Part II

Chapter 7
Spreadsheet

The standard commercial suite of programs consists of wordprocessors to create documents, database programs to store information and spreadsheets to perform calculations. A spreadsheet consists of a matrix of cells, which contain numeric values or formulas.
The formulas are expressions which can refer to other cells. When the spreadsheet is "run" the values of the formulas are calculated and displayed in the cells.
Both spreadsheets and functional programming are about evaluation of expressions. So it seems natural to use a functional programming language to denote the formulas of a spreadsheet and to construct a spreadsheet system. And on the other hand one can view an interactive functional programming interpreter as a spreadsheet with just one cell.
In this chapter we will examine the calculation model of a spreadsheet and map its formulas on a function. Then we will use this mapping to compile a spreadsheet to a functional program. Then we will add an expression evaluator to construct an interactive spreadsheet system.

7.1 The calculation model of a spreadsheet

As an example consider the following very small spreadsheet:

```
     |        1        2        3        4
-----+----------------------------------------
a    |        4        2        8        a1+a2+a3
b    |                                    b1+b2+b3
c    |                                    c1+c2+c3
```

```
d    |                                    a4+b4+c4
```

<vervangen door snapshot van designer>

It has four rows (from a to d), four columns (from 1 to 4) and 16 cells (a1, a2, a3, a4, b1 ...).
The cell a1 contains the numeric value 4.
The cell a4 contains the formula "a1+a2+a3". This means that, when the spreadsheet is run, the value of cell a4 should be calculated as the sum of the values of cells a1, a2 and a3 (giving 14).
The cell d4 contains the formula "a4+b4+c4". Its value depends on the calculated values of cells a4, b4 and c4.
So the value of cell a4 should be calculated before the value of cell d4.

This spreadsheet can be modeled by the following function:

```
calculate = [a1,a2,a3,a4,b1,b2,b3,b4,c1,c2,c3,c4,d1,d2,d3,d4]
where
  a1 = 4
  a2 = 2
  a3 = 8
  a4 = a1+a2+a3
  b1 = 0
  b2 = 0
  b3 = 0
  b4 = b1+b2+b3
  c1 = 0
  c2 = 0
  c3 = 0
  c4 = c1+c2+c3
  d1 = 0
  d2 = 0
  d3 = 0
  d4 = a4+b4+c4
```

Here we use the convention that an empty cell contains the numeric value 0.
In this description we need not worry about the order of calculation, because lazy evaluation (graph reduction) will do this for free. The only limitation is that a value may not depend on itself. Most spreadsheet systems do allow some form of selfdependency (often called recursion) but its semantics is generally not very clear.

Spreadsheet users make a distinction between formulas and numeric values. Generally they first design the formulas and then run the spreadsheet with different numbers in the numeric cells.

A more appropriate compilation of the example would be:

```
calculate [oa1,oa2,oa3,oa4,ob1,ob2,ob3,ob4,oc1,oc2,oc3,oc4,od1,od2,od3,od4]
 = [a1,a2,a3,a4,b1,b2,b3,b4,c1,c2,c3,c4,d1,d2,d3,d4]
where
  a1 = oa1
  a2 = oa2
  a3 = oa3
```

```
  a4 = a1+a2+a3
  b1 = ob1
  b2 = ob2
  b3 = ob3
  b4 = b1+b2+b3
  c1 = oc1
  c2 = oc2
  c3 = oc3
  c4 = c1+c2+c3
  d1 = od1
  d2 = od2
  d3 = od3
  d4 = a4+b4+c4
```

Here oa1 stands for the original value of a1, the value supplied by the user, and a1 stands for the actual calculated value.

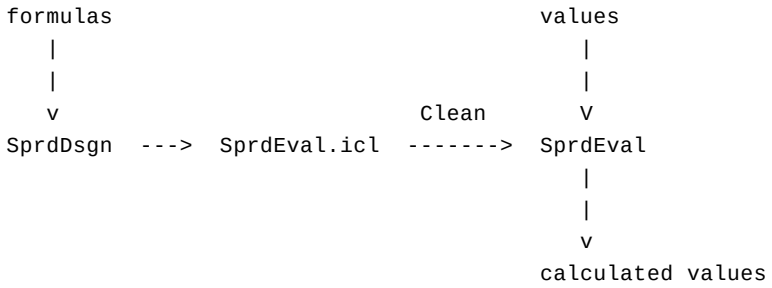
The general form of this compilation scheme is:

```
calculate [oa1,oa2,...] = [a1,a2,...]
where
  a1 = if"no formula for a1" oa1 "formula for a1"
  a2 = if"no formula for a2" oa2 "formula for a2"
 ...
```

7.2 A spreadsheet compiler

The spreadsheet compiler system consists of a program called "SprdDsgn" which displays a dialog window in which the designer of the spreadsheet can fill in formulas and which generates the text of the program called "SprdEval".
The text of SprdEval is not much more than the text of the calculate function according to the compilation scheme of the previous section. Both the compiler and the object code are Clean programs, so SprdEval should be compiled by the Clean compiler to generate the spreadsheet evaluator program, which displays a dialog window in which the spreadsheet user can fill in numeric values and which displays the calculated values.

In a picture:



By this scheme we can use all the features of the Clean programming language in the spreadsheet formulas. From a software engineering viewpoint this scheme is very useful because a casual user of a compiled spreadsheet cannot change its formulas.

An obvious drawback of this scheme is that it is not suited for interactive experimentation with formulas because the Clean compiler is too slow.

7.1.2 Compiler program

The programs SprdDsgn and SprdEval should display the same dialog window, though with different titles and different actions.
Therefore the compiler system consists of 3 modules:

The first module is SprdDsgn.icl, which only activates the designer function.

```
// spreadsheet designer

module sprddsgn

import sprdgen

Start :: *World -> *World
Start world = Designer world
```

The second module is SprdEval.icl, which is generated by the designer and which contains the calculation function.

```
// generated by sprddsgn

module sprdeval

import StdEnv, sprdgen

Start :: *World -> *World
Start world = Evaluator Calculate world

Calculate :: [Int] -> [Int]
Calculate [oa1,oa2,oa3,oa4,ob1,ob2,ob3,ob4,oc1,oc2,oc3,oc4,od1,od2,od3,od4]
 = [a1,a2,a3,a4,b1,b2,b3,b4,c1,c2,c3,c4,d1,d2,d3,d4]
where
  a1 = oa1
  a2 = oa2
  a3 = oa3
  a4 = a1+a2+a3
  b1 = ob1
  b2 = ob2
  b3 = ob3
  b4 = b1+b2+b3
  c1 = oc1
  c2 = oc2
  c3 = oc3
  c4 = c1+c2+c3
  d1 = od1
  d2 = od2
  d3 = od3
  d4 = a4+b4+c4
```

The third module is SprdGen.icl, which contains functions to construct the dialog window (parametrised with a window title, title of the button and action of the button), to run a spreadsheet, to generate a spreadsheet evaluator and some functions to perform file IO.
The constant function Ids lists the id's of the spreadsheet cells.
The constant function Names lists their names.
The function SpreadSheet does all the necessary window and dialog setup. It uses the filesystem as its only state variable.
Because of the parameters title, buttonTitle and buttonFunction it can be used both by the designer and the evaluator.

```
// spreadsheet generics

implementation module sprdgen

import StdEnv, StdFile, StdList, util,
       deltaDialog, deltaEventIO, deltaPicture


// exported functions

Designer :: *World -> *World
Designer world
  = SpreadSheet "Spreadsheet designer" "Build" Build world

Evaluator :: ([Int] -> [Int]) *World -> *World
Evaluator calcFunction world
  = SpreadSheet "Spreadsheet evaluator" "Calculate" (Calc calcFunction) world


// general framework

Ids   = [107,108,109,110,112,113,114,115,117,118,119,120,122,123,124,125]
Names =
["a1","a2","a3","a4","b1","b2","b3","b4","c1","c2","c3","c4","d1","d2","d3","d4"]

SpreadSheet title buttonTitle buttonFunction world1 = world5
where
  (events1, world2)  = OpenEvents world1
  (files1, world3)   = openfiles world2
  (formulas, files2) = ReadFileStrings "SprdDsgn.dat" files1
  (files3, events2)  = StartIO [WindowSystem [window], MenuSystem [menu]]
                               files2
                               [StartDialog]
                               events1
  world4             = closefiles files3 world3
  world5             = CloseEvents events2 world4

  window =
    FixedWindow 100 (0, 0) "Spreadsheet"
      ((0, 0), (150, 40))
      RedrawWindow
      [GoAway QuitMenu]

  RedrawWindow rects state
    = (state, [MovePenTo (40, 20), DrawString "Spreadsheet"])
```

```
  menu = PullDownMenu 11 "File" Able [
           MenuItem 12 "Quit" (Key 'Q') Able QuitMenu]

  QuitMenu files io = (files, QuitIO io)

  StartDialog state io = (state, OpenDialog dialog io)

  dialog = CommandDialog 1 title [] 2 (
    [StaticText 106 Left "a",
     StaticText 111 (Below 106) "b",
     StaticText 116 (Below 111) "c",
     StaticText 121 (Below 116) "d"] ++
    [EditText n (RightTo (n-1)) (Pixel 80) 1 formula \\
       n <- Ids &
       formula <- formulas ++ ["" \\ id <- Ids]] ++
    [StaticText 102 (Below 122) "1",
     StaticText 103 (Below 123) "2",
     StaticText 104 (Below 124) "3",
     StaticText 105 (Below 125) "4"] ++
    [DialogButton 2 (Below 103) buttonTitle Able buttonFunction,
     DialogButton 3 (Below 104) "Quit" Able QuitDialog])

  QuitDialog dialog files io = (files, QuitIO io)
```

The function Calc reads the values supplied by the user, applies the generated calculation function and displays the computed results.

```
// Calc function used by spreadsheet evaluator

Calc calcFunction dialog files io1 = (files, io2)
where
  vals = calcFunction [toInt(GetEditText n dialog) \\ n <- Ids]
  io2  = ChangeDialog 1
           [ChangeEditText n (DisplayValue val) \\
              n <- Ids &
              val <- vals]
           io1

DisplayValue 0 = ""
DisplayValue n = toString n
```

The function Build generates the calculation function, which comes down to a simple substitution of texts.

```
// Build function used by spreadsheet designer

Build dialog files1 io1 = (files3, io1)
where
  formulas = [GetEditText n dialog \\ n <- Ids]
  files2   = WriteFileString "SprdEval.icl" (GenerateEvaluator formulas) files1
  files3   = WriteFileString "SprdDsgn.dat"
               (FlatString[f +++ "\n" \\ f <- formulas])
               files2

GenerateEvaluator formulas
  = "// generated by sprddsgn\n\n" +++
    "module sprdeval\n\n" +++
```

```
    "import StdEnv, sprdgen\n\n" +++
    "Start :: *World -> *World\n" +++
    "Start world = Evaluator Calculate world\n\n" +++
    "Calculate :: [Int] -> [Int]\n" +++
    "Calculate [oa1,oa2,oa3,oa4,ob1,ob2,ob3,ob4,oc1,oc2,oc3,oc4,od1,od2,od3,od4]\n"
+++
    " = [a1,a2,a3,a4,b1,b2,b3,b4,c1,c2,c3,c4,d1,d2,d3,d4]\n" +++
    "where\n" +++
    FlatString["  " +++ s +++ " = " +++ DisplayFormula (s, f) +++ "\n" \\
                s <- Names &
                f <- formulas]

DisplayFormula (s, "") = "o" +++ s
DisplayFormula (s, fs) = fs
```

7.3 A spreadsheet interpreter

Up till now we have used the Clean compiler to transform a spreadsheet definition into a working spreadsheet. Despite the advantange of having the full Clean functionality available for the definition of cell expressions, this has the serious drawback of having to wait until the compiler is finished before being able to try out a spreadsheet. Furthermore, Clean offers much more functionality than we need for our spreadsheet expressions. Instead of using Clean we will now use the interpreter of chapter II.2 to evaluate the cell expressions.

The spreadsheet has two modes. In edit mode the user can enter the formulas in the cells. Pressing the Edit/Eval button will have the spreadsheet calculate the values of the formulas. In this mode the cells are disabled. Pressing the button again will have the spreadsheet turn back to edit mode.
For the interpreted spreadsheet we will not distinguish formulas and numeric values as we did for the spreadsheet compiler, because this simplifies things a bit.

<plaatje van spreadsheet>

The spreadsheet interpreter can reuse the function SpreadSheet from the previous section. The main difference is the state information. It should keep the filesystem, the mode of the spreadsheet and the formulas when the spreadsheet displays calculated values.

```
:: *ProgState =
   {
     files     :: Files,
     editing   :: Bool,
     formulas  :: [String]
   }
```

The function SpreadSheet should be changed so that it saves the formulas on exit.

```
SpreadSheet title buttonTitle buttonFunction world1 = world5
where
  (events1, world2)   = OpenEvents world1
  (files1, world3)    = openfiles world2
  (formulas1, files2) = ReadFileStrings "SprdDsgn.dat" files1
  ({files = files3, formulas = formulas2}, events2)
```

```
    = StartIO [WindowSystem [window], MenuSystem [menu]]
        initState
        [StartDialog]
        events1
  files4 = WriteFileString "SprdDsgn.dat"
             (FlatString[f +++ "\n" \\ f <- formulas2])
             files3
  world4 = closefiles files4 world3
  world5 = CloseEvents events2 world4

  initState = { files = files2, editing = True, formulas = formulas1 }
  ...
```

The Start rule should call the function SpreadSheet with a buttonFunction, which in Edit mode saves the formulas, calls the interpreter and displays the result.
And in Eval mode it restores the formulas.

```
Start world = SpreadSheet "Spreadsheet" "Eval/Edit" Calc world

Calc dialog state=:{editing=True} io
 = ({state & editing = False, formulas = fs, files = files2},
    ChangeDialog 1 (
      [ChangeEditText n (DisplayValue (toInt val)) \\ n <- Ids & val <- vals] ++
      [DisableDialogItems Ids])
    io)
where
  {files=files1} = state
  fs = [GetEditText n dialog \\ n <- Ids]
  (files2, vals) = Evaluate files1 (map DisplayFormula fs) Names
  DisplayValue 0 = ""
  DisplayValue n = toString n
  DisplayFormula "" = "0"
  DisplayFormula f  = f


Calc dialog state=:{editing=False, formulas} io
 = ({state & editing = True},
    ChangeDialog 1 (
      [ChangeEditText n f \\ n <- Ids & f <- formulas] ++
      [EnableDialogItems Ids])
    io)
```

The function Evaluate performs the call of the interpreter. The information the interpreter needs are the filesystem in order to read in the library file, the formulas of the spreadsheet and the names of the cells. As a result the interpreter delivers the changed filesystem and the calculated results of the formulas.
The interpreter should read in the library file, construct the first calculation function of section 7.1, and parse and evaluate it.
But sadly enough this will not work, because our interpreter can not handle where clauses.
To remedy this we could add where clauses to the interpreter (not even so difficult).
Or we could treat names of cells differently in the interpreter, thereby simulating a where clause (quite difficult).
Or we could use a different calculation model. Instead of modelling the calculation by one function we could define a function for every spreadsheet cell. As in the following example:

```
a1 = 4
a2 = 2
a3 = 8
a4 = a1+a2+a3
b1 = 0
b2 = 0
b3 = 0
b4 = b1+b2+b3
c1 = 0
c2 = 0
c3 = 0
c4 = c1+c2+c3
d1 = 0
d2 = 0
d3 = 0
d4 = a4+b4+c4
```

The serious drawback of this scheme is that there is no sharing of calculated values, so values can be computed more than once. But our spreadsheets are so small and the interpreter due to the quality of the Clean compiler is so fast, that this not noticeable.
The interpreter only needs to be extended with the function Evaluate.

```
Evaluate :: Files [String] [String] -> (Files, [String])
Evaluate files1 formulas names
 = (files2,
    [toString(parseEval(sysfuncs++recfuncs)(fromString name)) \\ name <- names])
where
  (_, files2,sysfuncs) = parsefile [] libfile files1
  recfuncs = map (fillin (sysfuncs ++ recfuncs)) funcs
  funcs = [(parsefunc o fromString)(name +++ " = " +++ f) \\
             name <- names &
             f <- formulas]
```

Exercise 7.1 (difficult)

Add where clauses to the language accepted by the interpreter and adapt the function Evaluate so that it uses the first spreadsheet calculation function.

Exercise 7.2

The current spreadsheet interpreter only reads in its library and no other script files.
Add a menu item 'open file' which using SelectInputFile asks for the name of a script file, that is given to the interpreter at every call of the function Evaluate.

Exercise 7.3

The current spreadsheet interpreter reads in its library on every call of the function Evaluate. By adding a list of function definitions to the program state this wasteful behaviour could be avoided. Implement this.