

# MBase demo: Spreadsheet DSL

Meta Alternative Ltd.

Nov. 2007

## Spreadsheet mini-language

S-expressions syntax is quite limiting, but it is still possible to use it for relatively complicated visual constructs. In this demo we will show how to implement a 2-dimensional language, an embeddable spreadsheet.

A source language is a flat list containing grid guides: symbolic column names and numerical row headers.

An implementation outline:

- Make a table (using grid guides)
- Parse cell values
- Build a dependency graph
- Build a sorted dependency list for a given return cell
- Compile a return cell and all its dependencies.

*First we have to build a table out of a flat list. MBase parser ignores whitespace characters, including EOLs, so we need guiding symbols to form a grid. The first row of a grid is a list of column titles, and each subsequent row must start with its number.*

```
(function ss-mktable ( src )
  (let* ((cls (partition symbol? src))
        (columns (car cls)))
```

*l is our flat list, c is a current collector, n is a current row number and cl is a list of remaining column titles to assign in the current row.*

```
    (let loop ((l (cdr cls)) (c nil) (n nil) (cl columns))
      (cond
        ((null? l) c)
        ((number? (car l))
         (loop (cdr l) c (car l) columns))
        (else
         (if (null? cl) (ccerror '(COLUMNS! ,@1)))
         (loop (cdr l)
               (cons '((, (car cl) ,n) ,(car l)) c)
               n (cdr cl))))
    )))
```

*Now, when all the cells are properly bound to the grid, we can parse their values, if necessary. Using an existing mini-language: Infix (see `arith` function).*

```
(function ss-parse-cells ( tbl )
  (map-over tbl
    (fmt ((col row) cl)
      '(, (Sm<< col row)
        ,(cond
          ((string? cl) (arith cl))
          (else cl))))))
```

*A little regular expression to distinguish normal variables from cell references.*

```
(define is-cell-r (<r> (p.ualpha +*)  
                      (p.digit +*)))
```

*Check if a given symbol is a cell reference.*

```
(function is-cell? (sym)  
  (matches? is-cell-r (symbol->string sym)))
```

*This function builds a list of immediate dependencies for a given cell value. E.g., “A1+A2” will give a list (A1 A2).*

```
(function ss-cell-depends (cell)  
  (collector (depends get)  
    (let loop ((l cell))  
      (p:match l  
        ($M (if (is-cell? l) (depends l)))  
        ((quote . $_) nil)  
        (($a . $b) (loop a) (iter loop b))  
        (else nil)))  
    (unifiq (get))  
  ))
```

*The following function converts a given parsed table in two hashtables: a table of immediate dependencies and a table of all the cells in a grid.*

```
(function ss-depends (tbl)  
  (let ((dh (mhash)) (vh (mhash)))  
    (iter-over tbl  
      (fmt (cn cl)  
          (hashput vh cn cl)  
          (hashput dh cn (ss-cell-depends cl))))  
    (cons dh vh)))
```

*Now, when we have a dependency graph and a top node, we can build an ordered list of cells to be evaluated. Cycles in a dependency graph are detected [right here](#).*

```

(function ss-depsort (dh top)
  (let ((nh (mhash)))
    ; Initialise a full-dependency table.
    (hashiter (fun (a b) (hashput nh a b)) dh)
    ; Fill this table properly, following all possible dependency paths.
    (let loop ((cell top) (ctops (list top)))
      (let* ((cd (hashget dh cell)))
        ; Update all the nodes in path
        (foreach (t ctops)
          (if (memq t cd) (ccerror '(CYCLE! ,t)))
          (hashput nh t (unifiq (append cd (hashget nh t))))))
        ; Go into all possible paths from this node
        (let ((npath (cons cell ctops)))
          (foreach (c cd)
            (loop c npath))))))
    ; Sort a dependency list for a given top cell in a proper order.
    (qsort (fun (c1 c2)
            (memq c1 (hashget nh c2)))
          (hashget nh top))))

```

Now, to bind everything together into a (let\* ...) construction:

```

(function ss-compile (src top)
  (let* ((tbl (ss-parse-cells (ss-mktable src)))
        (hshs (ss-depends tbl))
        (dh (car hshs))
        (vh (cdr hshs))
        (ds (ss-depsort dh top)))
    '(let* ,(foreach-map (d ds)
                        '(,d ,(hashget vh d)))
      ,(hashget vh top))))

```

And, as for any other MBase-targeting language, here is a macro wrapper:

```

(macro ssheet (name args . body)
  (let* ((ax (partition (fun (x) (not (eqv? x '->))) args))
        (pars (car ax))
        (top (caddr ax)))
    '(recfunction ,name ,pars ,(ss-compile body top))))

```

Now we can immediately use our new language extension. Here follows some examples:

A simple spreadsheet function with one parameter:

```

(ssheet stest ( x -> A20)
  A      B      C      D
  1  "1"    "x"    "A1*B1"  "C1/2"
  20 "A1+B20" "D1*D1"
)

```

*A spreadsheet-function can call another spreadsheet:*

```
(ssheet abc ( a b -> A1 )
  A          B          C
  1 "cons(B1,C1)" "a+b"      "A2*B2"
  2 "a*b"        "stest(B1+A2)"
)
```

*Spreadsheet-functions can be recursive (but remember: it is not a lazy language, all relevant cells are always evaluated).*

```
(ssheet fact ( n -> A1 )
  A          B
  1 "if (B2) then lazyref(B1) else 1" "lazy(n*fact(A2))"
  2 "n-1"        "n>1"
)
```