Chapter 1

Implementation

`clerk` can be used to produce a styled spreadsheet with some data and formulas on it. These formulas are evaluated when the document is loaded into a target spreadsheet system.

The library supports the following features:

- Typed cell references. Example: `Ref Double`;
- Type-safe arithmetic operations with them. Example: `(a :: Ref Double) + (b :: Ref Double)` produces a `Ref Double`;
- Constructing expressions with given types. Example: `("SUM" [a .: b]):: Expr Double` translates to `SUM(A1:B1)` (actual value depends on the values of `a` and `b`);
- Conditional styles, formatting, column widths.

Section 1.1 demonstrates the formula syntax and Section 1.2 provides an example of working with the library's data types.

1.1 Example 1. Formulas

This section demonstrates the formula syntax via several examples.

1.1.1 Imports

These are the necessary imports.

```
import Clerk
import Data.Text (Text)
import ForExamples (mkRef, showFormula)
```

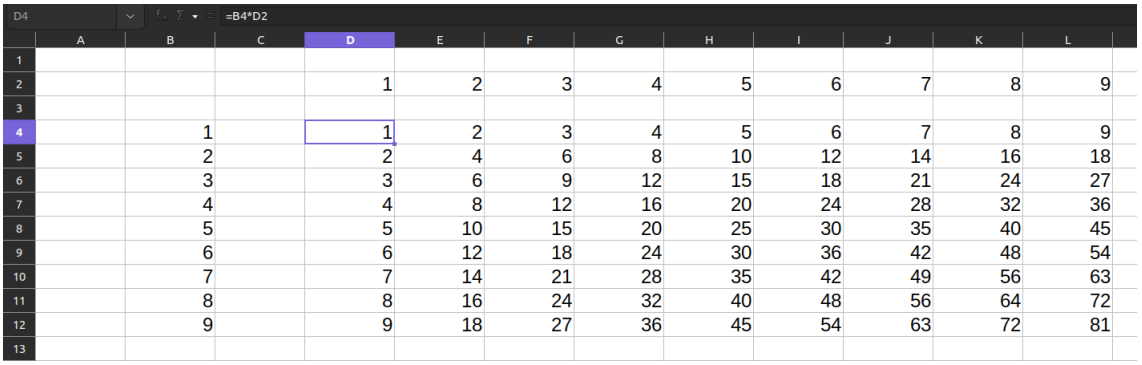
1.1.2 Sample formulas

Formulas consist of references, functions, and values. Here, I pretend that there are values with given types and that I can get references to them. I compose formulas using these references.

```
r1 :: Ref Int
r1 = mkRef 2 4

r2 :: Ref Int
r2 = mkRef 5 6

r3 :: Ref Double
r3 = mkRef 7 8

t1 :: Text
t1 = showFormula $ toFormula r2

-- >>>t1
-- "E6"

t2 :: Text
t2 = showFormula $ (r1 .* r2) .+ r1 .^ r2 ./ (ref r3)

-- >>>t2
-- "B4*E6+B4^E6/G8"
```

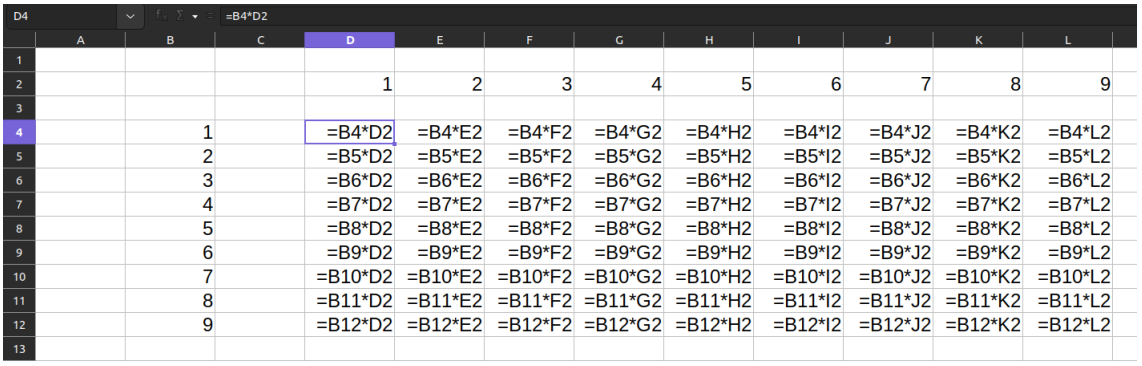
1.2 Example 2. Multiplication Table

This section shows how to describe a spreadsheet with a multiplication table. The program should produce an `xlsx` file with a multiplication table. Figure 1 demonstrates a desired multiplication table and Figure 2 shows the underlying formulas.



	A	B	C	D	E	F	G	H	I	J	K	L
1												
2				1	2	3	4	5	6	7	8	9
3												
4		1		1	2	3	4	5	6	7	8	9
5		2		2	4	6	8	10	12	14	16	18
6		3		3	6	9	12	15	18	21	24	27
7		4		4	8	12	16	20	24	28	32	36
8		5		5	10	15	20	25	30	35	40	45
9		6		6	12	18	24	30	36	42	48	54
10		7		7	14	21	28	35	42	49	56	63
11		8		8	16	24	32	40	48	56	64	72
12		9		9	18	27	36	45	54	63	72	81
13												

Fig. 1. Multiplication table



	A	B	C	D	E	F	G	H	I	J	K	L
1												
2				1	2	3	4	5	6	7	8	9
3												
4		1		=B4*D2	=B4*E2	=B4*F2	=B4*G2	=B4*H2	=B4*I2	=B4*J2	=B4*K2	=B4*L2
5		2		=B5*D2	=B5*E2	=B5*F2	=B5*G2	=B5*H2	=B5*I2	=B5*J2	=B5*K2	=B5*L2
6		3		=B6*D2	=B6*E2	=B6*F2	=B6*G2	=B6*H2	=B6*I2	=B6*J2	=B6*K2	=B6*L2
7		4		=B7*D2	=B7*E2	=B7*F2	=B7*G2	=B7*H2	=B7*I2	=B7*J2	=B7*K2	=B7*L2
8		5		=B8*D2	=B8*E2	=B8*F2	=B8*G2	=B8*H2	=B8*I2	=B8*J2	=B8*K2	=B8*L2
9		6		=B9*D2	=B9*E2	=B9*F2	=B9*G2	=B9*H2	=B9*I2	=B9*J2	=B9*K2	=B9*L2
10		7		=B10*D2	=B10*E2	=B10*F2	=B10*G2	=B10*H2	=B10*I2	=B10*J2	=B10*K2	=B10*L2
11		8		=B11*D2	=B11*E2	=B11*F2	=B11*G2	=B11*H2	=B11*I2	=B11*J2	=B11*K2	=B11*L2
12		9		=B12*D2	=B12*E2	=B12*F2	=B12*G2	=B12*H2	=B12*I2	=B12*J2	=B12*K2	=B12*L2
13												

Fig. 2. Multiplication table with formulas

The below sections describe how such a spreadsheet can be constructed.

1.2.1 Imports

These are the necessary imports.

```
import Clerk
import Control.Monad (forM, forM_, void)
import qualified Data.Text as T
import Lens.Micro ((&), (+~), (^.))
```

1.2.2 Tables

The tables that I construct are:

- A vertical header;
- A horizontal header;
- A table with results of multiplication of the numbers from these headers.

1.2.2.1 A vertical header

`clerk` provides the `RowI` monad. This monad takes some `i`nput, internally converts it into spreadsheet types, and outputs something, e.g., a cell reference. In background, it writes a template of a horizontal block of cells - a `row`. This row is used for placing the input values onto a sheet.

A vertical block of cells (Figure 3) can be represented as several horizontal blocks of cells placed under each other.

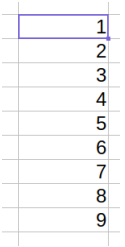


Fig. 3. A vertical header

As a template, I use a `RowI` with one integer as an input. Because I do not need any formatting, I use `blank` cells for templates. I place the rows for each input value and collect the references. Each row is shifted relative to the input coordinates.

```
mkVertical :: Coords -> [Int] -> Sheet [Ref Int]
mkVertical coords numbers =
  forM (zip [0 ..] numbers) $ \(idx, number) ->
    place1
      (coords & row +~ idx + 2)
      number
      ((columnRef blank (const number)) :: RowI Int (Ref Int))
```

1.2.2.2 A horizontal header

For a horizontal header (Figure 4), I make a row of numbers and collect the references to all its cells.

Fig. 4. A horizontal header

As the type of inputs is not important, I use the `Row` type. In the `Sheet` monad, I place this row starting at a specified coordinate.

```
mkHorizontal :: Coords -> [Int] -> Sheet [Ref Int]
mkHorizontal coords numbers =
  place
    (coords & col +~ 2)
    ((forM numbers $ \n -> columnRef blank (const n)) :: Row [Ref Int])
```

1.2.2.3 Table builder

For inner cells, I use single-cell rows for each input. I place the cells as in Figure 5.

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Fig. 5. A vertical header

Since the information about these cells is unnecessary, I use the `Row ()` type.

```
mkTable :: [(Ref Int, Ref Int)] -> Sheet ()
mkTable cs =
  forM_ cs $ \(r, c) -> do
    coords <- mkCoords (c ^. col) (r ^. row)
    place coords ((column blank (const (r .* c))) :: Row ())
```

1.2.3 Sheet

Here, I combine all functions to compose a complete `Sheet ()`.

```
sheet :: Sheet ()
sheet = do
  start <- mkCoords 2 2
  let numbers = [1 .. 9]
  cs <- mkHorizontal start numbers
  rs <- mkVertical start numbers
  mkTable [(r, c) | r <- rs, c <- cs]
```

1.2.4 Result

Finally, I write the result and get a spreadsheet like the one at the beginning of this example.

```
main :: IO ()
main = writeXlsx "example2.xlsx" [(T.pack "List 1", void sheet)]
```