EVENT: Start with the initial **nqthm** theory.

```
;                           A NOTE ON SHELLS
;                                  by
;                              John Cowles
;                    Department of Computer Science
;                        University of Wyoming

; The following is intended to give the reader some insight into SHELLS.

; Intuitively a nonempty SEQUENCE is an ordered list, possibly with
;  duplicates, of objects, ( Obj1 Obj2 ... ObjN ).

; There are two ways to recursively decompose sequences.
;
;  1. A SEQUENCE is either the EMPTY-SEQUENCE or a pair < Obj,Seq >.
;
;  2. A SEQUENCE is either the EMPTY-SEQUENCE or a pair [ Seq,Obj ].
```

```
;
;  Here Obj is an object, Seq is a sequence, and EMPTY-SEQUENCE is
;  the unique sequence which contains no objects.  Different pairing
;  brackets, < > and [ ], are used to enphasize which of the
;  decompositions is being used.

; Here the Shell Principle is used with decomposition 1 above to
;  add sequences as a "new" data type.
```

EVENT: Add the shell *cons-seq-first*, with bottom object function symbol *empty-seq*, with recognizer function symbol *seq-p*, and 2 accessors: *first*, with type restriction (none-of) and default value empty-seq; *final*, with type restriction (one-of seq-p) and default value empty-seq.

```
; default value

; ( CONS-SEQ-FIRST Obj Seq )     returns < Obj,Seq >.
; ( CONS-SEQ-FIRST Obj Non-Seq ) returns < Obj,EMPTY-SEQUENCE >.

; ( EMPTY-SEQ ) returns the EMPTY-SEQUENCE.

; ( SEQ-P Seq )     returns T.
; ( SEQ-P Non-Seq ) reTurns F.

; ( FIRST < Obj,Seq > ) returns Obj.
; ( FIRST (EMPTY-SEQ) ) returns (EMPTY-SEQ).
; ( FIRST Non-Seq )     returns (EMPTY-SEQ).

; ( FINAL < Obj,Seq > ) returns Seq.
; ( FINAL (EMPTY-SEQ) ) returns (EMPTY-SEQ).
; ( FINAL Non-Seq )     returns (EMPTY-SEQ).
; =================
; The next two functions "coerce" non-sequences into
;  behaving like the EMPTY-SEQUENCE.
```

DEFINITION:
empty-seq-p $(s) = ((s = \text{EMPTY-SEQ}) \lor (\neg\ \text{seq-p}\,(s)))$

DEFINITION:
coerce-seq (s)
$=$ **if** seq-p (s) **then** s
else EMPTY-SEQ **endif**

```
; The next three functions implement sequence decomposion 2 above.

; ( CONS-SEQ-LAST Seq Obj )     returns [ Seq,Obj ].
; ( CONS-SEQ-LAST Non-Seq Obj ) returns [ (EMPTY-SEQ),Obj ].
;     Here [ (EMPTY-SEQ),Obj ] is identified with < Obj,(EMPTY-SEQ) >.

; ( INITIAL [ Seq,Obj ] ) returns Seq.
; ( INITIAL (EMPTY-SEQ) ) returns (EMPTY-SEQ).
; ( INITIAL Non-Seq )     returns (EMPTY-SEQ).

; ( LAST [ Seq,Obj ] ) returns Obj.
; ( LAST (EMPTY-SEQ) ) returns (EMPTY-SEQ).
; ( LAST Non-Seq )     returns (EMPTY-SEQ).
```

DEFINITION:
cons-seq-last (s, c)
= **if** empty-seq-p (s) **then** cons-seq-first (c, s)
else cons-seq-first (first (s), cons-seq-last (final (s), c)) **endif**

DEFINITION:
initial (s)
= **if** empty-seq-p (s) **then** EMPTY-SEQ
elseif final (s) = EMPTY-SEQ **then** EMPTY-SEQ
else cons-seq-first (first (s), initial (final (s))) **endif**

DEFINITION:
last (s)
= **if** empty-seq-p (s) **then** EMPTY-SEQ
elseif final (s) = EMPTY-SEQ **then** first (s)
else last (final (s)) **endif**

```
; The next 12 rewrite rules and 1 elimination rule would have been
;  explicitly added as axioms to the data base by the shell principle
;  if sequence decomposition 2 had been used, in place of decomposition 1,
;  as the basis for the shell which added sequences as a new type.
```

THEOREM: initial-cons-seq-last
initial (cons-seq-last (s, c))
= **if** seq-p (s) **then** s
else EMPTY-SEQ **endif**

THEOREM: initial-nseq-p
($\neg$ seq-p (s)) $\rightarrow$ (initial (s) = EMPTY-SEQ)

THEOREM: initial-type-restriction
$(\neg \text{ seq-p}(s)) \rightarrow (\text{cons-seq-last}(s,\ c) = \text{cons-seq-last}(\text{EMPTY-SEQ},\ c))$

THEOREM: initial-lessp
$(\text{seq-p}(s) \land (s \neq \text{EMPTY-SEQ})) \rightarrow (\text{count}(\text{initial}(s)) < \text{count}(s))$

THEOREM: initial-lesseqp
$\text{count}(s) \not< \text{count}(\text{initial}(s))$

THEOREM: last-cons-seq-last
$\text{last}(\text{cons-seq-last}(s,\ c)) = c$

THEOREM: last-nseq-p
$(\neg \text{ seq-p}(s)) \rightarrow (\text{last}(s) = \text{EMPTY-SEQ})$

THEOREM: last-lessp
$(\text{seq-p}(s) \land (s \neq \text{EMPTY-SEQ})) \rightarrow (\text{count}(\text{last}(s)) < \text{count}(s))$

THEOREM: last-lesseqp
$\text{count}(s) \not< \text{count}(\text{last}(s))$

```
; The next two lemmas are obvious facts used only as
; hints for the proof of the lemma CONS-SEQ-LAST-EQUAL.
```

THEOREM: initial-apply-equals
$(x = y) \rightarrow (\text{initial}(x) = \text{initial}(y))$

THEOREM: last-apply-equals
$(x = y) \rightarrow (\text{last}(x) = \text{last}(y))$

THEOREM: cons-seq-last-equal
$(\text{cons-seq-last}(s1,\ c1) = \text{cons-seq-last}(s2,\ c2))$
$= \quad (\textbf{if } \text{seq-p}(s1)$
$\quad\quad \textbf{then if } \text{seq-p}(s2) \ \textbf{then } s1 = s2$
$\quad\quad\quad\quad \textbf{else } s1 = \text{EMPTY-SEQ } \textbf{endif}$
$\quad\quad \textbf{elseif } \text{seq-p}(s2) \ \textbf{then } \text{EMPTY-SEQ} = s2$
$\quad\quad \textbf{else t endif}$
$\quad\quad \land \quad (c1 = c2))$

THEOREM: cons-seq-last-initial-last
$\text{cons-seq-last}(\text{initial}(s), \text{last}(s))$
$= \quad \textbf{if } \text{seq-p}(s) \land (s \neq \text{EMPTY-SEQ}) \ \textbf{then } s$
$\quad\quad \textbf{else } \text{cons-seq-last}(\text{EMPTY-SEQ}, \text{EMPTY-SEQ}) \ \textbf{endif}$

THEOREM: initial-last-elim
$(\text{seq-p}(s) \land (s \neq \text{EMPTY-SEQ}))$
$\rightarrow \quad (\text{cons-seq-last}(\text{initial}(s), \text{last}(s)) = s)$

THEOREM: count-cons-seq-last
count (cons-seq-last (s, c))
= (1 + (**if** seq-p (s) **then** count (s)
else 0 **endif**
+ count (c)))

```
; The next rewrite rule would, in effect, have been implicitly added
;  as an axiom to the data base by the shell principle if sequence
;  decomposition 2 had been used, in place of decomposition 1, as the
;  basis for the shell which added sequences as a new type.
```

THEOREM: cons-seq-last-not-empty-seq
cons-seq-last (s, c) $\neq$ EMPTY-SEQ

```
;=====================================================================

; The next two functions give different versions of REVERSE.
```

DEFINITION:
reverse1 (s)
= **if** empty-seq-p (s) **then** s
else cons-seq-last (reverse1 (final (s)), first (s)) **endif**

DEFINITION:
reverse2 (s)
= **if** empty-seq-p (s) **then** s
else cons-seq-first (last (s), reverse2 (initial (s))) **endif**

```
; Use the Theorem Prover to verify the following proposed theorems.
;
; 1. For i=1 and i=2, ( EQUAL (REVERSEi (REVERSEi S)) S )
;
; 2. ( EQUAL (REVERSE1 S) (REVERSE2 S) )
; ==================================================================

; The next four functions give different versions of CONCATENATION.
```

DEFINITION:
concat1 ($s1$, $s2$)
= **if** empty-seq-p ($s1$) **then** coerce-seq ($s2$)
else cons-seq-first (first ($s1$), concat1 (final ($s1$), $s2$)) **endif**

DEFINITION:
concat2 (*s1*, *s2*)
= **if** empty-seq-p (*s2*) **then** coerce-seq (*s1*)
else concat2 (cons-seq-last (*s1*, first (*s2*)), final (*s2*)) **endif**

DEFINITION:
concat3 (*s1*, *s2*)
= **if** empty-seq-p (*s2*) **then** coerce-seq (*s1*)
else cons-seq-last (concat3 (*s1*, initial (*s2*)), last (*s2*)) **endif**

DEFINITION:
concat4 (*s1*, *s2*)
= **if** empty-seq-p (*s1*) **then** coerce-seq (*s2*)
else concat4 (initial (*s1*), cons-seq-first (last (*s1*), *s2*)) **endif**

```
; Use the Theorem Prover to verify the following proposed theorems.
;
; 1. For i=1, i=2, i=3, and i=4, CONCATi is associative.
;
; 2. For i=1 and i=2; and for j=1, j=2, j=3, and j=4;
;      ( EQUAL (REVERSEi (CONCATj S1 S2))
;              (CONCATj (REVERSEi S2) (REVERSEi S1)) )
;
; 3. For i and j such that 1 <= i < j <= 4,
;      ( EQUAL (CONCATi S1 S2) (CONCATj S1 S2) )
```

Index