101217	19,3228063095645
11	405	10000	740	9,71444746992774	12,6287817109061	19,4288949398555
12	406	10000	740	9,76773559282729	12,6980562706755	19,5354711856546
13	407	10000	740	9,82126827443338	12,7676487567634	19,6425365488668
14	408	10000	740	9,87504625068289	12,8375601258878	19,7500925013658

Figure 9: The worksheet 'Calculation'

Notice that the formula in the standard risk group is calling the RiskPremium function with only 4 arguments, while the formulas for the medium and high risk groups are calling the RiskPremium function with respectively 5 and 6 arguments, namely the e's and k's.

The user-defined function is easily set up. The only thing to remember is that the optional arguments must be represented as a cell area in the function signature. This is done in cell D5. See figure 10.



RiskPremium(Age(B1), Yearly premium(B2), Retirement age(B3), p(B4), e_k1, e_k2, ...)=B18

=B10*B11*B12*B15/B16

	A	B	C	D	E
1	Age	396		B1	Age
2	Yearly premium	10000		B2	Yearly premium
3	Retirement age	740		B3	Retirement age
4	p	1		B4	p
5	k	0.3		B5:B6	e_k
6	e	0.7		B18	Result
7					
8	f factor	2		=B6+B5+B4	
9					
10	Intermediate result	0.166666666666667		=B8*(1/12)	
11	Intermediate result	0.000503980847428141		=VLOOKUP(B1;MortalityTable!A1:B1322;2)	
12	Intermediate result	10000		=B2	
13	Intermediate result	17.6016072188794		=VLOOKUP(B1;LifeAnnuityTable!A1:D1322;4)	
14	Intermediate result	5.45719346055684		=VLOOKUP(B3;LifeAnnuityTable!A1:D1322;4)	
15	Intermediate result	12.1444137583225		=B13-B14	
16	Intermediate result	0.551655676167446		=VLOOKUP(B1;LifeAnnuityTable!A1:D1322;3)	
17					
18	Monthly risk premium	18.4914618600473		=B10*B11*B12*B15/B16	

Figure 10: A user-defined function that calculates risk premiums with optional arguments.

All in all we were easily able to create a spreadsheet that calculates risk premiums for the waiver of premium product for all three risk groups using the same function. With the use of user-defined functions we have encapsulated the important part of the calculation in a function and thereby avoided repetition, decreased the number of possible errors during maintenance, saved worksheet space and made it applicable for reuse. The spreadsheet could have been made without the use of optional arguments but it would not have been as nice and simple as this spreadsheet.

7 Prototype 3: Matrix functions

In this prototype we expand our second prototype to handle matrices in user-defined functions.

Our primary objectives are:

1. Design a structure that handles matrix values.
2. Implementation of user-defined matrix functions where both the argument and the result is a matrix.

As mentioned in [JBB03] the user will (according to the consistency concept) expect the user-defined functions to be able to handle matrices just as Excel does in several built-in functions (e.g. SUM) via the use of cell areas. It is therefore natural to extend the user-defined functions to include matrix functions.

7.1 Designing a matrix structure

Before deciding the structure and how to represent matrices in our user-defined functions, we have to consider the use of matrices in spreadsheets in a more general context.

Matrices are primarily used for handling large amount of data or calculations as regression analysis but they can also be used as a tool to outline results or statistics, e.g. sales figures. In Excel one often names cell areas or use several worksheets to obtain these matrix capabilities. The disadvantage in Excel when using the spreadsheet with matrices represented as cell areas is that you quickly end up with many (and often large) worksheets, making the spreadsheet difficult to handle.

When considering the user-defined functions some further problems must be addressed:

- If a user-defined function is given a matrix as argument how do we describe the function signature? We do not want to specify the size of the matrix as an argument.
- Even though the function sheet is limited in size (i.e. 255 columns), a matrix can be arbitrarily large.

In the next two subsections we will present two possible designs of a structure that handles matrix arguments and thereby solve the above problems.

7.1.1 Expansion of matrices

A possible solution is to let the cell area representing the arguments expand to be able to include the given matrix. By doing this one could have several invocations with different sizes of cell areas as arguments to the function.

At first the solution seems simple and consistent with the design concepts but there are some problems that need to be dealt with:

- A function sheet will definitely contain formula cells but how must these formulas react to the expansion of the argument area? Must they be copied or just moved? We could let the user decide via a right click menu or we could base the decision on the type of formula or a property of the cell containing the formula. We could end up writing formulas to find out which formulas to copy and which ones not to, a kind of conditional format copy.

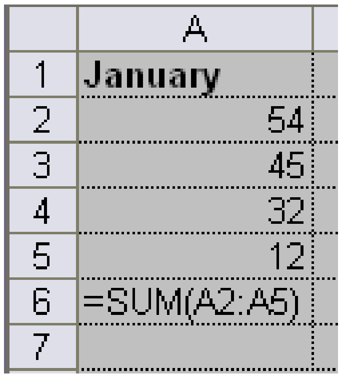
  Solving this problem will most likely result in overruling several of the design concepts, e.g. the abstraction gradient, the consistency, the error-proneness and the hidden dependencies concept.

- How do we handle multiple matrix arguments? We have only one function sheet so we have to expand the arguments in the same sheet but how and where? This will require a set of strict expansion rules that will conflict with e.g. the abstraction gradient and the consistency concept.

To illustrate the difficulties above we give an example.

Example

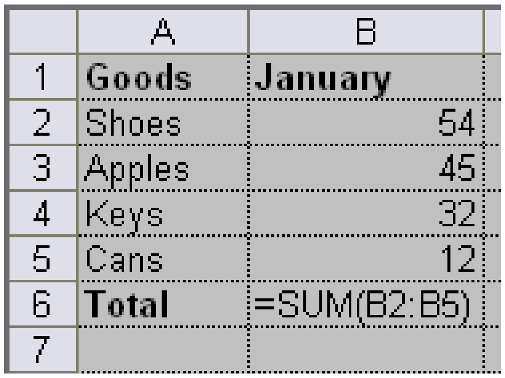
Let us consider a very simple function to calculate column sums of a matrix (consisting of a single column, or vector, in this case). We want to expand the argument area to a matrix which contains integers, e.g. number of goods.



	A
1	**January**
2	54
3	45
4	32
5	12
6	=SUM(A2:A5)
7	

Figure 11: Matrix containing integers.

Here it is obvious that the function must handle a matrix with two columns by copying the formula cell to the 'new' column. But what if the matrix not only contains integers but also string values.



	A	B
1	**Goods**	**January**
2	Shoes	54
3	Apples	45
4	Keys	32
5	Cans	12
6	**Total**	=SUM(B2:B5)
7		

Figure 12: Matrix containing both integers and string values.

Now we somehow need to define that the formula must be copied from the second column and not from the first in the case where we are dealing with a matrix with more than one column of numbers.

7.1.2 Matrices as first-class values

[JBB03] proposes another solution for handling a matrix argument in the user-defined functions. By letting the input matrix live in a single cell and treating matrices as first-class values (of a new run-time type 'matrix'), the major problems from section 7.1.1 are automatically dealt with. In doing this [JBB03] defines four ground rules:

- Any formula can have a matrix as its value.

- Matrices are two-dimensional.
- A vector is just a special case of a matrix.
- A scalar is implicitly promoted to be a 1x1 matrix in any context where a matrix is required.

These rules are obviously needed to respect the consistency and error proneness concepts.

By making matrices first-class values in line with integer, string, etc. the user will most likely expect some operations to follow naturally, e.g.:

- Obtain a specific element, row or column from a matrix.
- Add a column/row to a matrix.
- Obtain the number of rows/columns of a matrix.
- Make the transpose of a matrix.
- Multiplying matrices.
- Inverting matrices.

These must be implemented to respect the consistency and role expressiveness concepts.

The problems get more serious if we consider matrices containing other matrices. How shall a function or operation handle this situation? The function SUM can address the problem by applying SUM on the inner matrices before summing the outer matrix, but what if SUM is applied to two or more matrices, e.g. SUM(A1:A3) where there is a matrix in all of A1, A2 and A3? Should SUM operate like + and thereby result in an error if the matrices are of different dimensions or should it be redefined as a special matrix SUM?

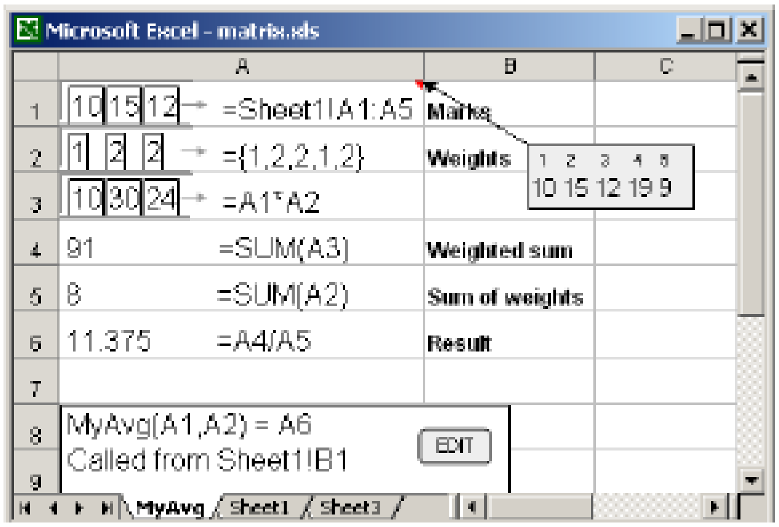
It is impossible to give a general answer, covering all functions and operations, to these questions. Depending on the function and operation one could think of several ways to deal with the problems. It could be an iterative/recursive approach or a top down approach, but one could also only consider the outer level and ignore the lower levels (this would be the preferred solution with the transpose function). The conclusion is that one has to go through every function/operation to find the optimal solution for that specific function/operation.

To use matrices to the desired extent, the user needs to be able to define and use functions on matrices that operate on each value (or specific values) of the matrices, that is, a map function on a matrix. This requires the presence of higher order functions and recursion which we will focus on the prototype 4, see section 8.

7.1.3 The matrix design

When considering the pros and cons of the two structures above we agree with [JBB03] that defining matrices as first-class objects (section 7.1.2) is the best way. By choosing this structure the problem regarding matrices as result values are automatically solved. The user-defined function just returns a matrix as a matrix type.

The structures do not automatically determine the visual representation of matrices. [JBB03] proposes to show a cell that contains a matrix as a visual queue of its first few values. Furthermore the user can hover the mouse over a cell and hereby bring up a scrollable floating panel that shows the entire matrix. See figure 13.


Microsoft Excel - matrix.xls

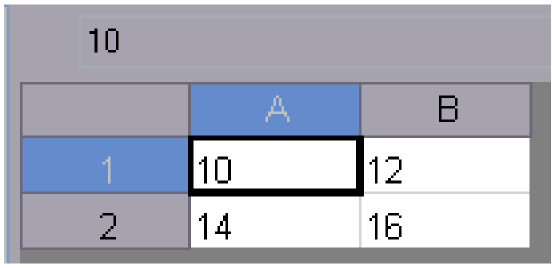
	A		B	C
1	10 15 12	=Sheet1!A1:A5	Marks	
2	1 2 2	={1,2,2,1,2}	Weights	
3	10 30 24	=A1*A2		
4	91	=SUM(A3)	Weighted sum	
5	8	=SUM(A2)	Sum of weights	
6	11.375	=A4/A5	Result	
7				
8	MyAvg(A1,A2) = A6			
9	Called from Sheet1!B1			

1	2	3	4	5
10	15	12	19	9

Figure 13: [JBB03]'s visual representation of matrices ([JBB03]'s figure 4)

This way to visually represent matrices might work on small matrices but will be unwieldy for larger matrices. As mentioned in section 7.1 matrices are primarily used for handling large amount of data. [JBB03] responds to this fact by allowing matrix values to be spread over a range of cells. But seen from our point of view this will only make things worse and one could end up with a visual chaos of matrices. All in all we think that the solution conflicts heavily with the visibility and juxtaposability concept.

Instead of trying to show all in the same window/sheet we just show what can be shown in the cell. To accommodate the visibility and juxtaposability concept and the premature commitment concept we add the feature to show a matrix in a separate window/sheet by pressing F12 when focus is on a cell containing a matrix. In figure 14 we have pressed F12 while focus was on a cell containing the matrix [10 12; 14 16].


10

	A	B
1	10	12
2	14	16

Figure 14: Matrix shown in a separate window/sheet.

It might conflict with the consistency concept to add a feature that is not already known by the users but we think it is the best way to do it.

7.2 Table Evolution Calculus

A paper by Martin Erwig, Robin Abraham, Steve Kollmansberger and Irene Cooperstein [EACK06] provides a possible alternative approach. It attempts to create a generator for correct spreadsheets, but also touches on the subject of matrix expansion. The table evolution calculus is a formal way to expand a cell area, referred to as a table. The template consists of a header, footer and an expandable group as shown in figure 15.

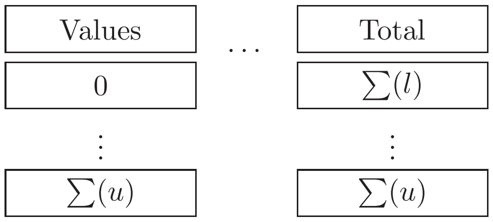


Figure 15: Horizontally expandable group template

These groups can be expanded horizontally and vertically and are referred to as hex groups and vex groups respectively. The footer could be a summation of an expanded group, and thus the notation for the formulas in the template are made relative to the group, e.g. $\sum(u)$ means $\sum(\text{"}up\text{"})$ where up refers the group above. Similarly the notation $\sum(l)$ or $\sum(\text{"}left\text{"})$ is using vex groups. An exponent k can be attached to k-fold repeated references, e.g. u^2 means two cells above. See the notation in the example in figure 16 where multiple vex groups are shown. The $\Delta(u^5, u^2)$ represents the difference between the sum of the income (u^5) and the sum of the sum of the expenses (u^2). This notation holds for different number of entries for incomes and expenses.

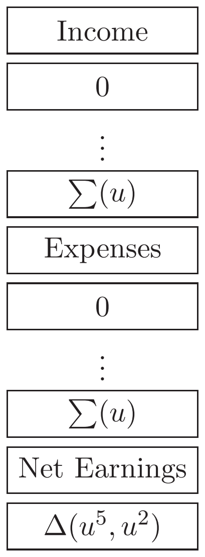


Figure 16: Template containing multiple vex groups

The template can contain any number of hex groups, and vex groups can be nested inside hex groups. To keep the expansion in one dimension, nesting is limited to having vex groups inside hex groups.

The template approach has the advantage of being a generator program and hence has its own user interface. Using the template approach with user-defined functions will mix the template interface with a normal spreadsheet interface, thus threatening the consistency concept adversely. Also, with this new notation for referencing groups we would mix two kinds of notation and thereby disregard the abstraction gradient concept. For our implementation we will explore this calculus no further and continue with the decision to use matrices as first-class values, but it is truly something to consider if one wants to implement some kind of expansion of matrix values.

7.3 Implementation of matrix user-defined functions

To implement the design structure found in the previous section we define a new 'matrix' type, raising matrices to first-class values. The 'matrix' type is called MatrixValue and is implemented as a 2-dimensional array. In the spreadsheet, matrices are indicated with the use of [ ]-brackets, e.g. [10 12; 14 16] represents a 2x2 matrix with first row containing 10 and 12 and with first column containing 10 and 14 (see figure 14).

In the following we have listed the matrix operations we have implemented.

MLookup: Obtains a specific value from a matrix M at a given row and column (this function is much like the build in function but work in absolute indices with the matrix).

CBIND and RBIND: Adds a column/row to a matrix. These are implemented rather inefficiently since all values from the original matrix are copied to a new array and a MatrixValue is returned. The running time depends linearly on the number of cells in the matrix.

CDIM and RDIM: Obtains the number of rows/columns of a matrix.

TRANSPOSE: Makes the transpose of a matrix. Implemented very much like CBIND and RBIND and has a running time of O(cols · rows). One could have made a solution where the MatrixValue class kept track of the row and column order of the values. Then it would only be a matter of changing this order when making the transpose and give a running time of O(1).

MMULT: Multiplying matrices. Again we have a rather inefficiently implementation, but this time it is because the values in MatrixValue are located in a 2-dimensional array of Values and must be cast to NumberValue to retrieve the values from it. It would be an optimization to raise matrices to 'NumberMatrixValue' when they should be used to mathematics (see section 7.3.1). The running time is of the order $O(R_1 \cdot C_1 \cdot C_2)$.

Furthermore it is relatively easy to modify +, - and * to be able to handle matrices. We have chosen these functions as a proof of concept. A complete implementation would require a much longer list of matrix function.

7.3.1 Performance

As indicated we have not gone deeply into the performance issues and our implementation does not represent optimal performance, even for a spreadsheet representation. The main reason for this is the MatrixValue representation. We will take a look at our options.

A normal cell area in Excel can represent a matrix containing values of different types but many of the matrix functions require the types to be convertible to numbers. If we are strict about the types we could guarantee that the resulting matrix only contained number values. The strictness can be introduced by returning an error value if any value in the matrix cannot be converted into a number. This is an alternative solution to just returning error values for the corresponding indices.

With this guarantee we could reduce one performance hindrance from the implementation. This could be achieved by introducing a special NumberMatrixValue which is the result of many of the matrix functions we would build into the spreadsheet. When a function encounters a NumberMatrixValue it is clear that all values are numbers and that all easily can be represented by doubles, which leaves us with a representation of the mathematical matrix.

Thomas Iversen explores further performance optimization in his thesis [Ive06].

7.3.2 CTRL+SHIFT+ENTER in Excel

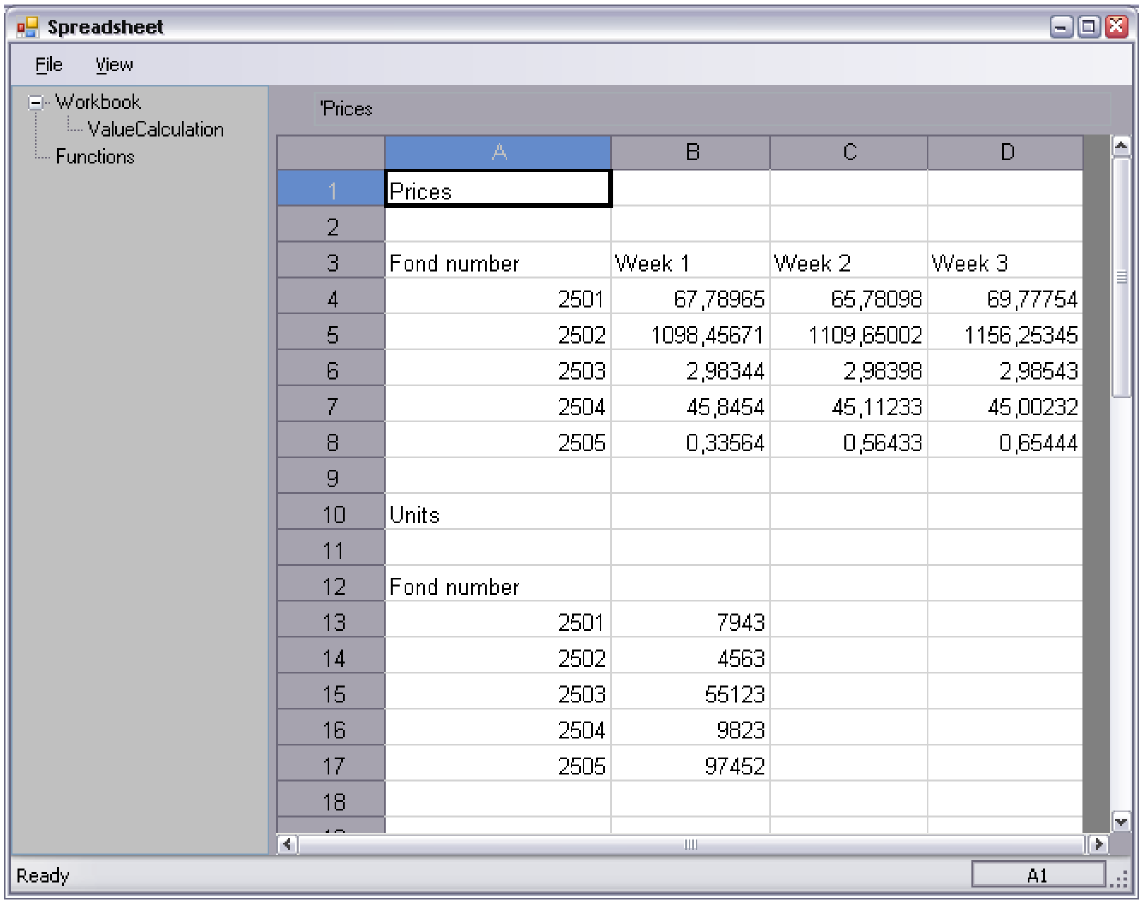
Some functions in Excel returns a matrix, for instance linear regression (LINEST in Excel). In order to see the entire result one has to enter the formula in a special way. A result cell area must be selected before the formula is entered, and upon completion of the formula the user must hit Ctrl+Shift+Enter instead of hitting the Enter key. Then Excel fits the result or part of it into the selected area. This approach should be extended in Excel for any user-defined functions that return matrices.

7.4 Case: Value calculation in Unit Link

In the recent an insurance type called Unit Link has become very popular. Opposed to traditional insurance Unit Link gives the insured the possibility to place his/hers savings in predefined funds. These funds could be e.g. Danish stocks, US stocks, Danish short-term bonds, European bonds, etc. On a pure Unit Link contract the insured has no guaranteed interest rate, meaning that the value of the insured's savings fluctuate with the prices of the funds the insured has chosen to invest in. In other words the insured's savings are placed in units of the funds he/she has chosen.

The insurance companies have to closely follow the development of both the total number of units in each fund, the price and the development of the total value. Seen from an actuary's point of view the interesting problem here is the development of the total value in the portfolio.

Let us consider a simple example where we want to follow the development of the total value for five funds in three weeks. Data consists of the prices each week and the total number of units per fund. See figure 17.



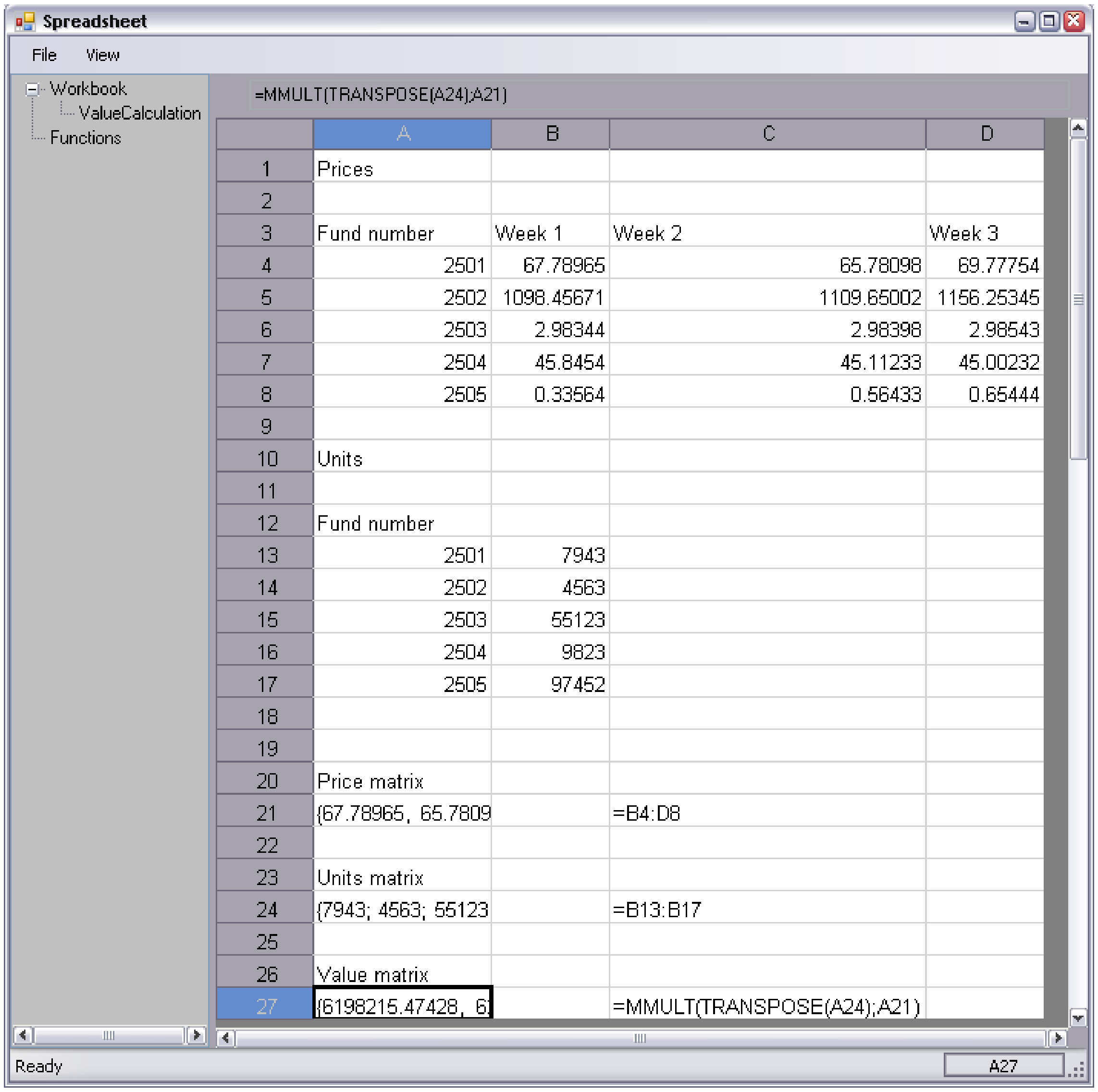
	A	B	C	D
1	Prices			
2				
3	Fond number	Week 1	Week 2	Week 3
4	2501	67,78965	65,78098	69,77754
5	2502	1098,45671	1109,65002	1156,25345
6	2503	2,98344	2,98398	2,98543
7	2504	45,8454	45,11233	45,00232
8	2505	0,33564	0,56433	0,65444
9				
10	Units			
11				
12	Fond number			
13	2501	7943		
14	2502	4563		
15	2503	55123		
16	2504	9823		
17	2505	97452		
18				

Figure 17: Prices and total number of units for each of the three weeks.

With the matrix features available we can calculate the total value each week. This is done by

1. Defining the two tables as matrices,
2. Transposing the units matrix and
3. Matrix multiplying the transposed units matrix with the price matrix.

This results in a 1x3-matrix of the total value each week. See figure 18.

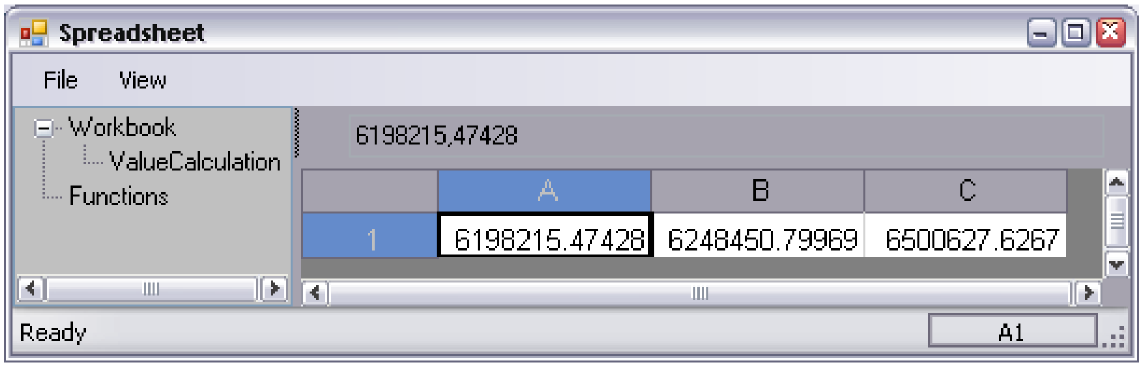


=MMULT(TRANSPOSE(A24);A21)

	A	B	C	D
1	Prices			
2				
3	Fund number	Week 1	Week 2	Week 3
4	2501	67.78965	65.78098	69.77754
5	2502	1098.45671	1109.65002	1156.25345
6	2503	2.98344	2.98398	2.98543
7	2504	45.8454	45.11233	45.00232
8	2505	0.33564	0.56433	0.65444
9				
10	Units			
11				
12	Fund number			
13	2501	7943		
14	2502	4563		
15	2503	55123		
16	2504	9823		
17	2505	97452		
18				
19				
20	Price matrix			
21	{67.78965, 65.7809		=B4:D8	
22				
23	Units matrix			
24	{7943; 4563; 55123		=B13:B17	
25				
26	Value matrix			
27	{6198215.47428, 6		=MMULT(TRANSPOSE(A24);A21)	

Figure 18: Value calculation in Unit Link.

By pressing $F12$ while having focus on the value matrix we get figure 19.



6198215,47428

	A	B	C
1	6198215.47428	6248450.79969	6500627.6267

Figure 19: Value calculation in Unit Link.

These calculations could of course have been done without the use of the matrix functionalities but it would not have been as nice and easy as with them. Besides from mathematical uses of matrices the introduction of matrix functions are not really impressive until we introduce higher-order function in the next chapter.

8 Prototype 4: Recursive and higher-order functions

In this prototype we expand our third prototype to handle recursive and higher-order user-defined functions.

Our primary objectives are:

1. Implementation of recursive functionality for the user-defined functions.
2. Design a structure that handles higher-order functions.
3. Implementation of higher-order user-defined functions.

As mentioned in section 3.2.2, more advanced users will expect and benefit from allowing user-defined functions to be recursive. Even though it is argued in [JBB03] not to allow recursive functions we find it natural to extend the user-defined functions to include recursive capabilities. See section 8.1.

In the case presented in prototype 3 (section 7.4) we realized that to be able to really use the potential of matrices in user-defined functions we needed higher-order functions. See section 8.2.

8.1 Recursive functions

As mentioned in section 2.1 a spreadsheet can be considered as an odd functional programming language. In functional languages, recursion is very common and thus a natural thing to explore in regard to spreadsheets. Recursive functions let the users define functions that invoke themselves, allowing the functions to be calculated over and over again.

Implementing recursive functions in our model is a relatively easy task. We have constructed the CalculationContext to represent the needed separation of data and formulas. Thus a function call to the function itself is no different from a call to any other function - except for the possibility of infinite recursion. To guard against infinite recursion we have two options:

1. We can wait for the runtime system (CLR) to throw a stack overflow exception. This will eventually happen, but is not a nice way to handle infinite recursion and it is very difficult to recover nicely from.
2. Alternatively we add a stack count to the CalculationContext which is incremented upon each call. If the counter exceeds a certain threshold we stop calculations. Microsoft Excel already has a threshold for iterations which we could reuse.

> If we increment the stack count for all calls to user-defined function we can prevent infinite calls that also involves other functions, e.g. function f calls function g which in turn calls function f.

In [JBB03] it is argued that recursive functions should not be implemented because of the possible confusion that arises from the representation of the calls and thereby threatens the consistency concept. If you decide to use arrows between sheets to represent the order of calls, a recursive call is either represented with an arrow back to the same sheet or a new representation of the same sheet. This can very easily create a blur of information on the screen.

Instead we propose that a call stack approach is used, which is more like traditional debuggers. When inspecting an instance of the function call we propose that the user is allowed to see the different levels of the recursion and jump to specific levels by replacing the current instance or opening a new instance. This solution is not perfect in regard to the consistency concept but it is simpler and visually far better than the linked-worksheet solution. We think, unlike [JBB03], that there is significant benefit for the users by having recursive user-defined functions available, but as discussed in section 3.2.1 we aim for a broader target audience.

8.2 Higher-order functions

To use the matrix functionality for something more than data collection one needs more advanced functions than just the trivial matrix functions. More specifically one needs well known functions as MAP and FOLD. The problem in prototype 3 was that both these functions took a function as argument and we therefore needed higher-order functions which we could not handle at that time. To make it up for that (among other things of course) we will in this section implement higher-order user-defined functions.

A higher-order function satisfies at least one of the following two criteria

1. The function takes one or more functions as arguments.
2. The function returns a function as the result.

To support these criteria we need to introduce another type of value.

8.2.1 Introducing a new type

The first step toward supporting higher-order functions in our framework is to introduce the function as a value. We introduce a new type, FunctionRefValue, which derives from

Value. The internal value of FunctionRefValue is a reference to an internal function or a user-defined function, represented by the type Function as shown in listing 5.

```
class FunctionRefValue : Value
{
  private Function function;
  ...
}
```

Listing 5: Partial class definition of FunctionRefValue

The simplest formula which returns a FunctionRefValue is of the form

```
=AgeFromDate
```

which returns a reference to the user-defined function AgeFromDate. We just have to decide how the reference is presented to the user.

Representing just the function name will confuse function references with normal text and representing it with something more in Excel style, like #FUNCTION(AgeFromDate), might lead to the user confusing it with errors (like #VALUE! or #NAME?). Alternatively a simple symbol (f_x:AgeFromDate) might suffice. The f_x symbol is already used in Excel as an icon bringing up the insert function menu so we are not introducing something completely new to the user and thereby maintaining the consistency concept.

In the case of AgeFromDate we know the name of the function that we refer to, but as we shall see later some functions create new functions as their result. These functions are anonymous functions which we represent with just #FUNCTION or f_x. Most functional languages do not display anonymous functions either. The SML notation would give us something on the form *val a = fn: int → bool*, where the variable a corresponds to a cell reference in our model. One could try to give more clever names to anonymous functions but we have not explored this any further.

8.2.2 Built-in higher-order functions

We propose a set of built-in higher-order functions inspired by many of the functional languages around.

MAP f m: Applies f to each cell in the matrix m, from left to right and top to bottom, and returns the matrix of f's results.

FOLDL f x m: Applies f to x and the first element in m, then applies f to the result and the second element, and so on.

FOLDR f x m: Applies f to x and the last element in m, then applies f to the second last element and the result, and so on.

These functions are inspired by their SML counterparts but are extended to use matrices instead of arrays. The array concept is not really present in spreadsheets, but we treat the matrix m as an array by concatenating rows top to bottom. However, when using matrices we have several other options to expand the number of functions available:

MAPROW f m: Applies f to each row in the matrix m and returns the one-dimensional matrix of f's results. The function f must take a number of arguments equal to the number of columns in m.

MAPCOL f m: Applies f to each column in the matrix m and returns the one-dimensional matrix of f's results. The function f must take a number of arguments equal to the number of rows in m.

For instance, using MAPROW(SUM, A1:D10) would give a 1×10 matrix with the sum of the rows. Any function taking 4 arguments and any function with variable number of arguments can be used instead of SUM.

Likewise we can define variants of functions like FOLDL and FOLDR.

8.3 Partial application and partial evaluation

One of the uses of higher-order functions is the use of partial application. We want to be able to bind arguments to a function to create a new function, but before we go into the details we will find a notation that is intuitive for spreadsheet users. In some functional languages (e.g. SML) we could have something like this, e.g.

```
fun add x y = x + y
fun add3 y => add 3 y
```

Here we bind the argument 3 to x in the function add to create the function add3. This notation is very unlike all other Excel notation and spreadsheet notation in general and would spoil the consistency concept. Instead we use a function like this, e.g.

```
BIND(function, arguments...)
```

In the example above we should write BIND(Add, 3) to create an anonymous function in the cell where we place this formula. This approach is more spreadsheet like, but has one deficiency. We cannot decide the order in which to bind the arguments. The functional language notation gives us a way to decide which arguments to bind, e.g.

```
fun minus x y = x - y
fun minus3 x = minus x 3
```

In this example it is very important to bind the y argument and not the x argument, or we will change the semantics of the function.

Several options are available:

BIND($f, x_1, \ldots, x_n$): Binds the first n arguments, where n depends on the number of arguments supplied in the call to BIND.

BINDL, BINDR: Like BIND but where BINDL binds the first n arguments and BINDR binds the last n arguments.

BIND1, BIND2, BIND3, BINDN: Binds the first, second, third or n'th argument, where n would be an argument in the call to BINDN. BIND1 would equal BIND above.

BIND + SWAP: Use SWAP to swap the order of the arguments and then BIND the first argument n arguments.

We have created a very simple implementation of BIND in our model. Assume that BIND binds the first n arguments like the first option above.

1. A new function is created that has a reference to the bound function and an array containing the value of the first n arguments.
2. We evaluate the arguments when the BIND function is called, thus the first n arguments are only evaluated once.
3. When the new function is invoked the bound parameters are prefixed to the arguments used in invocation.
4. The bound function is invoked with the new combined set of arguments and the result is returned.

Listing 6 shows an implementation of the above procedure.

```
class BindFunctionValue : FunctionRefValue
{
  public BindFunctionValue(Function boundFunction, Expr[] bEs)
    : base(
      new Function(
    delegate(CalculationContext context, Expr[] es, int col, int row)
    {
      Expr[] cEs = new Expr[bEs.Length + es.Length]; // combine arguments
      bEs.CopyTo(cEs, 0);                            // bound arguments first
      es.CopyTo(cEs, bEs.Length);
      return boundFunction.applier(context, cEs, col, row);
    }))
  {
  }
}
```

Listing 6: Class definition of BindFunctionValue

For consistency the BIND function always returns a function as its value, even though all parameters have been bound. In case all parameters have been bound, the result is a function that takes no arguments.

As discussed earlier we had to find away to name anonymous functions, but we also has to have a way to reference anonymous function. A call to BIND returns an anonymous function but this implies that it does not have a name. Say we have bound the birthday in our AgeFromDate function from earlier:

```
=BIND(AgeFromDate, 1974-08-24)
```

If we want to calculate the age at different points in time then we need a way to reference the result of the above call to BIND.

Two situations come to mind in regard to where we want to call an anonymous function

1. The anonymous function is in a cell in the form of a FunctionRefValue. This is the situation in the example above.
2. The anonymous function needs be to evaluated in the formula in which it was created. This is especially important when using SWAP (which returns a function) with BIND. The user will most likely want to use the SWAP in the argument to BIND.

We cannot use the cell address as the name of the function because it would confuse the user in the first situation (but it does make sense in the second).

Instead a function CALL is created which can evaluate any function if the appropriate number of arguments is supplied. The CALL function is the equivalent of calling the function directly, but supports function references in FunctionRefValue - especially the ones created by BIND.

8.4 Optimizations

One point in using partial evaluation is to optimize the calculations, e.g. the calculation of the n'th root (NROOT(x,n)) can be optimized into the square root when $n = 2$ for all values of x. When dealing with user-defined functions we have a special opportunity to optimize the calculations because we have 'the source code' of the function.

To handle this optimization we first have to be a little more concrete about how the BIND function works. The description above uses partial application where we construct a closure with the function and the bound arguments, e.g. (Add, x=3), and we do not do any evaluation based on x. With partial evaluation one would expect evaluations based on x.

In our model evaluation is handled by the CalculationContext, so to introduce partial evaluation in BIND we would have to evaluate cells based on the bound arguments and store the results in a special CalculationContext. This context is then used as a starting point for evaluations instead of an empty list.

One relatively simple way to obtain a context with evaluated arguments would be to evaluate the function with a special value for unbound arguments. The resulting CalculationContext would contain the special unbound value for all results depending on arguments other than the bound arguments. There is (at least) one exception: When using non-deterministic functions, such as RAND, the stored result will not change and the behavior is different from normal invocation of the function - this may or may not be what the user expects. This method implies a certain overhead when using BIND, so it is efficient only when there are heavy calculations depending on the bound argument only. The actual implementation would have to decide when to use partial evaluation, but other optimizations could apply if we treat the formulas as a functional language - we will not explore these optimizations.

8.5 Case: Calculating passives

In actuarial work one often need to calculate the unit price of a pension (called a passive) given a current age and the age of retirement. Depending on the type of product this can involve very complex calculations and therefore the values are often tabulated. Plenty of spreadsheets in life insurance companies and pension funds involve the tables and a number of complex formulas to find the right passive in the table.

We can lessen the complexity with our user-defined functions. We start by defining a function calculating the passive:

```
PASSIVE(AgeOfRetirement, CurrentAge)
```

The PASSIVE function is a user-defined function which handles the table lookup using built-in lookup functions and not higher-order functions.

Companies have several products and therefore several tables of passives. Let us create a function for each product and ignore that these products potentially have different parameterizations (e.g. we could introduce health rating parameters). Let us for simplicity only consider two products:

```
PASSIVE_PROD_A(AgeOfRetirement, CurrentAge)
PASSIVE_PROD_B(AgeOfRetirement, CurrentAge)
```

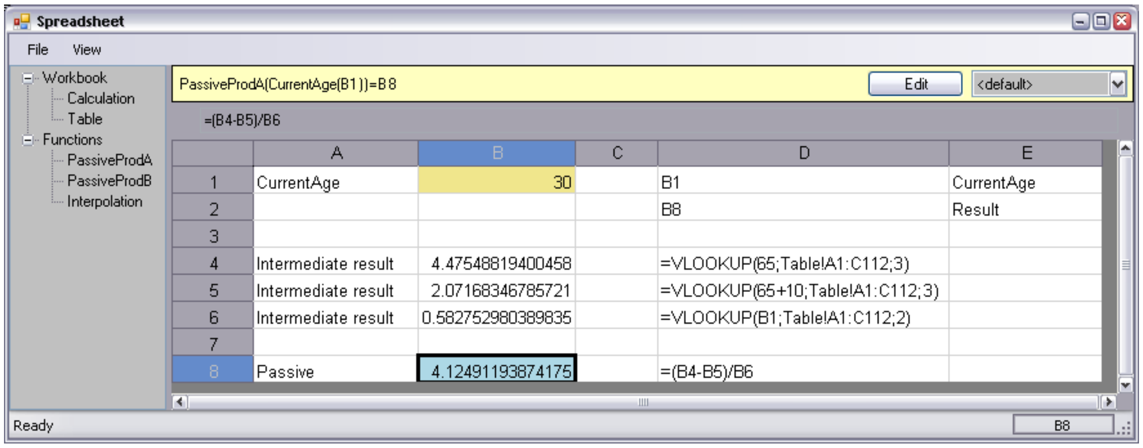
Product A could be benefits paid out for 10 years (called a deferred 10-year life annuity) and product B could be benefits paid outs for 15 years (called a deferred 15-year life annuity). The insured will select either product A or B but will have the same retirement age (say e.g. 65) on the two products, thus making the passive function dependent of only one variable - the current age.

```
PASSIVE_PROD_A_65 = BIND(PASSIVE_PROD_A, 65)
PASSIVE_PROD_B_65 = BIND(PASSIVE_PROD_B, 65)
```

This gives us

```
PASSIVE_PROD_A_65(CurrentAge)
PASSIVE_PROD_B_65(CurrentAge)
```

In figure 20 such a user-defined function is made.



PassiveProdA(CurrentAge(B1))=B8

=(B4-B5)/B6

	A	B	C	D	E
1	CurrentAge	30		B1	CurrentAge
2				B8	Result
3					
4	Intermediate result	4.47548819400458		=VLOOKUP(65;Table!A1:C112;3)	
5	Intermediate result	2.07168346785721		=VLOOKUP(65+10;Table!A1:C112;3)	
6	Intermediate result	0.582752980389835		=VLOOKUP(B1;Table!A1:C112;2)	
7					
8	Passive	4.12491193874175		=(B4-B5)/B6	

B8

Figure 20: A user-defined function that calculates yearly passives.

To make matters a little more complicated, the passive tables are usually defined on a yearly basis. The monthly passives are found using interpolation. Interpolation functions clutters the spreadsheet formulas we have seen until now, but once again user-defined functions comes to the rescue. This time, however, higher-order functions can be used.

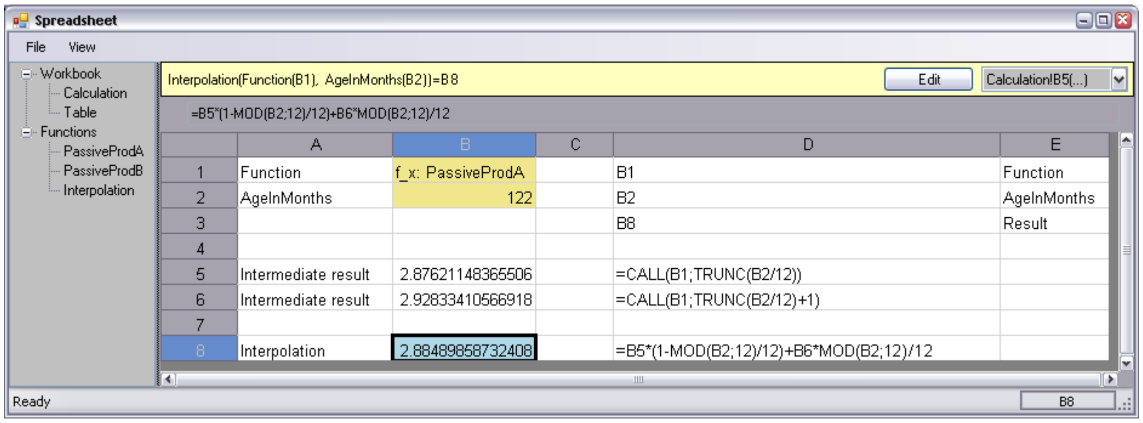
```
INTERPOLATION(Function, AgeInMonths)
```

The INTERPOLATION function uses CALL to calculate two values of the function using the age in years.

```
CALL(Function, INT(AgeInMonths / 12))
CALL(Function, INT(AgeInMonths / 12) + 1)
```

The result is the average of the two functions weighted with MOD(AgeInMonths, 12). This function can be used with different passive functions.

In figure 21 such a user-defined function is made.



Interpolation(Function(B1), AgeInMonths(B2))=B8

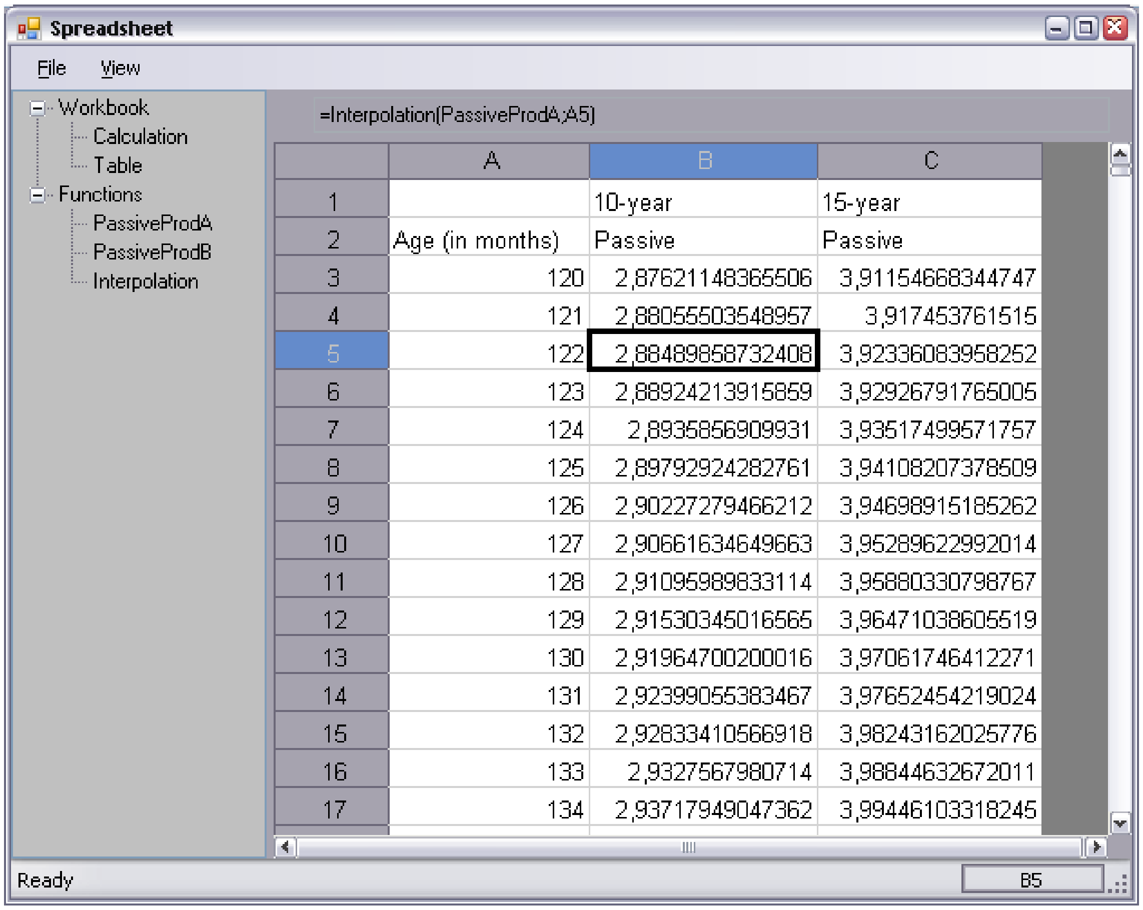
=B5*(1-MOD(B2;12)/12)+B6*MOD(B2;12)/12

	A	B	C	D	E
1	Function	f_x: PassiveProdA		B1	Function
2	AgeInMonths	122		B2	AgeInMonths
3				B8	Result
4					
5	Intermediate result	2.87621148365506		=CALL(B1;TRUNC(B2/12))	
6	Intermediate result	2.92833410566918		=CALL(B1;TRUNC(B2/12)+1)	
7					
8	Interpolation	2.88489858732408		=B5*(1-MOD(B2;12)/12)+B6*MOD(B2;12)/12	

Figure 21: A user-defined function that interpolate yearly passives to get monthly passives.

All in all we are able to calculate monthly passives with the use of our two user-defined functions. See figure 22.

Traditionally such spreadsheets are used by actuaries to verify the calculation system. We have seen many spreadsheets whose purposes are to calculate passives such as these. Either they are very large and complex or they are very simple but coded in e.g. VBA. With the user-defined functions we have created a spreadsheet that is easy to both understand and use.



=Interpolation(PassiveProdA;A5)

	A	B	C
1		10-year	15-year
2	Age (in months)	Passive	Passive
3	120	2,87621148365506	3,91154668344747
4	121	2,88055503548957	3,917453761515
5	122	2,88489858732408	3,92336083958252
6	123	2,88924213915859	3,92926791765005
7	124	2,8935856909931	3,93517499571757
8	125	2,89792924282761	3,94108207378509
9	126	2,90227279466212	3,94698915185262
10	127	2,90661634649663	3,95289622992014
11	128	2,91095989833114	3,95880330798767
12	129	2,91530345016565	3,96471038605519
13	130	2,91964700200016	3,97061746412271
14	131	2,92399055383467	3,97652454219024
15	132	2,92833410566918	3,98243162025776
16	133	2,9327567980714	3,98844632672011
17	134	2,93717949047362	3,99446103318245

Figure 22: Calculation of monthly passives.

9 Conclusion

By using nine design concepts from the usability and HCI world for their analytic approach in [JBB03], Blackwell, Burnett and Peyton Jones develop a theoretical basis for a design of user-defined functions in Excel. With this article as our theoretical basis and with our practical experience as actuaries, employed in companies where the use of spreadsheets is universal, as our practical basis, we have implemented user-defined functions in a spreadsheet.

9.1 The approach

Our methodological approach was an evolutionary implementation via iterative enhancements and we defined four prototypes that overall gave us a satisfying implementation of the user-defined functions. These prototypes included:

- Prototype 1: Simple function
  - Setting up a spreadsheet.
  - Import from other spreadsheets.
  - Simple function where we have N known arguments, the parameters are simple and the result is simple.
- Prototype 2: Advanced function
  - Simple function where we have a variable number of arguments and the number of arguments is known when the function is called.
  - Referring function. Function which refers to cells outside the function sheet.
- Prototype 3: Matrix function
  - Matrix function where both the argument and the result is a matrix.
- Prototype 4: Recursive and higher order function
  - Recursive function. Function that calls itself.
  - Higher order function. Function that has another function as argument.

Overall the method of evolutionary implementation via iterative enhancements turned out to be a very wise choice. It first of all gave us the possibility to see the user-defined functions work very early in the process but it also gave us the possibility of making later modifications and re-designs with relatively ease.

9.2 The prototypes

We successfully implemented what we aimed for in all four prototypes. But the challenge was not only to implement the four prototypes but also to do so while satisfying the nine design concepts described in [JBB03]. We believe that we have found technical solutions that satisfy the design concepts.

From our analysis and during the implementation of the prototypes we encountered some problems that we chose to solve differently than as proposed in [JBB03]. The most important of these were the function signature and the visual representation of the invocation list. In both cases we think we found more flexible solutions than proposed while still obeying the nine design concepts.

In prototype 4 (section 8) we implemented recursion and higher-order functionality into the user-defined functions. These two functionalities are not included in the proposed design in [JBB03]. The target audience we aimed for with these functionalities is the advanced spreadsheet users. This group will assume these functionalities to be available in the user-defined functions and adding them to the implementation did not spoil the user experience for the medium user. One can argue whether they spoiled any of the design concepts or not but we believe that even though some of the concepts may have suffered a bit recursion and higher-order functions are too essential not to include.

9.3 The cases

In all four prototypes we successfully used the user-defined functionalities on a case from 'the real world'. In comparison to the originals the spreadsheets were made simpler, easier to maintain, space were saved and the important parts were encapsulated in user-defined functions. The biggest effect (performance differences are not included) was on the more complex spreadsheets, which indicates that limiting the target audience to the 'medium' group might be wrong.

9.4 Test group

We have not carried out a full user test of our prototypes but we have shortly introduced the spreadsheet for some people with different backgrounds, mostly colleagues and friends of ours. The feedbacks were in general very positive.

The most noticeable was the feedback from the advanced Excel users (those with good knowledge of VBA), they quickly understood the concept and immediately came up with several ideas of how and where they could use it. The 'standard' group (more or less matching the target audience described in [JBB03]) needed more help to be able to understand the concept, but when this was achieved they were positive to the new ideas - even though they only had a few concrete ideas of what to use it for.

We also introduced the spreadsheet to novice Excel users, but with little outcome. They had difficulties understanding the concept and had trouble with the GUI.

All in all an interesting little test. Surely one needs to perform a full scale test to achieve any reliable results, but it indicates that the target audience might be a large proportion of the spreadsheet users.

9.5 Putting it all into perspective

With the outcome of the use for the user-defined functions in the cases and from our little user test we believe that the ideas presented in [JBB03] and this paper has a great potential, especially from the feedback from our colleagues in the pension and insurance business we think that many companies (in the financial sector) can benefit from the user-defined functions, not only seen from the single employees point of view but also from the company, e.g. encapsulation of critical business calculations in user-defined function libraries.

Since our goal was not to develop an end-user spreadsheet with user-defined functions available, one would have to look into, e.g. the GUI and the ’standard’ spreadsheet features to achieve that. Likewise, some of the functionality is implemented rather naively and inefficiently so one could look at the functionality performance.

Furthermore one could combine this work with [Ive06] that has some interesting thoughts regarding runtime code generation.

Appendix

A References, listings, figures and the project progress

References

[BT75] V.R. Basili and A.J. Turner. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering*, 1(4):390–396, 1975.

[EACK06] Martin Erwig, Robin Abraham, Irene Cooperstein, and Steve Kollmansberger. Gencel: A program generator for correct spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.

[Flo84] C. Floyd. A Systematic Look at Prototyping, Approaches to Prototyping. *Springer-Verlag: Heidelberg. Budde, R. and Kuhlenkamp, K. and Mathiassen, L. and Zullighoven, H.*, pages 1–17, 1984.

[Ive06] Thomas Iversen. Runtime code generation to speed up spreadsheet computations. *Master's thesis, DIKU, University of Copenhagen*, August 2006.

[JBB03] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A User-Centred Approach to Functions in Excel. *In ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, pages 165-176, New York, NY, USA, 2003. ACM Press.*, 2003.

[Mic] Microsoft Corporation. Office 2003 XML Reference Schemas. `http://www.microsoft.com/office/xml/default.mspx`.

[MWL] Hanspeter Mössenböck, Albrecht Wöss, and Markus Löberbauer. The Compiler Generator Coco/R. University of Linz. `http://www.ssw.uni-linz.ac.at/Coco/`.

[Ses05] Peter Sestoft. Spreadsheet notes. IT University of Copenhagen, 2005.

Listings

List of Figures

The project progress

In this section we will briefly describe the project progress.

Before the start of the project we made a project plan that included deadlines for each of the prototypes (implementation and description). These deadlines were:

- 1. March 2006
- 1. April 2006
- 8. May 2006
- 1. July 2006

The deadlines were almost reached every time and there was no need for any update of the project plan. The project plan also included our holidays etc.

The subject showed to be more and more interesting as we went along, but it also showed to be larger than first expected. We have also realized that the subject has a great potential and that Blackwell, Burnett and Peyton Jones has applied for a patent on their ideas in [JBB03].

It has not been necessary to change the approved project agreement.

All in all it has been seven tough months but we met our deadlines and are proud of the result.

B Selected source code

This section contains a selection of the source code for this paper. The presented code includes the core parts described in the paper, whereas many support classes and classes used for the visual representation are left out. Fell free to contact us for the complete sources.

Parts of the source code are produced by Peter Sestoft [Ses05].

License

ArgumentCell.cs

```
/// <summary>
/// Represents a function argument in a cell.
/// </summary>
class ArgumentCell : Cell
{
  private string name;
  private Cell defaultCell;
  private int index;

  public ArgumentCell(string name, int index, Cell defaultCell)
  {
    this.name = name;
    this.index = index;
    this.defaultCell = defaultCell;
  }

  public override Value Eval(CalculationContext context, int col, int row)
  {
    CellAddr ca = new CellAddr(col, row);
    Value v;
    if (context.TryGetValue(ca, out v))
      return v;

    if (context.IsFunctionInvocation)
      v = context.EvalArgument(index, col, row);
    else if (defaultCell != null)
      v = defaultCell.Eval(context, col, row);

    if (v != null)
      context[ca] = v;

    return null;
  }

  public override Cell MoveContents(int deltaCols, int deltaRows)
  {
    throw new NotImplementedException();
  }

  public override void InsertRows(Dictionary<Spreadsheet.AbstractSyntax.Expr,
      Adjusted<Spreadsheet.AbstractSyntax.Expr>> adjusted, Sheet modSheet,
      bool thisSheet, int aboveRow, int rows, int row)
  {
    throw new NotImplementedException();
  }

  public override string Show(int col, int row)
  {
    return "Argument: " + name;
  }

  public int Index
  {
    get { return index; }
  }
}
```

CalculationContext.cs

```
/// <summary>
/// Represents the calculation context.
/// </summary>
public sealed class CalculationContext
{
  private Sheet sheet;
  private string name;
  private Expr[] arguments;
  private CalculationContext invoker;
  private int stackDepth;

  private Dictionary<CellAddr, Value> values;

  public CalculationContext(Sheet sheet, string name)
  {
    this.sheet = sheet;
    this.name = name;
    this.values = new Dictionary<CellAddr, Value>();
  }

  public CalculationContext(Sheet sheet, string name,
    CalculationContext invoker, Expr[] arguments) : this(sheet, name)
  {
    this.invoker = invoker;
    this.arguments = arguments;
  }

  public bool HasValue(CellAddr ca)
  {
    return values.ContainsKey(ca);
  }

  public bool TryGetValue(CellAddr ca, out Value value)
  {
    return values.TryGetValue(ca, out value);
  }

  public void Reset()
  {
    values.Clear();
  }

  public Value EvalArgument(int index, int col, int row)
  {
    if (arguments != null && index < arguments.Length)
      return arguments[index].Eval(invoker, col, row);
    else
      return null;
  }

  public Value this[CellAddr ca]
  {
    get { return values[ca]; }
    set { values[ca] = value; }
  }

  public Sheet Sheet { get { return sheet; } }
  public string Name { get { return name; } }
  public int Count { get { return values.Count; } }

  public bool IsFunctionInvocation { get { return arguments != null; } }

  public int StackDepth
  {
    get { return stackDepth; }
    set { stackDepth = value; }
  }
}
```

Cell.cs

```
/// <summary>
/// A cell is a vay to obtain and show a value.
/// </summary>
public abstract class Cell
{
  public abstract Value Eval(
    CalculationContext invocationList, int col, int row);

  public abstract Cell MoveContents(int deltaCols, int deltaRows);

  public abstract void InsertRows(Dictionary<Expr, Adjusted<Expr>> adjusted,
    Sheet modSheet, bool thisSheet, int aboveRow, int rows, int row);

  /// <summary>
  /// Show computed value.
  /// </summary>
  public string ShowValue(CalculationContext invocation, int col, int row)
  {
    Value v = Eval(invocation, col, row);
    return v != null ? v.ToString() : "";
  }

  /// <summary>
  /// Show constant or formula or matrix formula.
  /// </summary>
  public abstract string Show(int col, int row);

  public static Cell Parse(string s, Workbook workbook, int col, int row)
  {
    Scanner scanner = new Scanner(MakeStream(s));
    Parser parser = new Parser(scanner);
    return parser.ParseCell(workbook, col, row);
  }

  public static Cell Create(Value value)
  {
    if (value == null)
      return null;
    NumberValue nv = value as NumberValue;
    if (nv != null)
      return new NumberCell(nv.value);
    DateTimeValue dtv = value as DateTimeValue;
    if (dtv != null)
      return new DateTimeCell(dtv.value);
    return new TextCell(value.ToString());
  }

  private 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 static Style GetStyle(Sheet sheet, int colIndex, int rowIndex)
  {
    Style style = sheet.Styles[colIndex, rowIndex];
    if (style == null)
      style = sheet.Workbook.Style;
    return style;
  }

  public static bool IsActive(Sheet sheet, int colIndex, int rowIndex)
  {
    return rowIndex == sheet.ActiveRow && colIndex == sheet.ActiveColumn;
  }
}
```

CellAddr.cs

```
/// <summary>
/// Represents an absolute, zero-based (col, row) cell adress.
/// </summary>
public struct CellAddr
{
  public static readonly CellAddr Invalid = new CellAddr(-1, -1);
  public static readonly CellAddr A1 = new CellAddr(0, 0);

  public readonly int col, row;

  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;
  }

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

  public override string ToString()
  {
    return Column.GetName(col) + (row + 1);
  }

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

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

CellArea.cs

```
/// <summary>
/// A CellArea expression is A1:C4 or $A$1:C4 or A1:$C4 or sheet1!A1:C4.
/// </summary>
class CellArea : Expr
{
  private readonly RARef ul, lr; // upper-left, lower-right
  private readonly Sheet sheet;

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

  /// <summary>
  /// Evaluate expression as if at cell address sheet[row, col].
  /// </summary>
  public override Value Eval(CalculationContext context, int col, int row)
  {
    if (this.sheet != null)
      context = sheet.Context;
    CellAddr ulCa = ul.Addr(col, row);
    CellAddr lrCa = lr.Addr(col, row);
    int cols = lrCa.col - ulCa.col + 1;
    int rows = lrCa.row - ulCa.row + 1;
    int col0 = ulCa.col;
    int row0 = ulCa.row;
    Value[,] values = new Value[cols, rows];
    for (int c = 0; c < cols; c++)
    {
      for (int r = 0; r < rows; r++)
      {
        Cell cell = context.Sheet.Cells[col0 + c, row0 + r];
        if (cell != null)
          values[c, r] = cell.Eval(context, col0 + c, row0 + r);
      }
    }
    return new MatrixValue(values);
  }

  public override Expr Move(int deltaCol, int deltaRow)
  {
    return new CellArea(sheet,
      ul.Move(deltaCol, deltaRow),
      lr.Move(deltaCol, deltaRow));
  }

  public override Adjusted<Expr> InsertRows(
    Sheet modSheet, bool thisSheet, int aboveRow, int rows, int row)
  {
    if (sheet == modSheet || sheet == null && thisSheet)
    {
      Adjusted<RARef> ulNew = ul.InsertRows(aboveRow, rows, row),
                      lrNew = lr.InsertRows(aboveRow, rows, row);
      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);
  }

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

  public CellAddr NextHorizontal(CellAddr ca)
  {
    int col = ca.col;
    int row = ca.row;

    col++;

    CellAddr calr = new CellAddr(lr, 0, 0);
    if (col > calr.col)
    {
      CellAddr caul = new CellAddr(ul, 0, 0);
      col = caul.col;
      row++;
      if (row > calr.row)
        return CellAddr.Invalid;
    }
    return new CellAddr(col, row);
  }

  public RARef UpperLeft { get { return ul; } }
  public RARef LowerRight { get { return lr; } }

  public bool IsSingleCell
  {
    get { return ul == lr; }
  }
  public override string ToString()
  {
    if (IsSingleCell)
      return ul.Show(0, 0);
    else
      return this.Show(0, 0, 0);
  }
}
```

CellRef.cs

```
/// <summary>
/// A CellRef expression is A1 or $A1 or A$1 or $A$1 or Sheet1!A1
/// </summary>
class CellRef : Expr
{
  public readonly RARef raref;
  private readonly Sheet sheet; // null if sheet-relative, else sheet-absolute

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

  public CellRef(Sheet sheet, bool colAbs, int colRef, bool rowAbs, int rowRef)
    : this(sheet, new RARef(colAbs, colRef, rowAbs, rowRef))
  {
  }

  public override Value Eval(CalculationContext context, int col, int row)
  {
    if (this.sheet != null)
      context = sheet.Context;

    CellAddr ca = raref.Addr(col, row);
    Cell cell = context.Sheet.Cells[ca];
    return cell == null ? Value.Null : cell.Eval(context, ca.col, ca.row);
  }

  public override Expr Move(int deltaCol, int deltaRow)
  {
    return new CellRef(sheet, raref.Move(deltaCol, deltaRow));
  }

  public override Adjusted<Expr> InsertRows(
    Sheet modSheet, bool thisSheet, int aboveRow, int rows, int row)
  {
    if (sheet == modSheet || sheet == null && thisSheet)
    {
      Adjusted<RARef> adj = raref.InsertRows(aboveRow, rows, row);
      return new Adjusted<Expr>(new CellRef(sheet, adj.e), adj.upper, adj.same);
    }
    else
      return new Adjusted<Expr>(this);
  }

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

ConstCell.cs

```
/// <summary>
/// A constant cell is completely immutable; its instances can be shared.
/// </summary>
public abstract class ConstCell : Cell
{
  public override void InsertRows(Dictionary<Expr, Adjusted<Expr>> adjusted,
        Sheet modSheet, bool thisSheet, int aboveRow, int rows, int row)
  {
  }

  public override Cell MoveContents(int deltaCols, int deltaRows)
  {
    return this;
  }
}
```

DataType.cs

```
public enum DataType
{
  Text,
  Number,
  DateTime,
  Boolean,
  Matrix,
  Error,
  Function
}
```

DateTimeCell.cs

```
/// <summary>
/// Represents a datetime constant in a cell.
/// </summary>
public class DateTimeCell : ConstCell
{
  private DateTimeValue v;

  public DateTimeCell(DateTime d)
  {
    this.v = new DateTimeValue(d);
  }

  public override Value Eval(CalculationContext invocationList, int col, int row)
  {
    return v;
  }

  public override string Show(int col, int row)
  {
    return v.value.ToString();
  }
}
```

DateTimeValue.cs

```
public class DateTimeValue : Value
{
  public readonly DateTime value;

  public DateTimeValue(DateTime value)
  {
    this.value = value;
  }

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

  public override DataType DataType
  {
    get { return DataType.DateTime; }
  }
}
```

ErrorValue.cs

```
class ErrorValue : Value
{
  private readonly string msg;

  public ErrorValue(string msg)
  {
    this.msg = "#" + msg;
  }

  public override string ToString()
  {
    return msg;
  }

  public override DataType DataType
  {
    get { return DataType.Error; }
  }
}
```

Formula.cs

```
/// <summary>
/// Represents a formula.
/// </summary>
public class Formula : Cell
{
  private Expr e;

  public Formula(Expr e)
  {
    if (e == null)
      throw new ArgumentNullException("e");
    this.e = e;
  }

  public Formula(Formula f)
    : this(f.e)  { }

  public override Value Eval(CalculationContext context, int col, int row)
  {
    CellAddr ca = new CellAddr(col, row);
    Value v;
    if (!context.TryGetValue(ca, out v))
    {
      context[ca] = Value.VisitMarker;
      v = e.Eval(context, col, row);
      if (v == null)
        v = new NumberValue(0);
      context[ca] = v;
    }
    if (v == Value.VisitMarker)
      throw new CyclicException("Cyclic␣cell␣reference:␣" + Show(col, row));
    return v;
  }

  public override Cell MoveContents(int deltaCols, int deltaRows)
  {
    return new Formula(e.Move(deltaCols, deltaRows));
  }

  public override void InsertRows(Dictionary<Expr, Adjusted<Expr>> adjusted,
    Sheet modSheet, bool thisSheet, int aboveRow, int rows, int row)
  {
    Adjusted<Expr> ae;
    if (adjusted.ContainsKey(e) && row < adjusted[e].upper)
    {
      // there is a valid cached adjusted expression
      //
      ae = adjusted[e];
    }
    else
    {
      // compute new adjusted expression and insert into the cache
      //
      ae = e.InsertRows(modSheet, thisSheet, aboveRow, rows, row);
      if (ae.same)
      {
        ae = new Adjusted<Expr>(e, ae.upper, ae.same);
      }
      adjusted[e] = ae;
    }
    System.Diagnostics.Debug.Assert(row < ae.upper, "Formula.InsertRows");
    e = ae.e;
  }

  public override string Show(int col, int row)
  {
    return "=" + e.Show(col, row, 0);
  }
}
```

Function.cs

```
public delegate Value Applier(CalculationContext context, Expr[] es, int col, int row);

public delegate R Fun<R>();
public delegate R Fun<A1, R>(A1 x1);
public delegate R Fun<A1, A2, R>(A1 x1, A2 x2);

class Function
{
  private static readonly IDictionary<string, Function> table;
  private static readonly Random rnd = new Random();

  public readonly string name;
  public readonly int fixity; // non-zero precedence of operator
  public readonly Applier applier;

  public static Function Get(string name)
  {
    Function function = null;
    if (table.TryGetValue(name, out function))
      return function;
    throw new Exception("Unknown␣function␣'" + name + "'.");
  }

  public static bool TryGetFunction(string name, out Function function)
  {
    return table.TryGetValue(name, out function);
  }

  static Function()
  {
    table = new Dictionary<string, Function>();

    new Function("RAND",
      MakeFunction(delegate() { return rnd.NextDouble(); }));
    new Function("PI",
      MakeFunction(delegate() { return Math.PI; }));

    new Function("TRUNC",
      MakeFunction(delegate(double x) { return (int)x; }));

    new Function("SIN",
      MakeFunction(delegate(double x) { return Math.Sin(x); }));

    new Function("+", 6,
      MakeFunction(delegate(double x, double y) { return x + y; }));
    new Function("*", 7,
      MakeFunction(delegate(double x, double y) { return x * y; }));
    new Function("/", 7,
      MakeFunction(delegate(double x, double y) { return x / y; }));
    new Function("^", 8,
      MakeFunction(delegate(double x, double y) { return Math.Pow(x, y); }));

    new Function("TYPE",
      delegate(CalculationContext invocation, Expr[] es, int col, int row)
            {
              if (es.Length != 1)
                return new ErrorValue("ARGCOUNT");

              Value v = es[0].Eval(invocation, col, row);
              return new TextValue(v.GetType().Name);
            });

    new Function("-", 6,
      delegate(CalculationContext invocation, Expr[] es, int col, int row)
            {
              if (es.Length != 1 && es.Length != 2)
                return new ErrorValue("ARGCOUNT");

              NumberValue v0 = es[0].Eval(invocation, col, row) as NumberValue;
              if (es.Length == 1)
                return new NumberValue(v0 == null ? 0 : -v0.value);
              else
              {
                NumberValue v1 = es[1].Eval(invocation, col, row) as NumberValue;
                return new NumberValue(
                  v0 == null ? 0 : v0.value
                   - (v1 == null ? 0 : v1.value));
              }
            });

    new Function("IF",
      delegate(CalculationContext invocation, Expr[] es, int col, int row)
            {
              if (es.Length != 3)
                return new ErrorValue("ARGCOUNT");

              NumberValue v0 = es[0].Eval(invocation, col, row) as NumberValue;
              if (v0 == null || v0.value == 0)
                return es[2].Eval(invocation, col, row);
              else
                return es[1].Eval(invocation, col, row);
            });
    new Function("MOD",
      MakeFunction(delegate(double x, double y) { return x % y; }));

    new Function("SUM",
      MakeFunction(delegate(Value[] vs)
      {
        double sum = 0.0;
        Apply(vs, delegate(double x) { sum += x; });
        return sum;
      }));

    new Function("YEAR",
      MakeFunction(delegate(DateTime d) { return d.Year; }));
    new Function("MONTH",
      MakeFunction(delegate(DateTime d) { return d.Month; }));

    new Function("VLOOKUP",
      delegate(CalculationContext invocation, Expr[] es, int col, int row)
      {
        if (es.Length != 3 && es.Length != 4)
          return new ErrorValue("ARGCOUNT");

        NumberValue v0 = es[0].Eval(invocation, col, row) as NumberValue;
        MatrixValue v1 = es[1].Eval(invocation, col, row) as MatrixValue;
        NumberValue v2 = es[2].Eval(invocation, col, row) as NumberValue;
        return v1[(int)v2.value - 1, v1.FindRowIndex(((NumberValue)v0).value)];
      });

    // register functions
    //
    Functions.MatrixFunctions.Register();
    Functions.HighOrderFunctions.Register();
  }

  public Function(string name, Applier applier)
    : this(name, 0, applier)
  {
  }

  private Function(string name, int fixity, Applier applier)
  {
    this.name = name;
    this.fixity = fixity;
    this.applier = applier;
    table.Add(name, this);
  }

  /// <summary>
  /// Initializes a new instance of the <see cref="T:Function"/> class.
  /// </summary>
  /// <remarks>
  /// The function is anonymous and is not added to the global function list.
  /// </remarks>
  public Function(Applier applier)
  {
    this.applier = applier;
  }

  private static Applier MakeFunction(Fun<double> dlg)
  {
    return
      delegate(CalculationContext invocation, Expr[] es, int col, int row)
      {
         if (es.Length == 0)
          return new NumberValue(dlg());
        else
          return new ErrorValue("ARGCOUNT");
      };
  }

  private static Applier MakeFunction(Fun<double, double> dlg)
  {
    return
      delegate(CalculationContext invocation, Expr[] es, int col, int row)
    {
      if (es.Length == 1)
      {
        NumberValue v0 = es[0].Eval(invocation, col, row) as NumberValue;
        if (v0 != null)
          return new NumberValue(dlg(v0.value));
        else
          return new NumberValue(dlg(0));
      }
      else
        return new ErrorValue("ARGCOUNT");
    };
}

private static Applier MakeFunction(Fun<DateTime, double> dlg)
{
  return
    delegate(CalculationContext invocation, Expr[] es, int col, int row)
    {
      if (es.Length == 1)
      {
        DateTimeValue v0 = es[0].Eval(invocation, col, row) as DateTimeValue;
        if (v0 != null)
          return new NumberValue(dlg(v0.value));
        else
          return new ErrorValue("ARGTYPE");
      }
      else
        return new ErrorValue("ARGCOUNT");
    };
}

private static Applier MakeFunction(Fun<double, double, double> dlg)
{
  return
    delegate(CalculationContext invocation, Expr[] es, int col, int row)
    {
      if (es.Length == 2)
      {
        Value v0 = es[0].Eval(invocation, col, row).ToNumberValue();
        Value v1 = es[1].Eval(invocation, col, row).ToNumberValue();

        if (v0 is ErrorValue)
          return v0;
        if (v1 is ErrorValue)
          return v1;

        return new NumberValue(dlg(
          ((NumberValue)v0).value, ((NumberValue)v1).value));
      }
      else
        return new ErrorValue("ARGCOUNT");
    };
}

private static Applier MakeFunction(Fun<Value[], double> dlg)
{
  return
    delegate(CalculationContext invocation, Expr[] es, int col, int row)
    {
      try
      {
        return new NumberValue(dlg(Eval(es, invocation, col, row)));
      }
      catch (ArgumentException)
      {
        return new ErrorValue("ARGTYPE");
      }
    };
}
```

```
  private static Applier MakePredicate(Fun<double, double, bool> dlg)
  {
    return
      MakeFunction(delegate(double x, double y)
        {
          return dlg(x, y) ? 1.0 : 0.0;
        });
  }

  private static Value[] Eval(Expr[] es, CalculationContext invocation, int col, int row)
  {
    Value[] vs = new Value[es.Length];
    for (int i = 0; i < es.Length; i++)
    {
      vs[i] = es[i].Eval(invocation, col, row);
    }
    return vs;
  }

  private static void Apply(Value[] vs, Act<double> act)
  {
    foreach (Value v in vs)
    {
      if (v != null)
      {
        if (v is NumberValue)
          act((v as NumberValue).value);
        else if (v is MatrixValue)
          (v as MatrixValue).Apply(act);
        else
          throw new ArgumentException();
      }
    }
  }
}
```

FunctionRefValue.cs

```
class FunctionRefValue : Value
{
  private Function function;

  public FunctionRefValue(Function function)
  {
    if (function == null)
      throw new ArgumentNullException("function");

    this.function = function;
  }

  public override DataType DataType
  {
    get { return DataType.Function; }
  }

  public override string ToString()
  {
    return "f_x:␣" + function.name;
  }

  public Function Function
  {
    get { return function; }
  }
}
```

FunctionSheet.cs

```
/// <summary>
/// Represents a function sheet.
/// </summary>
public class FunctionSheet : Sheet
{
  private FunctionSignature signature;
  private Dictionary<string, CalculationContext> invocations;

  /// <summary>
  /// Initializes a new instance of the <see cref="T:FunctionSheet"/> class.
  /// </summary>
  public FunctionSheet(Workbook workbook, string name, int cols, int rows)
    : base(workbook, name, cols, rows)
  {
    new Function(name, new Applier(ApplySheetFunction)) ;

    this.signature = new FunctionSignature();
    this.invocations = new Dictionary<string, CalculationContext>();
  }

  public string ShowSignature()
  {
    return signature.Show(this);
  }

  /// <summary>
  /// Applies the sheet function.
  /// </summary>
  private Value ApplySheetFunction(CalculationContext invoker,
                                   Expr[] es, int col, int row)
  {
    // create an invocation on the invocationlist
    //
    string name = string.Format("{0}!{1}(...)",
      invoker.Sheet.Name,
      new CellAddr(col, row)
      );

    CalculationContext newInvocation = new CalculationContext(this, name, invoker, es);
    newInvocation.StackDepth = invoker.StackDepth + 1;
    if (newInvocation.StackDepth > 1000)
      return new ErrorValue("STACK_OVERFLOW");

    // NOTE: StackDepth is hardcoded and should be configurable, however
    //       the current implementation will raise a StackOverflowException
    //       with at stackdepth of only 1634.

    invocations[name] = newInvocation;

    // apply arguments
    //
    if (!signature.ApplyArguments(invoker, newInvocation, es, col, row))
      return new ErrorValue("ARGCOUNT");

    // return result
    //
    Value v = signature.GetResult(newInvocation);
    signature.MarkUnevaluatedArguments(newInvocation);
    return v;
  }

  public FunctionSignature Signature
  {
    get { return signature; }
  }

  public Dictionary<string, CalculationContext> InvocationList
  {
    get { return invocations; }
  }
}
```

FunctionSignature.cs

```
public class FunctionSignature
{
  private class SignatureArgument
  {
    public CellAddr ca;
    public string name;
  }

  private List<SignatureArgument> arguments = new List<SignatureArgument>();
  internal CellArea Result;

  public FunctionSignature()
  {
    RARef nulref = new RARef(true, 0, true, 0);
    Result = new CellArea(null, nulref, nulref);
  }

  private CellArea optionalArguments;
  private string optionalArgumentsName;

  public void AddArgument(CellAddr ca)
  {
    AddArgument(ca, string.Empty);
  }

  public void AddArgument(CellAddr ca, string name)
  {
    SignatureArgument arg = new SignatureArgument();
    arg.ca = ca;
    arg.name = name;
    arguments.Add(arg);
  }

  internal void AddOptionalArguments(CellArea ca)
  {
    AddOptionalArguments(ca, string.Empty);
  }

  internal void AddOptionalArguments(CellArea ca, string name)
  {
    this.optionalArguments = ca;
    this.optionalArgumentsName = name;
  }

  public bool ApplyArguments(CalculationContext invoker, CalculationContext list,
                             Expr[] es, int col, int row)
  {
    // check number of arguments
    //
    if (es.Length < arguments.Count)
      return false;

    if (es.Length != arguments.Count && optionalArguments == null)
      return false;

    // run through the arguments
    //
    for (int i = 0; i < arguments.Count; i++)
    {
      list[arguments[i].ca] = es[i].Eval(invoker, col, row); ;
    }

    if (optionalArguments != null)
    {
      // apply optinal arguments
      //
      CellAddr ca = new CellAddr(optionalArguments.UpperLeft, 0, 0);
      for (int i = arguments.Count; i < es.Length; i++)
      {
        if (ca == CellAddr.Invalid)
          throw new Exception("Argument failed.");
        list[ca] = es[i].Eval(invoker, col, row);
        ca = optionalArguments.NextHorizontal(ca);
      }
    }
    return true;
  }

  public void MarkUnevaluatedArguments(CalculationContext invocation)
  {
    for (int col = 0; col < invocation.Sheet.ExpandedColumnCount; col++)
    {
      for (int row = 0; row < invocation.Sheet.ExpandedRowCount; row++)
      {
        CellAddr ca = new CellAddr(col, row);
        if (!invocation.HasValue(ca))
          invocation[ca] = new ErrorValue("NO_EVAL");
      }
    }
  }

  public Value GetResult(CalculationContext invocation)
  {
    Value v = Result.Eval(invocation, 0, 0);
    if (Result.IsSingleCell)
      return ((MatrixValue)v)[0, 0];
    else
      return v;
  }

  public string Show(Sheet sheet)
  {
    StringBuilder sb = new StringBuilder();
    sb.Append(sheet.Name);
    sb.Append("(");

    string prefix = "";
    for (int i = 0; i < arguments.Count; i++)
    {
      sb.Append(prefix);

      if (string.IsNullOrEmpty(arguments[i].name))
        sb.Append(arguments[i].ca);
      else
      {
        sb.Append(arguments[i].name);
        sb.Append("(");
        sb.Append(arguments[i].ca);
        sb.Append(")");
      }
      prefix = ", ";
    }
    if (optionalArguments != null)
    {
      for (int i = 0; i < 2; i++)
      {
        sb.Append(prefix);
        sb.Append(optionalArgumentsName);
        sb.Append(i + 1);
        prefix = ", ";
      }
      sb.Append(prefix);
      sb.Append(" ...");
    }
    sb.Append(")=");
    sb.Append(Result.ToString());
    return sb.ToString();
  }

  public void InitializeSignature(FunctionSheet sheet)
  {
    Style argumentStyle = new Style(sheet.Workbook);
    argumentStyle.Interior = new Interior(
        System.Drawing.Color.Khaki, System.Drawing.Color.Beige, PatternStyle.None);
    for (int index = 0; index < arguments.Count; index++)
    {
      sheet[arguments[index].ca] = new ArgumentCell(
        arguments[index].name,
        index,
        sheet[arguments[index].ca]);

      sheet.Styles[arguments[index].ca] = argumentStyle;
    }

    if (optionalArguments != null)
    {
      CellAddr ca = new CellAddr(optionalArguments.UpperLeft, 0, 0);
      int index = 0;

      while (ca != CellAddr.Invalid)
      {
        index++;
        sheet[ca] = new ArgumentCell(
          optionalArgumentsName + index,
          arguments.Count + index,
          sheet[ca]);

        sheet.Styles[ca] = argumentStyle;

        ca = optionalArguments.NextHorizontal(ca);
      }
    }

    Style resultStyle = new Style(sheet.Workbook);
    resultStyle.Interior = new Interior(
        System.Drawing.Color.LightBlue, System.Drawing.Color.Beige, PatternStyle.None);
    sheet.Styles[Result.UpperLeft.Addr(0,0)] = resultStyle;
  }
}
```

FunName.cs

```
class FunName : Expr
{
  private string name;
  private Function function;

  public FunName(string name)
  {
    if (name == null)
      throw new ArgumentNullException("name");

    this.name = name;
    Function.TryGetFunction(name, out function);
  }

  public override Expr Move(int deltaCols, int deltaRows)
  {
    return this;
  }

  public override Value Eval(CalculationContext context, int col, int row)
  {
    Function f;
    if (Function.TryGetFunction(name, out f))
      return new FunctionRefValue(f);
    else
      return new ErrorValue("NAME");
  }

  public override Adjusted<Expr> InsertRows(
    Spreadsheet.DOM.Sheet modSheet, bool thisSheet, int aboveRow, int rows, int row)
  {
    throw new NotImplementedException();
  }

  public override string Show(int col, int row, int ctxpre)
  {
    return name;
  }
}
```

HigherOrderFunctions.cs

```
internal static class HigherOrderFunctions
{
  private class BindFunctionValue : FunctionRefValue
  {
    public BindFunctionValue(Function boundFunction, Expr[] boundArgs)
      : base(
        new Function(
      delegate(CalculationContext context, Expr[] es, int col, int row)
    {
      Expr[] combinedArgs = new Expr[boundArgs.Length + es.Length];
      boundArgs.CopyTo(combinedArgs, 0);
      es.CopyTo(combinedArgs, boundArgs.Length);
      return boundFunction.applier(context, combinedArgs, col, row);
    }))
    {
    }
  }

  public static void Register()
  {
    new Function("MAP", new Applier(Map));
    new Function("BIND", new Applier(Bind));
    new Function("CALL", new Applier(Call));
  }

  private static Value Bind(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length < 2)
      return new ErrorValue("ARGCOUNT");

    // evaluate 1. argument to function
    //
    FunctionRefValue v0 = es[0].Eval(invocation, col, row) as FunctionRefValue;
    if (v0 == null)
      return new ErrorValue("VALUE");

    // evaluate all the other arguments to ValueConst
    //
    Expr[] es1 = new Expr[es.Length - 1];
    for (int i = 1; i < es.Length; i++)
    {
      Value v1 = es[i].Eval(invocation, col, row);
      es1[i - 1] = new ValueConst(v1);
    }
    return new BindFunctionValue(v0.Function, es1);
  }

  private static Value Call(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length < 2)
      return new ErrorValue("ARGCOUNT");

    // evaluate 1. argument to function
    //
    FunctionRefValue v0 = es[0].Eval(invocation, col, row) as FunctionRefValue;
    if (v0 == null)
      return new ErrorValue("VALUE");

    // evaluate all the other arguments to ValueConst
    //
    Expr[] es1 = new Expr[es.Length - 1];
    for (int i = 1; i < es.Length; i++)
    {
      Value v1 = es[i].Eval(invocation, col, row);
      es1[i - 1] = new ValueConst(v1);
    }
    return v0.Function.applier(invocation, es1, col, row);
  }

  private static Value Map(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length != 2)
      return new ErrorValue("ARGCOUNT");

    FunctionRefValue v0 = es[0].Eval(invocation, col, row) as FunctionRefValue;
    if (v0 == null)
      return new ErrorValue("VALUE");

    MatrixValue v1 = es[1].Eval(invocation, col, row) as MatrixValue;
    if (v1 == null)
      return new ErrorValue("VALUE");

    Value[,] v = new Value[v1.Cols, v1.Rows];
    Expr[] es1 = new Expr[1]; // function must have 1 argument!?!?!
    Function f = v0.Function;

    for (int r = 0; r < v1.Rows; r++)
    {
      for (int c = 0; c < v1.Cols; c++)
      {
        es1[0] = new ValueConst(v1[c, r]);
        v[c, r] = f.applier(invocation, es1, 0, 0);
      }
    }
    return new MatrixValue(v);
  }
}
```

MatrixFunctions.cs

```
internal static class MatrixFunctions
{
  public static void Register()
  {
    new Function("CBIND", new Applier(CBind));
    new Function("RBIND", new Applier(RBind));
    new Function("CDIM", new Applier(CDim));
    new Function("RDIM", new Applier(RDim));
    new Function("TRANSPOSE", new Applier(Transpose));
    new Function("MMULT", new Applier(MMult));
    new Function("MADD", new Applier(MAdd));
    new Function("MLookup", new Applier(MLookup));
  }

  private static Value CBind(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length != 2)
      return new ErrorValue("ARGCOUNT");

    MatrixValue v0 = es[0].Eval(invocation, col, row) as MatrixValue;
    MatrixValue v1 = es[1].Eval(invocation, col, row) as MatrixValue;
    if (v0 == null || v1 == null)
      return new ErrorValue("VALUE");

    if (v0.Rows != v1.Rows)
      return new ErrorValue("DIMS␣" + v0.Rows + "!=" + v1.Rows);

    int cols = v0.Cols + v1.Cols;
    int rows = v0.Rows;

    Value[,] v = new Value[cols, rows];
    for (int r = 0; r < rows; r++)
    {
      for (int c = 0; c < v0.Cols; c++)
      {
        v[c, r] = v0[c, r];
      }
      for (int c = 0; c < v1.Cols; c++)
      {
        v[c + v0.Cols, r] = v1[c, r];
      }
    }
    return new MatrixValue(v);
  }

  private static Value CDim(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length != 1)
      return new ErrorValue("ARGCOUNT");

    MatrixValue v = es[0].Eval(invocation, col, row) as MatrixValue;
    if (v == null)
      return new ErrorValue("VALUE");

    return new NumberValue(v.Cols);
  }

  private static Value RBind(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length != 2)
      return new ErrorValue("ARGCOUNT");

    MatrixValue v0 = es[0].Eval(invocation, col, row) as MatrixValue;
    MatrixValue v1 = es[1].Eval(invocation, col, row) as MatrixValue;
    if (v0 == null || v1 == null)
      return new ErrorValue("VALUE");

    if (v0.Cols != v1.Cols)
      return new ErrorValue("DIMS");

    int cols = v0.Cols;
    int rows = v0.Rows + v1.Rows;

    Value[,] v = new Value[cols, rows];

    for (int c = 0; c < cols; c++)
    {
      for (int r = 0; r < v0.Rows; r++)
      {
        v[c, r] = v0[c, r];
      }
      for (int r = 0; r < v1.Rows; r++)
      {
        v[c, r + v0.Rows] = v1[c, r];
      }
    }
    return new MatrixValue(v);
  }

  private static Value RDim(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length != 1)
      return new ErrorValue("ARGCOUNT");

    MatrixValue v = es[0].Eval(invocation, col, row) as MatrixValue;
    if (v == null)
      return new ErrorValue("VALUE");

    return new NumberValue(v.Rows);
  }

  private static Value Transpose(CalculationContext invocation,
    Expr[] es, int col, int row)
  {
    if (es.Length != 1)
```

```
      return new ErrorValue("ARGCOUNT");

    MatrixValue v = es[0].Eval(invocation, col, row) as MatrixValue;
    if (v == null)
      return new ErrorValue("VALUE");

    return v.Transpose();
  }

  private static Value MMult(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length != 2)
      return new ErrorValue("ARGCOUNT");

    MatrixValue v0 = es[0].Eval(invocation, col, row) as MatrixValue;
    MatrixValue v1 = es[1].Eval(invocation, col, row) as MatrixValue;

    if (v0 == null || v1 == null)
      return new ErrorValue("VALUE");

    return v0.Multiply(v1);
  }

  private static Value MAdd(CalculationContext invocation, Expr[] es, int col, int row)
  {
    if (es.Length != 2)
      return new ErrorValue("ARGCOUNT");

    MatrixValue v0 = es[0].Eval(invocation, col, row) as MatrixValue;
    MatrixValue v1 = es[1].Eval(invocation, col, row) as MatrixValue;

    if (v0 == null || v1 == null)
      return new ErrorValue("VALUE");

    return v0.Add(v1);
  }

  private static Value MLookup(CalculationContext invocation,
    Expr[] es, int col, int row)
  {
    if (es.Length != 3)
      return new ErrorValue("ARGCOUNT");

    MatrixValue v0 = es[0].Eval(invocation, col, row) as MatrixValue;
    NumberValue v1 = es[1].Eval(invocation, col, row) as NumberValue;
    NumberValue v2 = es[2].Eval(invocation, col, row) as NumberValue;

    if (v0 == null || v1 == null || v2 == null)
      return new ErrorValue("VALUE");

    int c = (int)v1.value;
    int r = (int)v2.value;

    if (c < 0 || c >= v0.Cols || r < 0 || r >= v0.Rows)
      return new ErrorValue("DIMS");

    return v0[c, r];
  }
}
```

MatrixSheet.cs

```
public class MatrixSheet : Sheet
{
  public MatrixSheet(Workbook workbook, MatrixValue value)
    : base(workbook, value.Cols, value.Rows)
  {
    for (int col = 0; col < value.Cols; col++)
    {
      for (int row = 0; row < value.Rows; row++)
      {
        Cells[col, row] = Cell.Create(value[col, row]);
      }
    }
  }
}
```

MatrixValue.cs

```
public class MatrixValue : Value
{
  private readonly Value[,] values;

  public MatrixValue(Value[,] values)
  {
    this.values = values;
  }

  public int Cols
  {
    get { return values.GetLength(0); }
  }

  public int Rows
  {
    get { return values.GetLength(1); }
  }

  public Value this[CellAddr ca]
  {
    get
    {
      return this[ca.col, ca.row];
    }
  }

  public Value this[int col, int row]
  {
    get
    {
      return values[col, row];
    }
  }

  public void Apply(Act<double> act)
  {
    foreach (Value v in values)
    {
      if (v != null)
      {
        if (v is NumberValue)
          act((v as NumberValue).value);
        else if (v is MatrixValue)
          (v as MatrixValue).Apply(act);
        else
          throw new ArgumentException();
      }
    }
  }

  public int FindRowIndex(double value)
  {
    for (int index = 0; index < Rows; index++)
    {
      NumberValue v = values[0, index] as NumberValue;
      if (v != null)
      {
        if (v.value > value)
        {
          return index - 1;
        }
      }
    }
    return Rows - 1;
  }

  public Value Add(MatrixValue m)
  {
    if (m.Rows != Rows || m.Cols != Cols)
      return new ErrorValue("DIMS");

    int rows = Rows;
    int cols = Cols;
    Value[,] v = new Value[cols, rows];
    for (int r = 0; r < rows; r++)
    {
      for (int c = 0; c < cols; c++)
      {
        NumberValue v0 = values[c, r] as NumberValue;
        NumberValue v1 = m.values[c, r] as NumberValue;
        if (v0 != null && v1 != null)
          v[c, r] = new NumberValue(v0.value + v1.value);
        else
          v[c, r] = new ErrorValue("VALUE");
      }
    }
    return new MatrixValue(v);
  }

  public Value Multiply(MatrixValue m)
  {
    if (m.Rows != Cols)
      return new ErrorValue("DIMS");

    int rows = Rows;
    int cols = m.Cols;
    int size = Cols;

    Value[,] v = new Value[cols, rows];
    double[] column = new double[size];
    for (int c = 0; c < cols; c++)
    {
      for (int k = 0; k < size; k++)
      {
        NumberValue nv = m[c, k] as NumberValue;
        if (nv == null)
          return new ErrorValue("VALUE");

        column[k] = nv.value;
      }
      for (int r = 0; r < rows; r++)
      {
        double s = 0;
        for (int k = 0; k < size; k++)
        {
          NumberValue nv = values[k, r] as NumberValue;
          if (nv == null)
            return new ErrorValue("VALUE");

          s += nv.value * column[k];
        }
        v[c, r] = new NumberValue(s);
      }
    }
    return new MatrixValue(v);
  }

  public override string ToString()
  {
    StringBuilder sb = new StringBuilder();
    sb.Append('{');
    for (int r = 0; r < Rows; r++)
    {
      for (int c = 0; c < Cols; c++)
      {
        Value v = values[c, r];
```

```
        sb.Append(v == null ? "[none]" : v.ToString());
        if (c < Cols - 1)
          sb.Append(",␣");
      }
      if (r < Rows - 1)
        sb.Append(";␣");
    }
    sb.Append('}');
    return sb.ToString();
  }

  public MatrixValue Transpose()
  {
    Value[,] t = new Value[Rows, Cols];

    for (int i = 0; i < Cols; i++)
    {
      for (int j = 0; j < Rows; j++)
      {
        t[j, i] = values[i, j];
      }
    }
    return new MatrixValue(t);
  }

  public override DataType DataType
  {
    get { return DataType.Matrix; }
  }
}
```

NullValue.cs

```
/// <summary>
/// Represent the value of null.
/// </summary>
public class NullValue : Value
{
  public override DataType DataType
  {
    get { throw new NotImplementedException(); }
  }

  public override Value ToNumberValue()
  {
    return new NumberValue(0);
  }

  public override string ToString()
  {
    return "<null>";
  }
}
```

NumberCell.cs

```
/// <summary>
/// Represent a floating-point constant in a cell.
/// </summary>
public class NumberCell : ConstCell
{
  private NumberValue v;

  public NumberCell(double d)
  {
    this.v = new NumberValue(d);
  }
  public override Value Eval(CalculationContext invocationList, int col, int row)
  {
    return v;
  }

  public override string Show(int col, int row)
  {
    return v.value.ToString();
  }
}
```

NumberValue.cs

```
public class NumberValue : Value
{
  public readonly double value;

  public NumberValue(double value)
  {
    this.value = value;
  }

  public override Value ToNumberValue()
  {
    return this;
  }

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

  public override DataType DataType
  {
    get { return DataType.Number; }
  }
}
```

RARef.cs

```
public sealed class RARef
{
  public bool colAbs, rowAbs;
  public 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;
  }

  public RARef(string s, int row, int col)
  {
    int i = 0;
    if (i < s.Length && s[i] == '$')
    {
      colAbs = true;
      i++;
    }
    int val = -1;
    while (i < s.Length && IsAToZ(s[i]))
    {
      val = (val + 1) * 26 + AToZValue(s[i]);
      i++;
    }
    colRef = colAbs ? val : val - col;
    if (i < s.Length && s[i] == '$')
    {
      rowAbs = true;
      i++;
    }
    val = 0;
    while (i < s.Length && char.IsDigit(s[i]))
    {
      val = val * 10 + (s[i] - '0');
      i++;
    }
    rowRef = (rowAbs ? val : val - row) - 1;
  }

  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;
  }

  public CellAddr Addr(int col, int row)
  {
    return new CellAddr(this, col, row);
  }

  public Adjusted<RARef> InsertRows(int aboveRow, int rows, int row)
  {
    int newRow;
    int upper;
    if (rowAbs)
    {
      if (rowRef >= aboveRow)
      {
        // absolute ref to cell above inserted
        newRow = rowRef + rows;
        upper = int.MaxValue;
      }
      else
      {
        // absolute ref to cell below inserted
        newRow = rowRef;
        upper = int.MaxValue;
      }
    }
    else // reltative reference
    {
      if (row >= aboveRow)
      {
        if (row + rowRef < aboveRow)
        {
          // relative ref from above insertion to cell below insertion
          newRow = rowRef - rows;
          upper = aboveRow - rowRef;
        }
        else
        {
          // relative ref from above insertion to cell above insertion
          newRow = rowRef;
          upper = int.MaxValue;
        }
      }
      else
      {
        if (row + rowRef >= aboveRow)
        {
          // relative ref from below insertion to cell above insertion
          newRow = rowRef + rows;
          upper = aboveRow;
        }
        else
```

```
        {
          // relative ref from below insertion to cell below insertion
          newRow = rowRef;
          upper = Math.Min(aboveRow, aboveRow - rowRef);
        }
      }
    }
    RARef rarefNew = new RARef(colAbs, colRef, rowAbs, newRow);
    return new Adjusted<RARef>(rarefNew, upper, rowRef == newRow);
  }

  public RARef Move(int deltaCol, int deltaRow)
  {
    return new RARef(
      colAbs, colAbs ? colRef : colRef + deltaCol,
      rowAbs, rowAbs ? rowRef : rowRef + deltaRow
      );
  }

  public string Show(int col, int row)
  {
    CellAddr ca = new CellAddr(this, col, row);
    return (colAbs ? "$" : "") + Column.GetName(ca.col)
        + (rowAbs ? "$" : "") + (ca.row + 1);
  }
}
```

Sheet.cs

```
public delegate void Shower(int col, int row, string value);
public delegate void SheetHandler(Sheet sheet);

public abstract class Sheet
{
  private int activeColumn;
  private int activeRow;
  private string name;
  private Workbook workbook;
  private CellCollection cells;
  private RangeSelection rangeSelection;
  private ColumnCollection columns;
  private RowCollection rows;
  private StyleCollection styles;

  private int leftColumnVisible;
  private int topRowVisible;
  private int defaultColumnWidth;
  private int defaultRowHeight;

  public event ActiveCellChangedEventHandler ActiveCellChanged;
  public event ColumnChangedEventHandler ColumnWidthChanged;
  public event EventHandler LeftColumnVisibleChanged;
  public event EventHandler RangeSelectionChanged;
  public event SheetHandler RecomputeComplete;
  public event RowChangedEventHandler RowHeightChanged;
  public event EventHandler TopRowVisibleChanged;

  private CalculationContext context;

  protected Sheet(Workbook workbook, int cols, int rows)
  {
    this.workbook = workbook;
    this.columns = new ColumnCollection(this);
    this.rows = new RowCollection(this);
    this.styles = new StyleCollection();
    this.cells = new CellCollection(cols, rows);

    defaultColumnWidth = Column.DefaultWidth;
    defaultRowHeight = Row.DefaultHeight;
  }
```

```
protected Sheet(Workbook workbook, string name, int cols, int rows)
  : this(workbook, cols, rows)
{
  this.name = name;
  this.context = new CalculationContext(this, name);
  workbook.AddSheet(this);
}

protected Sheet()
{
}

public void CopyCell(int fromCol, int fromRow, int col, int row, int cols, int rows)
{
  Cell cell = cells[fromCol, fromRow];
  if (cell is Formula)
  {
    // clone cache but share expression f.e between all target cells
    //
    Formula f = (Formula)cell;
    for (int c = 0; c < cols; c++)
      for (int r = 0; r < rows; r++)
        cells[col + c, row + r] = new Formula(f);
  }
  else if (cell is ConstCell)
  {
    // share constant cell between all target cells
    //
    for (int c = 0; c < cols; c++)
      for (int r = 0; r < rows; r++)
        cells[col + c, row + r] = cell;
  }
  else
    throw new Exception("Cannot␣copy␣cell:␣" + cell);
}

public void InsertMatrixFormula(
  Cell cell, int col, int row, CellAddr ulCa, CellAddr lrCa)
{
  throw new NotImplementedException();
}

public void InsertRows(int aboveRow, int rows)
{
  // check that this will not split a matrix formula
  //
  if (aboveRow > 1)
  {
    for (int col = 0; col < ExpandedColumnCount; col++)
    {
      Cell cell = cells[col, aboveRow - 1];
      if (cell is MatrixFormula)
        if (((MatrixFormula)cell).Contains(col, aboveRow))
          throw new Exception("Insert␣would␣split␣matrix␣formula.");
    }
  }

  // adjust formulas in all sheets. The dictionary record adjusted
  // expressions to preserve sharing of expression where possible.
  //
  Dictionary<Expr, Adjusted<Expr>> adjusted = new Dictionary<Expr, Adjusted<Expr>>();
  foreach (Sheet sheet in workbook)
  {
    CellCollection cs = sheet.cells;
    for (int r = 0; r < sheet.ExpandedRowCount; r++)
    {
      for (int c = 0; c < sheet.ExpandedColumnCount; c++)
      {
        Cell cell = cs[c, r];
        if (cell != null)
          cell.InsertRows(adjusted, this, sheet == this, aboveRow, rows, r);
      }
    }
  }
  // move the rows below aboveRow in current sheet
  //
  for (int r = ExpandedRowCount - 1; r >= aboveRow; r--)
  {
    for (int c = 0; c < ExpandedColumnCount; c++)
    {
      cells[c, r] = cells[c, r - rows];
    }
  }

  // finally, null out the fresh rows
  //
  for (int r = 0; r < rows; r++)
  {
    for (int c = 0; c < ExpandedColumnCount; c++)
    {
      cells[c, aboveRow + r] = null;
    }
  }
}

protected void OnActiveCellChanged(ActiveCellChangedEventArgs e)
{
  if (ActiveCellChanged != null)
  {
    ActiveCellChanged(this, e);
  }
}

protected internal void OnColumnWidthChanged(ColumnChangedEventArgs e)
{
  if (ColumnWidthChanged != null)
  {
    ColumnWidthChanged(this, e);
  }
}

protected void OnLeftColumnVisibleChanged(EventArgs e)
{
  if (LeftColumnVisibleChanged != null)
  {
    LeftColumnVisibleChanged(this, e);
  }
}

protected void OnRangeSelectionChanged(EventArgs e)
{
  if (this.RangeSelectionChanged != null)
  {
    this.RangeSelectionChanged(this, e);
  }
}

protected internal void OnRowHeightChanged(RowChangedEventArgs e)
{
  if (RowHeightChanged != null)
  {
    RowHeightChanged(this, e);
  }
}

protected virtual void OnRecomputeComplete()
{
  if (RecomputeComplete != null)
  {
    RecomputeComplete(this);
  }
}

protected void OnTopRowVisibleChanged(EventArgs e)
{
  if (TopRowVisibleChanged != null)
  {
    TopRowVisibleChanged(this, e);
  }
}

/// <summary>
/// Move cell from (fromCol, fromRow) to (col, row)
/// </summary>
public void MoveCell(int fromCol, int fromRow, int col, int row, int cols, int rows)
{
  Cell cell = cells[fromCol, fromRow];
  cells[col, row] = cell.MoveContents(col - fromCol, row - fromRow);
}

public void Recompute(CalculationContext invocation)
{
  for (int col = 0; col < ExpandedColumnCount; col++)
  {
    for (int row = 0; row < ExpandedRowCount; row++)
    {
      Cell cell = cells[col, row];
      if (cell != null)
      {
        cell.Eval(invocation, col, row);
      }
    }
  }
}

public void Recompute()
{
  Recompute(context);
  OnRecomputeComplete();
}

public void Reset()
{
  context.Reset();
}

public void SetActiveCell(int rowIndex, int colIndex)
{
  if (rowIndex != activeRow || colIndex != activeColumn)
  {
    ActiveCellChangedEventArgs args = new ActiveCellChangedEventArgs(
      activeRow, activeColumn);
    activeRow = rowIndex;
    activeColumn = colIndex;
    OnActiveCellChanged(args);
  }
}

public string Show(int col, int row)
{
  Cell cell = cells[col, row];
  if (cell != null)
    return cell.Show(col, row);
  else
    return null;
}

public void ShowAll(Shower show)
{
  for (int col = 0; col < ExpandedColumnCount; col++)
  {
    for (int row = 0; row < ExpandedRowCount; row++)
    {
      Cell cell = cells[col, row];
      show(col, row, cell != null ? cell.ShowValue(this.Context, col, row) : null);
    }
  }
}

public string ShowValue(int col, int row)
{
```

```
  Cell cell = cells[col, row];
  if (cell != null)
    return cell.ShowValue(this.Context, col, row);
  else
    return null;
}

public Cell this[int col, int row]
{
  get { return cells[col, row]; }
  set { cells[col, row] = value; }
}

public Cell this[CellAddr ca]
{
  get { return cells[ca.col, ca.row]; }
  set { cells[ca.col, ca.row] = value; }
}

public int ExpandedColumnCount
{
  get { return cells.Cols; }
}

public int ExpandedRowCount
{
  get { return cells.Rows; }
}

public CellCollection Cells
{
  get { return cells; }
}

public ColumnCollection Columns
{
  get { return columns; }
}

public RowCollection Rows
{
  get { return rows; }
}

public int ActiveColumn
{
  get { return activeColumn; }
  set { SetActiveCell(activeRow, value); }
}

public int ActiveRow
{
  get { return activeRow; }
  set { SetActiveCell(value, activeColumn); }
}

public int DefaultColumnWidth
{
  get
  {
    return defaultColumnWidth;
  }
  set
  {
    throw new NotImplementedException();
  }
}

public int DefaultRowHeight
{
  get
  {
    return defaultRowHeight;
  }
  set
  {
    throw new NotImplementedException();
  }
}

public bool DisplayColumnHeaders
{
  get { return true; }
}

public bool DisplayGridlines
{
  get { return true; }
}

public bool DisplayRowHeaders
{
  get { return true; }
}

public int LeftColumnVisible
{
  get
  {
    return leftColumnVisible;
  }
  set
  {
    if (value != leftColumnVisible)
    {
      this.leftColumnVisible = value;
      OnLeftColumnVisibleChanged(EventArgs.Empty);
    }
  }
}

public string Name
{
  get { return name; }
  set { name = value; }
}

public RangeSelection RangeSelection
{
  get { return rangeSelection; }
  set
  {
    if (rangeSelection != value)
    {
      rangeSelection = value;
      OnRangeSelectionChanged(EventArgs.Empty);
    }
  }
}

public int TopRowVisible
{
  get
  {
    return topRowVisible;
  }
  set
  {
    if (value != this.topRowVisible)
    {
      this.topRowVisible = value;
      OnTopRowVisibleChanged(EventArgs.Empty);
    }
  }
}

public Workbook Workbook
{
    get { return workbook; }
  }

  public CalculationContext Context
  {
    get { return context; }
  }

  public StyleCollection Styles
  {
    get { return styles; }
  }
}
```

TextCell.cs

```
class TextCell : ConstCell
{
  private readonly TextValue v;

  public TextCell(string s)
  {
    this.v = new TextValue(s);
  }

  public override Value Eval(CalculationContext invocationList, int col, int row)
  {
    return v;
  }

  public override string Show(int col, int row)
  {
    return "'" + v.value;
  }
}
```

TextValue.cs

```
public class TextValue : Value
{
  public readonly string value;

  public TextValue(string value)
  {
    this.value = value;
  }

  public override Value ConvertTo(DataType dataType)
  {
    switch (dataType)
    {
      case DataType.Number:
        double v;
        if (double.TryParse(value, out v))
          return new NumberValue(v);
        else
          return new ErrorValue("VALUE");
    }
    return base.ConvertTo(dataType);
  }

  public override string ToString()
  {
    return value;
  }

  public override Value ToNumberValue()
  {
    double v;
    if (double.TryParse(value, out v))
      return new NumberValue(v);
    else
      return base.ToNumberValue();
  }

  public override DataType DataType
  {
    get { return DataType.Text; }
  }
}
```

Value.cs

```
/// <summary>
/// Represent a value
/// </summary>
public abstract class Value
{
  private class VisitValue : Value
  {
    public override DataType DataType
    {
      get { throw new NotImplementedException(); }
    }
  }

  public static readonly Value VisitMarker = new VisitValue();
  public static readonly Value Null = new NullValue();
  public abstract DataType DataType { get; }
}
```

Workbook.cs

```
public sealed class Workbook : IEnumerable<Sheet>
{
  public List<Sheet> sheets;
  private bool cyclic; // if true, workbook may be inconsistent
  private Sheet activeSheet;
  private Style style;

  public Workbook()
  {
    this.sheets = new List<Sheet>();
    this.style = new Style(this);
  }

  public void AddSheet(Sheet sheet)
  {
    sheets.Add(sheet);
    if (activeSheet == null)
      activeSheet = sheet;
  }

  public void Recompute()
  {
    cyclic = false;
    try
    {
      foreach (Sheet sheet in sheets)
      {
        sheet.Recompute();
      }
    }
    catch (CyclicException)
    {
      foreach (Sheet sheet in sheets)
      {
        sheet.Reset();
      }
      cyclic = true;
      throw;
    }
  }

  public bool Cyclic
  {
    get { return cyclic; }
  }

  public Sheet ActiveSheet
  {
    get { return activeSheet; }
    set { activeSheet = value; }
  }

  public Sheet this[string name]
  {
    get
    {
      foreach (Sheet sheet in sheets)
      {
        if (sheet.Name == name)
          return sheet;
      }
      throw new Exception("No sheet named '" + name + "'.");
    }
  }

  public Sheet this[int index]
  {
    get
    {
      return sheets[index];
    }
  }

  public IEnumerator<Sheet> GetEnumerator()
  {
    foreach (Sheet sheet in sheets)
      yield return sheet;
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    foreach (Sheet sheet in sheets)
      yield return sheet;
  }

  public Style Style
  {
    get { return style; }
  }
}
```

Worksheet.cs

```
public class Worksheet : Sheet
{
  public Worksheet(Workbook workbook, string name, int cols, int rows)
    : base(workbook, name, cols, rows)
  {
  }
}
```

Parser.atg

```
using System.Collections.Generic;
using System.Globalization;

using Spreadsheet.AbstractSyntax;
using Spreadsheet.DOM;

COMPILER Spreadsheet

$L

  private int col, row;
  private Workbook workbook;
  private Cell cell;

  public Cell ParseCell(Workbook workbook, int col, int row)
  {
    this.workbook = workbook;
    this.col = col;
    this.row = row;
    Parse();
    return cell;
  }

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit = "0123456789".
  Alpha = letter + digit.
  cr = '\r'.
  lf = '\n'.
  tab = '\t'.
  exclamation = '!'.
  dollar = '$'.
  newline = cr + lf.
  strchar = ANY - '"'␣-␣'\\'␣-␣newline.
␣␣char␣=␣ANY␣-␣'"' - '\\' - newline.

TOKENS
  name = letter { letter }.
  number      =
               digit { digit }
               [                                /* optional fraction */
                [("." | ",") digit { digit }]    /* optional fractional digits */
                [ ( "E" | "e" )                  /* optional exponent */
                  [ "+" | "-" ]                  /* optinoal exponentsign */
                  digit { digit }
                ]
               ] .
  sheetref    = Alpha { Alpha } exclamation.
  raref = [ dollar ] letter { letter } [ dollar ] digit { digit }.
  string = "\"" { char } "\"".
  textcell = "\'" { char }.

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

IGNORE cr + lf + tab

PRODUCTIONS

AddOp<out string op>
=                           (. op = "+"; .)
  ( '+'
  | '-'                     (. op = "-"; .)
  | '&'                     (. op = "&"; .)
  ).

ArgumentList<out Expr[] es>   (. es = null; .)
=
  ( ')'                       (. es = new Expr[0]; .)
    | Exprs1<out es> ')'
  )
```

```
.

Expr<out Expr e>                        (. Expr e2; string op; e = null; .)

= "-" Expr<out e>          (. e = new FunCall("-", new Expr[] { e }); .)
  | (Term<out e>
  { AddOp<out op>
    Term<out e2>           (. e = new FunCall(op, new Expr[] { e, e2 }); .)
  }).

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(); .)
  .

Factor<out Expr e>         (. RARef r1, r2; Sheet s1 = null; double d;
                              string s; Expr[] es; e = null; .)
= (
    | sheetref             (. try
                              {
                                string sheetname = t.val.TrimEnd('!');
                                s1 = workbook[sheetname];
                              }
                              catch
                              {
                              }
                           .)
  )
    Raref<out r1> (          (. e = new CellRef(s1, r1); .)
      | ':' Raref<out r2> (. e = new CellArea(s1, r1, r2); .)
    )
    | Number<out d>        (. e = new NumberConst(d); .)
    | string               (. int len = t.val.Length-2;
                              e = new TextConst(t.val.Substring(1, len));
                           .)
    | '(' Expr<out e> ')'
    | Name<out s>
      ( '(' ArgumentList<out es>   (. e = new FunCall(s, es); .)
      |                            (. e = new FunName(s); .)
      )
  .

MulOp<out string op>
=                          (. op = "*"; .)
  ( '*'
  | '/'                    (. op = "/"; .)
  ).

Name<out string s>
= name                     (. s = t.val; .)
  .

Number<out double d>       (. d = 0.0; .)
= ( number                 (. d = double.Parse(t.val, CultureInfo.InvariantCulture); .)
  | "-" number             (. d = -double.Parse(t.val, CultureInfo.InvariantCulture); .)
  ).

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

Raref<out RARef raref>
= raref                    (. raref = new RARef(t.val, row, col); .).

Spreadsheet                (. Expr e; double d; .)
= ( '=' Expr<out e>        (. cell = new Formula(e); .)
  | textcell               (. cell = new TextCell(t.val.Substring(1)); .)
  | Number<out d>          (. cell = new NumberCell(d); .)
  ).

Term<out Expr e>           (. Expr e2; string op; .)
= PowFactor<out e>
  { MulOp<out op>
    PowFactor<out e2>      (. e = new FunCall(op, new Expr[] { e, e2 }); .)
  }.

END Spreadsheet.
```