An evaluation of sheet defined financial functions in Funcalc

Masters Thesis
IT University of Copenhagen
Supervisor: Peter Sestoft

Jens Zeilund Sørensen
(jzso@itu.dk)

March 1, 2012

Abstract

Spreadsheets are used heavily within many different fields. Many users are extending the spreadsheet functionality by user defined functions. However traditional spreadsheet implementations tend to support this extension of functionality through alien languages that takes you out of the spreadsheet metaphor. Different solutions has been suggested within academia that brings this extendability closer to the spreadsheet metaphor. Such a solution has been implemented by Peter Sestoft however it has not been tested along with larger spreadsheet libraries of functions. To address this, an evaluation has been performed based on a financial function library similar to the ones in Excel, that identified a range of observations and possible improvements to this new language. Further testing has also been performed that relates the performance of such functions against Excels built-in financial functions functions.

Contents

List of Figures

List of Tables

1 Introduction

Most spreadsheet software today supports writing your own functions using a language that takes you out of context of the spreadsheet metaphor. Examples are Excel that uses VBA[1] for scripting and Open Office Calc that uses their own version of Basic. With all these different languages, the end-user of a given spreadsheet program is taken out of context to perform a more complex computation on his data. Nunez[3] and Peyton Jones et al.[11] introduced the concept of defining functions within the spreadsheet metaphor to keep the end user inside the context of the spreadsheet.

Peter Sestoft implemented and extended Peyton-Jones idea of function sheets into Funcalc. Funcalc is a spreadsheet implementation written in C# which takes advantage of runtime code generation to speed up performance of user defined functions. It serves as a well documented reference implementation and a testbed for spreadsheet research[19][20]. The goal with Sestoft's implementation is to serve as an efficient open source spreadsheet implementation that allows to easily experiment with spreadsheet innovation. Secondly it also serves as a test bed for sheet defined functions to show the convenience and speed in these. The implementation is complete enough to perform benchmarks against other spreadsheet engines such as OpenOffice Calc and Microsoft Excel.

The goal of this thesis is to:

- evaluate the expressiveness and identify missing
- constructs in the functional language
- look at typical pitfalls that exist within the context of this programming model.

A correctness analysis will also be performed to see if the expected results match those of Excel. This evaluation may lead to recommendations for improvement of the concept of sheet-defined function and the Funcalc implementation. To reach the goal, this thesis will implement a subset of Excel's built in financial functions in the function sheets.

In other words the question would be:

"How do function sheets perform and can they conveniently express real world functions?"

[1] Visual Basic for Applications

2 Background

2.1 Spreadsheets

Spreadsheets have a long history dating back to the late 1970's. VisiCalc by Dan Bricklin[6] came as a product based around his days taking an MBA at Harvard Business School to help with financial calculations. VisiCalc introduced the concept a two dimensional grid of cells with row-numbers and column-letters containing formulas that could reference other cells and produce results based on these cells[6].

The power of VisiCalc was its interactiveness in a WYSIWYG[2] way with point and click to change values as well as automatic recalculation of dependent cells. It had an easy learning curve and people could use it without knowing the language completely.

The biggest player today is Microsoft with Excel which is used heavily within finance. Spreadsheets today are also used for scientific purposes in biology and other sciences as well as for home use. Excel was first released in 1983 and is still releasing new versions to this day.

The Spreadsheet programs today still maintain the same visual approach for representation which shows the power of the original design back in 1978.

Early versions of Spreadsheet programs did not include the possibility for users to write their own functions and Microsoft introduced this to Excel in 1993 with VBA. The open source world has also developed Spreadsheet programs with the biggest ones today being Open Office Calc and Gnumeric. Both are still under active development and is mostly used under linux. Implementation varies between the different spreadsheet engines and even within versions there can be differences. A good example is Excel in which behavior of certain functions differ from version to version which is not beneficial for companies on which part of their business rely on the results of their spreadsheets.

The spreadsheet formula language is a flat language that consist of equations and predefined functions. These consist of both conditional functions and computational functions.

By flat it is meant that the scope of the formula language is global. Global means that a cell can be referenced from everywhere. This can either be from other cells within the sheet, from cells across sheets or even across workbooks. Referencing sheets are built up by the following convention: $[Finance.xml]testTvm!B4$ starting with the filename (Workbook) followed by sheet name and finally the cell.

Conditional functions allow the user to create a control flow based on values from referenced cells. IF(condition, true, false) is an example of a conditional function. It evaluates the condition followed by an evaluation of either the true branch or the false branch. The condition is a boolean

[2]WYSIWYG is short for what you see is what you get and refers to a graphical interface that is simple and consistent

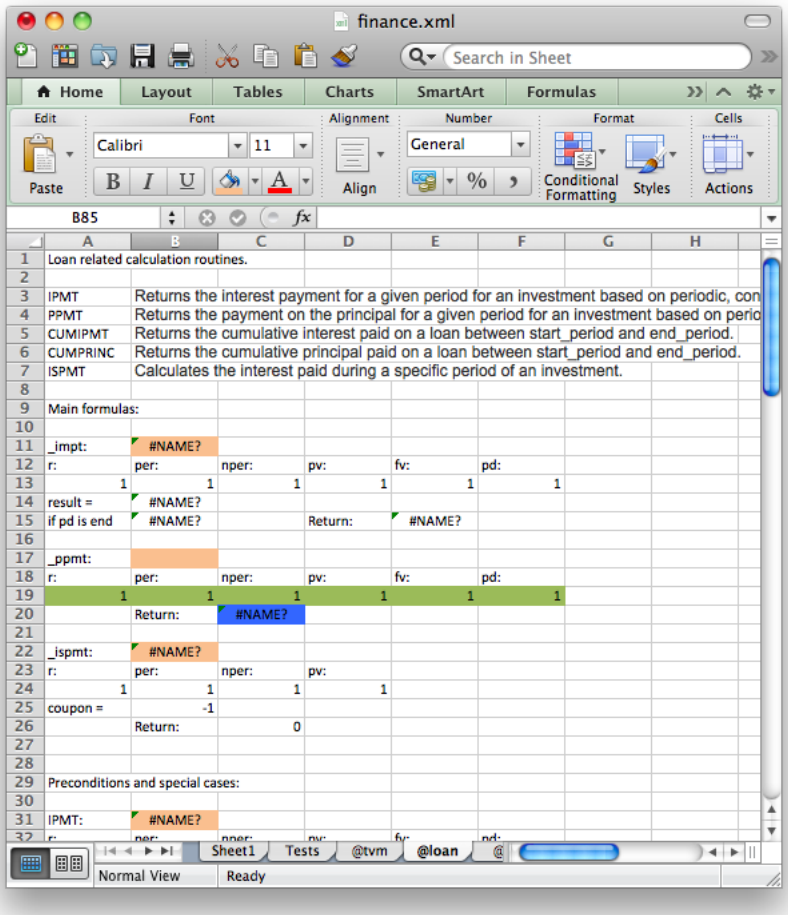


Figure 1: Screenshot of Excel for Mac 2011

expression evaluated on an equation such as bigger than > less than < and equality / inequality signs = or != or by using a group of boolean expressions with AND, OR and NOT functions to evaluate multiple boolean expressions.

Loops in Excel and similar spreadsheet engines all have to be performed using their extension languages such as VBA (Visual Basic for Applications) by Microsoft.

In Excel formulas can have relative or absolute references to other cells which has great effect when copying formulas around in a sheet. Absolute references written as `$A$1` to refer to cell A1 do not change on copy/paste, however relative references written simply as `A1` maintain the relative distance to the copied cell.[20][14]

Reusable abstractions do not exist in the formula language and users are bound to exploit

```
Function Die(n As Integer)
  Die = Application.WorksheetFunction.RandBetween(1, n)
End Function
```

Figure 2: VBA function of a die roll

```
=FLOOR(RAND()*6,1)
```

Figure 3: Excel Cell formula for a die roll

the clever copy paste functionality in Excel. Excel's copy paste functionality looks at the type of references whether they are relative or absolute. For relative references, Excel figures out the cell dependencies and updates cell references in the pasted cell. This mechanism also works the other way when you clip a cell that other cell reference to, dependent cells are updated to point to the new position.

This functionality leads to an easy way of reusing formulas multiple places spread across a workbook but leads to certain disadvantages regarding maintenance and bug fixing. One can think of a situation where a bug is found in an equation which has been pasted to ten other places in the workbook. Now that bug has to be corrected all places which leads to a maintenance nightmare for big workbooks.

Excel, Open Office and Gnumeric all support a feature allowing the user to define their own functions to accommodate the built in functions. However user defined functions can only be written in "alien languages" such as VBA (Microsoft Excel and OpenOffice).

These both have the disadvantage of complicated mechanics for communicating between VBA and the internal representation of the spreadsheet engine as a communication between the internal representation of the spreadsheet engine to the VBA runtime has to be performed. In Excel this communication is done through Microsofts Com architecture which slows down initial calls to VBA functions considerably. From a user perspective the language is alien compared to the formula language and requires extra effort for the user to master. Alien languages has a disadvantage with regarding to usability as mentioned in [11] as people are taken out of the spreadsheet metaphor and into a textual metaphor. Not only is this textual metaphor alien to the users but it also makes very little past knowledge from the formula metaphor reusable.

On Figure 2 and 3 you can see the difference of a simple function in VBA compared to a spreadsheet formula that performs the same calculation. Alone in this simple function an end user developer suddenly has to understand the concept of types, Assignments to variables and methods on objects.

2.2 Sheet Defined Functions

A sheet defined function is a user defined function that has been defined within the spreadsheet metaphor.

The concept of sheet defined functions has been proposed by both Nuñez[3] and Peyton-Jones, Blackwell and Burnett[11]

An approach that keeps sheet defined functions closely similar to existing spreadsheet usage has been suggested by Peyton-Jones[11]. This approach is based on a study made that shows that end-user programmers outnumber professional programmers and renders it infeasible to use traditional programming languages to express abstractions as usability for end-user programmers drop.

The main concern with traditional approaches (VBA) is that a user is taken out of the spreadsheet context and in to a context of an ordinary programming language. This has a high cost in terms of learning and reusability of knowledge. Users need to learn a secondary language as well as a set of accompanying tools for debugging and development. The idea is that it should be possible to define functions within certain function sheets which acts like any other worksheet. Styling and structure should be the same as normal worksheets.

Peyton-Jones [11] has decided not to allow recursive functions in the formula language because of consistency. The reason of this design choice is that recursive functions are more difficult to grasp for the regular spreadsheet user and is in itself difficult to read. Arguments against this choice are that a recursive function can be described in math and that various scientists both within biology, math and computer science are already used to the idea of recursion.

Nuñez took a slightly different approach augmenting the classical spreadsheet paradigm with his ViSSH implementation putting graphical elements into cells to represent sheet defined functions with arguments and its implementation. Nuñez uses Scheme as the language to express your functions supporting both recursion and higher order functions.

2.3 Funcalc

Funcalc is a modern implementation of a spreadsheet engine testing the concept of sheet defined functions as well as runtime code generation for optimized performance. Funcalc comes with a small set of core functions for doing various data manipulations.

Funcalc is an extension of Corecalc which was an interpretive implementation of a spreadsheet engine. Funcalc also introduces sheet defined functions to support functionality similar to that of VBA in Excel. Sheets denoted with an at(@) sign are so called function sheets where functions can be defined. The defined functions can then be used between all sheets in a workbook or (in the future) between workbooks.

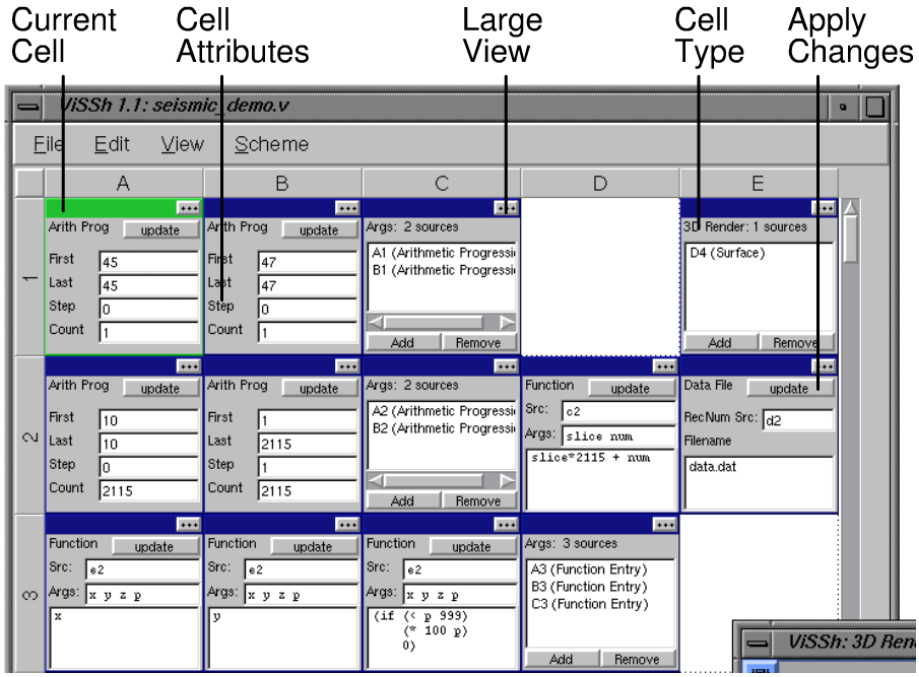


Figure 4: Sheet defined functions written in VISSH by Nuñez[3]

Functions that differ from Excels built-ins are `APPLY`, `DEFINE`, `EXTERN` and `CLOSURE`. `DEFINE` is used to define a function with the parameters function name, output cell and a list of input cells. `CLOSURE` is used to create a closure that can be passed as a parameter to another function hereby supporting higher order functions. APPLY is a function that applies a function value to the number of arguments given.

A closure is a mechanism for referencing non local variables outside the scope of the original caller. These values are bound to the closure at creation time and can be used when the function value is used for execution. Goalseek can then use the given function closure with the free variable being the interest rate.

Funcalc also include a few built-in higher order functions such as `MAP` and `REDUCE`.

`MAP(func, array)` is a function that maps the result of a function to a list. It is a generic function that supports both `MAP2` by `MAP(func, array, array)`, `MAP3` by `MAP(func, array, array, array)` etc. depending on the number of lists that is passed to it. `MAP` is currently the only possible way to apply calculations to a list and return the result in a new list in Funcalc. While `MAP` applies a function to a number of arrays, `REDUCE(fv, a0, arr)` applies a function to each element in an array and accumulates the result.

Funcalc does not include loops known from imperative languages but supports recursion to

achieve similar functionality.

In order for Funcalc to provide decent performance of functions an interpretive implementation is not an option. Funcalc takes advantage of Microsoft .NET which makes it possible to compile sheet defined functions to byte code at runtime. Recalculation times in Funcalc are comparable to those of Excel in many cases as proven by Iversen[9].

2.4 Financial Functions

Financial Functions is a part of all major spreadsheet programs today. These are heavily used in the bank sector to calculate bonds, interest rates, payback rates etc. These functions are used daily in the banking sector when we have to take both fixed rate loans and flex rate loans. We are told when our loan will be paid back and in how much we need to pay back each month. We also use these to figure out how many interests that run on top of the initial loan before it is paid back.

Other functions in the library are used to calculate Securities as a means for companies to raise capital which has certain advantages over traditional bank loans. This can be the demand for certain characteristics in the market or the lack of need for securing extra protection to banks for loans. Governments uses these as well to gain capital by issuing securities.

The financial library also contains functions for calculating depreciation of assets over time. You can connect this to for example housing where interior such as kitchens and bathrooms loose value over time.

The reason for choosing financial functions for implementation in Funcalc is:

- Spreadsheets are widely used in finance
- Excel's financial functions are already implemented in third party languages (F#)
- The function style of F# is similar to that of Funcalc.
- Financial functions has a size that can stress and locate bugs and missing features in Funcalc.

Excel currently has 52 financial functions implemented covering Bonds, Loan, Time value of money, Depreciation etc.

A list of functions with descriptions and implementation status in Funcalc can be found below. The descriptions are taken from the Microsoft documentation located at their office documentation website[15].

Function	Description	Implemented
ACCRINT	Returns the accrued interest for a security that pays periodic interest	No
ACCRINTM	Returns the accrued interest for a security that pays interest at maturity	No
AMORDEGRC	Returns the depreciation for each accounting period by using a depreciation coefficient	Yes
AMORLINC	Returns the depreciation for each accounting period	Yes
COUPDAYBS	Returns the number of days from the beginning of the coupon period to the settlement date	No
COUPDAYS	Returns the number of days in the coupon period that contains the settlement date	No
COUPDAYSNC	Returns the number of days from the settlement date to the next coupon date	No
COUPNCD	Returns the next coupon date after the settlement date	No
COUPNUM	Returns the number of coupons payable between the settlement date and maturity date	No
COUPPCD	Returns the previous coupon date before the settlement date	No
CUMIPMT	Returns the cumulative interest paid between two periods	Yes
CUMPRINC	Returns the cumulative principal paid on a loan between two periods	Yes
DB	Returns the depreciation of an asset for a specified period by using the fixed-declining balance method	Yes
DDB	Returns the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify	Yes
DISC	Returns the discount rate for a security	No
DOLLARDE	Converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number	No
DOLLARFR	Converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fraction	No
DURATION	Returns the annual duration of a security with periodic interest payments	No
EFFECT	Returns the effective annual interest rate	No
FV	Returns the future value of an investment	Yes
FVSCHEDULE	Returns the future value of an initial principal after applying a series of compound interest rates	Yes
INTRATE	Returns the interest rate for a fully invested security	No
IPMT	Returns the interest payment for an investment for a given period	Yes

Function	Description	Implemented
IRR	Returns the internal rate of return for a series of cash flows	Yes
ISPMT	Calculates the interest paid during a specific period of an investment	Yes
MDURATION	Returns the Macauley modified duration for a security with an assumed par value of $100	No
MIRR	Returns the internal rate of return where positive and negative cash flows are financed at different rates	Yes
NOMINAL	Returns the annual nominal interest rate	No
NPER	Returns the number of periods for an investment	Yes
NPV	Returns the net present value of an investment based on a series of periodic cash flows and a discount rate	Yes
ODDFPRICE	Returns the price per $100 face value of a security with an odd first period	No
ODDFYIELD	Returns the yield of a security with an odd first period	No
ODDLPRICE	Returns the price per $100 face value of a security with an odd last period	No
ODDLYIELD	Returns the yield of a security with an odd last period	No
PMT	Returns the periodic payment for an annuity	Yes
PPMT	Returns the payment on the principal for an investment for a given period	Yes
PRICE	Returns the price per $100 face value of a security that pays periodic interest	No
PRICEDISC	Returns the price per $100 face value of a discounted security	No
PRICEMAT	Returns the price per $100 face value of a security that pays interest at maturity	No
PV	Returns the present value of an investment	Yes
RATE	Returns the interest rate per period of an annuity	Yes
RECEIVED	Returns the amount received at maturity for a fully invested security	No
SLN	Returns the straight-line depreciation of an asset for one period	Yes
SYD	Returns the sum-of-years' digits depreciation of an asset for a specified period	Yes
TBILLEQ	Returns the bond-equivalent yield for a Treasury bill	No
TBILLPRICE	Returns the price per $100 face value for a Treasury bill	No
TBILLYIELD	Returns the yield for a Treasury bill	No

Function	Description	Implemented
VDB	Returns the depreciation of an asset for a specified or partial period by using a declining balance method	Yes
XIRR	Returns the internal rate of return for a schedule of cash flows that is not necessarily periodic	Yes
XNPV	Returns the net present value for a schedule of cash flows that is not necessarily periodic	Yes
YIELD	Returns the yield on a security that pays periodic interest	No
YIELDDISC	Returns the annual yield for a discounted security; for example, a Treasury bill	No
YIELDMAT	Returns the annual yield of a security that pays interest at maturity	No

Table 1: Financial functions included in Microsoft Excel 2010[15]. Financial functions implemented in this thesis are marked with Yes.

The reference implementation that this work will be based on is written in F# [1]. This reference implementation covers 50 financial functions and has been run against 201,349 automated test cases that matches the results of Excel version 12.

3 Related Literature

To better understand the context in which ideas for improvements can be based upon it is worth taking a look at how traditional languages solves design issues. On the same hand it is worth investigating what users best practices are in spreadsheets. This is looked upon in the coming two subsections.

3.1 Programming Language Design

When designing new programming languages there are a range of criteria to take into account.

Niklaus Wirth discusses in his paper [21] different approaches in programming languages. He mentions the importance of abstractions as part of the language to abstract from the underlying implementation. In the formula language integers, and floating point numbers are all abstracted away to just be a number. The syntax of the formula language is also kept simple as it does only contain the bare essentials to perform computations.

Many languages contain keywords to express loops, branches function definitions etc. The formula language does not contain any keywords and uses functions to express similar behavior.

Wirth sums up some lessons learned from early languages[21]:

- The designer must have a clear idea of the intended usage.
- The language should be kept as simple as possible and be provided with clear and readable documentation
- The notation must be familiar to existing standards.

The third point in particular notes the importance of keeping sheet defined functions familiar and in the setting of the Excel formula language.

Hoare[8] gives a list of hints to what to focus on in programming language design. Simplicity in a language gives the user an easy grasp of the language and connected to modularity a user will be able to use the language even without knowing everything. Modularity can be dangerous if not designed with simplicity in mind. It could lead to parts of code that the user does not understand and do not pick up easily[8] A language has to be readable for a human being and the syntax should not be optimized for the machine. However the syntax should still allow for accurate pinpointing of errors from the compiler. Comment conventions is according to Hoare a primary concern. Today most languages support both multiline comments and single line comments. In a spreadsheet context comments are merely cells with text in. This has the benefit that the user is free to structure his comments himself. This however is also a disadvantage and show a need of conventions internally in teams on how to write comments. The spreadsheet

metaphor does not support comments inline with function calls and other calculations. This is the preferred behavior based on Excel's hiding of the formulas as default.

3.2 End user Programming /End user development (EUD/EUP)

End user Development is a hot topic these days as more people use computers in their daily work and needs the ability to extend the functionality of the programs they use. The goal of EUD can therefore said to be: "Empowering end-users to develop and adapt systems themselves"[12]. The focus is therefore to make adaptation unobtrusive so that the primary task of the user is not disturbed by a cumbersome alien way of extending the functionality. The cognitive load for the user should be as small as possible when switching from using the software to adapting the software[12].

The trade-off in EUD is where to place the balance between the cost of learning and the scope of the language. Java and C++ can be seen as having high cost of learning and a high scope as well. End users however might end up being confused with the scope of these languages and tend to need solutions with lower cost of learning. Ideally the best solution would be a language having high scope and low cost of learning[7].

Examples on EUD can be seen both as textual languages and visual languages. Legos visual language for programming their Lego Mindstorms robots is a good example on how the task of applying logic to the robots is simplified.

Nardi and Miller writes in their paper The spreadsheet interface[16], that some of the properties that makes the spreadsheet interface work so well from an EUD point of view is the tabular view and the simplicity in the formula language. The language contains high level task-specific functions as well as very simple control constructs. They praise the balance between expressiveness and simplicity based on the low learning curve.

End user Computing (EUC) is not without issues. Today EUC is seen a lot within spreadsheets. There are a lot of Errors in applications written non programmers. However the error rate is not substantially higher than that of programmers that lack poor user development practices. Usually lack of testing is the reason for the error rates in these applications. According to Panko and Port[17] many corporations see some of these end user developed applications as mission critical for the business which is positive. However mission critical software should hopefully not contain errors or being unavailable at times. The relevance of EUC is clearly shown in the paper as Panko and Port sum up that a large financial services company had 180000 spreadsheets which they were dependent upon[17]. A range of other examples are also shown such as a large government agency had 630000 spreadsheets and 2500 Access databases. This is a massive amount of spreadsheets developed by end users. The paper end up concluding that 2-

5% of all formula cells in a spreadsheet contains errors. And since cells most of the time are dependent on each other it leads to wrong numbers on the bottom line. What is most scary is that with these numbers, nearly all larger spreadsheets contain errors. Testing would help solving the amount of errors however end users rarely performs extensive testing of their spreadsheets. The corporations also tend not to have strong policies on this subject.

3.3 Spreadsheet best practices

Spreadsheets are easy to get to and it is intimidating to just start filling in constants and formulas. However the grid based design of spreadsheets also allow for messy spreadsheets as there is little control over where the user can put in data.

This in terms can lead to spreadsheets containing a lot of errors. A messy spreadsheet is harder to grasp if cells refer to each other in no organized manner. However with a nicely organized spreadsheet one could say that the visual complexity is minimized. In a paper by Barry Boehm et al.[4] they highlight ways of reducing errors in software development which in terms can be directly translated to spreadsheet development. A good analysis and design of your problem should be done before you start implementing your solution. It is good to have a clear vision of what you want to solve with your spreadsheet and test it out on paper before starting to implement your idea in the spreadsheet. For important parts of the spreadsheets peer reviews can be used to minimize errors. Boehm states that 60% of defects can be caught by peer reviews.

A set of 50+ best practices has been suggested in a paper by Philip l. Bewig. [2]. As Philip mentions, spreadsheets has a history of being prone to errors that can have dire consequences economically. Some of the best practices Bewig mentions are:

Think before you write Makes the point that you should think out your solution before starting to implement it and sketch it out first.

Make your spreadsheet as simple as possible, but no simpler This point suggest to hide complex logic in user-defined functions.

Keep input, logic, and reports in separate sections of a spreadsheet Bewig mentions that it is important to keep things separated in a spreadsheet. So presentation for the end user of the spreadsheet is separated from the underlying implementation that does the calculations. He suggest that you split the sections in tabs and if this is not possible, arranging the structure so that insertion of rows or columns does not insert other areas of the spreadsheet. Bewig also calls separate worksheets for false modularity as cells inherently are all accessible through the global scope and is therefore not hidden.

Trap formula errors Unexpected errors should be propagated but formula errors should be converted into appropriate errors. The example given is that a #DIV/0 error should be thrown as an error saying something about the input value instead of giving the impression that it is the implementation that is at fault.

Break up a complicated expression into multiple cells with intermediate results A complex expression that is written in one cell can be difficult to read and it is therefore suggested to split this into multiple cells. Also intermediate results can be useful in terms of assertions.

Build a library of frequently-used macros and functions Bewig suggest that one should create a library of functions you frequently use in a separate workbook but also warns that these functions are not available to other users unless you ship this workbook with your other spreadsheets.

These practices will be used as a base to implement sheet defined financial functions.

4 Implementation

This implementation is based upon an F# implementation of financial functions matching those implemented in Excel[1].

The resulting workbook of the implementation is split in function sheets named according to the group they fall under. These groups are based on the grouping done in the F# reference implementation. The workbook contains 24 fully implemented financial functions along with a range of partially implemented functions. The total number of functions in the workbook is above 100 when helper functions are taken into account and 509000 cells containing either data or formulas.

The general convention of structuring the functions can be seen on figure 5. It structures arguments in a column followed by optional local variables for use inside the function. An implementation (IMPL) column is used to contain the main logic of a function. Arguments and variables are annotated with comments in the cell left of them. These names maps to the names in the F# reference implementation. The return cell is always to be found in the IMPL column and colored darker blue by Funcalc.

CUMPRINC:	FUN CU...				_ccumprinc_rec	FUN...			
ARGS:		VARS:	IMPL:		ARGS:		VARS		IMPL
r	1		-0,5		startPeriod	1	s	123	-3,5
nper	1		-0,5		endPeriod	3	accresult	-0,5	
pv	1		-0,5		r	1			
startPeriod	1		-0,5		nper	1			
endPeriod	1		-0,5		pv	1			
pd	1		-0,5		pd	1			
			-0,5		acc	0			
			-0,5		index	1			
			-0,5						
			-0,5						

Figure 5: General structure of a function in the workbook finance. Shows a function `CUMPRINC` to the left and its helper function cumprinc_rec to the right with columns of arguments, internal variables and the implementation. The return value is always the first cell in the implementation column.

When a function contains an inner function, the inner function is written to the right of the main function to clarify that it is not meant to be used outside. On Figure 5, `_cumprinc_rec` is such an inner function. Empty cells are used to separate functions as well as layout of functions.

The workbook contains a range of different spreadsheets which are organized similar to the F# implementation.

- @DAYCOUNTBASIS is a worksheet containing helper functions for date calculations mostly related to bonds functions. This sheet contains the different financial calendars that exist.

- @TVM contains functions related to Time Value of Money calculations.
- @LOAN contains functions related to calculations of Loans, payback rate etc.
- @IRR contains functions related to calculations on the Internal rate of return
- @DEPRECIATION contains functions related to calculations on assets
- @COMMON contains common functions used across all other function sheets that does not belong to any specific group
- @GOALSEEK contains a Funcalc implementation of Goalseek provided by Peter Sestoft.
- @CALENDAR contains an implementation of Gregorian calendar functions provided by Peter Sestoft but originally developed by two MSc students Hui Xu and Mohammad Mainul Liton[13].

A full list of financial functions in excel and which are implemented in Funcalc can be seen in section 2.4.

Some of the functions are similar in implementation and do not give extra insights in implementation issues or inconveniences, only functions of interest will be listed below. An example is that between FV and PV these share many of the same helper functions and one does not bring any extra value for the observations in section 6.

Underscore (_) is used as a convention for naming helper functions. A further explanation for this choice can be found in section 6 Functions named using this convention are not made to be used outside the Financial Functions library and should be considered as "Private" functions. Most functions containing "rec" in their name are to be considered inner functions only relevant for the matching function without "rec" in its name.

Observations spotted during the implementation are discussed in the Observations chapter and referred to under the individual implementation descriptions.

A thorough explanation of how the actual formulas work for each function will not be described in this section and is considered to be general knowledge for finance students after their first year. The formulas used in this implementation are a reimplementation of those of F# but deeper explanations of these can be found in Fundamentals of Corporate Finance[5] and in Financial management and analysis[18].

4.1 FV - Future Value

FV is one of the simpler among the financial functions. It calculates the future value of an investment based on fixed payments and a fixed interest rate. The future value of an asset for a given period in the future is calculated by assuming a fixed rate of return.

Implementation of FV consist of a range of helper functions, some which are used across other financial functions, as well as the main FV function which does input error checking.

$$-(pv * (1+r)^{nper} + pmt * \frac{(1+r*pd)*(1-\frac{1}{(1+r)^{nper}})}{r} * (1+r)^{nper}$$

The complete formula for calculating the future value is shown above. This function however is only valid if the rate is bigger than 0 as shown in the implementation of the private function _annuitycertainpvfactor below. This formula has been split into a series of helper functions which can be seen below.

_ **FVFACTOR** takes two arguments, Rate and Number of periods. It is a "Private" function that is used across multiple worksheets in the financial functions library and returns $1 + r^{nper}$ from the formula above.

```
Function _FVFACTOR
A3 = <input rate>
B3 = <input nper>
B4 = <output>
   = 1+A3^B3
```

_**PVFACTOR** takes two arguments, Rate and Number of periods shown below.

```
Function _PVFACTOR
A8 = <input rate>
B8 = <input nper>
B9 = <output>
   = 1/_fvfactor(A8,B8)
```

_**ANNUITYCERTAINPVFACTOR** shown below takes three arguments, Rate, Number of periods and Payment due. Payment due is a flag that tells if payment is due in the end of the month (0) or in the start of a month (1).

```
Function _ANNUITYCERTAINPVFACTOR
A13 = <input rate>
B13 = <input nper>
C13 = <input paymentDue>
B14 = <output>
    = IF(A13 = 0, B13, (1 + A13*C13)*(1- _pvfactor(A13,B13))/A13)
```

_**ANNUITYCERTAINFVFACTOR** shown below takes three arguments, Rate, Number of periods and Payment due.

```
Function _ANNUITYCERTAINFVFACTOR
A18 = <input rate>
B18 = <input nper>
C18 = <input paymentDue>
B19 = <output>
    = _annuitycertainpvfactor(A18, B18, C18) * _fvfactor(A18, B18)
```

_FV shown below takes five arguments, Rate, Number of periods, Number of payments, Payment and Type.

```
Function _FV
A38 = <input rate>
B38 = <input nper>
C38 = <input pmt>
D38 = <input pv>
E38 = <input type>
B39 = <output>
    = -(D38*_fvfactor(A38, B38) + C38
        * _annuitycertainfvFactor(A38, B38, E38))
```

This function is the core function for calculating the future value and is called by the public `FV` function described later.

The helper functions:
`_fvfactor`, `_pvfactor`, `_annuitycertainpvfactor`, `_annuitycertainfvfactor` and `_fv` are used across the entire @tvm function sheet.

They are also referenced from other function sheets within the workbook as other calculations depends on these. These helper functions are primarily written to prevent code duplication which in terms minimizes chances of errors and keeping code one place in cases where there is detected errors. The core function `_FV` is used to calculate Future Value which takes 5 arguments. Rate, Number of Periods, PMT, Present Value and a Payment Due 0 or 1 for End or beginning of month. The public function of FV is listed below and contains mimicing of pattern matching built from if/else constructs since Funcalc do not contain a constructs for pattern matching. When errors are met, an `ERR` function is used that correctly propagates the given error up the hierarchy. Here they are used to give the user proper error messages when input values are wrong.

In Excel, both Present Value and Type is optional. Both Present Value and Type is assumed to have the value 0 if omitted. However this functionality is not supported in the current version of Funcalc and has to be declared explicitly.

FV (rate,nper,pmt,pv,type) takes five arguments, Rate, Number of periods, Number of payments, Payment and Type. It does error checking and returns the future value based on the input.

```
Function FV
B92 = <input rate>
B93 = <input nper>
B94 = <input pmt>
B95 = <input pv>
B96 = <input type>
D92 = <output>
    = if(_raisable(B92,B93),D93,ERR("r is not raisable to nper")
D93 = if(or(not(B92=-1),and(B92=-1,BB93>0),D94,
         ERR("r cannot be -100% when nper is <= 0")))
D94 = if(or(not(B94=0),not(B95=0)),D95,
         ERR("pmt or pv need to be different from 0"))
D95 = if(and(B92=-1,B96=1), -(B95*_fvfactor(B92,B93)), D96)
D96 = if(and(B92=-1,B96=0), -(B95*_fvfactor(B92,B93)+B94), D97)
D97 = _FV(B92,B93,B94,B95,B96)
```

If we look at cell D92 above it should be read that if rate is raisable to nper then evaluate D93, otherwise throw an error message to the caller of the function. In D95 and D96 we handle special cases and in D97 we calculate FV.

Observations during implementation of FV are Pattern Matching Section 6.1, Naming Conventions Section 6.3, Private Functions Section6.2, Constants Section6.5 and Optional Arguments Section 6.4.

4.2 Rate

Rate calculates the interest rate per period for an annuity. Rate takes six arguments: Number of Periods(nper), Payment(pmt), Present Value(pv), Future Value(fv), Payment Due(type) and Guess. Payment is the amount of payment done each period and Payment Due is either 0 or 1 depending of whether the payment is due in start of the month(1) or in the end of the month(0). In the Excel implementation of RATE, fv, type and guess are optional. The default values in Excel are for fv and type 0 and guess is assumed to be 10% if omitted.

```
Function _SFV:
D66 = <definition>
    = DEFINE(D92,B93,B94,B95,B96,B92)

Function RATE:
```

```
B63 = <input nper>
B64 = <input pmt>
B65 = <input pv>
B66 = <input fv>
B67 = <input type>
B68 = <input guess>
D68 = <output>
    = GOALSEEK(CLOSURE("sfv",B63,B64,B65,B67),B66,B68)
```

RATE is calculated using `GOALSEEK(funcval, targetval,initialval)`. Goalseek is a function that iteratively tries to find an unknown input value based on a known output value. In this current example we are trying to find the interest rate based on the future value(targetval) we specify. The current implementation, copied from workbook testsdf.xml which is included with the current Funcalc source, initially developed by Peter Sestoft is implemented using Divide and Conquer to estimate the best result. The F# library differs by implementing both newton and bisection root finding algorithms to find the result. First the fast newton algorithm is tried and if the result is not sensible, a slower but more precise bisection algorithm is tried.

Looking at the function `_SFV` above, it is a redeclaration of `_FV` but with arguments ordered differently. This is done as Funcalc not yet supports arbitrary arguments to be omitted and since we want to find rate, we will need to have that as the last argument.

Since `GOALSEEK` is a higher order function we need to use `CLOSURE(function name, args ... )` to generate a function closure.

Observations during implementation of RATE are: Arguments Section 6.4, Naming Convensions Section 6.3, Function Hiding Section 6.2 and Constants Section 6.5

4.3 FVSCHEDULE

FVSCHEDULE calculates a future value based on a series of interest rates. The use of this function when having a variable interest rate to calculate the future value for a given asset.

FVSCHEDULE takes two arguments, A principal value and an array of variable interest rates. Since Funcalc uses a functional language we are using recursion to represent loops.

In order to recursively iterate over the interest rates, we use a helper function `_REFCV` to accumulate the result and a public function `FVSCHEDULE` that calls the helper function with an argument of array position. Since arrays are cell areas they can either be Horizontal (1,x), Vertical (x,1) or Two dimensional (x,y). This implementation only handles cell areas defined as (x,1).

Looking at cell E77 in the _RECFV implementation below, it returns the accumulated sum if we reach the maximum number of rows in the array. Otherwise we call itself with an accumulator of itself multiplied with the next index in the array and increment the array position.

```
Function _RECFV
E76 = <input accumulator>
E75 = <input data>
E74 = <input position>
E77 = <output>
    = IF(E74>ROWS(E75), E76,
         _recfv(E76*(1+INDEX(E75,E74,1)),E75,E74+1))

Function FVSCHEDULE
B74 = <input pricipal>
B75 = <input schedule>
E77 = <output>
    = _RECFV(B74,B75,1)
```

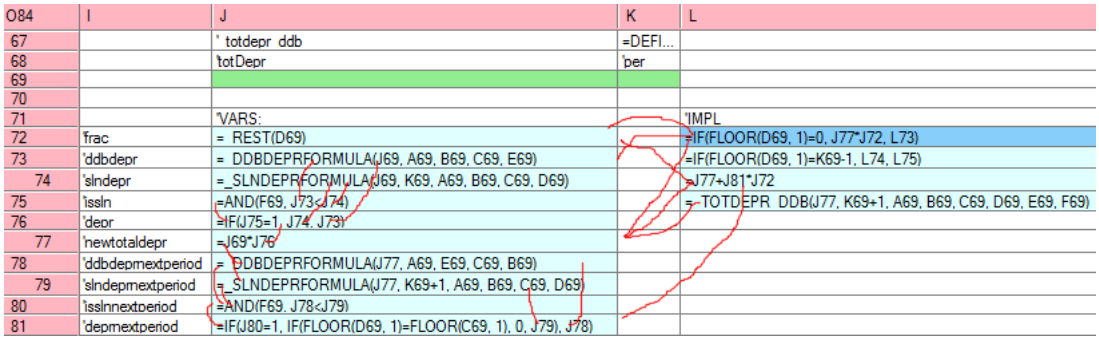
Noteworthy observations in the implementation of FVSCHEDULE are Recursion discussed in Section 6.7 and Arrays discussed in Section6.8

4.4 Total Depreciation

Total Depreciation _totaldepr is a helper function for several of the functions in the Depreciation sheet such as DDB and VDB. It consist of a recursive inner function _totaldepr_ddb which also has two inner functions _ddbdeprformula and _slndeprformula. The implementation of _totaldepr_ddb shows a more rigorous implementation of recursion with a long range of internal variables and many referencing cells.

Sometimes referencing cells can get quite hectic which is visualized on Figure 6 which contain many cell dependencies. The implementation column initiates the tree of inter depended cells referring to the variables defined in the Vars column. Since variables can not be named the variable name corresponding to that of the F# implementation is written to the left of the cell containing the formula.

Both the function _totaldepr_ddb and its inner functions use lambda lifting as a means to refer to the arguments from their caller functions. Lambda lifting is necessary because all functions are located in the global scope and can not refer to arguments in other functions. Hence inner functions is not supported[10]. If we look at the function definitions below we can see that they all share cells.



O84	I	J	K	L
67		' totdepr_ddb	=DEFI...	
68		'totDepr	'per	
69				
70				
71		'VARS:		'IMPL
72	'frac	= REST(D69)		=IF(FLOOR(D69, 1)=0, J77*J72, L73)
73	'ddbdepr	= DDBDEPRFORMULA(J69, A69, B69, C69, E69)		=IF(FLOOR(D69, 1)=K69-1, L74, L75)
74	'slndepr	=_SLNDEPRFORMULA(J69, K69, A69, B69, C69, D69)		=J77+J81*J72
75	'issln	=AND(F69, J73<J74)		=_TOTDEPR_DDB(J77, K69+1, A69, B69, C69, D69, E69, F69)
76	'depr	=IF(J75=1, J74, J73)		
77	'newtotaldepr	=J69+J76		
78	'ddbdepnextperiod	=_DDBDEPRFORMULA(J77, A69, E69, C69, B69)		
79	'slndepnextperiod	=_SLNDEPRFORMULA(J77, K69+1, A69, B69, C69, D69)		
80	'isslnnextperiod	=AND(F69, J78<J79)		
81	'depnextperiod	=IF(J80=1, IF(FLOOR(D69, 1)=FLOOR(C69, 1), 0, J79), J78)		

Figure 6: View of cell references with totaldepr's inner function _totaldepr_ddb from Funcalc. Only internal cells are visualized and references to function arguments are left out.

```
DEFINE("_totaldepr", B70, A69, B69, C69, D69, E69, F69)
DEFINE("_totaldepr_ddb", L72, J69, K69, A69, B69, C69, D69, E69, F69)
DEFINE("_ddbdeprformula", O68, J69, A69, B69, C69, E69)
DEFINE("_slndeprformula", O73, J69, N72, A69, B69, C69, D69)
```

It is important to note that cell B69 in `_totaldepr` and cell B69 in `_totaldepr_ddb` does not share the same value in the underlying representation as visually. B69 should be seen merely as a placeholder reference for the argument taken in the definition so the implementation of the function can refer to the argument. If B69 is not passed onto the inner functions the value that will be referenced when calling the function will be treated as the constant value that is represented visually in B69 instead of the argument that was passed when the outer function was called. Currently it is suggested to use unique cells for function arguments to avert this misunderstanding however a visual solution to scoping is discussed in section 6.2.

The main observations related to Total Depreciation are Scoping Section 6.2 and Visual representation of recursive functions 6.7

4.5 Modified Internal Rate of Return

Modified Internal Rate of Return `MIRR (cfs, financeRate, reinvestRate)` found in sheet @IRR takes three arguments. The first argument being a list of payments, financeRate the interest you pay and reinvestRate the interest you get on the cash flows when reinvesting in them. It calculates the modified internal rate of return for a series of cash flows considering both the cost of investing and the interest of the reinvestments.

Part of the error checking of `MIRR` is using the higher order function `MAP` to get all negative payments for use in the calculation of Net Present Value(NPV) in cell D67. To filter out positive

values in Funcalc we are defining a helper function _mirr_map which does the comparison for MAP. Note that Error message text has been omitted from the code below.

```
Function _MIRR_MAP
K68 = <input cf>
L68 = <output>
    = if(K68<0,K68,0)

Function MIRR
B67 = <input cfs>
B68 = <input financeRate>
B69 = <input reinvestRate>
D67 = <output>
    = if(not(B68=-1),D68, err("..."))
    = if(not(B69=-1),D69, err("..."))
    = if(not(ROWS(B67)=1),D70, err("..."))
    = if(not(npv(B68,
           map(closure("_mirr_map"),B67))=0),
           D71,err("..."))
    = _mirr(B67, B68, B69)
```

Since it is both tedious and polluting the worksheet declaring such a simple function, a form of lambda notation is suggested in the observations chapter 6.6.

4.6 Financial Calendars

In finance there are five types of calendars which has influences different calculations related to Bonds. These calendars are USPSA30_360, ACTUAL/ACTUAL, ACTUAL360, ACTUAL365 and EUROP30_360. Not only are the length of a year different in these but also whether to include leap years, length of months etc. These calendars are represented using an interface in the F# library. One can get a particular calendar by calling a function `dayCount` with a given basis(type of calendar). In Funcalc inheritance is not supported so member functions are implemented with an extra argument.

Looking at an example of the DaysInYear member function in F# it looks like

```
dc.DaysInYear(issue, settl) = 365.
```

whereas in Funcalc it is expressed as shown below.

bs_daysinyear

```
B32 = <input basis>
B33 = <input issue>
B34 = <input settl>
E32 = <output>
    = IF(B32 = 0, 360, E33)
E33 = IF(B32 = 1, ..., E34)
E34 = IF(B32 = 2, 360, ...)
```

As seen, a comparison of the basis has to be done in each function related to the calendar operations instead. These calendar functions are only partially implemented and can be found in the sheet named @DAYCOUNTBASIS.

4.7 CUMIPMT & CUMPRINC

Both `CUMIPMT` and `CUMPRINC` uses fold and list comprehensions in their F# implementation. To create a list of specific numbers, one can write a recursive function that returns an array containing these numbers.

This function takes 3 arguments. Start, End and Step. It calls a helper function called _genlist_rec which recursively loops through from start to end and builds up an array. The array is built up by concatenating the result of the previous iteration with an extra element.

_genlist_pos

```
F28 = <input start>
F29 = <input end>
F30 = <input step>
F31 = <input data>
H28 = INDEX(F31, ROWS(F31), 1)
I28 = <output>
    = IF(H28+F30<=F29, _GENLIST_POS(F28,F29,F30,VCAT(F31, H28+F30)),F31)
```

Looking at the implementation above it has a runtime complexity $O(N^2)$ as `VCAT` creates a a new matrix with length of F31 + 1. However a $O(N)$ solution is possible in Funcalc using the `VSCAN (fv, c1, n)` function.

```
Function CUMIPMT
G61:G68 = <input startPeriod,endPeriod,r,nper, pv, pd, acc, index>
I61 = <var s>
      IF(CEILING(G61, 1)<=FLOOR(G62, 1), _GENLIST(CEILING(G61, 1),
      FLOOR(G62, 1), 1), _GENLIST(FLOOR(G62, 1), CEILING(G61, 1), -1))
I62 = <var accresult>
      G67+_IPMT(G63, INDEX(I61, G68, 1), G64, G65, 0, G66)
```

```
J61 = <output>
      IF(G68<=ROWS(I61),
      _CUMIPMT_REC(G61, G62, G63, G64, G65, G66, I62, G68+1), G67)
```

The F# implementation uses fold to calculate the result of `CUMIPMT`. I have implemented this by write a specialized function that recursively runs over the array and and accumulates the final result. The accumulator can be seen in cell I62 above with the returned output in cell J61 above.

Funcalc however also comes with the built in functions `MAP` and `RECUCE` where `REDUCE` can be used to perform the same calculation but in a cleaner way. Since `REDUCE` only takes functions that has two values one has to specify these at the end of the argument list. If we have a function `CUMIPMT_FOLD` shown below, then `REDUCE` can be called with:

```
REDUCE(CLOSURE("CUMIPMT_FOLD",G61,...,G66),0,I61).
```

where I61 corresponds to the formula in I61 in the above CUMIPMT function. Since the last two arguments is left out in the closure, we respect the two argument rule set by `REDUCE`.

```
Function CUMIPMT_FOLD
G61:G68 = <input startPeriod,endPeriod,r,nper, pv, pd, acc, index>
J61     = G67+ _IPMT(G63, G68, G64, G65, 0, G66)
```

5 Testing

5.1 Testing Correctness

To test the correctness of the implemented financial functions i have run a long range of tests against the functions. The test values used for the tests are extracted from the F# library and imported into the workbook with one test sheet for each group of functions. The total amount of tests executed is 52931 with errors in 16816 of these. A more thorough explanation of the type of errors is explained later.

During implementation of these functions many more errors appeared. These have been found through the unit tests and corrected. Debugging these functions has been cumbersome both due to missing named variables but also due to error propagation in Funcalc not telling which cell that initially caused the error. I consider this missing feature a bug in the current implementation of Funcalc.

The F# library has an extensive test suite that runs it's tests in an automated manner against Excels built in functions. These tests are based on combinations of a small set of values with some functions tested more thoroughly than others. The test suite only covers correct input and compares it with the result of Excel.

	A	B	C	D	E	F	G	H	I	J
1	'IPMT								Total:	4256
2	rate	per	nper	pv	fv	type	Excel Builtin	Result	Matches:	4256
3	-1,5	1	1	-300	-300	1	0	0	1	
4	-1,5	1	1	-300	-100	1	0	0	1	
5	-1,5	1	1	-300	-5,4	1	0	0	1	
6	-1,5	1	1	-300	0	1	0	0	1	
7	-1,5	1	1	-300	100	1	0	0	1	
8	-1,5	1	1	-300	150,5	1	0	0	1	

Figure 7: Screenshot of testing functions in Funcalc. All test cases are presented in rows and a sum of results are calculated in the top right corner.

To get decent coverage of my financial functions i modified the test suite to dump the combinations of arguments computed by the test-suite along with the computed result of the Excel function executed by F#. Output for each function has then been imported into Finance.xml as shown on Figure 7. First the range of input is written, followed by the result of Excel. A call to the sheet defined function is then performed in column H and compared with column G in column I. A match is present if the formula in column I shown below returns true.

$$x = y \vee \frac{|x-y|}{|x|+|y|} < \epsilon \vee x = 0 \vee y = 0 \wedge |x-y| < \epsilon$$

This formula checks for equality of two cells with a precision of ϵ set to 1^{-9}.

In Excel this formula gives an unexpected behavior since the OR function evaluates all inputs before returning, resulting in #DIV0 errors in cases where $x = 0 \wedge y = 0$. Funcalc implements the OR function with early stopping, not evaluating the rest of the expressions when it meets an expression that evaluates to TRUE.

Function	Tests	Errors
Tvm		
PV	1783	0
FV	796	0
PMT	1750	0
NPER	853	0
RATE	4	1
FVSCHEDULE	12	0
Loan		
IPMT	4256	0
PPMT	4130	0
CUMIPMT	208	36
CUMPRINC	208	36
ISPMT	624	0
Irr		
IRR	9	4
NPV	21	21
MIRR	147	0
XNPV	1	1
XIRR	18	15
Depreciation		
DB	396	132
SLN	24	0
SYD	132	0
DDB	456	246
VDB	2544	1924
AMORLINC	11519	1440
AMORDEGRC	23040	12960
Total	52931	16816

Table 2: This table shows the number of unit tests run against these functions and the number of errors the unit tests has found.

Table 2 above shows errors in RATE. These errors are are result of `RATE` using `GOALSEEK` to iteratively find the unknown input value that gives the known result. As there is no documen-

tation on how GOALSEEK is implemented in Excel we can not make sure that they are equal. Differences in algorithms can be a result of the mismatching results. If RATE is tested with a precision of 1E-7 then results match up with Excel. Both `CUMIPMT` and `CUMPRINC` has errors which is assumed to be caused by Payment Due set to end of month. Errors in `AMORDEGRC` is a result of a partial implementation of the financial calendar. None of these errors is caused by wrong calculations. Errors in `XIRR` are caused by NumErrors reported by Funcalc. `XIRR` uses `GOALSEEK` internally to calculate the result. `DB` errors are caused when Period is above or equal 2. These are Both `VDB` and `DDB` uses a helper function total depreciation which is the suspect of the errors. This helper function is explained in more detail in Section 4.4

5.2 Testing Performance

Performance testing is done by comparing the performance of funcalc using the built-in `BENCHMARK` function which takes a closure to the function that needs to be benchmarked along with how many iterations it should run. It returns an average of these iterations. Each function has been tested with 5*100000 iterations as shown on Figure 8.

The performance tests has been run on an Intel Core2 Duo E6550 @ 2.33GHz with 2.00GB RAM running Windows 7 Professional SP1 64-bit.

Excel does not come with benchmark tools however benchmarking can be simulated through VBA. Since VBA has a big overhead in terms of runtime one will have to use a clever mechanism to take this overhead into account.

To overcome this overhead, a calculation on a simple function that is expected to have almost instant execution time is performed. The time it takes for this simple function to execute is subtracted from the function that is measured against and the delta time between the two shows a rough estimate on how long time it takes for a given function. Sestoft has provided a spreadsheet that does this which i have modified to test the built-in financial functions running time.

The resolution of the timer in VBA is rather poor so using less than 10.000 iteration will result in a bigger variation in the results while too many iterations makes excel crash. To get a good estimation of the running time, 5*100000 cycles has been run and compared to the Funcalc results as shown in Table 3.

Function	Excel Ns	Funcalc Ns
Tvm		
PV	1461	804
FV	1445	1138
NPER	1055	472
RATE	2297	44864

PMT	1523	664
FVSCHEDULE	2960	928
Loan		
IPMT	1593	1732
PPMT	1805	1292
CUMIPMT	3117	3400
CUMPRINC	2742	4072
ISPMT	468	170
Irr		
IRR	4750	79804
NPV	2156	2060
MIRR	3515	8328
XNPV	-	-
XIRR	-	-
Depreciation		
DB	-	-
SLN	125	158
SYD	453	212
DDB	-	-
AMORLINC	14921	2054
AMORDEGRC	16343	4444

Table 3: This table shows the running time in nano seconds in Excel and Funcalc. All numbers are calculated by averaging 500.000 function calls.

Some tests have been omitted based on too many errors in the correctness test.

In general, Funcalc performs better than Excel for most of the functions as seen in Table 3. It is difficult to argue for the exact reasons of this performance difference, however in some cases we can assume the reason for the difference. If we look at `RATE` and `IRR` these both make use of the `GOALSEEK` function provided by Sestoft. Excels implementation performs this calculation much faster which is believed to be based on the choice of algorithm. Difference in precision might also be the reason behind the extra run time cost as Sestofts `GOALSEEK` might compute with unnecessary much precision.

Two other functions are worth noting namely `AMORLINC` and `AMORDEGRC` which are considerably faster than Excels built-in functions. The Funcalc implementation expresses dates using financial calendar functions built on top of gregorian calendar sheet defined functions all of which are calculations on numbers as described in Section 4.6. Excel on the other hand is believed to use Date objects for calculation on dates. The cost of instantiating these objects are believed to have a certain overhead.

BenchMark	Iterations:	100000	Optimal is 10...				
	Average	closure	bench	bench	bench	bench	bench
PV	2240	PV(-1,5,-2,50,-30...	2400	2040	2410	1960	2390
FV	2418	FV(-1,5,-2,50,-30...	2270	2520	2480	2450	2370
NPER	1186	NPER(-0,4,50,-3...	1170	1340	1110	1190	1120
RATE	77314	RATE(5,20,120,-...	92180	97950	0	96240	100200
PMT	1694	PMT(-0,4,-2,-300,...	1700	1500	1800	1690	1780
FVSCHEDULE	2666	FVSCHEDULE(-3...	2710	2610	3190	2470	2350
Loan:							
IPMT	4960	IPMT(-0,4,1,10,-3...	4420	4340	4620	6760	4660
PPMT	3764	PPMT(-0,4,1,1,-3...	3270	2930	3430	6040	3150
CUMIPMT	9042	CUMIPMT(0,6,2,...	7950	7730	9710	11580	8240
CUMPRINC	10442	CUMPRINC(0,6,...	10060	9490	12750	9770	10140
ISPMT	436	ISPMT(-1,5,1,2,-...	380	390	630	400	380
Irr:							
IRR	#ERR: ...		#ERR: ArgT...	#ERR: ...	#ERR:...	#ER...	#ERR:...
NPV	6656	NPV(-1,5,-20001...	6610	6400	6830	7220	6220
MIRR	#ERR: ...		#ERR: ArgT...	#ERR: ...	#ERR:...	#ER...	#ERR:...
XNPV	#ERR: ...		#ERR: ArgT...	#ERR: ...	#ERR:...	#ER...	#ERR:...
XIRR	#ERR: ...		#ERR: ArgT...	#ERR: ...	#ERR:...	#ER...	#ERR:...
Depreciation:							
DB	#ERR: ...		#ERR: ArgT...	#ERR: ...	#ERR:...	#ER...	#ERR:...
SLN	486	SLN(100,10,12,7)	470	500	420	600	440

Figure 8: Screenshot of benchmarking in Funcalc. Each function is ran 500000 times and an average is calculated.

6 Observations

During development a range of observations has been spotted followed by solutions that would help enhance Funcalc.

Conventions within structure are suggested as well as within naming conventions. With naming conventions a particular important one is naming of private functions which start their name with an underscore (_). Functions is suggested to be written in column layout and function scope is suggested to be solved visually with a box surrounding these. A solution to visualizing recursion in functions is also presented.

6.1 Pattern matching

Pattern matching can be a good abstraction in cases where different branches are possible. In my implementation it is used by means of if else constructs to do error checking on functions. This is represented on figure 9 as a range of cells in a column which break out and returns a given error if a pattern is not matching. I see this as the current way of representing such situations in

Funcalc. It matches the best practices in spreadsheet development which is mentioned in section 3.3. The use of multiple cells in a column makes it easier readable than if everything was written in one cell.

	A	B	D
46	'PPMT:	=DEFINE("P...	
47	'ARGS:		'IMPL:
48	'r	1	=IF(_RAISABLE(B48, B50), D49, "r is not raisable to nper (r is negative and nper not an integer)")
49	'per	1	=IF(_RAISABLE(B48, B49-1), D50, "r is not raisable to (per - 1) (r is negative and nper not an integer)")
50	'nper	1	=IF(OR(NOT(B52=0), NOT(B51=0)), D51, "fv or pv needs to be different from zero")
51	'pv	1	=IF(OR(NOT(B48=-1), AND(B48=-1, B49>1, B50>1, B53=0)), D52, "r cannot be -100% when nper is <= 0")
52	'fv	1	=IF(NOT(_ANNUITYCERTAINPVFACTOR(B48, B50, B53)=0), D53, "1 * pd + 1 - (1 / (1 + r)^nper) / nper has to be <> ...
53	'pd	1	=IF(AND(B49>=1, B49<=B50), D54, "per must be in the range 1 to nper")
54			=IF(B50>0, D55, "nper must be more than 0")
55			=IF(AND(B49=1, B53=1), 0, D56)
56			=IF(B48=-1, -B52, D57)
57			=_IPMT(B48, B49, B50, B51, B52, B53)

Figure 9: A representation of pattern matching using if else constructs.

Comparing Figure 9 to a traditional language it can be noted that the return value of the function is bound to the first cell. This can be confusing at first since a traditional language has its return value at the end. This is also the case here however in order to execute the column of if else functions the first one has to be the return value and it evaluates the chain of cells below.

6.2 Scope

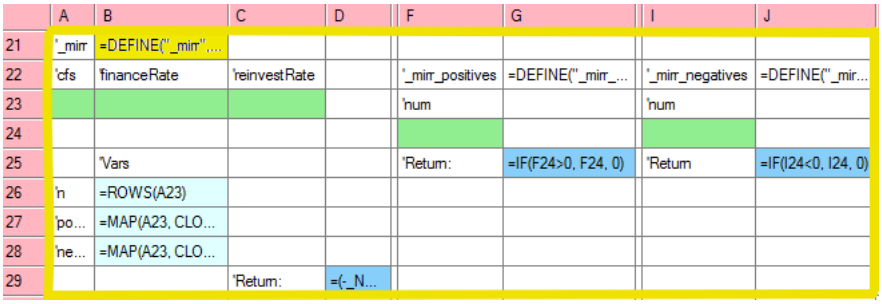
Funcalc does not have scope of function definitions. All function definitions are global to a Workbook. This gives usability issues regarding end users of your sheet defined functions. If another user needs a financial function suite and downloads a workbook containing these function sheets, the list of functions available will contain all the helper functions exposed to the end user as well as the ones targeted as end functions. Naming clashes can partly be solved by using naming conventions mentioned in Section 6.3.

Possible solutions to the scoping problem could be solved with a visual scope by boxing functions. This could be particularly helpful for functions that has helper functions which is only used within the caller function function.

On the image below an example of scope could be shown visually by surrounding a function with an outline grouping inner functions with the main function. This can help prevent pollution of a namespace.

To enforce private functions not leaking into normal sheets the spreadsheet engine could support functions prepended with underscore (_) to only be available within function sheets.

Scoping of functions visually can be a benefit for the tree representation discussed in chapter 6.7 as well in terms of formula cells that are not referenced can act as a warning to the developer



	A	B	C	D	F	G	I	J
21	'_mirr	=DEFINE("_mirr",...						
22	'cfs	'financeRate	'reinvestRate		'_mirr_positives	=DEFINE("_mirr_...	'_mirr_negatives	=DEFINE("_mir...
23					'num		'num	
24								
25		'Vars			'Return:	=IF(F24>0, F24, 0)	'Return	=IF(I24<0, I24, 0)
26	'n	=ROWS(A23)						
27	'po...	=MAP(A23, CLO...						
28	'ne...	=MAP(A23, CLO...						
29			'Return:	=(-_N...				

Figure 10: This shows how scoping can be grouped visually by putting a border around a a given area

that there might be an error here.

Visual scoping like this is not only beneficial for hiding helper functions but could also allow for referencing to an outer functions variables to omit the use of lambda lifting.

Boxing functions like this puts constraints on the developer to where he can define his helper functions. However these constraints do not break functionality and if following the best practices shown in section 3.3 then this will not be an issue.

6.3 Naming Conventions

Another problem in the current implementation that arises is naming clashes. Currently all function names share same scope. It would be beneficial if functions could be called with their sheet name e.g. `@tvm!FV(-1.5,-2,50,-300,1)` to call a specific function within a specific sheet as you can in Excel when referencing to cells. This can be compared to Namespaces in traditional languages with workbook name and sheet name acting as natural namespace names. Currently there has been no need for it during implementation of the financial functions but one can imagine that when people start using multiple workbooks made by others, name clashes will occur. From an end user point of view the optimal solution would be that you get an error if you call a function that is defined twice, telling you to specify the sheet and optionally the workbook it belongs to. In cases where there is only one function defined with a name, it will automatically map to that function omitting the namespace.

6.4 Optional Arguments

Currently Funcalc is missing the functionality of being able to specify optional arguments. This missing functionality has shown that it is impossible to model the behavior of Excels functions completely without. I suggest to include this in the definition of a function so one could write:

```
DEFINE("PV", C2, A1, A2, A3, A4 = 0, A5 = 0)
```

The pros with this solution is that it is defined explicitly that some arguments has a default value. However the cons with this solution is that single equal signs in the spreadsheet formula language is also used as comparators and therefore creates a possibility for people to misread the meaning. Another solution is to use the value of cell A4 and A5 as default values thereby saying that if argument cells has data in them, this will be used as default values. One downside with this solution is that default values are declared implicitly and it is easier to mess up the behavior when testing functions. Both solutions does still not show a solution to arbitrary arguments having default values. To support arbitrary optional arguments, one would have to introduce named arguments such that argument order does not matter. Python solves this problem by using the argument names such as `def help(object, b=10, c=1)` can be called like `help(o, c=3)` leaving out b which then has a default value of 10. However since arguments in Funcalc are cell names this does not make it clear for the user. Consider line 1 below while calling the function with `PV(3,4,1,A5=1)` is possible leaving A4 to use the default value 0 it is not necessarily clear for the user to write.

```
1.  DEFINE("PV", C2, A1, A2, A3, A4=0, A5=0)
2.  DEFINE("PV", C2, A1, A2, A3, fv:A4=0, type:A5=0)
```

A cleaner solution is suggested on line 2 where optional arguments are named before a colon. Calling the function on line 2 would then look as `PV(3,4,1,type=1` leaving A4 to the default value of 0 again.

6.5 Constants

Named constants can enhance clarity of function definitions at times. Examples where constants would have been beneficial during development are for example in calendar types. So instead of giving numbers 0-4 to such functions it would make it more clear to use a named constant such as ACTUAL360 for the given calendar.

Excel has the feature of renaming cells which would work as a fix. Funcalc however lacks this functionality at the given moment.

6.6 Lambda lifting

In Funcalc it is currently not possible to scope inner functions within a given parent function as mentioned in section 6.2. This is circumvented using a technique called lambda lifting where the parent functions arguments are parsed through to the inner function[10].

It is clear that this is not a viable solution. Especially not in cases with deeply nested functions as seen in the implementation of the helper function totaldepr described in Section 4.4.

As discussed in subsection 6.2 a visual scoping of functions can solve this making it possible to refer to argument cells from an outer function in an inner function without passing it along. Scoping however makes the underlying implementation more complex but is considered to be a necessary extension of Funcalc in the future.

6.7 Recursion

As mentioned earlier, the spreadsheet concept hides implementation by default and shows results immediately. While this makes good sense in a normal spreadsheet setting when doing calculations on data, it hides the logic behind a recursive function. Luckily Funcalc has the ability to show formulas in the cells of a given sheet which alleviates this issue somewhat. Implementing recursive functions in the spreadsheet paradigm stresses the importance of a good structure of your spreadsheet. In traditional languages a recursive function is read from top to bottom having a return value calling itself. If a recursive function is structured as shown on figure 5 then it can be read reasonably well however the dependency between the cells are still not clear. Excel has the ability to draw arrows to show how cells are connected. However this can quickly become useless to look at visually since you quickly end up with arrows overlapping each other. A visual tree representation of a function can be beneficial for debugging purposes to spot infinite loops in recursive functions or just to get a view of a function that resembles that of a traditional programming language. This tree idea is based upon the point made by Bewig[2] where he suggest that you draw a dependency graph for debugging your spreadsheet.

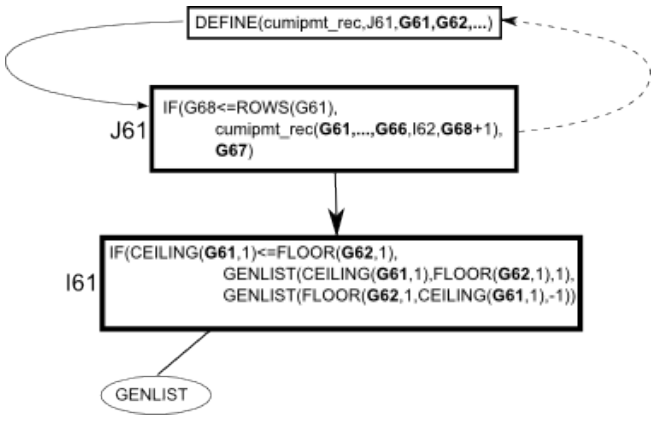


Figure 11: Showing a graph of a function with dependence to another cell.

Such a tree is shown on figure 11. With visual scope implemented as shown in section 6.2, it is possible to single out formula cells which are not referred and highlight these in the tree.

6.8 Arrays

Arrays in Funcalc and in other Spreadsheet engines are represented by a row/column matrix. This brings up some implementation difficulties as you will have to check the input to your function regarding the direction of your array. As you would have to check `ROWS` and `COLUMNS` to figure out how to `INDEX` over the array. In my implementation i have omitted these checks and made a convention to always expect vertical arrays.

A solution is to write a function `FLATTEN (arr)` that takes one argument: `arr` being the cell range to into a row of values.

6.9 Anonymous Functions

In some cases Anonymous Functions can be useful. The implementation of `MIRR` in section 4.5 is a good example, where we use map to apply a function to a range of values calculating an end result. In `MIRR` the function to be applied is `IF(K68<0,K68,0)`. This formula is very simple but currently takes up 3x3 cells including description of the helper function. For such cases defining it inline will have a benefit on clarity of program structure.

While this can be an advantage from developers with knowledge from traditional programming languages, it can be a problem for end users that are not used to e.g. lambda notation. It can also be argued how well lambda notation fits into the formula language by breaking its simplicity. Keeping the formula language simple is the main concern. Most traditional programming languages that supports lambda notation is some variant of the arrow notation. C# uses `lambda x -> x*x` to note that the function takes one argument x and returns $x * x$. F# uses `fun x -> x*x`. Others use a double arrow or other similar notations with different keywords.

To keep such a feature true to the spreadsheet paradigm, i suggest using a function `FUN("x -> x*x")` which returns a closure for the given function. Another syntax could be `FUN(x,x*x)` which is preferred. If this is used in connection to scoping discussed in Section 6.2, one could refer to cell arguments so anonymous functions also could be written as `FUN(x, x*A5)` if cell `A5` was an argument.

6.10 Documentation

Documentation of functions for end users are necessary both in terms of argument types but also descriptions of functions. Consider the excel function:

```
IPMT(interest_rate, period, number_payments, PV,FV,Type).
```

This function is defined in Funcalc as `DEFINE("IPMT",out,in,in,in,in,in,in)` where `in` are input cells and `out` is the return cell. For an end user there is not much help to get from

the definition and the only way to get help is looking in the "library" of the function which might contain comments about it's arguments.

I suggest a solution to introduce a function:

```
DOCFUN( A11 ,"IPMT", "interest rate", "period", "number
            payments", "PV", "FV", "Type")
```

to document function arguments to a function. The first argument to `DOCFUN` A11 is a more thorough description of the function. This argument can both be a text string or cell reference to a text string. The advantage of this approach is to keep documentation of user defined functions inside the spreadsheet metaphor on which the underlying spreadsheet engine can use to guide end users in using function libraries. I imagine a representation of the help as tooltips when writing functions much like Excel. A user could also get a more thorough explanation of the defined function through the description in cell A11.

Internal documentation and comments of functions should be done following the same practices on documentation of normal spreadsheets as mentioned in Section 3.2. Namely using the power of the spreadsheet by using empty cells as comment fields.

6.11 Unit Tests

Testing user defined functions showed to be an important step in order to ensure correctness. Implementation of user defined functions using the spreadsheet metaphor has showed that the errors during development is common. Most functions even contained small errors that was only found after running the test cases extracted from the F# library against the implemented functions.

Based on this observation a method is needed to formally specify unit tests for your sheet defined functions. I suggest introducing a third type of sheet marked with an exclamation mark(!) which is used only for tests. There are certain advantages of introducing a third sheet type for tests. Doing thorough unit testing requires many combinations of input to the functions which results in a long range of calculations that needs to be performed. Test sheets can be loaded in a lazy manner reducing load time considerably. Test sheets can then be loaded when selected in the spreadsheet program.

For users that include a package of functions from a third party it might be feasible to hide test sheets from the user interface so these do not litter the UI if a user is only using functionality from a package.

6.12 General Observations

6.12.1 Error Propagation

Error propagation in Funcalc is working well and the errors are propagated to the top. However if an error is caused from deep within a hierarchy of dependent cells, it is currently not possible to see which cell that originally caused the error. A reference to the bad cell should be prepended to the error message to solve the issue of locating the error.

A stack trace of the call hierarchy could be beneficial for certain errors that are caused by for example division by zero errors or other subtleties that should be handled by some of the calling functions higher up the hierarchy.

6.12.2 Cell reassignment

In traditional languages you have the possibility of reassigning variables or updating values of variables.

```
Traditional language
a = 15
a = a + 4

Funcalc.
A1 = 15
A2 = A1 + 4
```

In Funcalc however you will have to use a new cell to represent the updated value as shown in the code sample above. Funcalc is doing the right thing not to allow for updating cells based on the visual grid structure of a spreadsheet since evaluation order can not be guaranteed. In function declarations however order can be guaranteed but the visual order of cells might differ from the underlying order of the function implementation.

6.12.3 Ease of use

Understanding the concepts behind sheet defined functions seemed to have a low learning curve. The syntax is well defined and fits well into the basic formula language. As noted in section 6.7, recursive functions were difficult to read when debugging. Debugging proved to be an issue both within the formula language but also within function definitions. Funcalc color highlights cells that are used in functions which helps partly. However order of definition of cells are not directly shown visually and is up to the user to visually lay out. This lack of structure in the

language puts a lot of responsibility in the hands of the user. However this seems difficult to change when staying in the spreadsheet metaphor. Therefore the guidelines for writing sheet defined functions are:

- Invent a structure for how you want to write functions and stick to that.
- Keep inner functions on the same rows as their parent functions leaving rows above and below to new top level functions.
- Always write `DOCFUN` documentation for your functions.
- Name private functions with a prepending underscore.

7 Discussion

In Section 6 a range of suggestions was highlighted to improve the functional language in Funcalc. To sum up, these are:

Scope would give certain advantages both in terms of function hiding so helper functions do not litter the list of functions available. The other advantage implementing the visual scope lies in that it is suddenly possible to refer to arguments from outer functions and thereby making helper functions truly nested to the outer function.

Naming Conventions was used to separate internal library functions from external functions using underscore prepended to the function name. This convention is similar that of C.

Lambda Lifting can be solved using the scoping method that was discussed in Section 6.2.

Recursion showed to quickly be cluttered visually with invisible cell references, especially in larger recursive functions that used a lot of temporary variables. The part solution for this will be to introduce a tree representation as an alternative to how the cells are interconnected in a recursive function. More over with the scoping, unreferenced cells within a scope could be highlighted in the tree representation since these would most likely be errors.

Anonymous Functions showed to be a nice to have in situations where the function given to the higher order function is rather simple. A syntax was suggested that maps well into that of the spreadsheet metaphor.

Documentation of spreadsheets proves to be important when future users wants to use a library developed by somebody else. To make it easy for the end users to use your library, a mechanism was suggested on how to document your sheet defined functions as well as your package.

Unit Tests was a big part of this implementation of financial functions. They showed a need for special sheets containing such tests. The reason was based on the fact that during package distribution it suddenly becomes irrelevant to load in unit tests every time you include a package in your own sheet.

7.1 Workbook library distribution

If you follow the vision where you can see people sharing workbooks with function sheets between each other we quickly end up in a versioning nightmare and making sure everyone

gets bug fixes etc. There are already many solutions to this problem within the open source world and most newer programming languages seem to get tools for solving this issue. These are mostly referred to Package Repositories on which a user can browse and download new packages. Similar technology can be very beneficial with regards to maintainability for user defined functions. For corporations one would most likely use an in house repository to keep in house developed libraries up to date however for more general libraries public repositories would also be available.

I suggest Workbooks for such a package distribution to enforce that:

Function names with a prepended underscore (_) to be private for a given package. For a package i mean a workbook containing a number of function sheets with user defined functions. The `DOCFUN` function should be written for all public functions exposed from the package.

Normal sheets in a package should only be used for examples on how to use a given package or as an introduction to what this package does. No test cases of functions should be written in normal sheets.

Test sheets in packages are not evaluated or read when including a package. A practical example of why this is a good idea is the loading time of finance.xml which is 8-10 seconds in Funcalc. Real world Excel documents can be hundreds of megabyte and takes up to minutes to load. Having excessive test data will only add to the loading time.

Funcalc will need to have a UI for browsing/searching a repository for available packages as well as selecting repositories. The architecture will consist of a webserver listing available packages for download maintaining meta information on packages such as version, author etc. Users will include workbook packages in their own workbooks and have read only access to these. Inclusion of workbook packages will be handled as part of Funcalc UI and not through cells. When a package has been included in your workbook, all public functions will be browsable along with their documentation. When downloading third party packages, security comes in mind as Funcalc like Excel allows arbitrary code execution. Funcalc allows this through the `EXTERN` function which calls to should not be executed unless a user permits this.

When package repositories grow, package dependencies will occur where for example a package for doing 3D graphics calculations needs an underlying package for doing linear algebra calculations. When this occurs you might not want to pollute the end users Document with references to packages that the end user does not directly use. In scenarios where an end user wants to make a backup of his/her spreadsheet, a backup of dependency packages should also be made. Backing up dependencies with a backup of the main Document prevents the packages from being unavailable in the future.

7.2 Future

The future of user defined functions in the spreadsheet metaphor relies heavily on the implementation of this feature into popular spreadsheet programs such as Excel and Open Office Calc. Funcalc in itself currently works as a testbed for new ideas related to user defined functions and the internal implementation behind. The learning curve for sheet defined functions feels lighter than that of VBA. The syntax is very similar to the formula language however some concepts are still alien to end users. Namely higher order functions and recursion. Scope in itself is also alien if it does not show a clear way of why it is necessary.

8 Conclusion

This thesis has proved that most functions can be implemented in Funcalc similarly to Excel and F#. However not all functionality could be mimicked due to missing language constructs that we touched upon in Observations. During testing, it was shown that human errors during implementation of functions in Funcalc are common. It stressed the importance of mechanics of writing tests easily for user defined functions. We also looked at a range of other observations that could improve various functionality and introduce new concepts for handling package distribution. This thesis also highlighted various bugs in Funcalc, most of which has been corrected during the writing of this thesis.

References

[1] Excel financial functions for .net. `http://archive.msdn.microsoft.com/FinancialFunctions`.

[2] Philip L. Bewig. How do you know your spreadsheet is right? 2005.

[3] Edwin H. Blake, Fabian Nuñez, and Fabian Nuñez. An extended spreadsheet paradigm for data visualisation systems, and its implementation, 2002.

[4] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34:135–137, January 2001.

[5] Richard A. Brealey, Stewart C. Myers, and Alan J. Marcus. *Fundamentals of corporate finance*. McGraw-Hill series in finance. McGraw-Hill, New York, NY [u.a.], [5. dr.], internat. ed edition, 1997.

[6] Dan Bricklin. *Bricklin on technology*. Wiley, 2009.

[7] G. Fisher, E. Giaccardi, Y. Ye, A. G. Sutcliffe, and N. Mehandjiev. Meta-design: a manifesto for end-user development. *Commun. ACM*, (9):33–37.

[8] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford, CA, USA, 1973.

[9] Thomas S. Iversen. Runtime code generation to speed up spreadsheet computations. master's thesis, diku, university of copenhagen, 2006.

[10] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. pages 190–203. Springer-Verlag, 1985.

[11] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in excel. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 165–176, New York, NY, USA, 2003. ACM.

[12] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. End-User Development: An Emerging Paradigm. In *End User Development*, chapter 1, pages 1–8. Springer Netherlands, Dordrecht, 2006.

[13] Mohammad Mainul Liton and Hui Xu. Spreadsheet with user-definable functions, 2009.

[14] M. MacDonald. *Excel 2007: The Missing Manual*. Missing Manual. O'Reilly, 2007.

[15] Microsoft. Excel financial functions. http://office.microsoft.com/en-us/excel-help/CH006252825.aspx.

[16] Bonnie A. Nardi and James R. Miller. The spreadsheet interface: A basis for end user programming. In *Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*, INTERACT '90, pages 977–983, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[17] Raymond R. Panko and Daniel N. Port. End user computing: The dark matter (and dark energy) of corporate it. *Hawaii International Conference on System Sciences*, 0:4603–4612, 2012.

[18] P.P. Peterson, F.J. Fabozzi, and W.D. Habegger. *Financial management and analysis workbook: step-by-step exercises and tests to help you master financial management and analysis*. Frank J. Fabozzi Series. Wiley, 2004.

[19] Peter Sestoft. Implementing function spreadsheets. In *Proceedings of the 4th international workshop on End-user software engineering*, WEUSE '08, pages 91–94, New York, NY, USA, 2008. ACM.

[20] Peter Sestoft. Compiling spreadsheet-defined functions. 2010.

[21] Niklaus Wirth. On the design of programming languages.