EVENT: Start with the library "mlp" using the compiled version.

```
; counter.bm: a simple clock counter, x is the clock.
;     similar to acc when we take our output from the combinational elt.
;
; RESULTS:
; if we look at the combinational output (Ycomb)
;   with initial value of reg: 0 , OK can prove Ycomb = len . [log: counter]
;   with initial value of reg: 1 , fails (expectedly) in COUNT-CORRECT-AX
;        in case p x = e, [log: counter2]
; if we look at Yreg (i.e. register output) then:
;   with initial value of reg: 0 , fails (expectedly) in COUNT-CORRECT-AX
;        in case p x = e, [log: counter4]
```

```
;   with initial value of reg: 1 , OK can prove Yreg = len . [log: counter3]
; ANALYSIS:
;   The issue is a confusion of the intent of the SPEC: if we are counting the
; number of "pulses" then indeed we can look at the Reg-output, and initialize
; it with zero, because #pulse = len(clk) - 1 = len(any input string) - 1 .
; In other words, the issue is with NUMER-COUNT x =LEN x.  If we want to count
; the number of pulses, then we should have number-count x = len x -1, and
; then initialize w/ 0 and look at Reg-out. Check: OK!
; CONCLUSION:
;   The model (essentially Mealy) is fine even for looking at Reg-outs, but
; be careful of translating specs involving number of "clock ticks".  My tics
; are really "periods", and the engineer's tick are "pulses" of which there are
; always one less when based on an operational semantics looking at things
; "at the end of clock periods".
;
; COMPARISON w/ PAILLET, and PAILLET inferred spec:

;;; DEFINITION OF CIRCUIT:
#|
(setq sysd '(sy-count (x)
(Ycomb S Inc Yreg)
(Yreg R 0 Ycomb)
))

(setq counter '( |#
; BM DEFINITIONS and A2 LEMMAS, generated by BMSYSD:
; comb_inc.bm: INCrement combinational element
; U7-DONE
```

DEFINITION: $\mathrm{inc}\,(u) = (1 + u)$

```
; Everything below generated by: (bmcomb 'inc '() '(x))
```

DEFINITION:
s-inc$\,(x)$
= **if** empty$\,(x)$ **then** E
else a (s-inc (p$\,(x)$), inc (l$\,(x)$)) **endif**

```
;; A2-Begin-S-INC
```

THEOREM: a2-empty-s-inc
empty (s-inc$\,(x)$) = empty$\,(x)$

THEOREM: a2-e-s-inc
$(\text{s-inc}\,(x) = \text{E}) = \text{empty}\,(x)$

THEOREM: a2-lp-s-inc
$\text{len}\,(\text{s-inc}\,(x)) = \text{len}\,(x)$

THEOREM: a2-lpe-s-inc
$\text{eqlen}\,(\text{s-inc}\,(x),\ x)$

THEOREM: a2-ic-s-inc
$\text{s-inc}\,(\text{i}\,(c\_x,\ x)) = \text{i}\,(\text{inc}\,(c\_x),\ \text{s-inc}\,(x))$

THEOREM: a2-lc-s-inc
$(\neg\ \text{empty}\,(x)) \rightarrow (\text{l}\,(\text{s-inc}\,(x)) = \text{inc}\,(\text{l}\,(x)))$

THEOREM: a2-pc-s-inc
$\text{p}\,(\text{s-inc}\,(x)) = \text{s-inc}\,(\text{p}\,(x))$

THEOREM: a2-hc-s-inc
$(\neg\ \text{empty}\,(x)) \rightarrow (\text{h}\,(\text{s-inc}\,(x)) = \text{inc}\,(\text{h}\,(x)))$

THEOREM: a2-bc-s-inc
$\text{b}\,(\text{s-inc}\,(x)) = \text{s-inc}\,(\text{b}\,(x))$

THEOREM: a2-bnc-s-inc
$\text{bn}\,(n,\ \text{s-inc}\,(x)) = \text{s-inc}\,(\text{bn}\,(n,\ x))$

```
;; A2-End-S-INC
```

```
; eof:comb_inc.bm
```

DEFINITION:
topor-sy-count (ln)
= **if** ln = `'ycomb` **then** `1`
elseif ln = `'yreg` **then** `0`
else `0` **endif**

DEFINITION:
sy-count (ln, x)
= **if** ln = `'ycomb` **then** s-inc (sy-count (`'yreg`, x))
elseif ln = `'yreg`
then if empty (x) **then** E
else i (`0`, sy-count (`'ycomb`, p (x))) **endif**
else sfix (x) **endif**

```
;; A2-Begin-SY-COUNT
```

THEOREM: a2-empty-sy-count
empty (sy-count (ln, x)) = empty (x)

THEOREM: a2-e-sy-count
(sy-count (ln, x) = E) = empty (x)

THEOREM: a2-lp-sy-count
len (sy-count (ln, x)) = len (x)

THEOREM: a2-lpe-sy-count
eqlen (sy-count (ln, x), x)

THEOREM: a2-pc-sy-count
p (sy-count (ln, x)) = sy-count (ln, p (x))

```
;; A2-End-SY-COUNT

;;; SPEC definition:
; FIRST (misunderstood) spec:
; note: written when looking at the combinational output..
;
;(defn numer-count (x)
;  (if (empty x)
;      0
;      (inc (numer-count (p x)) )))
;
; intent verification: should have numer-count = len; prove but don't use.
;(prove-lemma numer-count-len ()
;(equal (numer-count x) (len x))
;)
```

DEFINITION:
numer-count (x)
= **if** empty (x) **then** 0
else len (x) $-$ 1 **endif**

```
;a cheap way: (len (p x))

; this is the standard extension from last-char-fun to MLP-string-fun,
; see theta.bm .
```

DEFINITION:
spec-count (x)
$=$ **if** empty (x) **then** E
else a (spec-count (p (x)), numer-count (x)) **endif**

```
; Paillet's spec do not define an additional function, just a relation
; that must be verified by the circuit output. See below.

;;; Circuit CORRECTNESS:
; THIS is where it matters which line we take as output!

; Count-correct-ax is a "predicative correctness statement", i.e. what we would
; do if we didn't have functional equality as a specification method, but
; instead used a purely axiomatic approach.
```

THEOREM: count-correct-ax
$(\neg$ empty $(x)) \rightarrow ($l (sy-count (`'yreg`, x)) $=$ numer-count $(x))$

```
; to go to a functional equality once we have the "last" (ax) statement is
; a trivial induction, if we start out with an P-L split which is unnatural
; for BM, so we force it w/ a USE hint of A-p-l-split
```

THEOREM: a-p-l-split
$(\neg$ empty $(x))$
$\rightarrow$ (sy-count (`'yreg`, x)
$=$ a (p (sy-count (`'yreg`, x)), l (sy-count (`'yreg`, x))))

THEOREM: count-correct
sy-count (`'yreg`, x) $=$ spec-count (x)

```
; PAILLET CORRECTNESS:
```

EVENT: Disable count-correct.

EVENT: Disable count-correct-ax.

```
; count-paillet-correct obtained trivially (without recursion) as expected.
```

THEOREM: count-paillet-correct
$(\neg$ empty $(x))$
$\rightarrow$ (sy-count (`'yreg`, x) $=$ i (0, s-inc (p (sy-count (`'yreg`, x)))))

```
; my (more intuitive) version of "simple" (no recursion req'd) correctness:
```

THEOREM: count-correct-simple

$$((\neg\ \text{empty}\,(x)) \wedge (\neg\ \text{empty}\,(\text{p}\,(x))))$$
$$\rightarrow \quad (\text{l}\,(\text{sy-count}\,(\texttt{'yreg},\ x)) = \text{inc}\,(\text{l}\,(\text{p}\,(\text{sy-count}\,(\texttt{'yreg},\ x)))))$$

```
; eof: counter.bm
;))
```

Index