Spreadsheet Programming with User Defined Types and Functions

John Williams

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2013

Abstract

A system for defining functions and data types using spreadsheets is presented. *Jeksy* is spreadsheet application which provides a way to create function and type definitions using spreadsheets themselves. These are referred to as *FunctionSheets* and *TypeSheets*. This gives users only familiar with spreadsheets access to some of the power available in classic programming languages.

By solving real world problems using *Jeksy* it was shown how these new features can be used to build spreadsheets with a better structure and as a result reduce the likelihood of errors in the program.

Acknowledgements

I would like to take this opportunity to thank my supervisor Don Sannella for the guidance and support he has provided throughout the course of the project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*John Williams*)

Table of Contents

Chapter 1

Introduction

"A Dallas Oil and Gas company's spreadsheet error resulted in millions of dollars being lost. Several executives were fired."[1]

Classic programmnig languages have addressed and solved many of the issues that cause errors in spreadsheets today. This project investigates how these features can be made available to users only familiar with spreadsheets. Two concepts are introduced; *FunctionSheets* and *TypeSheets*. Through these a spreadsheet user can build structured function defintions and data types with the intention that these tools will help them to produce more robust programs.

In chapter **two** the motivation for the project is presented and the existing work into the classification of spreadsheet errors discussed. From here different approaches to extending spreadsheets are analysed culminating in the proposal of *FunctionSheets* and *TypeSheets*. Finally, goals for these features and the project as a whole are constructed.

Chapter **three** focuses on the design of *FunctionSheets* and *TypeSheets*. Various design considerations are presented and their relative merits compared until a final design is selected. This serves as a specification for how the new features appear and ultimately how the user interacts with them.

The spreadsheet application *Jeksy* is introduced in chapter **four**. Building upon an existing application, *Jeksy* is the implementation of the specification constructed in chapter three. In this section details of the algorithms and data structures used to implement *FunctionSheets* and *TypeSheets* are given. Problems faced during implementation are discussed and ultimately why the prevailing solutions was chosen from amongst the candidates. The chapter ends with a summary of remaining work providing justifications for why these features should be added to *Jeksy*.

Evaluation of *Jeksy* is carried out in chapter **five** of the report, there are two components of the evaluation phase. First, *Jeksy* is used to create a definition of a binary tree. This demonstrates how *TypeSheets* and encapsulation can be used to guarantee invariants that define properties of the data structure. Furthermore *Jeksy* is evaluated on how well it adheres to the expected user experience when working with spreadsheets.

The second half is focused on how *Jeksy* can be used to build more robust spreadsheets. By implementing aspects of an industry level spreadsheet using *Jeksy* it is shown that *FunctionSheets* and *TypeSheets* can create solutions that are easier to understand and less likely to contain certain types of error.

In the final chapter the project's achievements are summarised and future directions discussed. Potential areas for expanding *Jeksy* include collections and higher order function as well as enhancing the user interface when working with *FunctionSheets*.

Chapter 2

Background

2.1 Motivation for the Project

Spreadsheets are now ubiquitous with business and finance however despite their importance there are still a startling number of errors found in these documents. It has been reported that error rates in industry can be as high as 90%[2], with the consequences of errors possibly leading to million dollar omissions[3].

While there is awareness surrounding these errors their likelihood is not set to reduce. Corporate spreadsheets are only increasing in complexity as studies have found that documents and formulas are doubling in size every three years[4]. In addition to the increase of "innocent errors"[2] a more sinister phenomenon is intentional fraud through malicious spreadsheet manipulation. The established methods for detecting accidental errors are not especially effective at identify those caused deliberately[2] and so new techniques need to be developed. With a survey conducted in 2004 by the Association of Certified Fraud Examiners revealing that the median fraud loss for financial fraud being one million dollars the incentive for corporations to improve their spreadsheet practices is clear.

2.2 Existing Work

2.2.1 Research into Spreadsheet Errors

At a high level spreadsheet errors have been placed into two categories: qualitative and quantitative errors[5]. Quantitative errors are described as numerical errors that produce the wrong result from a calculation. The second group of errors - qualitative - are flaws that do not necessarily manifest themselves immediately. They *"degrade the quality of the spreadsheet model"*[5] and as result potentially cause quantitative errors later on in the life-cycle of a spreadsheet.

Most of the work into researching spreadsheet errors has gone into the analysis of

quantitative errors as techniques such as testing and code inspection have generally found better success dealing with those[6]. Quantitative errors have been grouped into three different classes[5]:

- Mechanical: Generally classed as incorrectly entering a value, they can exist in the form of typing or pointing errors. While their rate of occurrence is high they are likely to be detected by the user making the error.
- Logical: These are errors in which the wrong algorithm was chosen to complete the task or it was incorrectly implemented. Logic errors may require domain knowledge to find and solve.
- Omission: This type of error is created when a user misses something from the spreadsheet model which may be due to a miss-interpretation of the problem. Omission errors are recognised as being very problematic because they are difficult to detect.

When approaching the problem of preventing and correcting errors the generally accepted view is that any solution should be subtle and un-intrusive[6]. They should also mirror the behaviour of spreadsheets themselves, providing immediate feedback to the user as changes are made within the program.

2.2.2 Extension of the Spreadsheet Concept

Attempts to improve the current spreadsheet paradigm have taken various different approaches. One such approach focuses on extending existing spreadsheet applications to support new features that reduce error rates and increase probability of detection. The paper *"A User-Centred Approach to Functions in Excel"*[7] introduces the idea of using spreadsheets themselves to create user-defined functions. It establishes the rule that spreadsheets must be used to define functions because it is the only paradigm of computation that the user understands.

The term *"Function Instance Sheet"* is used to refer to a spreadsheet that is the invocation of a particular function. The sheet contains cells designated as input and output to the function along with formulas used as intermediate computation. In figure 2.1 a function instance sheet is presented in addition to its invocation.

Notable features include the source of the function's arguments being displayed in the sheet, this provides the user with an easy way to identify where the function is being called from. As is to be expected with spreadsheets, any changes are immediately propagated both inside the function instance sheet and to its result. A large emphasis is placed on offering the user these features with as little resistance as possible. This requirement is fulfilled by allowing the user to create a function instance sheet from a formula in a cell. The arguments are inferred from the formula and the expression is replaced by a call to the function.

As each sheet refers to a single instance of the function there is an issue when it comes to altering the definition. This is solved by giving the user two choices; either make the changes to all invocations of the function or limit it to that sheet. If the first option is

chosen then all function instance sheets are simply update to reflect the new definition. In the case of local changes the function is renamed and the formula calling it updated also, leaving all other instances of the function unaltered.

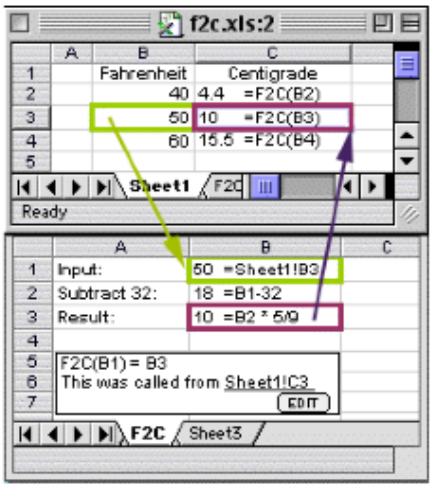


Figure 2.1: Function Instance Sheet

A limitation on the idea proposed in the paper is the ability to handle programming techniques such as recursion. As each sheet represents a single call to the function a recursive operation could potentially generate hundreds of sheets. They chose to prevent recursion altogether in order to prevent this problem, a justification being that the only inductive type is integers.

Much of the focus is on converting existing formulas in a spreadsheet to function instance sheets however what is not explicitly clear is how existing definitions can be used when building a spreadsheet from scratch. As each sheet represents a single call it is unclear how a function could be distributed as a library or plugin to a new spreadsheet. Although the emphasis was on improving legacy systems if this approach became widely adopted then transitioning existing definitions to *"greenfield"* spreadsheets would become increasing common.

At the other end of the spectrum there exist spreadsheets that make use of other programming languages to enhance their functionality. Two examples of this are *Haxcel*[8] and *Pyspread*[9].

Solutions such as this add a great amount of power to a spreadsheet by exposing features such as abstract data types and object based cell values. While impressive examples can be created using these tools they are not suitable for a corporate environment because of the knowledge required to operate them. Most users of spreadsheets in industry have no formal programming experience, their only window to programming being through spreadsheets themselves. The knowledge barrier required to use an application such as this would deter most regular users from utilising the features and so it does not address the issue of reducing errors in real word spreadsheets. There is no disputing that if used properly these types of spreadsheets could produce well

structured programs however for mainstream adoption the interface must be closer to the expectation of a regular spreadsheet user.

2.3 Goals of the Project

Taking into account the ways of approaching spreadsheet extension discussed in 2.2.2 the aims of the project were to produce a solution that bridged the gap between spreadsheets as an interface and classic programming language features. The goals can be summarised by two main points:

- Offer features available in classic programming languages to users of spreadsheets to aid in the design of more robust programs.
- Provide these features in such a way that the spreadsheet appearance and behaviour is not compromised.

In order to satisfy the goals two new features were proposed; *FunctionSheets* and *TypeSheets*.

FunctionSheets were heavily inspired by function instance sheets proposed in [7] with one distinct difference. *FunctionSheets* are a definition and do not represent a particular invocation of a function. As a result their distribution as a library which can be re-used in multiple spreadsheets is more intuitive.

TypeSheets take this concept and extends it to data types; exposing object oriented principles through a spreadsheet based interface.

Chapter 3

Design

The objective for the design of *FunctionSheets* and *TypeSheets* is to expose the power of standard programming features through an interface that is consistent with spreadsheets. This section details what these features look like and how they are used within the application.

3.1 Spreadsheet Structure

When encouraging users to adopt new techniques and features, reducing the amount of additional knowledge required to operate them is important; this was the motivation to make the structure of *FunctionSheets* and *TypeSheets* as similar as possible. It is reasonable to expect that the spreadsheet is the interface the user is most familiar with and therefore it makes sense to design these features to reflect that.

Should a user with to define a new type or function they do so using the spreadsheet itself. The details of the definition are stored within the cells of the spreadsheet and it is up to the application to extract the information and create the type/function. For the program to be able to complete the definition the cells must be labelled with their purpose, for example cells that contain function arguments. When designing how the cells were identified there were two main choices.

The first approach is to assign cells roles through graphical interface elements. For instance a user could select cell *A1* with the contents *"square"* and upgrade this to a *function name* cell. This notifies the application that the current definition is for a function called *"square"*. An advantage of this is that it makes clear to the system how the definition has changed without requiring the entire sheet to be parsed again.

Alternatively, cells could be identified by headers that indicate where a particular part of a definition begins. A parser can then extract the values from the following cells and assign them to their correct roles. Under a naive implementation this would require the entire sheet to be processed each time a change is made however optimisations can be made later to only check particular sections of a sheet. The main advantage of this approach is that its appearance is very similar to how spreadsheets are currently used;

namely a tabular layout with data labelled using headers and titles. This was the chosen method because it was aligned more closely with the goal of producing an experience for regular spreadsheets users. Headers are identified using a special character, the choice here being the symbol '#' however this is customisable by the user. The result is that a very basic function definition could start as follows:

	A	B	C
1	*#FUN_NAME*		
2	addSquares		

...
...

Figure 3.1: Beginning of a Definition

3.2 FunctionSheets

In this report the term *FunctionSheet* refers to the definition of a function using a spreadsheet, it does not necessarily limit a sheet to a single definition. It may be preferable to include multiple related functions within a single spreadsheet as this makes their distribution easier.

3.2.1 Sheet Headers

The headers available to the user when defining a *FunctionSheet* are:

#FUN_NAME Marks the name of the function being defined.

#ARGS Start of function argument declarations.

#NAME The names of individual function arguments.

#TEST_VALUE When testing the function the arguments are replaced with these values.

#DEFUN Start of any intermediate calculations used by the function.

#OUTPUT The output of the function.

Only the *#DEFUN* header is optional, it may be omitted when the entire function definition is located inside the output cell. Initially the function name header was also optional if the sheet only contained one definition (the name was based on the file name). This was later revoked in favour of consistency.

3.2.2 Function Arguments

The function argument section of a definition has two purposes; to indicate the name of the arguments and also their values when testing the function. Although not strictly necessary to produce a valid function, the *#TEST_VALUE* header was made mandatory as it encourages the user to test the function as they write it. This in turn reduces the chance of implementing the function incorrectly.

	A	B	C
6	*#ARGS*		
7	*#NAME*	*#TEST_VALUE*	
8	x	9	
9	y	10	

Figure 3.2: Argument Definition

Figure 3.2 shows an example of how function arguments can be specified. This would follow a name declaration such as 3.1. The user would be able to include the symbols x and y in expressions and when evaluated they take the value 9 and 10 respectively.

3.2.3 Function Output

The output of the function is taken from the cell below the *#OUTPUT* header. This must take the form of an expression and may include cell addresses. When it comes to determining the function definition as used by the interpreter a recursive process of substituting cell addresses for expressions begins. Starting from the output cell, any cell references in the expression are replaced with the contents of that cell. This is done depth-first until there are no more cell addresses remaining in the expression. Finally the function signature is added to the expression to complete the process.

	A	B	C
11	*#DEFUN*		
12	Square x	*=x*x*	
13	Square y	*=y*y*	
14			
15	*#OUTPUT*		
16	*=B12 + B13*		

Figure 3.3: Computing the Function Definition

The substitution process is as follows:

Start with output	$= B12 + B13$
Replace $B12$	$= (x * x) + B13$
Replace $B13$	$= (x * x) + (y * y)$
Add signature	$addSquares(x,y) = (x * x) + (y * y)$

3.2.4 Testing

Unlike a normal spreadsheet the default display setting for an expression is its definition, not the result of its evaluation. This is because when developing a *FunctionSheet* the user will want to know the previous calculations used. A toggle is available that will switch to "*test mode*" whereby all expressions display their values. In this mode the function parameters assume their test values so the user can inspect the output. By changing the test values (3.4(b)) the user can immediately see the output of their function with different arguments. Its worth noting that the cells *A12* and *A13* are simply comments and do not interfere with the compilation of the function. This is an advantage of using *FunctionSheets* as it allows documentation of the steps within a function.

	A	B	C
6	*#ARGS*		
7	*#NAME*	*#TEST_VALUE*	
8	x	9	
9	y	10	
10			
11	*#DEFUN*		
12	Square x	81	
13	Square y	100	
14			
15	*#OUTPUT*		
16	181		

(a) Switching 3.3 to Test Mode

	A	B	C
6	*#ARGS*		
7	*#NAME*	*#TEST_VALUE*	
8	x	5	
9	y	6	
10			
11	*#DEFUN*		
12	Square x	25	
13	Square y	36	
14			
15	*#OUTPUT*		
16	61		

(b) Changing Test Values.

3.3 TypeSheets

TypeSheets can be considered as a superset of *FunctionSheets* as they too are capable of containing multiple function definitions. In addition to that they also allow the user to create a type definition which is analogous to a record or struct.

3.3.1 Sheet Header

As well as the *FunctionSheet* headers the following are all valid within a type definition:

#FIELDS Marks the starting point of the field definitions.

#LABELS Values in this column will become named attributes of the type.

#ACCESS Provide optional access modifiers to the attributes.

#DEFAULT Specify optional default values for the type attributes.

Only #FIELDS and #LABELS are compulsory headers because they are the minimum amount of information required to define a named record. The other properties allow the user to customise the behaviour of the type if desired.

Unlike functions there is no header to specify the name of the type, instead the file name is used. This is so that it is clear to the user that only one type may be defined per sheet, the motivation for this being that the sheet acts as the scope for accessing private attributes (more information in 3.3.3).

	A	B	C
1	*#FIELDS*		
2	*#LABELS*		
3	x		
4	y		

Figure 3.4: A Basic Type Definition in File 'point.jks'

3.3.2 Using Type Instances

Functions using the bracket notation *[e.g. SUM(...)]* is a concept extensively used in spreadsheets so it makes sense to use this as the method for creating a type instance. It can be thought of as a function that takes arguments intended to be the attribute values and returns an instance of type *foo*. When a new type is defined a function is created with the same name that takes an argument for each public attribute. In the case of 3.4 a new function *point(x,y)* will be created that returns a *point* instance.

The visual representation of type instance is based on its values. By default only public attributes will be displayed however this can be changed be overriding certain properties of the type, see 3.3.5 for further detail on this. In the *point* example, the

result of the expression *=point(3,4)* will return a type instance which is represented to the user as *"point {x: 3, y: 4}"*.

Instead of supplying all the arguments it is possible to use a default value assuming this has been defined in the type definition. The notation used for this is very simple, it consists of a special symbol that replaces the function argument. The term used here is *DEFAULT* because it is self descriptive however this can be changed. Given 3.5, an example using default values is the expression *=point(9,DEFAULT)* which would return an instance that looks like *"point {x: 9, y: 7}"*.

	A	B	C
1	*#FIELDS*		
2	*#LABELS*	*#DEFAULT*	
3	x	5	
4	y	7	

Figure 3.5: Using Default Values

Dereferencing a type instance is one of the most common uses of a type. The choice of notation was based on what appears most natural to the user and fits with what is perceived to be happening. Using function notation was considered however when nesting multiple dereferences it becomes very verbose quickly. Instead the *!* symbol also used to reference cells in other sheets was selected. If a sheet is considered as a type and cells its attributes this operation is conceptually the same and makes for a consistent syntax. Furthermore it is very concise when dereferencing nested types. Again using the *point* example, an expression that adds the co-ordinates of a point in cell *A1* would be *=A1!x + A1!y*. This transitions easily when using multiple sheets, for example *=OtherSheet!A1!x + OtherSheet!A1!y*.

3.3.3 Encapsulation

There are two levels of access for type attributes; *public* and *private*. By default all attributes are public but can be made private using the *#ACCESS* header. The scope of private attributes is within the defining sheet of that type meaning it is only possible to dereference private attributes if both the type instance and calling expression are within the *TypeSheet*. As mentioned in section 3.3.2 type constructor functions only accept public attributes. In addition to that function there is also a private constructor function which allows the user to set all attributes. As with private fields, this can only be accessed from within the *TypeSheet* definition itself.

In the case of 3.6 two constructor functions are created; *point(x)* and point*(x,y)*. While the first function can be used anywhere the latter can only be invoked from within the defining sheet. Details on how private attributes can be used within public functions are included in section 3.3.4.

	A	B	C
1	*#FIELDS*		
2	*#LABELS*	*#DEFAULT*	*#ACCESS*
3	x	5	public
4	y	7	private

Figure 3.6: Access Modifiers

3.3.4 Nested FunctionSheets

The term nested *FunctionSheet* refers to a *FunctionSheet* that occurs within a type definition. The main incentive for constructing a function in a *TypeSheet* rather than on its own is that it can access the protected attributes and constructor function for that type. A simple example would be a "getter" function that returns the value of a private attribute. Another common pattern would be a custom constructor function enabling the caller to set the private attributes when creating a type instance.

In the case of *point* types, the user could specify a constructor called two_x which only allows the creation of points that exist on the line $y = 2x$. This could be done using a nested *FunctionSheet* with definition *two_x(x) = point(x,2*x)*.

Another reason for using nested *FunctionSheets* is that it allows the user to define custom behaviour for types, this is discussed further in section 3.3.5.

3.3.5 Operator Overriding

In classic programming languages the ability to specify the behaviour of a common operator with user types can lead to more readable code. With one of the objectives of this project being to integrate user types into spreadsheets it makes sense to provide an interface for the user to override some of the common operators in the syntax; this is done by using nested *FunctionSheets* to implement this behaviour.

Taking inspiration from *Haskell* type classes, the functions available to override are *Show*, *Eq* and *Ord*. The *Show* function is analogous to the *Java* method *toString()* with the exception that it takes the type instance as an argument instead. *Eq* takes two arguments and is used to override the = operator for that type. Finally, the purpose of the *Ord* function is to specify how two type instances should be ordered with respect to each other. Its return values are either *-1*, *0* or *1* and indicate the first argument is less than, equal to or greater than the second argument respectively.

To summarise, the signatures for these functions are as follows:

$Show(UserType) \rightarrow String$
$Eq(UserType, UserType) \rightarrow Boolean$
$Ord(UserType, UserType) \rightarrow Integer \in \{-1, 0, 1\}$

Chapter 4

Implementation

The design chapter provided a specification for how function and type sheets appear and how the user can interact with them. It did not go into any real detail about how these features were implemented. The purpose of this chapter is to expand on this and explain how they were built and highlight possible future additions.

4.1 Building on *Jeks*

Creating a spreadsheet application from scratch would require significant work just to produce a basic program that introduces nothing new to the concept. This was the motivation for finding an existing application to build on, allowing development of the new features to begin almost immediately.

The application which served as a platform is called *Jeks*[10], a *Java* spreadsheet implementation licensed under the *GNU General Public License*[11]. From the project page some of *Jeks's* features include:

- Formulas in cells using a rich set of operators and functions.
- Parameterized functions using expressions defined by users.
- Checking of circular reference in formulas.
- Optimised update of formulas referencing other cells.
- Cut / Copy / Paste with automatic shift of cell references in formulas.
- Save / Open files at spreadsheet format.

From this point on *Jeksy* refers to the application that includes the features discussed in this report, it can be considered as a fork of *Jeks*.

4.2 General Additions

Before work could commence on the main features of *Jeksy* some core functionality had to be added, namely the ability to use multiple sheets. Previously the program only worked with a single spreadsheet at any one time, obviously this was a problem when trying to build a system that uses numerous sheets to create definitions.

Adding multiple sheets also required the syntax to be extended, permitting references to cells in other sheets. In order to be consistent with other spreadsheet implementations the *!* notation was used.

Another alteration was extending the encoding system to handle different types of sheets, these were spreadsheets, functionsheets and typesheets. As function/type definitions were really just spreadsheets themselves the application needed to know which type of sheet it was processing. One alternate solution was to use the file extension however incorrect file naming could lead to errors. The more robust (and chosen) approach was to add meta-data into the file itself and take the responsibility from the user and give it to the application.

4.3 FunctionSheets

4.3.1 Function Parameters

FunctionSheets use named arguments in their expressions which are assigned test values by the user. In *Excel* this could be implemented using cell labels however *Jeks* did not support this meaning it had to be added in. There are two components of a function parameter in *Jeksy*, its name and the test cell associated with it. When the expression parser encounters a token that it does not recognise (e.g. a function argument) it queries the spreadsheet model for a list of defined parameters for that sheet. It then replaces the token with the cell address that corresponds to that parameter. The interpreter can then evaluate the expression using then test values specified by the user.

Function parameters are constructed when the file is loaded and updated after each subsequent save. As each parameter is linked with a cell address rather than a value it means that expressions are automatically updated when a user changes the contents of the test cell, the sheet does not require re-parsing.

There is one limitation with how this is currently implemented. When creating a new definition the user must save the file after completing the arguments definition but before using any of the parameters in expressions. This is to force *Jeksy* to compile the sheet and load the parameters into the run-time. Unless this is done the parameters will be undefined and expressions using them will not evaluate. A better solution would be to automatically compile the file after any changes to the sheet are made so the user does not need to save explicitly.

	A	B	C
6	*#ARGS*		
7	*#NAME*	*#TEST_VALUE*	
8	x	9	
9	y	10	

Figure 4.1: *FunctionSheet* Parameters

Using the parameters from 4.1, the process of evaluating the expression $x + y$ would be as follows:

Encounter unknown symbol x, search defined parameters for an entry x and replace with the test cell address.
$x + y \rightarrow B8 + y$

Encounter unknown symbol y, search defined parameters for an entry y and replace with the test cell address.

$B8 + y \rightarrow B8 + B9$

4.3.2 Compiling Functions

Section 3.2.3 gave a brief overview of the recursive algorithm used by the *Jeksy* compiler to build the function definition, *Algorithm 1* shows the implementation of this function. One consideration was ensuring that the algorithm would always terminate and could not loop indefinitely. This would be caused by two cells that reference each other in their expressions. Fortunately this cyclical referencing is forbidden by the expression parser so it is not possible for the user to create expressions that would cause the algorithm to infinitely loop.

Algorithm 1 Generating the Function Definition from a *FunctionSheet*

```
1:  function EXPANDEXPRESSION(expression)
2:      for all token ∈ expression do
3:          if token is cellAddress then
4:              contents ← GETCELLCONTENTS(token)
5:              if contents is cellExpression then
6:                  expansion ← EXPANDEXPRESSION(contents)
7:              else
8:                  expansion ← contents
9:              end if
10:             expression ← REPLACE(expression, token, expansion)
11:         end if
12:     end for
13:     return expression
14: end function
```

4.3.3 Recursive Functions

When creating a recursive function using a *FunctionSheet* the user will need to create an expression within the sheet that calls said function. One problem was that when first constructing the sheet the function would not exist in the run-time, this meant the user was unable to enter the expression issuing the recursive call.

The solution to this was based on the similar problem faced when using function parameters. The user must save the file after completing the argument definition but before implementing the actual function. Doing so will generate an empty signature for the function using the name and specified arguments, it simply returns null. The user can then enter expressions using the function and make recursive calls.

Once the user has completed the definition saving the file will update the existing empty function in the run-time to reflect the implementation in the *FunctionSheet*.

4.4 TypeSheets

4.4.1 Type Instances

The basic data types that were initially available as cell values were *integers*, *floats*, *strings* and *booleans*. As the application was built in *Java* they were mapped to *Java* classes; *Long*, *Double*, *String* and *Boolean* respectively. To support instances of user defined types another class was created called *UserTypeInstance*. Fundamentally this is a map from *String* keys to any permitted cell value; it also includes meta-data such as the definition of the type it represents.

Another import data structure used in *Jeksy* is the *UserType* class which represents a type definition. Important features of this include a collection of compiled functions corresponding to the overridden operators. When a *UserTypeInstance* is involved in an operation that can be overridden the *UserType* object is queried to check if a compiled function for that operation exists. The other main feature of a *UserType* is the sequence of attribute descriptors. An attribute descriptor is essentially a 3-tuple that holds an attribute name, its access level and default value.

As mentioned in section 3.3.3 about type encapsulation there are two functions that create type instances. One only sets the public attributes and the other sets every attribute. Originally the intention was to allow multiple functions with the same name but different argument counts to be defined but due to time constraints this was **not** implemented. The justification for not implementing it was the fact that it did not add new functionality but rather improved the user experience. In the current implementation there are still two constructor functions per type but they have different names. The public attribute only function is named after the type whereas the function for all attributes is named after the type with the suffix "*_unprotected*". For example a *point* type would have two constructor functions *point* and *point_unprotected*.

4.4.2 Dereferencing

Syntactically dereferencing is done using the *!* operator as previously discussed in section 3.3.2. When using types with attributes that are also user defined types it is possible to apply multiple *!* operations to access fields in a concise way. A basic example might be *line!point_one!x* which returns the *x* value of the first point on the line.

In the actual implementation dereferencing is done using a function called *deref*. Before an expression is evaluated any *!* operations are transformed into *deref* calls. The reason for using a function was twofold; firstly it was easier to right rules for applying string based transformations than it was to change the expression parser to handle the dereference operator. The other reason was the derefencing requires additional information such as scope and using a function was a convenient way to pass this information without the user being aware.

The signature for the *deref* function is: *deref(env, type, fields)*.

env is the sheet from which the expression was called, its purpose is discussed further in 4.4.3. *type* is a *UserTypeInstance* and *fields* is an array of attribute names. Examples of how user expressions map to *deref* calls are as follows:

A1!x $\rightarrow$ *deref(env, A1, [x])*
Sheet!A1!x $\rightarrow$ *deref(env, Sheet!A1, [x])*
Sheet!A1!x!y!z $\rightarrow$ *deref(env, Sheet!A1, [x,y,z])*

It is worth clarifying that the *!* operations remaining in the *deref* call (e.g. *Sheet!A1*) are references to cells in other sheets and so handled separately. These arguments are replaced by the cell contents prior to the expression being evaluated.

4.4.3 Encapsulation

To provide a way to encapsulate type attributes *deref* calls must be aware of the context from which they are called. This is done using the variable *env* which is set by the interpreter when the expression is evaluated. Algorithm 2 provides an abstracted view of how this process works. For these examples expressions and functions can be considered equivalent, in practice the main difference is that expression arguments are mapped to cell addresses.

Algorithm 2 Binding the *env* Variable

```
1: function EVALUATE(expression, args)
2:     env ← GETCALLINGSHEET()
3:     return COMPUTEVALUE(env, expression, args)
4: end function
```

The function *computeValue* returns the result of evaluating *expression* with parameters *args*. Any *deref* function calls that occur within *expression* take the context to be

the *env* argument. When an attempt to dereference a private attribute occurs the *env* argument is compared with the sheet that defined that type. If they are the same then the expression is being called within the scope of the private attribute and a value is returned, if there is a mismatch then an access error is returned.

Users are able to provide controlled access to private attributes by defining functions within *TypeSheets*. The process of how these public functions are given access to private fields is presented in algorithm 3.

Algorithm 3 Encapsulation with User Defined Functions

```
1: function COMPILE(function)
2:     env ← GETCALLINGSHEET()
3:     return λx.COMPUTEVALUE(env, function, x)
4: end function
```

In the *compile* function a closure is formed over the *env* variable, its value is the context from which the function was compiled. The returned value is an anonymous function that accepts a sequence of arguments and computes the function value using these. This anonymous function is then associated with the function name (specified by the *#FUN_NAME* sheet header) to complete the compilation.

The purpose of the closure is to add state to the compiled function. In this case a value for *env* is now associated with that anonymous function. Compiling a function within a *TypeSheet* will set the value of *env* to be that sheet for all invocations of the function. As a result the scope of the function will include private attributes for that type regardless of where the function is called.

4.5 Remaining Work

Features that were not implemented in the final application but were discussed in the design include functions that can have multiple signatures and automatic compilation of sheet definitions.

Allowing multiple functions to have the same name but different parameter counts would unify the type constructors for user defined types. It would also make *FunctionSheets* more versatile enabling users to implement functions such as *get(map, key)* and *get(map, key, key-not-found)* whereby a default value can be specified if a key does not exist in a map.

Another enhancement to *FunctionSheets* would be to provide local scoping of function parameters. Currently if there are multiple function definitions in one sheet then parameter names must be unique within the sheet for test values to work correctly. Local scoping within functions is a standard across most programming languages and adding this to *Jeksy* would tighten the integration between spreadsheets and programming languages.

Chapter 5

Evaluation

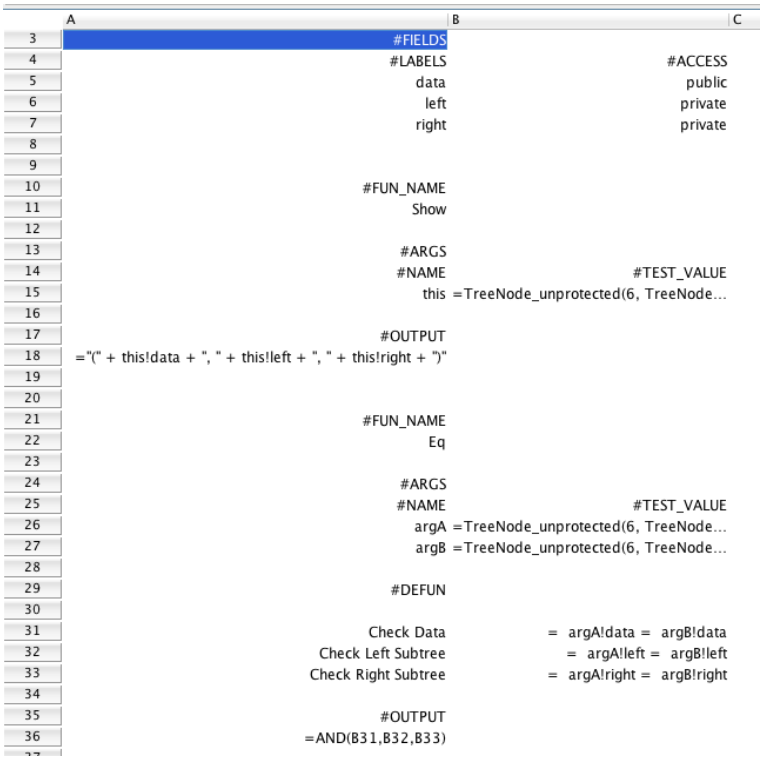
The evaluation of *Jeksy* is separated into two sections; the first looks at the capabilities of the new features and ultimately how well they integrate into the spreadsheet concept. This is done by examining non-trivial examples that utilise the range of new tools available to the user. The second component of the evaluation investigates how *Jeksy* can be used to improve existing industry level spreadsheets.

5.1 Spreadsheet Characteristics of *Jeksy*

As one of the goals of project was to not only add new features but to ensure they fit within the spreadsheet paradigm, it is critical to define what actually constitutes the *"look and feel"* of a spreadsheet. The most prominent aspect of spreadsheets is the tabular layout where the contents of a sheet are structured into rectangular blocks, this is unlike a conventional programming language that opts for a tree structured syntax.

Other traits of a spreadsheet include the feedback loop whereby a user can enter a value and immediately see the effect of its evaluation. Not only does this make interactive development fast, but it makes it easier to understand the program because data can be seen flowing through it actively.

Another constituent to the appearance of spreadsheets is the notation and syntax used in expressions. Ensuring that the syntax is consistent reduces the amount of content that must be remembered in order to operate the program; it also aids the user in understanding what new operators do because intuitively features that look similar should behave in a similar manner.



	A	B	C
3	#FIELDS		
4	#LABELS		#ACCESS
5	data		public
6	left		private
7	right		private
8			
9			
10	#FUN_NAME		
11	Show		
12			
13	#ARGS		
14	#NAME		#TEST_VALUE
15	this	=TreeNode_unprotected(6, TreeNode...	
16			
17	#OUTPUT		
18	="(" + this!data + ", " + this!left + ", " + this!right + ")"		
19			
20			
21	#FUN_NAME		
22	Eq		
23			
24	#ARGS		
25	#NAME		#TEST_VALUE
26	argA	=TreeNode_unprotected(6, TreeNode...	
27	argB	=TreeNode_unprotected(6, TreeNode...	
28			
29	#DEFUN		
30			
31	Check Data	= argA!data = argB!data	
32	Check Left Subtree	= argA!left = argB!left	
33	Check Right Subtree	= argA!right = argB!right	
34			
35	#OUTPUT		
36	=AND(B31,B32,B33)		

Figure 5.1: Tree Node *TypeSheet* Definition showing type attributes and overridden functions

5.1.1 Binary Tree Definition

When implementing traditional programming language features in a spreadsheet it seemed fitting to implement a classic data structure in a spreadsheet to demonstrate the potential of *Jeksy*. The following example consists of the definition for a positive integer binary tree along with basic operations *search* and *insert*. These examples use the value *-1* to represent the null pointer though a more general binary tree could use a *NULL* user defined type.

The first component of the tree implementation is the definition of the *TreeNode* type as shown in 5.1. Upon inspection it looks like any normal spreadsheet with the evaluation of expressions suppressed. The structure itself is concise; expressions have annotations to give them context and the components of the definition (fields and functions) are separated clearly. Cell expressions within the sheet - in particular the definition of the function *Eq* - are consistent with formulas likely to be found in a typical *Excel* spreadsheet. Where features specific to *Jeksy* are introduced the appearance and fundamental meaning of the expressions are still within expectations of a regular spreadsheet. For example the *!* operator commonly used to refer to cells in other sheets is also used by *Jeksy* to dereference type attributes. Although technically different, the meaning of *"look up the value stored at x in structure y"* is common across both applications of the operator and so conceptually it is easy to transition between the two.

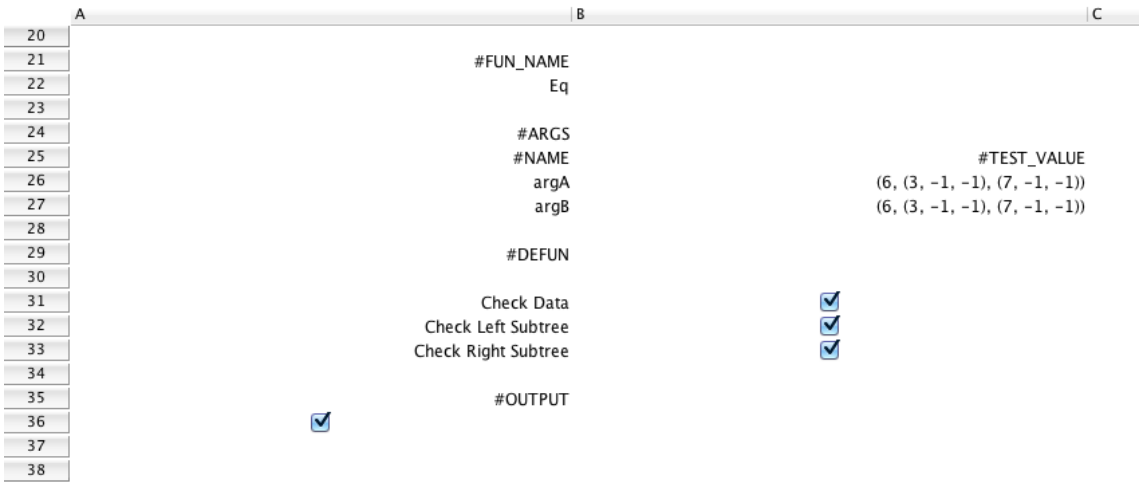


Figure 5.2: Interactive Debugging Part A

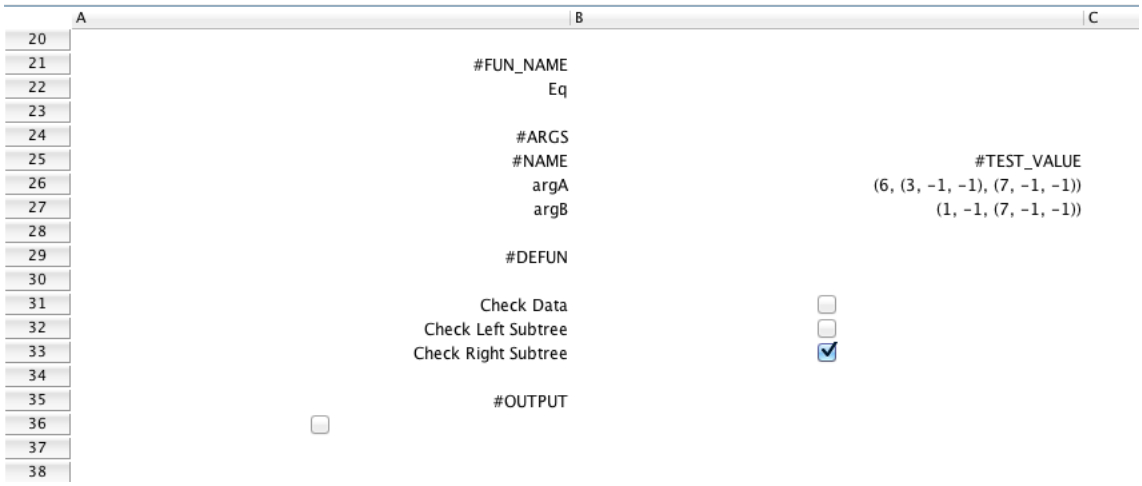


Figure 5.3: Interactive Debugging Part B

As discussed previously the ability to enter data and see the results immediately is crucial to spreadsheets and *Jeksy* embraces that idea when it comes to *FunctionSheets*. Figures 5.2 and 5.3 provide an example of how test arguments can be used to ensure the function is correct. Here a user can change the test values to check that each conditional in the function *AND* is computed properly. Another situation where visual feedback is available immediately is when defining the string representation of the type. Instead of having to save or compile the file first, the result of changing the *Show* function can be viewed immediately by examining the contents of the output cell. This behaviour utilises the benefits of spreadsheets whilst still providing a structured way to define functions.

5.1.2 Binary Tree Functions

For the functions that operate on a binary to work correctly certain invariants must hold. In this implementation those conditions are:

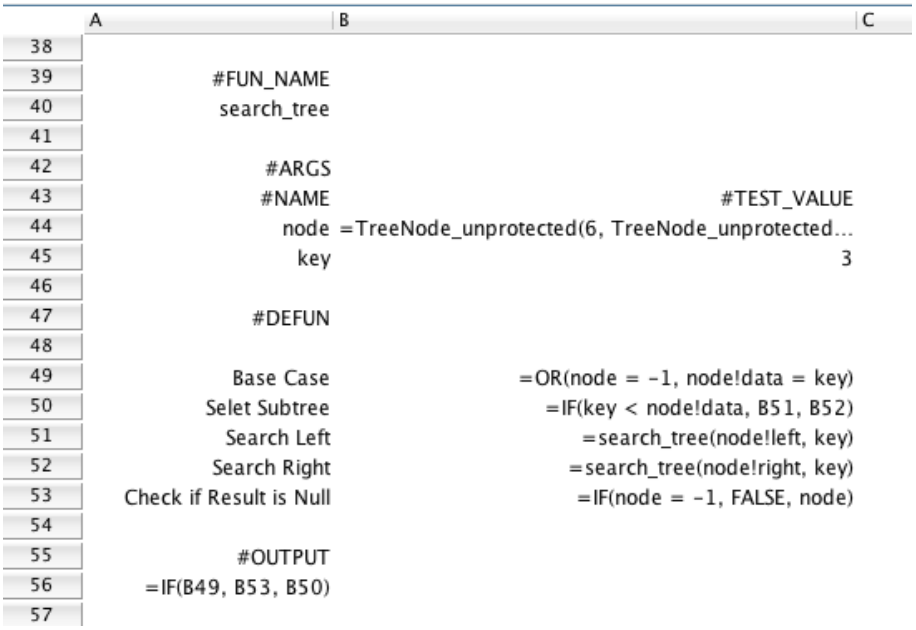
- Let *x* be a node in a binary tree. If *y* is a node in the left subtree of *x*, then *y.key* is strictly less than *x.key*
- If *y* is a node in the right subtree of *x*, then *y.key* is strictly greater than *x.key*.
- Both left and right subtrees are binary trees.
- All keys are unique.

If it were the case that the *left* and *right* attributes were publicly accessible it would be possible for a user to create a tree where the invariants do not hold, causing search operations to behave unexpectedly. *Jeksy* is able to enforce these properties by making the pointer attributes private and only allowing access to them from within the *TypeSheet* definition itself.

	A	B
59	#FUN_NAME	
60	insert_tree	
61		
62	#ARGS	
63	#NAME	#TEST_VALUE
64	current_node	=TreeNode_unprotected(6, -1, -1)
65	value	6
66		
67	#DEFUN	
68		
69	Node to Insert	=TreeNode_unprotected(value, -1, -1)
70		
71	Base Case	=current_node = -1
72	Check if 'value' Exists	=IF(value = current_node!data, TreeNode_unprotected(current_node!data, current_node!left , current_node!right), B73)
73	Select Subtree	=IF(value < current_node!data, B74, B75)
74	Recurse Left	=TreeNode_unprotected(current_node!data, insert_tree(current_node!left, value), current_node!right)
75	Recurse Right	=TreeNode_unprotected(current_node!data, current_node!left, insert_tree(current_node!right, value))
76		
77	#OUTPUT	
78	=IF(B71,B69,B72)	

Figure 5.4: Binary Tree Insert

Figure 5.4 shows the nested *FunctionSheet* definition for the function *insert_tree*. The term *nested* refers to the fact that the definition occurs within the *TypeSheet* rather than on its own. This gives the function access to the private attributes of the *TreeNode* type and the unprotected constructor function. Through (and only through) this function can a user build a tree. The result is that it is therefore not possible to break the properties of the structure at any time. The definition for the function *search_tree* (5.5) is also placed within the *TypeSheet* because it requires access to the both pointer attributes.



	A	B	C
38			
39	#FUN_NAME		
40	search_tree		
41			
42	#ARGS		
43	#NAME		#TEST_VALUE
44	node	=TreeNode_unprotected(6, TreeNode_unprotected...	
45	key		3
46			
47	#DEFUN		
48			
49	Base Case	=OR(node = -1, node!data = key)	
50	Selet Subtree	=IF(key < node!data, B51, B52)	
51	Search Left	=search_tree(node!left, key)	
52	Search Right	=search_tree(node!right, key)	
53	Check if Result is Null	=IF(node = -1, FALSE, node)	
54			
55	#OUTPUT		
56	=IF(B49, B53, B50)		
57			

Figure 5.5: Binary Tree Search

As the complexity of functions increase the benefit of *FunctionSheets* become more apparent. Implementing the functions in 5.4 and 5.5 as single line formulae would be unwieldy and exceptionally hard to read. Furthermore *FunctionSheets* encourage the user to separate components of the function into logical steps which in turn make it easy to inspect the value of intermediate expressions.

In practice when using an application such as *Microsoft Excel* functions of this size may be implemented using the *Visual Basic* editor. It must be acknowledged that this does have advantages over the approach of *FunctionSheets*; namely that the language itself is more expressive with access to more control statements as well as typed variables.

Using the spreadsheet allows the user to view the evaluation of statements within the function on the fly as changes are immediately propagated throughout the sheet. Another feature that contributes to the advantage of using spreadsheets to define functions is that it requires no further knowledge beyond that of the application itself. In practice most spreadsheet users do not have programming experience and so cannot make full use of the *Visual Basic* integration with *Excel*, it is reasonable however to assume that users of spreadsheets understand the spreadsheet language to some extent.

5.1.3 Limitations & Improvements

The binary tree example demonstrates how it is possible to build data structures in a way that is in keeping with spreadsheets while still retaining much of the functionality of a standard programming language; for example operator overloading and structured function definitions. However there are also some missing features in *Jeksy* which would improve the user experience and make the integration of *TypeSheets* and *FunctionSheets* into the spreadsheet environment more natural.

Using the binary tree as an example, there is no easy way to create a large instance of the data structure. Currently it relies on many repeated calls to the function *insert_tree* which quickly becomes cumbersome. A more idiomatic approach would be to allow *FunctionSheets* to operate on collections of data or cells, constructing the tree from a single data source. As spreadsheets are essentially functional programming languages an intuitive solution would be to provide higher order functions to the user which in conjunction with the ability to create collections, would greatly extend the power of *Jeksy*.

Although *Jeksy* does have a basic concept of a collection, the *cell set*, there is currently no way to use these within a *FunctionSheet*. Operations that make use of collections are currently limited to those that already ship with the application such as *SUM*. Providing higher order functions that work with sequences would enable the user to create *FunctionSheets* that take collections as well as individual values as arguments. Under this new system creating a binary tree from a large collection of values can be as simple as one function call:

Define seed value	*Cell* A1 *contains the value* -1
Construct tree	$reduce(insert\_tree, A1, A2:A100)$

The function *reduce* would be a predefined function that comes with *Jeksy* allowing the user to combine a collection into a single result. Its implementation would be based on a left fold operation because this mirrors the cell range notation *:* which operates left to right. In addition, left folds are tail recursive which is important when trying to avoid stack overflows caused by especially large collections. An example implementation might be:

$$reduce(function, val, []) = val$$
$$reduce(function, val, [x:xs]) = reduce(function, function(val, x), xs)$$

where $[x:xs]$ does not denote a cell range but a destructuring bind of the collection to the variables x (the head) and xs (the tail).

Calling *reduce(insert_tree, A1, A2:A100)* would then expand to:

$$insert\_tree(\ldots insert\_tree(insert\_tree(insert\_tree(A1, A2), A3), A4)\ldots A100)$$

Another shortcoming with the current version of *Jeksy* is when creating *FuntionSheets*. Currently a user will not be allowed to enter an expression that contains unknown symbols, for example an undefined function. This is an issue when making recursive calls or using a function parameter because the symbol will not have been loaded into the application at that time, therefore preventing the user from entering the expression.

The current solution to this problem is the user to save the file after completing the function signature, that is, after the name and arguments to the function have been specified. This allows the user to then continue with the function definition, issuing recursive calls if necessary as now a signature exists in the run-time. Unfortunately this has two main drawbacks; the first being that it exposes some of the internal workings of the application and requires knowledge of the tool that is not immediately apparent. The second issue is that the process of having to save the file explicitly does not sit comfortably with the spreadsheet concept of evaluating expressions instantly; a spreadsheet user should expect the contents of a sheet to become active straight away.

A better solution would be for *Jeksy* to compile the signature on the fly, thereby removing the need for the user to save the file before continuing with the definition. This could be triggered by a listener that checks for *FunctionSheet* headers being entered into the current sheet, particularly *#DEFUN*. At this point the sheet will contain the function arguments and name so it is possible for *Jeksy* to create a signature and load it into the running program. For small spreadsheets it would even be possible to compile the file any time the contents of a cell changes.

The final issue raised by the binary tree example is the difference between the evaluation of test values and the actual invocation of the function. Figure 5.6 shows the intermediate expressions for the *FunctionSheet insert_tree*. This image shows the expression for "Recurse Left" being evaluated to *(6, (8, -1, -1), -1)* however when the function is actually called this statement is never reached. Currently there is no notification to the user that this statement would not be executed given the current test values, 5.6 presents a solution. Statements that are reached given the test cases are given colour overlays to signify that they are relevant, the numbers show the order of execution.

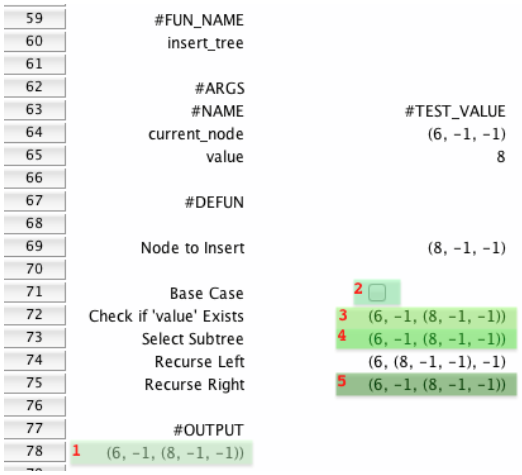


Figure 5.6: Path of Execution Overlay

5.2 Real World Applications

The underlying motivation for adding new features was to create a spreadsheet application that would produce better programs. The definition of a *better* spreadsheet in this context is largely based on two aspects; how well structured the program is and the probability of it producing errors. The following example is a real spreadsheet obtained from industry along with a version implemented using *Jeksy* to demonstrate the benefits of the new system and its features.

5.2.1 *Excel* Data Triangles

This spreadsheet uses a sheet as a data source from which the values are then imported into other sheets and used in calculations. The data (5.7) is structured into rows where each rows has a key that is used in the look up process. Figure 5.8 displays the triangle of keys that are computed using a formula and parameters defined at the top of the sheet. Once the desired keys have been determined, another triangle is built using the keys to look up values in the data source, in the example 5.9 the target is the *Paid* column.

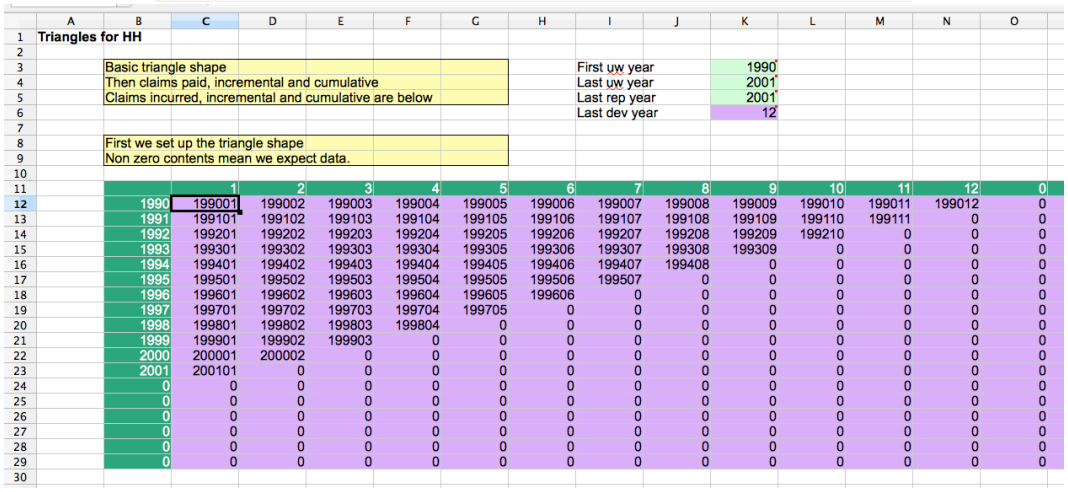
A10 =B10*100+C10-B10+1

	A	B	C	D	E	F	G	H	I	J
1	**HH data from import**									
2								Used in offset formulae		
3		Assign keys to the imported data						Key column		0
4		Key is uwyr*100+dev yr						paid column		3
5								incurred column		4
6								UwYr column		1
7								premium column		5
8								Rep yr column		2
9	Key	Uw year	Rep year	Paid	Incurred	Premium				
10	199001	1990	1990	16,294	26,860	30,000				
11	199101	1991	1991	18,777	29,990	30,648				
12	199201	1992	1992	18,860	31,717	34,724				
13	199301	1993	1993	22,254	34,124	41,391				
14	199401	1994	1994	26,222	40,209	47,765				
15	199501	1995	1995	30,073	51,576	57,701				
16	199601	1996	1996	37,149	59,987	63,817				
17	199701	1997	1997	42,954	62,719	72,241				
18	199801	1998	1998	46,744	74,673	81,777				
19	199901	1999	1999	56,647	87,899	98,336				
20	200001	2000	2000	61,245	96,879	105,023				
21	200101	2001	2001	69,467	110,951	116,786				
22	200201	2002	2002	72,939	108,783	122,742				
23	200301	2003	2003	73,495	116,255	129,861				

Figure 5.7: Data Source

The method to look up data by key used in this example relies on the *Excel* function *SUMIF*; it takes two ranges (*range* and *sum_range*) and a *criteria*. If the current element in *range* matches the *criteria*, then the corresponding element in *sum_range* is added to the running total. In this context it means if the current element in the *Key* column matches the supplied key, add the corresponding element in the *Paid* column to the result.

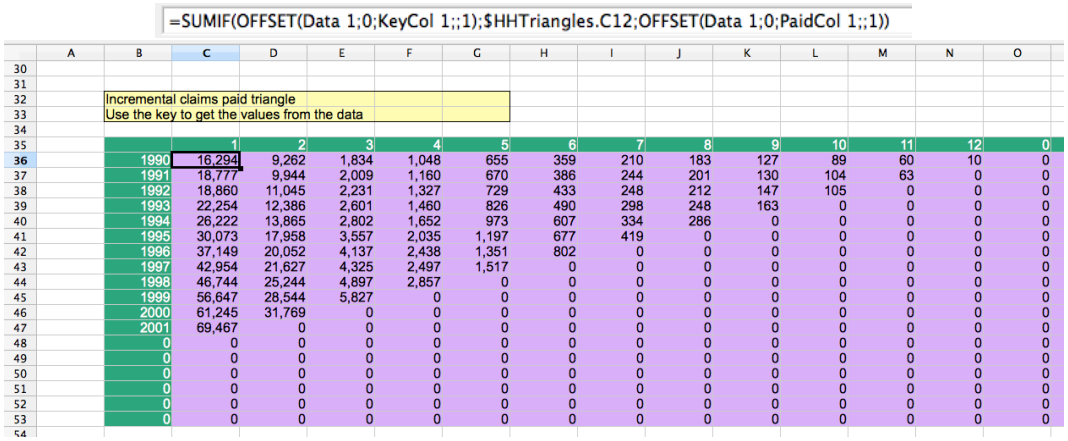
=IF(AND($B12>0;C$11>0;$B12+C$11<=LastRepYr 1+1);$B12*100+C$11;0)



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	**Triangles for HH**														
2															
3		Basic triangle shape							First uw year		1990				
4		Then claims paid, incremental and cumulative							Last uw year		2001				
5		Claims incurred, incremental and cumulative are below							Last rep year		2001				
6									Last dev year		12				
7															
8		First we set up the triangle shape													
9		Non zero contents mean we expect data.													
10															
11			1	2	3	4	5	6	7	8	9	10	11	12	0
12		1990	199001	199002	199003	199004	199005	199006	199007	199008	199009	199010	199011	199012	0
13		1991	199101	199102	199103	199104	199105	199106	199107	199108	199109	199110	199111	0	0
14		1992	199201	199202	199203	199204	199205	199206	199207	199208	199209	199210	0	0	0
15		1993	199301	199302	199303	199304	199305	199306	199307	199308	199309	0	0	0	0
16		1994	199401	199402	199403	199404	199405	199406	199407	199408	0	0	0	0	0
17		1995	199501	199502	199503	199504	199505	199506	199507	0	0	0	0	0	0
18		1996	199601	199602	199603	199604	199605	199606	0	0	0	0	0	0	0
19		1997	199701	199702	199703	199704	199705	0	0	0	0	0	0	0	0
20		1998	199801	199802	199803	199804	0	0	0	0	0	0	0	0	0
21		1999	199901	199902	199903	0	0	0	0	0	0	0	0	0	0
22		2000	200001	200002	0	0	0	0	0	0	0	0	0	0	0
23		2001	200101	0	0	0	0	0	0	0	0	0	0	0	0
24		0	0	0	0	0	0	0	0	0	0	0	0	0	0
25		0	0	0	0	0	0	0	0	0	0	0	0	0	0
26		0	0	0	0	0	0	0	0	0	0	0	0	0	0
27		0	0	0	0	0	0	0	0	0	0	0	0	0	0
28		0	0	0	0	0	0	0	0	0	0	0	0	0	0
29		0	0	0	0	0	0	0	0	0	0	0	0	0	0
30															

Figure 5.8: Computing the Keys Required

=SUMIF(OFFSET(Data 1;0;KeyCol 1;;1);$HHTriangles.C12;OFFSET(Data 1;0;PaidCol 1;;1))



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
30															
31															
32		Incremental claims paid triangle													
33		Use the key to get the values from the data													
34															
35			1	2	3	4	5	6	7	8	9	10	11	12	0
36		1990	16,294	9,262	1,834	1,048	655	359	210	183	127	89	60	10	0
37		1991	18,777	9,944	2,009	1,160	670	386	244	201	130	104	63	0	0
38		1992	18,860	11,045	2,231	1,327	729	433	248	212	147	105	0	0	0
39		1993	22,254	12,386	2,601	1,460	826	490	298	248	163	0	0	0	0
40		1994	26,222	13,865	2,802	1,652	973	607	334	286	0	0	0	0	0
41		1995	30,073	17,958	3,557	2,035	1,197	677	419	0	0	0	0	0	0
42		1996	37,149	20,052	4,137	2,438	1,351	802	0	0	0	0	0	0	0
43		1997	42,954	21,627	4,325	2,497	1,517	0	0	0	0	0	0	0	0
44		1998	46,744	25,244	4,897	2,857	0	0	0	0	0	0	0	0	0
45		1999	56,647	28,544	5,827	0	0	0	0	0	0	0	0	0	0
46		2000	61,245	31,769	0	0	0	0	0	0	0	0	0	0	0
47		2001	69,467	0	0	0	0	0	0	0	0	0	0	0	0
48		0	0	0	0	0	0	0	0	0	0	0	0	0	0
49		0	0	0	0	0	0	0	0	0	0	0	0	0	0
50		0	0	0	0	0	0	0	0	0	0	0	0	0	0
51		0	0	0	0	0	0	0	0	0	0	0	0	0	0
52		0	0	0	0	0	0	0	0	0	0	0	0	0	0
53		0	0	0	0	0	0	0	0	0	0	0	0	0	0
54															

Figure 5.9: Key Look Up

5.2.2 *Jeksy* Data Triangles

In the *Jeksy* based implementation the structure of the data is captured in a user defined type *DataRow*. The signature of the type is as follows:

DataRow

Attributes	Functions
key - private	*build(uw_year, rep_year, paid, next)*
uw_year	*Show(data_row)*
rep_year	*key(data_row)*
paid	*paid(data_row)*
next - private	

There are two points of interest in the *DataRow* type; the custom constructor function *build* and the *next* attribute. The attribute *next* is a pointer to another *DataRow* instance. This allows the user to combine related *DataRows* into linked lists which can then be traversed recursively. The purpose of the *build* function is to compute the key based on both year parameters and set it accordingly. As *key* is private a user cannot create a *DataRow* instance where the key is specified explicitly, this guarantees that two *DataRow* types with the same *uw_year* and *rep_year* will always have the same key. Another property of the *build* function is that it ensures that the keys are unique. When a new *DataRow* is created, the list starting from *next* is searched for the current *key*. The function only constructs a new type if its key cannot be found in the list.

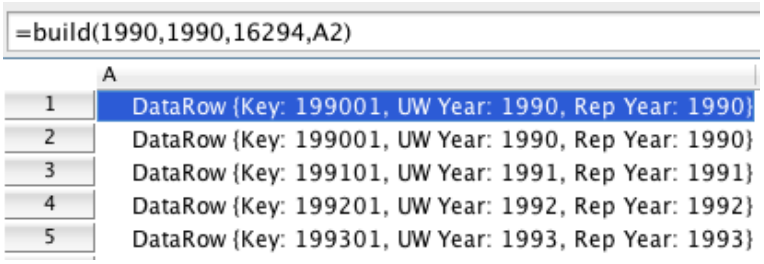


Figure 5.10: *Jeksy* Triangles Data Source. Note the *Show* function has been overridden to allow the user to view the private attribute *key*.

Figure 5.10 demonstrates how the data is represented in *Jeksy*. Cell *A1* serves as the head of the list with the next list element appearing in *A2* and so on. The values themselves are inserted directly into the build function however they could just as easily be imported from other cells or sheets (as is the case in the *Excel* version).

Unlike the spreadsheet built in *Excel*, *Jeksy* does not use the *SUMIF* function to look up the data. Instead the linked list is traversed recursively returning the row with the matching key (5.11).

```
get(node,k) = IF(node!key= k, node, IF(node!next = -1, "ERROR", get(node!next,k)))
```

Figure 5.11: Function to find element with matching key

```
=paid(get(Data!A1,B4))
```

Figure 5.12: *Jeksy* Triangle Formula

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
20															
21		1	2	3	4	5	6	7	8	9	10	11	12	0	0
22	1990	16294	9262	1834	1048	655	359	210	183	127	89	60	10	0	0
23	1991	18777	9944	2009	1160	670	386	244	201	130	104	63	0	0	0
24	1992	18860	11045	2231	1327	729	433	248	212	147	105	0	0	0	0
25	1993	22254	12386	2601	1460	826	490	298	248	163	0	0	0	0	0
26	1994	26222	13865	2802	1652	973	607	334	286	0	0	0	0	0	0
27	1995	30073	17958	3557	2035	1197	677	419	0	0	0	0	0	0	0
28	1996	37149	20052	4137	2438	1351	802	0	0	0	0	0	0	0	0
29	1997	42954	21627	4325	2497	1517	0	0	0	0	0	0	0	0	0
30	1998	46744	25244	4897	2857	0	0	0	0	0	0	0	0	0	0
31	1999	56647	28544	5827	0	0	0	0	0	0	0	0	0	0	0
32	2000	61245	31769	0	0	0	0	0	0	0	0	0	0	0	0
33	2001	69467	0	0	0	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
38															

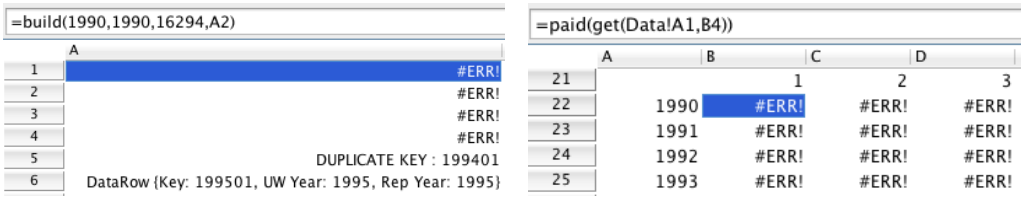
Figure 5.13: *Jeksy* Triangle

Figure 5.12 shows the formula that is used to populate the triangle in 5.13. The cell *B4* corresponds to the key required to fill that position in the triangle and *Data!A1* is the head of the linked list, this is located in a sheet called *Data*. Function *paid* is simply an accessor function that returns the value of the *paid* attribute for the given type instance. It could be replaced with a dereference operation however this approach allows the user to apply additional computation over the value (e.g. rounding) if desired.

5.2.3 Comparison

Whilst the *Excel* solution works correctly there are some aspects of the design which could lead to future errors when maintaining or re-using the program; the *Jeksy* spreadsheet addresses some of these issues.

When selecting the data from the source to import into the triangles an assumption is made that all the keys in the source will be unique. Should this assumption be broken and duplicate keys exist in the data, the resulting look up will be the summation of these values. As the return types of both correct and incorrect results are numeric there is no way to ensure that the value of the expression is in fact right without checking the data. This means that to be guaranteed of a correct result the user must check for duplicates themselves; this is both time consuming and potentially redundant. Whilst *Excel* based solutions exist to remove repeated values in a range, they are not particularly well suited to this task. Should new data be added to source the de-duplication process must repeated again over the entire collection. Here it is possible that a user forgets this process or incorrectly selects the range, missing repeated keys.



(a) List becomes invalid with duplicates.

(b) Value look up fails as a result.

Figure 5.14: Duplicate Detection

The *Jeksy* solution is superior because it is visually apparent to the user when the uniqueness invariant is broken. If a duplicate key is added to the data set the head of the linked list will become invalid, as shown in 5.14(a). Fetching any values using the function *get* will return an error instead of a number, see 5.14(b). Furthermore the replicated key will be marked in the data set so tracking the source of the error is relatively quick.

Another benefit of this solution is that it is not coupled to the formatting of a particular region in a sheet, rather it is bound to the data itself. Simply using the *DataRow* type and *build* function are enough to guarantee that values returned from the function *get* are unique.

In addition to reducing the likelihood of errors the spreadsheet can also be evaluated on its structure and ease of use. Although there is some complexity in defining the types and recursive look up functions used in the *Jeksy* solution, much of this can be hidden to the spreadsheet user by deploying as a library or API. The actual functions used to fetch and display the data are considerably simpler and easier to understand. When comparing the formulas:

- *=SUMIF(OFFSET(Data 1;0;KeyCol 1;;1);$HHTriangles.C12;OFFSET(Data 1;0;PaidCol 1;;1))*
- *=paid(get(Data!A1,B4))*

it is self evident that the *Jeksy* expression is clearer; its purpose to fetch the *paid* value. At first glance a user might believe that the *Excel* formula is summing values however only with greater understanding of the domain does it become obvious that it is intended to select a single value.

A further advantage of using the type driven approach is that it does not force the user to store the data in a contiguous block. This means that it is possible to structure the data in a way that is more manageable to the spreadsheet user whilst still being able to operate over it as single unit. In this example for instance, a user could easily use multiple sheets of *DataRows* by simply linking them together via pointers rather than combining the results into a single sheet. Not only does this save time but it also maintains the original separation of the data.

Chapter 6

Conclusion

This report presents the spreadsheet application *Jeksy*; an attempt to offer spreadsheet users classic programming functionality through a convincing spreadsheet interface. Two features were introduced: *FunctionSheets* and *TypeSheets*. Using these it is possible to define structured function and type definitions in a style that is in keeping with common spreadsheet design patterns.

6.1 Main Achievements

Summary of the main achievements using *FunctionSheets* and *TypeSheets*:

- Define a binary tree user type where invariants are guaranteed by the encapsulation of type attributes.
- Implement some of the standard binary tree operations using recursive *FunctionSheets*, all using the spreadsheet syntax.
- Interactively test and debug binary tree functions using the test mode available in *FunctionSheets*. The main example is the implementation of the function *Eq*.
- Derive sections of an existing industry spreadsheet using *FunctionSheets* and *TypeSheets*.
- Greatly simplify the formulas used in the example spreadsheet.
  e.g. *=paid(get(Data!A1,B4))*.
- Mitigate potential errors caused by the use of the *SUMIF* function in the example spreadsheet. Incorrect values are now returned as errors rather than numeric values.

6.2 Future Directions

There are multiple directions of further work available, some are due to insufficient implementation time and others as a result of analysing the project outcomes. They are:

- Permitting functions to have multiple signatures that have different parameter counts. This would unify public and private type constructors as well as making for interesting examples with *FunctionSheets*.
- Local scoping of function parameters inside *FunctionSheets*.
- Automatic compilation of definitions to further enhance the spreadsheet appearance of *Jeksy*. A user should expect features declared in a definition to become active immediately.
- Support for collections of raw values such as vectors and matrices. In conjunction with user types this could make for very advanced data structures such as hash tables.
- Higher order functions such as *map* and *reduce*, or more generally the ability to pass functions as arguments. As discussed in the evaluation, functions such as *reduce* are a very good way of avoiding many nested function calls when creating large data structures.
- Enhancing test mode for *FunctionSheets*. In particular providing visual feedback to the user indication the path of executing within a function. This is important when the definition contains conditional statements. A candidate solution is the *Path of Execution Overlay* presented in the evaluation.

Bibliography

[1] R. R. Panko, "Reports of spreadsheet errors in practice." http://panko.shidler.hawaii.edu/ssr/Cases.htm. Accessed: 2013-04-04.

[2] R. R. Panko, "What we know about spreadsheet errors," *Journal of End User Computing's (Special issue on Scaling Up End User Development)*, vol. 10, no. 2, pp. 15–21, 1998.

[3] R. Freeman, "A slip of the chip on computer spread sheets can cost millions," *The Wall Street Journal*, August 1984.

[4] D. Whittaker, "Spreadsheet errors and techniques for finding them," *Management Accounting*, pp. 50–51, October 1999.

[5] *Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks*, (29th Hawaii Int. Conf. on System Sciences), 1996.

[6] R. Abraham, M. Burnett, and M. Erwig, "Spreadsheet programming," *Encyclopedia of Computer Science and Engineering*, 2009.

[7] S. P. Jones, A. Blackwell, and M. Burnett, "A user-centred approach to functions in excel," June 2003.

[8] J. Malmström, "Haxcel: A spreadsheet interface to haskell written in java," Master's thesis, Mlardalen University, March 2004.

[9] M. Manns, "Pyspread." http://manns.github.com/pyspread/index.html/.

[10] E. Puybaret, "Jeks." http://www.eteks.com/jeks/en/. Accessed: 2013-04-04.

[11] F. S. F. Inc., "GNU GPL." http://www.gnu.org/copyleft/gpl.txt. Accessed: 2013-04-04.