Designing a spreadsheet in a pure functional graph rewriting language

Walter de Hoon

October '93

Master Thesis 300

University of Nijmegen

Department of Informatics

Faculty of Mathematics and Informatics

Preface

Finally I have reached the point where I am about to finish my study and leave the University to go into the 'real' world. What you see here is the result of my final exam. But, of course, I could not have done it all by myself.

Therefore I would like to thank my advisors Rinus Plasmeijer and Marko van Eekelen who put me back on track when I needed it. I would also like to thank Peter Achten who was always willing to help me when I came in with questions concerning the IO-System. Then I would like to thank Luc Rutten, who let me use his program and was always there to answer questions about it. And finally I would like to thank all the others working at the department and especially my fellow students who created a pleasant spot for me to go to work.

A special thank goes out to my girlfriend who was extra supportive and understanding for the past year.

Nijmegen, October 1993,
Walter de Hoon.

Abstract

Input/Output (IO) and efficiency have always been a handicap when using functional languages. Therefore large applications written in such a functional language are rare. At the University of Nijmegen research has been done to develop an efficient functional language called Clean and in addition to this language an IO-System has been developed and implemented to make IO possible in an elegant way.

What we did was taking a critical look at the practical applicability of Clean. We wanted to check whether it fulfils the needs of the programmer and whether the IO-System can live up to our high expectations?

We will first describe spreadsheets to give a general impression about the properties they have in common. Then we describe the interpreter that is used to evaluate the spreadsheet language. Then, using both these parts, we will show the development a the functional design of the spreadsheet. This showed us that the IO-System had to be extended with an extra feature. The design was adjusted to the extension and tested by implementing the spreadsheet.

We came to the conclusions that Clean is a elegant language that can be used fairly easily for writing huge programs. And the IO-System was quite easy to work with and using it we came to a stimulating result. Therefore the IO-System containing the extension does live up to our expectations.

Contents:

Chapter 1
Introduction

Input/Output (IO) has always been a difficulty (you can say that it was virtually impossible) when using functional languages and the same goes up for efficiency. So functional languages just could not match imperative languages. Therefore large applications written in such a functional language are rare.

At the University of Nijmegen research has been done to develop a functional programming language called Clean. By optimising just about every aspect of the language they tried to make it comparably efficient to imperative languages. In addition to this language an IO-System has been developed to make IO possible in an elegant way. Cleans efficiency was tested by running small programs, which lead to results, matching those of imperative languages like C. However, they did not have any big test cases so it was hard to see whether Clean does live up to the expectations when it is used for writing large applications.

This master's thesis is written in order to do research to the practical applicability of Clean. The language was tested to see whether there are any advantages or disadvantages to the use of Clean when writing an application and whether the properties of the language can be used to our advantage. We have also tested the IO-System to see whether it lives up to the expectations.

In this testcase we chose for developing a spreadsheet because it is an application that many people are familiar with and because it contains a virtual interface it will be a good testcase for the IO-System. We started with first looking at some spreadsheets and then designing one ourselves but from a different (functional) point of view. This resulted in a spreadsheet with as elegant feature Clean as the spreadsheet language.

Normal spreadsheets have their own 'language' in which entries (the input strings) are defined. These ad-hoc notations and languages do fulfil the needs but they are not really satisfying. It is much better to have a spreadsheet in which an existing programming language has to be used for the entries. Using Clean and having an interpreter written in Clean to our disposal, the spreadsheet was based on that interpreter which provides us with a very powerful tool that can evaluate Clean and Miranda expressions. Therefore this tool can even evaluate higher order functions like ZF-expressions, something no other spreadsheet can do.

This thesis has the following structure: Chapter 2 is used to give a description of the basics; it contains a short introduction on how to work with Clean and its IO-System, needed for displaying the contents of the spreadsheet and for disc access, and a description of spreadsheets in general, about their properties and their looks. In chapter 2 we will see that a language is used to formulate the entries. This is followed in Chapter 3 with a thorough explanation of the interpreter that is used to evaluate the spreadsheet language. This is done by describing what has to be done when an input string is evaluated. In chapter 4 we describe the special aspects of using a functional language for developing a spreadsheet. The use of the interpreter and the definition of column-functions are introduced here. This description leads to the design of the spreadsheet.

This design shows however that the current IO-System does not live up to our needs. Therefore chapter 5 is used to describe the extension of the IO-System and accordingly the adjustments to the design. Then, in chapter 6, the result of testing the design, the implementation, is described. At the end some conclusions will be formulated on the work that has been done.

Chapter 2
The Basics

This chapter is used to give a description of Clean which is used for the design and implementation of the spreadsheet and a description of spreadsheets in general.

2.1 Clean

Clean[1] is a lazy, pure functional, high order language. This means respectively that values are only calculated when they are needed, it is not possible to make assignments and that functions themselves can be given to other functions as parameter and functions can have functions as a result. An example of a function written in Clean is '`Fib x`' which computes the x-th Fibonacci number. This function can be defined in several ways. Here are given two examples. The first one makes use of pattern matching and the other one uses guards:

```
::  Fib1 INT -> INT;
    Fib1 0 -> 0                          ;
    Fib1 1 -> 1                          ;
    Fib1 2 -> 1                          ;
    Fib1 x
    -> + (Fib1 (-- x)) (Fib1 (- x 2))    ;

::  Fib2 INT -> INT;
    Fib2 x
    -> + (Fib2 (-- x)) (Fib2 (- x 2))    ,IF > x 2
    -> 1                                 ,IF > x 0,
    -> 0                                 ;
```

The lines starting with a double colon are the *type rules*. These rules are used by the compiler but also are very useful for the user to determine if his program is correct. The part starting with '`IF`' is the *guard*. When alternatives are followed by a guard this alternative will only be executed when the guard is TRUE. The evaluation is done from top to bottom (the upper rule containing a TRUE-guard is executed) and from left to right.

To execute a function a *starting rule* '`Start`' is needed. So if you want to calculate the 10th Fibonacci number the following function must be obtained in the program:

---

[1]Here is only given a short introduction into Clean. For further details you can take a look at references [1] and [2].

```
::   Start -> INT;
     Start -> Fib1 10;
```

Clean also makes use of *sharing*. This means that if a certain value is needed by more than one function, this value (node) can be shared. As a result computations have to be done only once. This sharing is used in the following example (the shared possibility of `'Fib1 x'`):

```
Fib1 x
->   + (Fib1 y) (Fib1 (-- y)) ,
     y:   -- x                ;
```

In this example 'y' is only a small computation, but you can imagine that it can also be a huge computation like `Fib1 1000`.

An other property of Clean is the use of *Unique types* (notation `UNQ`). This means that if a certain part of a program is unique, it can be used only once, so it can have only one reference. These unique nodes are very important for the IO-System. When you know that, for example, a window or a data-structure is unique an update of that element can be done destructively meaning that no new node has to be build but the old node can be adjusted.

Writing big programs can result in a program that is no longer clear to the programmer. Therefore Clean, like many other languages, offers the possibility of *modular programming*. This kind of programming forces the use of two kinds of modules: the '.icl'-files and the '.dcl'-files. The first ones are used to write down the actual program , defining the functions (like Fib) and the second ones are used to report which rules and synonym-types can be used by other modules. This is done by taking the typing-rules from the functions that are exported. Splitting up a program like this results in an hierarchical structure of that program.

Finally we have the *strictness annotations* (notation: '!'). They are used to evaluate arguments eagerly in stead of lazily. They are used in the type-definitions of functions and in the synonym-types:

```
TYPE
::   Complex -> (!REAL, !REAL);

RULE
::   +C !Complex !Complex -> Complex;
     +C (r1, i1) (r2, i2) -> (+R r1 r2, +R i1 i2);
```

2.1.1 The IO-System

The functional IO-System[2] implemented for Clean is based on the UNQ-type attribute. Clean uses three kinds of environments to define its IO-System. First of all there is the `(UNQ) WORLD` which is initialised in the starting rule:

[2]Here only a short explanation is given. For further details I would like to refer to [2] and [3].

```
::   Start (UNQ WORLD) -> (UNQ WORLD,....);
```

From this world the environments (UNQ) FILES and EVENTS can be withdrawn.

The first one is used for the FILE-IO. Files can be opened from this FILES-environment, which can be done in different modes (Read-only, Write-only, etc.). In addition functions are defined to read or write characters, lines, ints or reals to or from these files. Notice that when two strings have to be written to a file this has to be done one by one. First one string is written to a file, delivering a file', which is used to write the second string to, delivering a file''. To make sure you do not write twice to the same file it is made unique.

The other environment is used for defining the WindowBased-IO (the EVENT-IO). This IO manipulates visual objects like windows, menus and dialogs. These manipulations take place after interactions with the user. These actions generate events that are evaluated by the program and result in a manipulation of one of the objects.

This EVENT-IO is started using the function StartIO. This function takes an IO-System (the definition of the initial windows, menu and dialogs), an initial state, a list of functions that have to be evaluated when the IO System is initialised and the EVENTS-environment. QuitIO is used to quit. This function results in termination of the StartIO, resulting in the final state and the final events.

2.2 Spreadsheet Specification

A spreadsheet can be seen as a graphical representation of computations. You can imagine that if you want to know the profits of a large company with an enormous turnover, you want to be able to see which divisions made profits and which did not. Well this is where a spreadsheet comes in handy. It can give you a good overview about what is going on. An important property of a spreadsheet is that when some of the data change you do not have to do all the computations again (they are done by the spreadsheet) and the intermediate results can also be shown in the sheet and used for further computations.

This section will be used to tell a little about spreadsheets in general and about their properties. To get a good overview the manuals of Lotus 123, Excel and Quatro Pro ([4], [5] and [6]) were used.

2.2.1 What is a Spreadsheet?

When you look at a spreadsheet you will see that it is divided into rows and columns. The intersections of these rows and columns are called cells. These cells are indexed, horizontally by one or two capitals (A or BD) and vertically by integers, can be selected and strings (entries) can be assigned to them when they are. These entries will be evaluated and the results are displayed. When a user adds an entry to the sheet, this entry together with some information obtained by the interpreter are stored in the selected cell. The definition of a simple cell could have the following syntax[3]:

```
:: CELL        -> (STRING, VALUE);
:: String CELL -> STRING;
:: Value CELL  -> VALUE;
```

where `String` and `Value` are functions to get the information from the cell.

A lot of entries can be assigned to a spreadsheet and therefore it has to consist out of a set of cells. If one looks at the graphical representation of the spreadsheet it is clear that the most easy way to group a set of cells would be in a matrix, resulting in the following definition of a sheet:

```
:: SHEET  -> [[CELL]]
```

To get information from cell (x, y) the address is passed to a select function that selects that cell from the sheet that is referred to.

The entries of the cells can be simple strings ("1", "Index" "'c'", etc.) that can be easily evaluated (to (Int 1), (String "Index"), (Char 'c'), etc.), but they also may contain *functions* ("1+1", "sum [1,2,3,4], etc.) which evaluates to the results of the functions (to (Int 2) and (Int 10)).

In addition to this we have the *references* that can be used in the entries. This is one of the most important features of a spreadsheet. They allow us the use of values defined in other cells. Furthermore spreadsheets contain the possibility to use predefined functions like average and sum and they allow the user to define functions or macros himself. A thorough explanation of these two will follow in the following sections.

The entries are dependent on certain rules that together describe a language that is used for the entries of the spreadsheet. The spreadsheet contains a tool to evaluate entries according to these rules. The way the entries are evaluated is dependant on the first character (-s) of that entry. So when an entry is given to the sheet as input the evaluator manipulates that string to a value and assigns this result to the cell that is selected. Finally the evaluated value is displayed on screen.

2.2.2 References used in entries

One of the most important features of a spreadsheet is the use of references. There are two kinds of references, the direct references and the label references.

As told before the cells are indexed and the entries of the cells can be functions. These two are combined in the direct references. It is possible to use the index (A1 of G6) in the entry and therefore it can also be used as a parameter of a function. The entries can therefore be "A1" or "A1 + A2".

To be able to use label references a label has to be assigned to a certain (group of) cell (-s). This label then can be used in the entry.

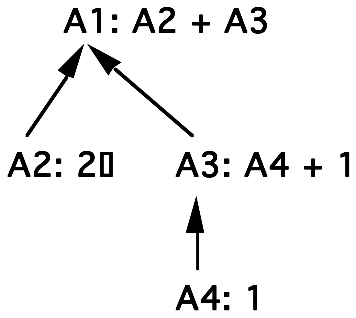
---

[3]Clean notation is used because it is also used in later chapters for the design and implementation and therefore the same notation is used there.

Because of these references an hierarchical structure is created in which a cell at level 'x' depends on a cell on level 'x+1'. The top elements (the leafs) of the graph contain entries without references. When a cell in layer x is changed, the cells in the lower levels that can be reached from that cell also have to be updated. As a result of this changing the value of one cell might have a chain reaction which recursively causes the change of large group of cells.

Example:

A1= "A2 + A3"
A2= "2"
A3= "A4 + 1"
A4= "1"



If we look at the following example we
can see that if A2 changes, A1 has to change too. If on the other hand A4 changes A3 and recursively A1 have to change.

Usually the spreadsheet has a property to prevent having to update the whole sheet when the entry of a cell changes. This can be done by saving the addresses of the cells that depend on that particular cell (the dependencies). If a certain cell changes you just have to look at its dependencies to find out which cells have to change too.

Using references to other cells the possibility exists to define cycles. Normally however these cycles do not terminate. Therefore spreadsheets usually contain a cycle detector to prevent the program from trying to evaluate an infinite recursion. As a result the spreadsheet will exist out of non-cyclic graphs.

2.2.3 Functions

Spreadsheets also contain a set of predefined functions that can be used in the entries. You could say that they are a part of the language and therefore can be used in those entries. The functions that are predefined are functions that are used often like sum or average. However, not all of the functions that are needed by the user are predefined. Therefore the user is offered the possibility to develop his own functions and macro's, which is useful when he uses a particular function several times but that function is not predefined. These user defined functions can be used the same way as the predefined functions.

2.2.4 The Interface

The userinterface is the part of the program that is directly used by the user. There is no need for him to know what is going on underneath the surface. He does not have to think about the computations and evaluation of values. All he knows (and has to know) is (or can be) shown by the userinterface. It is the only way for him to access the program.

A spreadsheet is menu driven which means that from the menu various functions[4] can be applied on the sheet. First of all the functions that perform IO. Those are functions to save or read sheets, to change the names of the sheets or to quit a sheet. Then you have the functions that can be used while editing a sheet (Cut/Copy Paste, Undo, Delete). Spreadsheets offer the opportunity to hide rows and columns and they have the possibility to make diagrams in addition to the sheet.

There are functions that can change the format of a sheet, to change the row height, the column width, the extension (whether the number implies $ or Fl and whether it must have 2 or 3 digits behind the comma etc.). It also offers the possibility to define home-made functions and to select (pre-) defined functions and macro's. And finally it is possible to define, delete and select labels.

Furthermore it has a part in which the evaluated values and the entries are displayed. This is done in a window. The values are displayed in the so-called "cells", squares separated by horizontal and vertical lines. To keep track of the address an index is displayed in the window constantly.

The cells in the sheet can be selected. This can be done by mouse or by the arrows defined on the keyboard .

Furthermore, the window contains a part in which the text that is edited by the user is displayed. After the text is edited the computed value is shown in the selected square. The string and its evaluated value are assigned to the cell matching the selected square.

Finally it is also possible to have more than one sheet open. This is done by opening multiple windows.

[4]Notice that these functions are other functions than the ones used in the entries. These functions are functions that can be applied by the user on the sheet, or on elements of that sheet like cells or the environment.

Chapter 3
The interpreter

An interpreter for a certain language is a program that receives a string as input and evaluates this string according to the rules that describe that language.

As stated in chapter 2 the entries (input strings) of a spreadsheet are also strings that have to be evaluated and this evaluation is also dependent on certain rules. Therefore spreadsheets need a tool to parse and evaluate the entries. This tool will have to recognise the predefined functions and it has to evaluate them. Furthermore a spreadsheet offers the user the possibility to define functions that also have to be recognised and evaluated by the tool.

One way to achieve this is developing a language that describes the entries of the cells and then build an interpreter that is able to evaluate these strings. However we had an interpreter[5] to our disposal and we thought it is preferable to use an existing interpreter instead of developing one ourselves because it is more easy to adjust an available interpreter to our needs than developing a new one. It also makes it possible to formulate the entries in a Clean/Miranda like language which is more elegant and easy to work with than using ad-hoc notations.

This interpreter mentioned here is able to evaluate Clean and Miranda code. It is only possible to use one language at a time and therefore the program contains a function that tells which language is used. In addition the interpreter contains two sets of operators describing the syntax of both the languages (see Appendix A). These operators are used to define the predefined functions (see Appendix B). The interpreter evaluates the strings and recognises the operators, the functions and their parameters. Functions that are not defined are given back as a result. Whenever the interpreter comes across a part of an input string that does not add up to the syntax, it will produce an error.

3.1 Description of the interpreter

An interpreter receives a string as input. To evaluate this string the interpreter uses an environment. This environment consists of an ordered set of rules: the evaluated predefined functions read from a file. The syntax of the defined functions depends on certain rules. The following grammar defines the rules that the functions have to live up to.

[5] The interpreter used here was developed by Luc Rutten for his PHD-thesis at the University of Nijmegen.

<u>Grammar for the predefined functions</u>:

```
Rules:          []      |
                [Rule,Rules2];
Rules2:         []      |
                [";",Rule,Rules2];
Rule:           [Expression,"=",Expression];
Expressions:    []      |
                [Expression,Expressions];
Expression:     Basic |
                [Expression,Op,Expression] |
                [Identifier,Expressions];
Basic:          Int         |
                Real        |
                Char        |
                String      |
                Bool        ;
Identifier:     see Appendix A;
Op:             see Appendix A;
```

For the evaluation of the entries the interpreter first has to evaluate his environment, needed for that evaluation. This evaluation can be separated into the following stages: The reading of the string that has to be evaluated, the parsing of the read string, converting the value to an environment (when needed), lambda-lifting the value (eliminating the where-clauses that can be obtained in the value) and calculating the result (when needed).

Because the interpreter is written in Clean the reading and evaluation of the environment is done lazy. So only when the input string contains one of the functions defined in the environment for the first time the evaluator has to check the environment and therefore read and evaluate it.

In the following sections the working of these five stages will be explained.

3.2 Reading the string

The string that has to be evaluated must be read. The (standard) environment is read from a file, containing the predefined functions. This is done using the following function:

```
:: ReadLines UNQ FILE -> STRING;
```

The input string that the user feeds to the interpreter is read from the standard input using:

```
:: FReadLine UNQ FILE -> (STRING, UNQ FILE);
```

Here the file is taken from `FILES`-environment using the function `StdIO`.

3.3 The parser

The next step is transforming the string into a value. This is done by the parser using the following function:

```
:: String2Value STRING -> Value;
```

To evaluate the string it first has to be scanned. The scanner splits up the string and adds a constructor to each part describing its type. The scanning of the string results in a list of `Lex`-items. These `Lex`-items are defined as follows:

```
:: Lex ->
     LexInt     STRING |
     LexReal    STRING |
     LexString  STRING |
     LexChar    STRING |
     LexId      STRING | if not an operator
     LexOp      STRING | recognised from operator list
     Lbrack     STRING |
     Rbrack     STRING ;
```

The string is split up according to the grammar defined in Appendix C.

The result of the scanner is passed on and transformed to a tree. At this stage the operands are recognised and connected to their arguments in the constructor Fx. This results in a value of type ParseTree:

```
:: ParseTree ->
     Fx STRING [ParseTree]      |
     In STRING                  |
     Re STRING                  |
     C STRING                   |
     S STRING                   |
     ParseError STRING          |
     B STRING                   ;
```

Finally the ParseTree is transformed to a Value. Here the arguments of the values are recursively evaluated and the operands and function symbols become compound expressions:

```
:: Value ->
     F Id [Value]               |
     Int INT                    |
     Real REAL                  |
     Char CHAR                  |
     Bool BOOL                  |
     ValueError STRING          ;

:: Id -> STRING;
```

The only remarkable thing is that `Value` does not contain a constructor for type string, because they are parsed as lists of characters.

3.4 From Value to Environment

After the predefined functions are read an environment is evaluated from the parsed value. This is done using the following function:

```
:: Value2Env Value -> Env;
```

This section is used to describe this environment, to show how it is evaluated out of a `Value` and what the result looks like.

As described in the grammar of the functions the rules are separated by semicolons. These semicolons are evaluated as identifiers. The string `[rule,";",rules]` is parsed to:

```
F ";" [evalrule | Evaluate rules],
evalrule: F "=" [F f args, rhs];
```

where '`f`' is the function name, `args` is a list of values (the arguments), `rhs` is the righthandside (the result) of the function and `Evaluate rules` is a list of other rules.

When the parsed value is transformed to an environment the value is first split up into a list of values where each value is a parsed rule. Then this list of values is transformed to an environment. This is done by first taking the first rule from the list, which is the first argument of a certain function 'f'. Then the other arguments of that same function 'f' that are situated on top of the rest of the rules are taken from this list and these definitions are combined into a rule. This is done by making an alt (a tuple containing the arguments and the righthandside of one alternative) of all the selected alternatives and then connecting these alts in a list to the function name (the Id).

Finally the rest is recursively transformed to an environment and connected to the rule.

This results in an environment looks like as follows:

```
:: Env    -> [Rule];
:: Rule   -> (Id, [Alt]);
:: Alt    -> ([Value], Value);
```

3.5 The lambda-lifter

The program contains a function `TheLanguage` which selects the language that is used. There are two languages that can be selected, Clean or Miranda. When Miranda is chosen the strings can contain where-clauses. During evaluation these clauses have to be eliminated before the value can be evaluated any further. This where-elimination is called lambda-lifting.

This lambda-lifting has to be done for both the value and the environment. Therefore the program contains the following two functions for the job:

```
:: Lambda_lift_env Env [Id] -> Env;
:: Lambda_lift Value Env -> (Value, Env);
```

They are, however, comparable in the way they work. The `Lambda_lift_env` gets an environment, which might contain where-clauses in its rules and the `Lambda_lift` gets a `Value` that might contain a where-clause. They both rewrite their input to an output without these where-clauses. The environment is updated according to the functions defined in the where-clauses.

Notice that the environment is passed to the lambda-lifter as parameter. They both need that environment to be able to update the value. The lambda-lifter needs it to determine which symbols have a meaning.

3.6 The evaluator

Finally the evaluator delivers a value that can not be evaluated any further. It delivers the final result of the initial input string as a value. This value may be symbolic, containing identifiers that are not defined.

```
:: E Env Value -> Value;
```

3.7 Modular structure

The description defined above leads to the following modular structure of the interpreter. Notice here that Runtime contains the reading functions and the rest (except the parsing part) is obtained the module miranda.

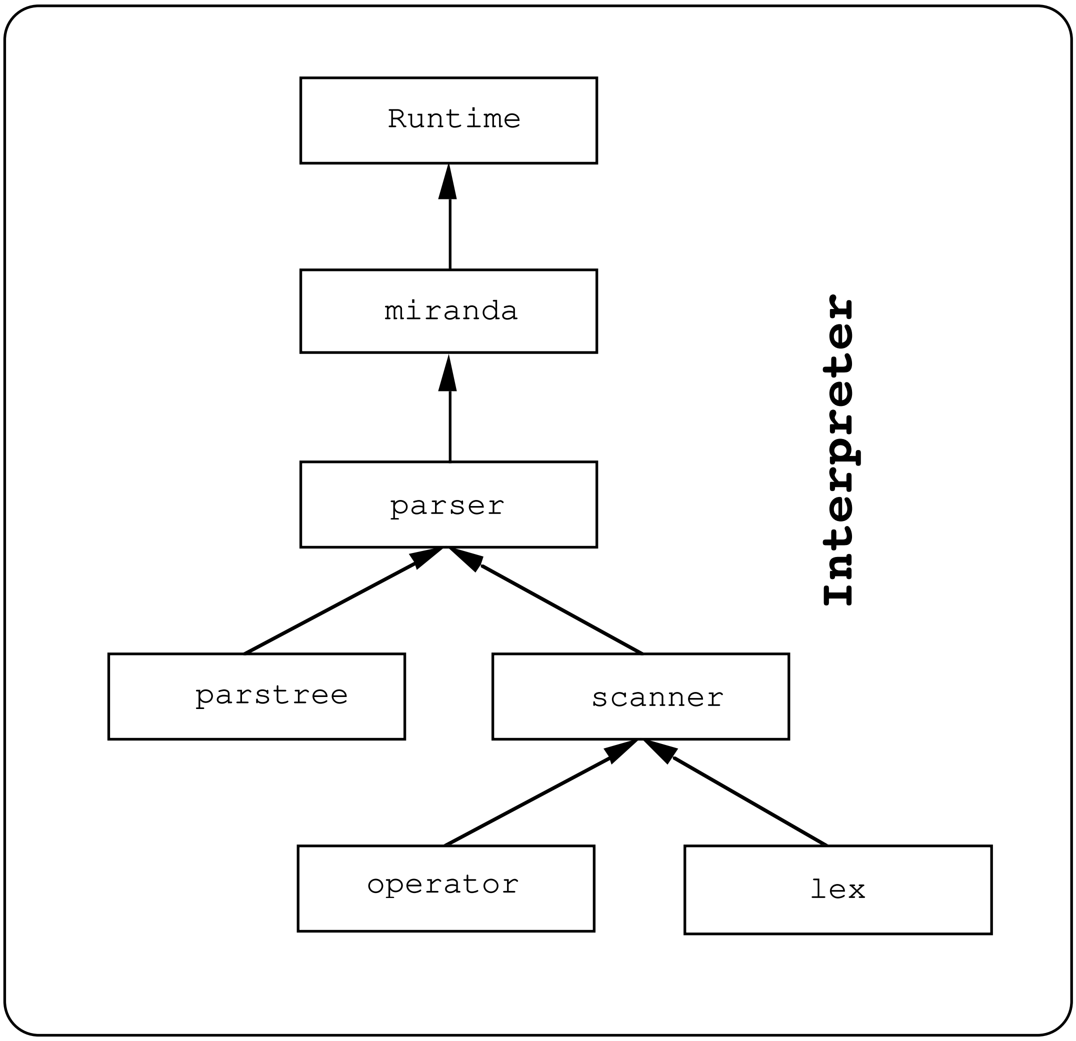


Figure 1: The modular structure of the interpreter

Chapter 4

Designing a Spreadsheet

This chapter is used to give a design of a spreadsheet. To do so we will first give a functional definition of a spreadsheet. Then, after formulating the internal structure of the spreadsheet together with its modular (hierarchical) structure the design will be formulated.

4.1 A functional definition

Looking at a spreadsheet from a functional point of view the environment used by the interpreter offer us an interesting angle for designing a spreadsheet and the definition of the references.

Spreadsheets do not use tuples containing two integers for their addressing. Instead they use references that exist out of one or two capitals followed by one ore two digits. When these references are used they will have to be changed to an address that can be given as an argument of the select function.

An other way to look at references is to see them as functions. A sheet will then consist out of a list of functions. The first part of the references, existing out of one or two capitals, will be the function names. These functions get an integer as argument and result in the evaluated value of a entry. Now we can define a type `SHEET` as follows:

```
::   SHEET      -> [=> INT Value];
```

Functions of type (`=> INT Value`) will be called column-functions.

Now this definition can be compared to the environment of the interpreter. As explained in the previous chapter the environment consists of a set of rules (`Env -> [Rule]`). If we can transform the column-functions to such a rule the spreadsheet can be implemented as a set of rules (Environment) too. This would lead to the following definition of a sheet:

```
SHEET -> Env;
```

An Env is a list of rules. Therefore the column-functions will have to be made of type Rule. This can be done easily by making the function name the 'id' of the rule and adding it together with the arguments in a tuple. The arguments will have to be defined adding the evaluated function-argument (the y-index) to a list and adding the evaluated entry to it in a tuple (see chapter 3).

Take a look at the following example where A1 contains value (Int 4) and A2 delivers (Int 5). When these alternatives of function "A" are add to the environment the matching rule of function "A" will be defined as follows:

```
("A",  [  ([Int 1], Int 4),
```

```
                ([Int 2], Int 5)])
```

This definition will simplify getting information from the sheet. A reference "A1" will result in the application of function `A' with his argument `1'. And when the value of cell A1 is needed the value (`F "A" [Int 1]`) has to be evaluated (`E Env Value -> Value`). Because the function is defined for the given argument the result (Int 10) can be gained from the sheet immediately, without further computations.

But what are the advantages of defining the sheet this way. Well, defining the columns as functions results in the possibility (for the user) to combine them with the predefined functions (described in chapter 2) which offers the possibility to use the columnfunctions in high-order functions like ZF-expressions:

```
A5 = [A i | i <- [1..4]];
A6 = sum [A i | i <- [1..5]];
A7 = sum (map A [1..6]);
```

And an other advantage is that for evaluating entries these column-functions can be used. This way you do not have to search through the matrix of cells (as defined in the previous section).

4.2 Using the definition

In the previous section we gave a functional definition of a spreadsheet. We saw that columns can be described as functions that takes an integer and delivers a value, and that these functions can be add to an environment. But what happens if a string is assigned to a cell and the environment has to be updated, and what happens if the entry is changed, and how can the corresponding function be adjusted and how can it be saved?

4.2.1 A separate environment

As told before the string that is assigned to a cell has to be parsed and evaluated. This will be done by the interpreter, described in the previous chapter. This interpreter uses an environment for the evaluation of strings. All the functions that are available for the user are defined in this environment.

The problem we have using this environment is that more than one sheet can be opened at the same time. When this is the case it will be no longer clear which function (or which alternative) belongs to which sheet. This can be resolved by adding a new environment to each sheet (This environment will be called the local environment and the other the global. Be aware however that in Clean there is no such thing as a global definition). The column- and label-functions can now be added to this environment and not interfering with each other.

4.2.2 Adding an entry to a the environment

Once we have got an (local) environment defined in our program we can also add the column-functions to that environment. When a string 's' is assigned to a cell A1 this string has to be parsed and transformed to a value. This value together with its address are then used to update the local environment as follows. If defined, the rule with Id "A" has to be updated by adding the new alternative between the old ones. The arguments are sorted to optimise the searching algorithm. When an argument already exists it is replaced by the new alternative. If this function is not defined yet the environment is expanded with this rule. Once the function is defined in the environment other cells can use that function.

Take a look at the example above. When A1 is assigned entry "10" function A will have to be updated with this new argument. But A1 is already defined and therefore is has to be replaced. The new definition of "A" will then look like as follows:

```
("A",  [  ([Int 1], Int 10),
          ([Int 2], Int 5)])
```

Now adding entry "3" to cell A4 would result in:

```
("A",  [  ([Int 1], Int 10),
          ([Int 2], Int 5),
          ([Int 4], Int 3)]
```

4.2.3 Adding a label to the environment

What goes up for entries also goes up for labels. Labels are used to name blocks of cells. When "label" is assigned to block x, this block will be evaluated to a value. Adding the label to the environment can be done by placing it on top of the old environment because labels have only one alternative (one with no arguments).

4.2.4 Saving a sheet

Saving the sheet is now a peace of cake. The entries and labels can be written to a file the way the predefined functions are defined in the standard environment (the same way as they are evaluated).

4.3 An internal representation

A spreadsheet contains more information than we saw in the previous section. This can be information that has to do with the efficiency of the program as well as information concerning the contents of the cells and the visual aspects of the sheet. All this information has to be stored in the spreadsheet. This is done in information structures. These structures are also hierarchical in structure. In this section first will be explained what information has to be stored and then will be shown the structure of this information.

4.3.1 The Cell

The most important information stored in the cells are the entries. These are the input strings given by the user. The user must be able to adjust these entries and in order to get them they have to be saved in the cells.

Then there is the parsing information. For efficiency reasons it could be stored in the cells as well as the environment. This offers us some possibilities:

- The first is to save the parsed information in the cell together with the final value. This way however functionality of the spreadsheet will be lost, because now it is not possible to define the column functions. Therefore this solution is not adequate.
- Another possibility is to store this information in the environment. This offers us two options. We can add the final result to the environment, which is not preferable because changing one entry would result in having to parse the dependant entries again and add their values to the environment again. Or we can parse the entries upto the point that the used column functions and label functions are not evaluated yet. This option however is not very useful either because this would result in having to recalculate functions each time they are used. So, again, the solution is not adequate.
- Then there is the option to save the final result (the result that has to be printed in the sheet) in the cell and to add a function that is dependent on other column functions to the environment. This has as advantage that the environment only has to be updated with the new entry and the functionality is preserved. The matrix of cells however has to be updated with the new values of all the cells that changed because of that initial change. This causes the program to run through the matrix many times. And again functions will have to be evaluated more then once.
- Finally we can do it the other way around. The cell keeps the function and the rule in the environment has the final result. When the environment has to change we can get the parsed functions out of the cells and evaluate these functions again and add the result to the environment.

In this case the best solution is the latter. The updating of the environment can be done fairly easy (just put the new alternative on top of the stack as described in the previous section) and the access of the matrix has been brought back to a minimum. When the sheet is read again the double elements vanish from the environment.

Changing the entry of a certain cell might affect a large group of cells in the sheet. Other cells can refer to this particular cell with labels or direct references. To avoid having to check the entire sheet for references to this particular cell a list of references is obtained in the cell. This way it is easy to change the dependencies of a cell and recursively also their dependencies.

Example:

If cell A1 contains the entry "A2", cell A2 contains at least a reference to A1 in his `used-by` list. If A2 changes the value of A1 also has to change. If the entry of A1 changes the `used-by` list of A2 and the `uses` list of A1 have to be updated too (the old ones are removed and then the new ones are add).

When the entry of cell A1 contains a reference to cell A2 the dependency list of cell A2 will have to be extended with the reference to cell A1. However when this cell A1 gets an other entry the old references have to vanish. We do not want to have to parse the entry again to get its references. Therefore the cell also contains a list of references to which it refers. With these two lists of references updating a cell is fairly easy and fast.

Finally there is the format information. This is information that can be different for every cell. Therefore this information is saved in the cell other than the information that tells something about a row or a column of cells or the whole sheet that is stored somewhere else. The properties meant here are properties like the font of a cell, the type of a cell (the forced type) which can be used if you know what the type of that cell has to be. Furthermore information about the representation of the represented values (like defined in chapter 3).

4.3.2 The Sheet

The cells and the (local) environment are on their turn stored in the sheet. To achieve this the sheet contains a matrix of cells (`[[CELL]]`) and an environment. Because (as we will see later) the user-defined functions are stored in the same file as the rest of the sheet, a text string that can be written to a file is added to the environment.

Furthermore the information about the format of groups of cells (rows and columns) is stored here. Cells can be hidden and their height and their width can be adjusted. However when a cell is hidden the whole row (or column) it is part of has to be hidden. The same can be said about the height and the width of a cell. To avoid having to change these properties in every pointed cell, these properties of a row (column) will be stored in a separate list defined in the sheet.

It also has a part which contains information concerning the interactions between the user and the program. This is information about the mouse and concerning the entries of the selected cells.

It contains information about the labels. The labels are also added to the environment, but when information is needed by the user about the labels he has to get this information from the sheet.

Since a spreadsheet offers the possibility to open more than one sheet at a time. To keep track of the sheet the `WindowId` also is kept in the sheet as well as the name of the sheet which has to be known when the sheet is saved.

Finally it contains format information concerning the whole sheet in stead of one cell. This is information like which font is used.

4.3.3 The State

Finally there is the total view on the information. Of course it contains a list of sheets defined above. Furthermore it contains information that is sheet-independent. Here you must think of a standard (global) environment that is read and evaluated during the initialisation of the program, together with its identifiers and information that is needed for IO.

4.3.4 The representation

Writing a spreadsheet in an imperative language this information would all be stored in global variables. Clean however is a pure functional language and therefore has no global variables. This problem is solved by storing the information in the `ProgramState`. This results in the following data structures:

Cell-Definition

```
::   Cell           -> UC (Entry, Format,
ParsedInput,                                    Uses,
IsUsedBy)
                    -> EC;
== (EC stands for EmptyCell and UC for UsedCell)
::   Format         -> (Font, Type)
::   ParsedInput    -> Value
::   Uses           -> ([Address], [STRING]);
                         == direct refs and labels
::   IsUsedBy       -> ([Address], [STRING]);
```

Sheet-Definition

```
::   Sheet          -> ( Identify, Matrix Cell,
                         Interface, Row, Column,
                         [Label], ParseEnv, Format);
::   Identify       -> ( WindowId, Name);
::   Interface:     -> ( Input, Frame);
::   Input          -> (
     BOOL,RefBlock,STRING,STRING);
::   Row            -> [(Height, Hidden)];
::   Column         -> [(Width, Hidden)];
::   Height         -> INT;
::   Width          -> INT;
::   Hidden         -> BOOL;
::   Label          -> (STRING, RefBlock, IsUsedBy);
::   RefBlock       -> Frame;
::   ParseEnv       -> (STRING, Env);
```

State-Definition

```
::   UNQ State      -> (!MyFiles, !Env, [Sheet]);
::   UNQ MyFiles    -> EFiles
```

```
                          -> UFiles !FILES;
    == Efiles means EmptyFiles and UFiles means UsedFiles
```

The `FILES` and the `Environment` of the state are made eager. The files because all actions on files have to be evaluated eagerly. When you write a sheet to a file (make a backup of it) you want to make sure this is done so that power failures will not result in losing all information. The environment is made eager because these are the initial predefined functions. You do not want them to be evaluated when they are used but you want them evaluated before the program is started.

As said before these information structures have an hierarchical structure. Such a hierarchy can be implemented in an hierarchical way. This will result in the modular structure displayed in figure 1

4.4 The design

We have seen the functional description and the internal representation of the spreadsheet. Now we will give the design, containing a thorough explanation of the interfaces between the modules will be given in the next sections.

4.4.1 The main module

This main module contains the starting rule.

```
::   Start UNQ WORLD -> (UNQ WORLD, State);
```

This function initialises the IO-system using `StartIO` which defines the initial windows, dialogs and the menu system.

This module also contains an initial window definition containing windowattributes. The functions defined by these attributes are defined in an underlying layer in the module `InteractiveFunctions` which contains the mouse- and keyboardfunction and the module `Update` that contains the updatefunction of the windows.

Finally it also contains some small functions that are defined by the menu.

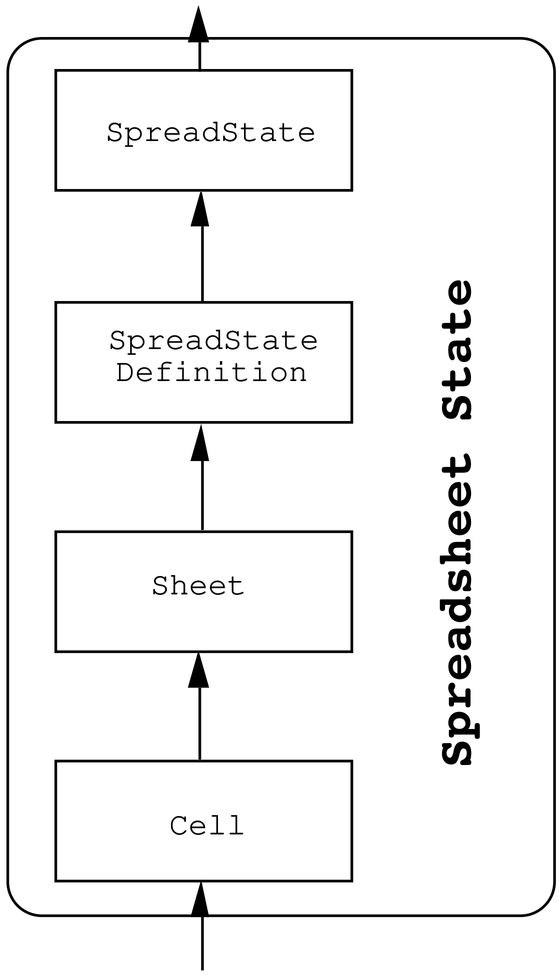


Figure 1: The modular structure of the Spreadsheet State

The MenuSystem

The menu-system contains menu-items that executes a function when selected. These functions can be separated into three kinds. First of all the 'file'-functions. These are functions like Open, Close, Quit and Save. Then there are the functions that are concerned directly with the environment. Here we can select functions from the environment or define functions of our own. The function for defining macros is captured in a new separate module because it is a function that uses `NestIO and needs a` new `IO-System` and a different `ProgramState`.

Then there are the `'edit'-functions` that are used to manipulate entries and labels. These are the Cut/Copy, Paste functions, the UndoTyping function, a function to select remote values (values defined in other sheets) and functions to add, select and delete labels.

Another part is the functions that perform disc-access. These are functions to read a sheet from disc and to save a sheet, defined in the module `DiscIO`.

Finally there are the `FormatFunctions`. These are functions that change the format of the sheet.

4.4.2 The Function editor

As told in a previous chapter the user must to be able to define functions of his own. The function editor is a part of the program that provides the user with this possibility. This module exports only one function to the layer above (the main module):

```
::   DefineFunc ParseEnv Env IO -> (ParseEnv, IO);
```

The ParseEnv is the local environment of the active sheet (window). This environment is updated and returned. The Env, the global environment, is needed to evaluate the new environment.

This function uses the functions `NestIO`. Therefore it needs the old IO-State to be able to define a new one. And it also needs a new State. This State is defined in an underlying module.

To be able to define this editor this module imports the used types from the module `SpreadStateDef`

4.4.3 The Update

The update-functions take care of the way everything is put on the screen. For the spreadsheet this means that it has to draw lines separating the cells, it has to print the defined values, it must have an area where the entries of the selected cells are shown and it has to show the index of the cells.

The lines and the contents are drawn using the update function, which is an argument of the window definition:

```
::   UpdateFunction UNQ s
          ->   =>   UpdateArea     (=>   s   (s,
[DrawFunction]));
::   UpdateArea      -> [Rectangle];
::   DrawFunction    -> => Picture Picture;
```

The `UpdateArea` are rectangles that have to be updated in the window. The drawfunctions are functions that do the actual updating executed by `DrawIn(-Active-)Window.`

The others on the other hand can not be defined in the current windowdefinition because it is not possible to have a certain part of the window constantly visible. Therefore the update possibilities of the windows will have to be expanded or changed. This will be described in the following chapter.

Evaluating expressions can be very time-consuming. Therefore you do not want values that are not visible and not used by visible cells to be evaluated. This can be solved by adding the addresses of the invisible cells to a list and, when scrolled, checking whether changed cells have become visible and then evaluate them. Using Clean this problem already is solved because it uses lazy evaluation and therefore values are only evaluated when needed.

4.4.4 The Interaction

This module is used to define the functions that have an effect on the (local) environment. This can mean adding and evaluating entries as well as adding labels to the environment.

An important part of a program is the way the program communicates with the user. How can the user assign entries to cells in a spreadsheet? Well, the Clean IO-System provides the program with windows. Such a window contains a list of attributes which can be used to define functions that are useful for the user. The attributes important here are the `Mouse` and `Keyboard` defining the mouse- and keyboardfunction:

```
:: MouseFunc WindowId        -> MouseFunction    State
IO;
:: MouseFunction MouseState State IO       ->
(State,IO);
:: KeyboardFunc WindowId    -> KeyboardFunction State
IO;
:: KeyboardFunction KeyboardState State IO ->
(State,IO);
```

The `MouseFunction` receives the co-ordinates of the place the mouse was clicked. Out of these co-ordinates the frame and the index of the matching cell are evaluated using the height and width of the cells. This information is stored in the input part of the sheet. When a new cell is selected the entry of the old cell is evaluated (only when it is changed) and added to the environment. As a result other cells may have to be updated too because they depend on the changed cell. Therefore a dependency-tree is build (see module `RefTree`) and all elements are updated. The windowframe will be adjusted if the selected cell is not completely visible. The entry of the selected cell must be displayed in the input-line. If the selected cell has evaluated to an error the type of error (contained in the value, the evaluated entry) is displayed in the error-line. In the window the newly selected cell will be marked as a rectangle (looks like a column that came towards the user out of the sheet).

The `KeyboardFunction` receives as input a character and four modifiers (`Shift`, `Option`, `Command`, `Control`). There are two kinds of characters that can be added. The first kind of characters are the ones that change the entry by adding or deleting a character or by replacing the cursor (`Command-Arrow` combination). The others are characters that force the entry to be evaluated, and (if done by typing an arrow) to select of the connecting cells. This will result in an update of the window which will be done the same way as described by the `MouseFunction`.

This module also contains a function manipulating labels:

```
::   LabelFunc State IO -> (State,IO);
```

LabelFunc opens a dialog. This dialog contains an edit text to be able to give a label, a scrolling list to select a label and buttons for adding, selecting and deleting labels:

```
:: SaveLabel RefBlock WindowId DialogInfo State IO
```

```
                                                   ->
(State,IO);
:: DelLabel WindowId DialogInfo State IO->
(State,IO);
:: SelectLabel WindowId DialogInfo State IO
                                                   ->
(State,IO);
```

To evaluate the strings and labels an evaluation function and the Environment-type have to be imported from the module Evaluator and to be able to update the window the updatefunction is gained from the module `Update`.

4.4.5 Cycle detection

This module provides the program with the possibility to build a tree containing all the cells depending on a certain cell. To achieve this the following types and function are defined:

```
TYPE
::   TreeRef   -> AR Address | LR STRING;
::   RefTree   -> TNode TreeRef [RefTree];

RULE
::   CheckCycle Address [Label] (Matrix Cell)
                                         ->
(BOOL,STRING,RefTree);
```

`CheckCycle` gets the `address` of the changed cell, the `LabelList` (labels can also be part of a cycle) and the `Matrix of Cells` (containing the dependencies). This function delivers a `Bool` telling whether a cycle is created by adding the new entry and, if it does contain a cycle, `STRING` contains the addresses of the cells obtained in the cycle and finally `RefTree` contains the dependency-tree.

The RefTree is evaluated by taking the dependencies of the address and checking whether that address is equal to the initially given address (this is not done for the initial step because an address is always equal to himself). If the addresses are equal a cycle is detected the path from that cell to the root (the initial address) is converted to a string and if not recursively his `RefTree` is build. The `RefTrees` belonging to a certain address are connected to that address in a `RefTree`. Notice that if no cells depend on a certain cell his list of `RefTrees` is empty.

4.4.6 The Evaluator

When an entry is inserted in the sheet it first is evaluated by the interpreter using the standard (global) environment, not containing the label- and column-functions. The result, a parsed function, will be saved in the cell that is selected to prevent having to parse the entry again when one of the cells it depends on changes.

Then the parsed function will be evaluated again with the local environment. This is done by the function `EvalRef`:

```
::   EvalRef Value Env Env State -> (Value,State);
```

At this point the remote references (references to a sheet situated on disc) are evaluated first. Then the other references (inside the sheet) are removed by evaluating that value using the local environment and finally that result is evaluated using the standard environment because the result may contain predefined functions that had Refs as their argument and therefore are not evaluated yet. This will result in the final value that will be inserted in the local environment.

4.4.7 The IO

In this module functions are defined to save sheets and to read sheets. It is also used when values are needed that are stored in sheets situated on disc. To achieve this the following functions are defined:

```
::   Save State IO  -> (State,IO);
::   Read STRING WindowId State IO
                    -> (BOOL,State,IO);
::   ReadSheet UNQ FILE Sheet Env
                    -> (BOOL,STRING,UNQ FILE,Sheet);
```

The separate function `ReadSheet` is needed by the function `EvalRef` in the matching module. This function does not get the IO as a parameter and already has selected which file has to be read. However the function `ReadSheet` is used by the function `Read`. Notice also that saving the sheet must be done eagerly. This is forced by checking whether is succeeded to put the file that is saved back into the `FILES`-environment again.

The entries and labels can be saved and read fairly easily (according to the grammar defined by the interpreter). Entry A1 can be written as ["A1 = ",entry,";"]. The same goes up for the labels. Reading this information can now be done easily by reading and evaluating this file the same way as you read the standard environment. However there is one slight problem. The entry itself is also needed because it can be changed. And the spreadsheet contains more information like format-information and dependency-information.

Therefore we have chosen for a different solution. We decided to make one file that contains all this information. But not all information stored in the sheet is stored on disc because some information can be restored more easily when they are gained again from the surrounding information. Therefore we have chosen to store the following elements:

- Row
- Column
- Font
- Labels
- Matrix
- User defined functions

The cells themselves obtained in the matrix are not saved completely either. Again only the necessary information is stored:

- format information
- the addresses of the cells that depend on this cell
- the entry
- the parsed value (if the string made from the value can be changed back to that same parsed value again)

The parsed value is added to the file because the entries that are saved have to be read and evaluated to add them to the environment. These entries can depend on other entries that are not evaluated yet. In the case that one of those other entries are evaluated, that entry recursively has to be evaluated too. So if a entry depends on many other entries it can happen that it has to be evaluated many times. But we do not want to evaluate entries more than once. Therefore the parsed value information is added when it is changeable which means that the value can be evaluated to a string and then evaluated back to the same value (the most important problem is the error-handling).

4.4.8 The interpreter

The interpreter described in the previous chapter is very useful for the spreadsheet. There are however some minor adjustments that have to be taken care of. Take for instance the scanner. It can recognise labels and column-functions as identifiers but references (i.e. "`A1`" or "`A1..B4`") that have their argument connected to the function name can not be recognised because functionnames may contain digits. Therefore an extra `Lex`-item has to be added to the scanner. These references will be evaluated to a list of `Value` in the final result of the evaluation. To achieve this the grammar (defined in Appendix C) has to be expanded.

Finally the runtime module had to be adjusted. It was changed to a module `Interpreter` that imports all information and functions necessary for the spreadsheet. This is the only module that is imported by the spreadsheet.

Chapter 5

Extension of the IO System

As we saw in the previous chapter we had a problem defining the fixed part of a spreadsheet. The current window definition did not support that kind of use. As a solution a new attribute has been added to the definition. First the definition of this attribute will be explained and then will be shown how it fits into the design.

5.1 The Margin Attribute

The problem we had using the existing window definition was that it did not offer us the possibility to keep a certain part visible. This problem was solved by adding a new attribute to the windowdefinition, the `Margin`, which can be used for exactly that function. This `Margin` is defined as follows:

```
::    WindowAttribute UNQ s UNQ io
      -> .......
      -> Margin MarginDomain (Marginalia s)
                                     (MarginScroll s);
::    Marginalia UNQ s
      -> => MarginState (=> s (s,[DrawFunction]));
::    MarginScroll UNQ s
      -> => ScrollState (=> s (s,[DrawFunction]));
::    MarginDomain
      -> (!INT, !INT, !INT, !INT);
::    MarginState
      -> (!MarginSize, !ThumbValues);
::    ScrollState
      -> (!MarginSize, !ThumbValues, !Vector);
::    MarginSize
      -> (!INT, !INT);
::    ThumbValues
      -> (!INT, !INT);
::    Vector
      -> (!INT, !INT);
```

The `MarginSize` is a tuple containing the height and the width of the margin. The `ThumbValues` are needed to see what is the left top value of the window. The `Vector` that is given to the `MarginScroll` shows in which direction was scrolled. This way you do not have to update the whole margin when scrolled.

This `Margin` fulfils the needs when developing a spreadsheet. In this margin it is possible to show the index (which is dependent on the `ThumbValues`) and the entry of the selected cell.

One of the current restrictions of the margin is that it is not possible to use the mouse in the margin-area. If that would be possible the mouse could be used to place the cursor somewhere in the middle of the entry and to define buttons in the `Margin`. This is something that can be added to the `Margin` in future adjustments.

5.2 The UpdateFunctions

Having this new attribute results in having to define two updatefunctions, one for the window and one for the margin. Because it is possible to open more then one window the `WindowId` is given to the updatefunctions. This results in the following functions:

```
:: UpdateWin      WindowId  -> UpdateFunction   State;
:: UpdateMargin   WindowId  -> Marginalia       State;
:: TrackScroll    WindowId  -> MarginScroll     State;
```

`UpdateWindow` only has to make sure that the values (and the lines separating them) are displayed. The values that have to be displayed together with the co-ordinates (derived from the height and width saved in the row and column (section 4.2.2)) are taken from the `ProgramState`.

`UpdateMargin` has to make sure that the index is displayed, together with the entry-line (at the top of the sheet) and the error-line (at the bottom of the sheet). When a cell is selected the index-line is adjusted by inverting the corresponding area.

`TrackScroll` does the same a `UpdateMargin` except for the fact that it gets a vector that indicates the direction of the scrolling. When both directions (x and y) are not equal to zero `UpdateMargin` is used. Otherwise only the index is changed of the scrolled direction.

5.3 Modular structure

Finally we will give the modular structure gained from the design in chapter 4 and the adjustment in this chapter:

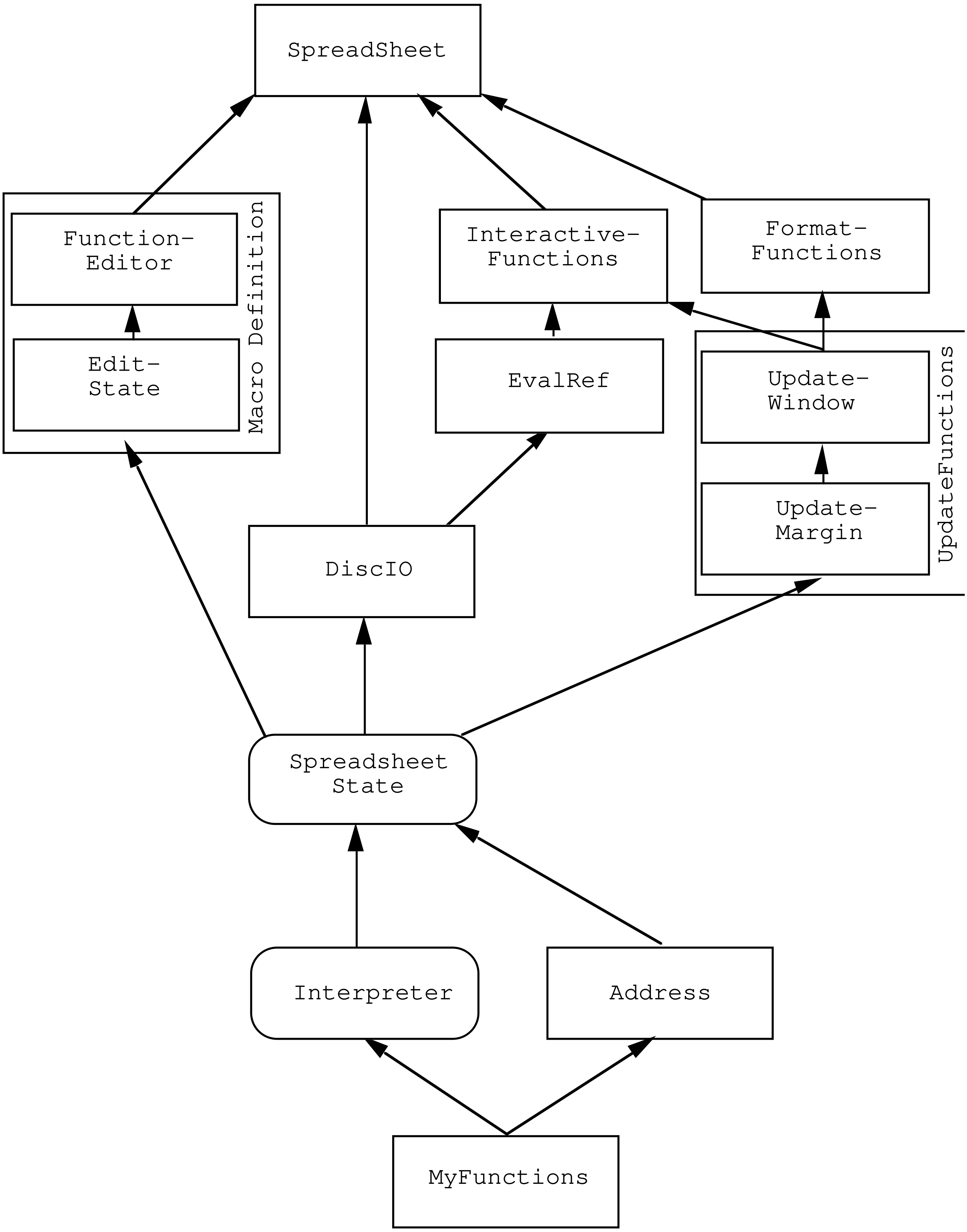


Figure 2: The modular structure of the Spreadsheet

Chapter 6
Implementation

After formulating the design of a spreadsheet now its implementation will be explained. This is done by first explaining about the ProgramState, followed by the other modules: the main module, the function editor, the updatefunctions, the interface, the evaluator, the IO and the interpreter.

6.1 The ProgramState

The IO of a program written in Clean is done using a **`ProgramState`**. This `ProgramState` is used to pass information from one function to another.

In the previous chapter we saw a description of the types that define the `ProgramState` of the spreadsheet. These types are, together with their access functions, defined in separate modules. These types are made abstract to make sure that little changes of these types will not result in having to change the whole program.

The module `CellDefinition` is used to define the type `Cell` together with its access functions. These functions are defined starting with `Get`... and `Insert`... (where ... stands for the value they have to get).

```
TYPE
::   Cell       -> UC (Entry, Format, ParsedInput,
                                     Uses, IsUsedBy)
                -> EC;

RULE
::   InsertEntry Entry Cell -> Cell;
::   GetEntry Cell -> Entry;
```

This is done for all the elements of the cell. For the `Uses` and `IsUsedBy` some extra update functions are defined. These are functions to add or remove an element to or from the list.

The type `Matrix` is also defined here, together with its access functions `Update` and `GetElt`. This is done because it is possible to get or update a cell that is not defined yet. In that case it must return (or insert) an empty value (in this case an empty cell).

Secondly we have the type `Sheet`. The access-functions for this type are defined as `S`... and `IS`... (get_from_Sheet and Insert_in_Sheet). This results in the following definitions:

```
TYPE
::   Sheet      -> ( Identify, Matrix Cell,
                     Interface, Row, Column,
                     [Label], ParseEnv, Format);

RULE
::   SName Sheet -> STRING;
::   ISName STRING Sheet -> Sheet;
```

The `WindowId` can not be set (this is done when initialising the sheet), and therefore `SId` is the only access-function for it. This module contains however a function `NewSheet` to make a new sheet where only the `Id` and the `Name` are defined and the rest are empty values:

```
::   NewSheet WindowId Name -> Sheet;
```

Finally we have the definition of the State. It contains a function that creates the initial ProgramState

```
TYPE
::   UNQ State -> (!MyFiles, !Env, [Sheet]);

RULE
::   SetInitState UNQ FILES -> State;
```

This function gets `Files` (which he got from the world) and delivers a `State`. These files are connected to the constructor `UFiles` to be able to get these files from the state again. When it is taken from the state it is replaced by `EFiles`. This has to be done this way because the files are unique and it is not possible to make a copy of a unique element.

6.2 The Main Module

As told in the design the main module contains the Starting-rule (see chapter 2) that begins the IO-session. Initially it will only define the menu-system. To start the IO the function `StartIO` needs a initial `ProgramState` (`SetInitState` from module `SpreadState`), a menubardefinition and a list of functions that have to be executed when initialising the IO-System.

For the spreadsheet one function is defined to read and evaluate the environment. This way the initialising takes longer, but when this is finished the user does not have to wait too long when entries are evaluated.

6.3 The Function editor

The editor is defined using NestIO. This is done because it is the only way to define a new independent menu-system, a state that contains the right information needed for the editing of text and a separate update-function.

This FunctionEditor is (in this case) a simple editor. It is a normal editor with some extra utilities to check whether the edited function is correct and to evaluate the functions to an environment. An other possibility is using an existing editor. The only thing that has to be done to adjust this editor to the needs is expanding the menu-system with the checking and evaluating functions. Furthermore some adjustments may be needed to see whether and how the functions can be saved and evaluated.

Finally, when quitting the NestIO, the SpreadState has to be updated with the right information taken from the EditState.

6.4 The Update

As made clear in Chapter 5 an extra windowattibute is needed to be able to define a spreadsheet. This resulted in a second updatefunction, the UpdateMargin. In the following sections both functions will be explained.

6.4.1 The UpdateWindow

As told in the previous section we do not want to update the whole window but the rectangles (UpdateArea). This results in the following function:

```
::   UpdateWin WindowId UpdateArea State
                              -> (State, [DrawFunction]);
```

The information necessary for the updating (the matrix, row, column, font and the mouseframe) is collected out of the ProgramState and is given to the UpdateWin together with the rectangles. This function updates the rectangles by passing them to the function UpdateRect one by one and delivers an list of drawfunctions. These lists are concatenated and finally the selected cell is drawn.

The UpdateRect is defined as follows:

```
:: UpdateRect Rectangle Env Row Column Font
                              -> [DrawFunction];
```

Each rectangle has to draw the following elements:

- Horizontal and vertical lines separating the cells.
- The values of the cells defined in that rectangle.

To update a rectangle information about the height and width of cells (the distances between the lines) is collected from the row and column. The Env is used to get the values that have to be printed and the font is used for displaying the values.

6.4.2 The UpdateMargin

```
::   UpdateMargin WindowId MarginState State
                              -> (State, [DrawFunction]);
```

The `UpdateMargin` receives a `MarginState`. In this state the `Windowsize` and the `Thumbs` are defined.

The `Margin` can be split up in three parts:

- The input-line
- The error-line
- The index

To draw the input-line the following function is used to get the drawfunctions:

```
InputLineFuncs INT WindowId State -> ([DrawFunction],State)
```

This function first takes the selected cell out of the `ProgramState`. Then the entry is taken from this cell and this entry is displayed at the top of the window. Initially the cursor is placed at the end of the entry.

If evaluation of the entry of the selected cell produces an error, this error is shown at the bottom of the sheet.

The index is calculated by using the `ThumbValues` and the height and width (defined in the row and column). This is done by first getting the index of the `ThumbValues` and then using that index for displaying it in the `Margin` (horizontally by Capitals and vertically by integers).

Finally the index of the selected cell has to be inverted. These UpdateFunctions define a window that looks like as figure 3.

6.5 The Interactive functions

The windowdefinition defined in section 6.2 contains two functions that provide the interface, the `KeyboardFunction` and the `MouseFunction`. Both will be explained in this section.

6.5.1 The MouseFunction

The `MouseFunction` is the function that has to be evaluated when the mouse is clicked. The information about the mouse is stored in the `MouseState` which contains the position of the mouse where it was clicked, the state of the button

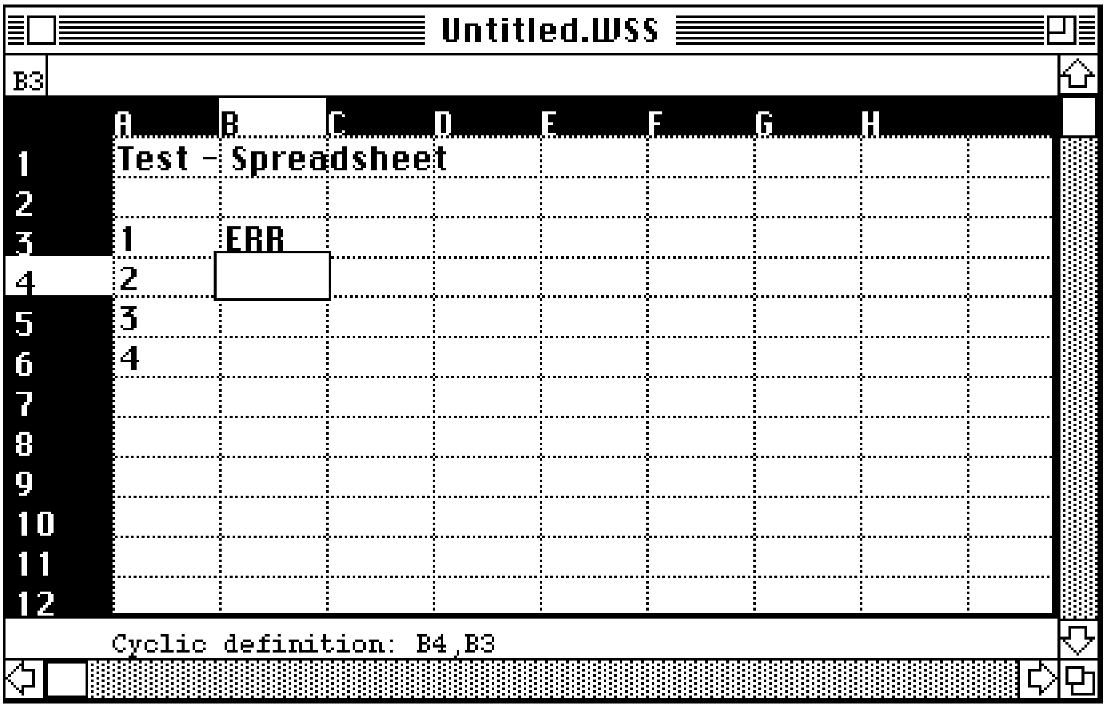


Figure 3: The spreadsheet the way it looks like

(`ButtonUp`, `ButtonDown` and `ButtonStillDown`) and the modifiers, the keys that were pressed at the same time as the mouse was clicked.

What happens if the mouse is clicked? Well if the mouse is clicked outside the window nothing happens.

If the `MouseFunc` receives a `ButtonDown` the mousefunction of the window that is active is changed to the mousefunction `GetFrame` which gets the initial position of the mouse and the old position of the mouse (initially this is the same as the initial position). Before this is done the pen is set to `Xor Mode` (when a rectangle overwrites an other rectangle the double lines disappear) and the first rectangle is drawn ((`pos, pos`)).

The `GetFrame` is defined as follows. If it receives a `ButtonUp` together with the modifier command the reference of the block is add to the entry of the cell that was selected. This results in an update of the input-line, the rectangle has to be removed (`DrawRectangle initpos oldpos`) and the penmode must be changed back.

If it receives a `ButtonUp` (no matter what modifiers are selected) the string displayed in the input-line of the `Margin` is evaluated as well as the entries of the cells that use the selected cell (this results in an update of the environment) and the index of the newly selected cell is inverted. Then the new frame (`initpos,newpos`) is stored in the state. The rectangle has to be removed from the window too. If the initpos and the new pos are situated in the same cell this cell is selected and its entry is displayed in the `Margin`. If this is not the case the block of cells that are captured by the rectangle is inverted and the input-line of the `Margin` is emptied.

6.5.2 The KeyboardFunction

The `KeyboardFunction` is the function that has to be evaluated when one of the buttons of the keyboard is pressed. The information about the keyboard is stored in the `KeyboardState` which contains the `KeyCode` (the character that was pressed), the state of the key (`ButtonUp, ButtonDown and ButtonStillDown`) and the modifiers (see `MouseFunction`). But what happens if a key is pressed?

Well if a `KeyUp` is received this action is ignored. If an arrow is pressed then first is command-modifier is checked. If it is not pressed the entry is saved and a new cell is selected according to the arrow that is pressed. If it is pressed the input-tuple from the sheet is adjusted according to the arrow (`Arrowleft` will cause an action if the cursor is not situated at the beginning of the inputline and `Arrowright` will cause an action if the cursor is not situated at the end of the line). A backspace removes the character before the cursor (if the cursor is not at the beginning of the inputline). If the character is a return the entry also is saved and if it is any other character this character is placed before the cursor. If any action has taken place the input-line in the `Margin` is updated according to this new entry.

An entry is saved as follows. First the address of the newly selected cell is evaluated. The old cell is de-selected in the `Margin` (inverted again) and the frame of the new address is stored in the `ProgramState`. Then is checked if the new frame is situated in the current window, if not the window is adjusted.

Then the old entry is removed from the sheet. The usedbys of the cells that use the changed cell are updated by removing its references. This is followed by parsing the new input string using the standard environment resulting in a function dependant on labels and columnfunctions. This function is saved in the cell together with the uses. The usedbys of the cells referred to by the uses are also updated (the address is add to it).

Then the spreadsheet is updated according to the entry. This results in a new environment where the new entry is added to the old and recursively all the columnfunctions that use this cell are adjusted. This is also a good moment to detect cycles. If one of the usedbys that is recursively reached is the cell that was assigned a new entry you know that a cycle has developed. If the cell does not reach itself you also know that no cycle has occurred because only the new entry may cause a cycle and therefore it has to be part of that cycle. When a cycle has occurred the environment can simply be update by assigning an error to the changed cell and leaving the usedbys unchanged.

6.6 The Evaluator

The implementation of this module is done exactly the way if was defined in the design (section 4.2.5).

6.7 The IO

The IO is used for saving and reading the sheets. The format of the files will be described here.

The first things stored are the row and column information. This is done in separate lines. Then the labels and the global sheet font are stored (also in a separate line), then the information stored in the matrix of cells is saved and finally the user-defined functions are stored.

This matrix will be stored in the following format. It is stored column by column marking each new column using "@@X", where X stands for the number of empty columns that follow the current column. A column is stored cell by cell. They are separated by newlines and when the current cell is followed by empty cells this is marked using "@X". Finally the different elements of the cells are separated by "&".

6.8 The interpreter

The interpreter described in one of the previous sections can be added to the spreadsheet with some minor adjustments. One of those things is adjusting the scanner. This can be done easily by adding the following extra elements to the `Lex`-items:

```
:: Lex+ ->
     Lex                  |
     LexPt STRING         |
     LexRPt STRING        |
     LexRLabel STRING     ;
```

To recognise these items the grammar (defined in Appendix C) also has to be expanded.

```
Pt ->
     And [Capital,Option Capital,Digit,Option Digit,
          Option  (And  [The  "..",Capital,Option
     Capital,  Digit,Option Digit])];

Remote ->
     And [The "\"",Option Path,FileName,The ";",
          Or [Pt,Identifier],The "\""];
```

```
FileName ->
     StarPlus Character2;


Path ->
     And [FileName,The ":",Option Path];

Character2 ->
     And [Not (The "\""),Not (The ":"),Not (The ";"),
     Or [And [The "\\", Or [The "n", The "r", The
     "f", The "b", The "t", The "\\", The "\'", The
     "\"", And [Od, Od, Od]]], And [Not (The "\\"),
     SkipChar]]];
```

At this point the label- and column-functions are recognised as identifiers. But when the references are written together as one ("A1") the interpreter recognises the total as a function. It does not split up the actual function and its parameter. Therefore we needed an extra construct (the `Pt`) to recognise these column-functions.

`Lexes` are rewritten to values, so the extra `Lex`-item has to be evaluated to a value too. This is done by evaluating the argument of the `Pt` to a `RefBlock` that then is evaluated to a list of column-functions with their arguments.

```
P2V (Pt s) -> Ref2Value (STORef s);
```

Finally the interpreter has to be connected to the spreadsheet. To achieve this the main module `Runtime` is modified. The interaction using the `StandardOutput` is not needed anymore and therefore removed. The definition module of the interpreter imports all the functions that are needed to evaluate, check, read or save entries and environments. This way the only module that has to be imported is the interpreter.

```
IMPORT MiraTypes,OperTypes,LexTypes;
FROM Miranda IMPORT Evaluate,E;
FROM Miranda2 IMPORT
           Defined_functors, Lambda_lift, Val2Env;

RULE
::   ReadFile STRING FILES [Id]
                         -> !(!FILES,[Id], Env);
::   ReadLines UNQ FILE -> (STRING,UNQ FILE);
::   MakeEnv STRING [Id] -> ([Id],Env);
::   StdEnv UNQ FILES -> (Env,[Id],UNQ FILES);
::   ValueToString Value -> STRING;
::   SLTOS [STRING] -> STRING;
::   SortIds [STRING] -> [STRING];
::   EvalCheck STRING -> (BOOL,STRING);
::   IsError Value -> (BOOL,STRING);
::   HasError [Value] -> (BOOL,STRING);
```

Chapter 7
Conclusions

After all the work that is done we made some conclusions about Clean and the interpreter.

7.1 Clean

The IO-System: This system (including the Margin) is a very clear, useful and stable system. Once you got used to it, it is very pleasant to work with. The result of using it is also very stimulating. All of its elements (windows, dialogs and menu's) can be made exactly the way you want to and this can be done in a clear, elegant way.

The use: Clean is a language that can be used very well for developing interactive applications. The way the program has to be written comes close to the way people think and this can be written down very compact which results in a small program (4000 lines of code for the spreadsheet and an additional 1200 for the interpreter). Important here is that the smaller the program, the lesser mistakes are made. Therefore the program will be more reliable and more easy to maintain. Furthermore software can be developed faster and therefore cheaper. And finally lazy evaluation also has shown its advantages which is clear when using windows; not visible parts are not evaluated, unless you force it.

The efficiency: It is hard to tell whether this program is equally efficient as a program written in a imperative language like 'C'. What we can say is that the program as defined here is reacting pleasantly to the input of the user. He does not have to wait too long to get the results back to the screen and that is the most important part. So therefore we can say that Clean is very useful for writing interactive programs.

Because of the clearness of the programs it is quit easy to use programs written by others. This is shown by the use of an interpreter that was already implemented, but did not give us difficulties when we had to adjust it to our needs.

The use of structures (that are being developed at this moment) would have made the work done more easy. It would have been more easy to implement the information structure because no access functions would have to implemented. And also the access itself would have been more easy.

7.2 Spreadsheet

The spreadsheet developed here is a very basic spreadsheet with as remarkable aspect the interpreter that is used for evaluation of the spreadsheet language (the entries). This interpreter is used for designing the spreadsheet. As told before it fitted in quite easily. But more importantly here is that it offers a very powerful tool that can evaluate more complex entries than the spreadsheets I examined before I started.

A: Defined operators.

```
:: OpsOf Language -> [[Op]];
OpsOf Miranda ->
  [  [R "->"],
     [N "|"],
     [R ";"],
     [N ".."],
     [R ","],
     [N "<-"],
     [R "where", R "="],
     [N "if"],
     [R ":", R "++", R "--"],
     [L "\\/", L "&"],
     [Pre "~"],
     [N ">", N ">=", N "~=", N "<=", N "<"],
     [L "+", L "-"],
     [Pre "-"],
     [L "*", L "/", L "div", L "mod"],
     [R "^"],
     [R ".", R "~.", R "..."],
     [Pre  "#",  Pre  "!!"],       ==  !!  denotes  the
     strictness annotation
     [N "!"],
     [L  ""]                           ==  denotes  the
     application function
  ];
OpsOf Clean ->
  [  [R ";"],
     [R "::"],
     [N "IF"],
     [L "&&", L "||"],
     [R ","],
     [N "|"],
     [L ""],
     [N ":"]
  ];
```

B: Predefined functions.

```
stdenv = "stdenv";

map f (a : as) = AP f a : map f as;
map f [] = [];

filter f (a : as)
 = a : filter f as if AP f a
 = filter f as;
filter f [] = [];

(t if True = e) = t;
(t if False = e) = e;

(t if c) = t if c = err "no value";

(f . g) x = AP f (AP g x);

f ~. g  = g . f;

(...) = (.) . (.);

(a : as) ++ b = a : as ++ b;
[] ++ b = b;

# (a : as) = 1 + # as;
# [] = 0;

x ~= y = ~ (x = y);

[a, b .. c] = a : [b, b + b - a .. c] if (a <= b & a <= c) \/ (a >= b & a >= c)
            = [];
[a .. c]   = [a, a + 1 .. c];

[a, b ..] = a : [b, b + b - a ..];
[a ..]    = [a, a + 1 ..];

(a : as) ! 0 = a;
(a : as) ! i = as ! (i - 1);

True & x = x;
False & x = False;

True \/ x = True;
False \/ x = x;
```

```
~ True = False;
~ False = True;

abs x
 = - x if x < 0
 = x;

all f (a : as) = AP f a & all f as;
all f [] = True;

and (a : as) = a & and as;
and [] = True;

concat (a : as) = a ++ concat as;
concat [] = [];

const x y = x;

converse f a b = AP f b a;

curry f a b = AP f (a, b);

drop 0 l = l;
drop n (a : as) = drop (n - 1) as;

dup f x = AP f x x;

fold f [a] = a;
fold f (a : b) = AP f a (fold f b);

foldl f c (a : b) = foldl f (AP f c a) b;
foldl f c [] = c;

foldr f c (a : b) = AP f a (foldr f c b);
foldr f c [] = c;

fst (a, b) = a;

hd (a : as) = a;

id x = x;

merg (a : b) (c : d)
 = a : merg b (c : d) if a < c
 = c : merg (a : b) d if c < a
 = c : merg b d;

merge (a : b) (c : d)
 = a : merge b (c : d) if a < c
```

```
 = c : merge (a : b) d;
merge x [] = x;
merge [] y = y;

map2 f (a : as) (b : bs) = AP f a b : map2 f as bs;
map2 f [] [] = [];

max2 a b = a if a >= b = b;

min2 a b = a if a <= b = b;

neg x = 0 - x;

or (a : as) = a \/ or as;
or [] = False;

pi = 4 * arctan 1;

rep n x = take n (repeat x);

repeat x = x : repeat x;

snd (a, b) = b;

subtract x y = y - x;

sum (a : as) = a + sum as;
sum [] = 0;

take 0 l = [];
take n (a : as) = a : take (n - 1) as;

tl (a : as) = as;

twice f x = AP f (AP f x);

uncurry f (a, b) = AP f a b;

zip2 (a : as) (b : bs) = (a, b) : zip2 as bs;
zip2 [] [] = [];
```

C: Grammar for the entries.

```
TYPE

:: LexGrammar ->
     The STRING | AnyChar STRING | Range CHAR CHAR |
     SkipAnd [LexGrammar] | Or [LexGrammar] |
     Not LexGrammar | Star LexGrammar | StarPlus
     LexGrammar | Option LexGrammar |Char;

Digit      -> Range '0' '9';

Capital    -> Range 'A' 'Z';

Integer    -> StarPlus Digit;

RestReal ->
     And [The ".", Integer,
     Option (And [The "e", Option (AnyChar "+-"),
     Integer])];

Identifier ->
     And [FirstIdentifierChar, Star IdentifierChar];

FirstIdentifierChar ->
     Or [Range 'a' 'z', Capital, AnyChar Special];

IdentifierChar ->
     Or [FirstIdentifierChar, Digit];

Space      -> StarPlus (AnyChar " \n\t");

String     ->
     And [The "\"", Star Character, The "\""];

Char       ->
     Or [The "'\"'", And [The "'", Character, The
"'"]];

Character ->
     And [Not (The "\""),
     Or [And [The "\\", Or [The "n", The "r", The
     "f", The "b", The "t", The "\\",The "\'", The
     "\"",
     And [Od, Od, Od]]], And [Not (The "\\"),
     SkipChar]]];

Od         -> Range '0' '7';
```

References

[1] Plasmeijer, van Eekelen: *Functional Programming and Parallel Graph Rewriting*, 1993.

[2] van Eekelen, Huitema, Nocker, Smetsers, Plasmeijer: *Concurrent Clean Language Manual* , version 0.8.3, February 1993.

[3] Achten, Plasmeijer: *The beauty and the beast*, Technical Report No.93-03, March 1993.

[4] *Lotus 1 2 3 Manual*, Release 2, Upgrade.

[5] Mary Campbell: *Mary Campbell's Quatro Pro Handbook*, 1990.

[6] *Microsoft Excel User's Guide*, Version 3.0, 1990,1991.