This copyright applies to all the scripts presented here.

Appendix A

NFSA ≡ DFSA Appendices

A.1 Without ϵ-transitions

This is the proof script for the constructive proof. Start NQTHM and submit the following script.

```
;; Proof of the equivalence of nondeterministic and deterministic automaton
;;
;; Debora Weber-Wulff, Berlin, February 1997

(boot-strap nqthm)

;; ------------------------------------------------------------------------
;;      Library functions
;; ------------------------------------------------------------------------
;; These functions were copied from the basis.events, bags-and-sets.events,
;; and lists.events for nqthm-1992 that I obtained in Austin 1993.
;; Only these were needed for the proof, and it speeds up the proof
;; considerably to keep unnecessary stuff out - they do not need to be
;; considered at every step.

(defn setp (l)
  (if (not (listp l))
      t
      (and (not (member (car l) (cdr l)))
           (setp (cdr l)))))

(prove-lemma setp-cons (rewrite)
  (equal (setp (cons x l))
         (and (not (member x l))
              (setp l)))
  ((enable setp)))

(defn consl (x l)
   (if (listp l)
       (cons (cons x (car l))
             (consl x (cdr l)))
       nil))

(prove-lemma member-consl (rewrite)
  (equal (member x (consl v lst))
         (and (listp x)
              (equal (car x) v)
              (member (cdr x) lst))))

(prove-lemma member-non-list (rewrite)
  (implies (not (listp l))
           (not (member x l))))

(prove-lemma member-union (rewrite)
  (equal (member x (union a b))
```

```
         (or (member x a)
             (member x b))))

(defn length (l)
  (if (not (listp l))
      0
      (add1 (length (cdr l)))))

(prove-lemma length-cons (rewrite)
  (equal (length (cons a x))
         (add1 (length x)))
  ((enable length)))

(defn plistp (l)
  (if (not (listp l))
      (equal l nil)
      (plistp (cdr l))))

(prove-lemma plistp-cons (rewrite)
  (equal (plistp (cons a l))
         (plistp l))
  ((enable plistp)))

(defn all-subbags (l)
   (if (listp l)
       (let ((x (all-subbags (cdr l))))
         (union x (consl (car l) x)))
       (list nil)))

;; ---------------------------------------------------------------------------
;;      Basic functions
;; ---------------------------------------------------------------------------

;; some-member
;; This is a witness for existence of an element in the intersection

(defn some-member (l1 l2)
 (if (nlistp l1)
     F
     (if (member (car l1) l2)
         T
         (some-member (cdr l1) l2))))

(prove-lemma some-member-member (rewrite)
  (implies (and (member n y)
                (member n x))
           (some-member x y)))

;; ---------------------------------------------------------------------------
;;      subsetp
;; ---------------------------------------------------------------------------
;; I will use a weak definition of subset: all elements of a are in b
;; (some may be double).

(defn subsetp (a b)
  (if (nlistp a)
      T
      (and (member (car a) b)
           (subsetp (cdr a) b))))

(prove-lemma subsetp-union (rewrite)
 (equal (subsetp (union a b) c)
        (and (subsetp a c)
             (subsetp b c))))

(prove-lemma subsetp-union2 (rewrite)
```

```
 (subsetp y (union y x)))

(prove-lemma subsetp-cons (rewrite)
 (implies (subsetp x y)
          (subsetp x (cons z y))))

(prove-lemma subsetp-reflexive (rewrite)
 (subsetp x x))

(prove-lemma member-subsetp (Rewrite)
 (implies (and (member x a)
               (subsetp a b))
          (member x b)))

(prove-lemma some-member-and-subsetp (rewrite)
 (implies (and (some-member a b)
               (subsetp b c))
          (some-member a c)))

;; ---------------------------------------------------------------------------
;; --- theorems about setp -----

;; consl preserves setp
(prove-lemma setp-consl (Rewrite)
 (implies (setp x)
          (setp (consl y x))))

(prove-lemma setp-union (rewrite)
 (implies (and (setp x)
               (setp y))
          (setp (union x y))))

;; ---------------------------------------------------------------------------
;; --- theorems about consl -----

(prove-lemma setp-union-consl (REWRITE)
   (IMPLIES (SETP Y)
            (SETP (UNION Y (CONSL V Y)))))

;; --- theorems about all-subbags -----

(prove-lemma listp-all-subbags (rewrite)
 (listp (all-subbags d)))

;; nil is a member of all power sets.

(prove-lemma nil-member-all-subbags (rewrite)
 (member nil (all-subbags x)))

;; Singleton lists of members of a list are members of the power set.

(prove-lemma member-list-all-subbags (rewrite)
  (implies (member x y)
           (member (list x) (all-subbags y))))

;; If x is a set, the 'power-set' is a set

(prove-lemma setp-all-subbags (rewrite)
 (implies (setp x)
          (setp (all-subbags x))))

(defn all-subbags-hint1 (x y)
  (if (nlistp y)
      T
      (and (all-subbags-hint1 (cdr x) (cdr y))
           (all-subbags-hint1 x (cdr y)))))
```

```
(prove-lemma member-all-subbags (rewrite)
 (implies (member x (all-subbags y))
          (subsetp x y))
 ((induct (all-subbags-hint1 x y))))

;; --- Theorems about union

(prove-lemma nothing-member-nil (rewrite)
             (not (member x nil)))

;; ---------------------------------------------------------------------------
;;      order
;; ---------------------------------------------------------------------------
;; To simulate set equality we order the list representation of
;; a set (x) along a basis (lst).

;; Order the elements of x according to lst, a finite total order for x.

(defn order (x lst)
  (if (nlistp lst)
      nil
      (if (member (car lst) x)
          (cons (car lst) (order x (cdr lst)))
          (order x (cdr lst)))))

(prove-lemma order-preserves-member (rewrite)
 (implies (member x (order y z))
          (and (member x y)
               (member x z))))

(prove-lemma member-both-member-order (rewrite)
 (implies (and (member z x)
               (member z v))
          (member z (order x v))))

; ---- order and all-subbags

(prove-lemma helper1 (rewrite)
 (implies (equal (order x z) y)
          (member y (all-subbags z))))

(prove-lemma ordered-member-all-subbags (rewrite)
 (implies (equal (order x z) x)
          (member x (all-subbags z))))

(disable helper1)
(disable ordered-member-all-subbags)

(prove-lemma member-order-all-subbags (rewrite)
 (member (order w d) (all-subbags d)))

(prove-lemma some-member-order1 (rewrite)
 (implies (some-member (order a b) c)
          (some-member (order (cons x a) b) c))
 ((disable member-subsetp)))

(prove-lemma subsetp-order-2 (rewrite)
 (subsetp (order a b) (order (cons c a) b)))

(prove-lemma some-member-order (rewrite)
 (implies (some-member (order w d) z)
          (some-member w z)))

(prove-lemma some-member-order-2 (rewrite)
 (implies (and (subsetp b c)
```

```
              (some-member a b))
         (some-member (order a c) b))
 ((induct (length a))))

;; ---------------------------------------------------------------------------
;;      all-member
;; ---------------------------------------------------------------------------

;; This is a key definition for the proof - it turns out to be the same
;; as my definition of subsetp.

(defn all-member
      (a b)
      (if (nlistp a)
          t
          (and (member (car a) b)
               (all-member (cdr a) b))))

;; ---------------------------------------------------------------------------
;;       Definedp
;; ---------------------------------------------------------------------------
;; needed to explain the workings of next-state

(defn definedp (x table)
  (if (nlistp table)
      F
      (or (equal x (caar table))
          (definedp x (cdr table)))))

;; ---------------------------------------------------------------------------
;; This is the basic automaton recognizer for NQTHM.

(add-shell fsa* nil fsap*
  ((alphabet (none-of) zero)
   (states   (none-of) zero)
   (starts   (none-of) zero)
   (table    (none-of) zero)
   (finals   (none-of) zero)))

;; A "real" finite state automaton is an NQTHM-automaton with the following
;; properties:

(defn fsap (auto)
  (let ((al (alphabet auto))
        (st (states auto))
        (s0 (starts auto))
        (tr (table auto))
        (fi (finals auto)))
    (and (fsap* auto)
         (listp al)
         (listp st)
         (listp s0)
         (subsetp s0 st)
         (setp st)
         (setp fi)
         (subsetp fi st))))

;; A transition is a list ((state . input) nexts) where nexts is a list
;; of following states, for example (mk-transition ´q ´a ´(q r s)).

(defn mk-transition (state input nexts)
  (cons (cons state input) nexts))

;; Selector functions, will open up immediately

(defn state (trans) (caar trans))
```

```
(defn input (trans) (cdar trans))
(defn nexts (trans) (cdr trans))

;; A good transition is one where the input is a member of the alphabet,
;; and the states and all the elements of nexts are members of states.
;; nexts must also be a proper list.

(defn transitionp (trans alphabet states)
 (and (member (input trans) alphabet)
      (member (state trans) states)
      (subsetp (nexts trans) states)
      (plistp (nexts trans))))

;; A table is well-formed if all the entries are transitions
;; with respect to an alphabet and a table.

(defn wf-table (table alphabet states)
  (if (nlistp table)
      (equal table nil)
      (and (transitionp (car table) alphabet states)
           (wf-table (cdr table) alphabet states))))

;; A nice nondeterministic automaton is a well-formed one

(defn ndfsap (a)
  (and (fsap a)
       (wf-table (table a) (alphabet a) (states a))))

;; ----- Construct the new automaton

;; Looking up the next states is done like this. nil is returned if
;; there is no next state.

(defn next-states (table st a)
  (if (nlistp table)
      nil
      (if (equal (cons st a) (caar table))
          (nexts (car table))
          (next-states (cdr table) st a))))

;; If M is a well-formed table, then the result of next-states
;; is a subset of nstates.

(prove-lemma subsetp-next-states (rewrite)
 (implies (wf-table M alphabet nstates)
          (subsetp (next-states M state symbol)
                   nstates)))

;; If (cons st a) is not defined, the next-state is nil.

(prove-lemma non-definedp-next-state (rewrite)
 (implies (not (definedp (cons st a) table))
          (equal (next-states table st a) nil)))

;; If the next state is not in the first part, look in the second

(prove-lemma next-states-append (rewrite)
 (equal (next-states (append a b) s x)
        (if (definedp (cons s x) a)
            (next-states a s x)
            (next-states b s x))))

:; The deterministic start states is a list with each element
:; a singleton list containing a nondeterministic start state.

(defn dfsa-starts (l nstates)
```

```
  (if (nlistp l)
      nil
      (list (order l nstates))))

;; The deterministic next state is the closure of dstate in nfsa over symbol.

(defn dfsa-next-state (dstate symbol M)
  (if (nlistp dstate)
      nil
      (union (next-states M (car dstate) symbol)
             (dfsa-next-state (cdr dstate) symbol M))))

;; The calculated next state for the deterministic machine is a subset
;; of the nstates.

(prove-lemma subsetp-dfsa-next-state (rewrite)
 (implies (and (wf-table M alphabet nstates)
               (subsetp dstate nstates))
          (subsetp (dfsa-next-state dstate symbol M) nstates)))

;; I construct the deterministic transition by calculating the
;; deterministic next state and ordering it according to nfsa-states,
;; which is the basis for constructing the power-set, so that I
;; have a real member of the power-set as the next step.

(defn dfsa-next-transition (dstate symbol nfsa-table nfsa-states)
  (mk-transition dstate
                 symbol
                 (list (order
                        (dfsa-next-state dstate symbol nfsa-table)
                        nfsa-states))))

;; cdring down states, which is the powerset of all nfsa-states.

(defn dfsa-table-for-symbol (symbol states nfsa-table nfsa-states)
  (if (nlistp states)
      nil
      (cons (dfsa-next-transition (car states) symbol nfsa-table nfsa-states)
            (dfsa-table-for-symbol
             symbol (cdr states) nfsa-table nfsa-states))))

;; cdring down the alphabet...

(defn dfsa-table (alphabet states nfsa-table nfsa-states)
  (if (nlistp alphabet)
      nil
      (append (dfsa-table-for-symbol
               (car alphabet)
               states
               nfsa-table
               nfsa-states)
              (dfsa-table (cdr alphabet) states nfsa-table nfsa-states))))

(prove-lemma not-defined-next-states-nil (rewrite)
  (implies (not (definedp (cons s x)
                  (dfsa-table-for-symbol x b c d)))
           (equal (next-states (dfsa-table z b c d) s x) nil))
 ((do-not-generalize t)))

;; This is needed for some following lemmata, but it is a bad rewrite rule,
;; as it is considered in every test for equality. It is disabled.

(prove-lemma definedp-means-equal (rewrite)
  (implies (definedp (cons s a) (dfsa-table-for-symbol x b c d))
           (equal (equal a x) t)))
```

```
(disable definedp-means-equal)

(prove-lemma next-states-dfsa-table (rewrite)
 (implies (member a alphabet)
          (equal (next-states (dfsa-table alphabet b c d) s a)
                 (next-states (dfsa-table-for-symbol a b c d) s a)))
 ((enable definedp-means-equal)))

;; All final states with at least one non-deterministic final are
;; deterministic final states.

(defn dfsa-final-states (dstates nfsa-finals)
  (if (nlistp dstates)
      nil
       (if (some-member (car dstates) nfsa-finals)
          (cons (car dstates)
                (dfsa-final-states (cdr dstates) nfsa-finals))
          (dfsa-final-states (cdr dstates) nfsa-finals))))

;; *******************************************
;; This is the function that generates the DFSA
;; *******************************************

(defn generate-dfsa (nfsa)
  (let ((nstates (states nfsa)))
    (let ((dstates  (all-subbags nstates))
          (alphabet (alphabet nfsa)))
      (fsa* alphabet
            dstates
            (dfsa-starts (starts nfsa) nstates)
            (dfsa-table alphabet dstates (table nfsa) nstates)
            (dfsa-final-states dstates (finals nfsa))))))

;; ----- Some theorems about the DFSA -----

;; Need to show that the deterministic starts are members of the powerset.

(prove-lemma dfsa-final-states-subsetp (rewrite)
 (subsetp (dfsa-final-states x y) x))

;; Extremely bad rewrite rule! Must be disabled!

(prove-lemma dfsa-final-states-member (rewrite)
 (implies (member z (dfsa-final-states x y))
          (member z x)))

(disable dfsa-final-states-member)

;; Need to show that the deterministic finals are going to be a set.

(prove-lemma setp-dfsa-final-states (rewrite)
  (implies (setp dstates)
           (setp (dfsa-final-states dstates nfinals)))
 ((enable dfsa-final-states-member)))

;; When is an automaton a deterministic automaton?

;; A transition is deterministic if it is a transition over the alphabet
;; and the states, and there is at most one state in the list of nexts.

(defn deterministic-transition (tr alphabet states)
  (and (transitionp tr alphabet states)
       (leq (length (nexts tr)) 1)))

;; A table is deterministic if all the transitions are.
```

```
(defn deterministic-table (table alphabet states)
  (if (nlistp table)
      T
      (and (deterministic-transition (car table) alphabet states)
           (deterministic-table (cdr table) alphabet states))))

(defn dfsap (d)
  (and (fsap d)
       (deterministic-table (table d) (alphabet d) (states d))))

;; ---------------------------------------------------------------------------
;; ------------- The theorems ---------------------

;;  If nfsa is a nondeterministic fsa, and dfsa the one constructed
;;  from nfsa, then
;;  1) dfsa is deterministic,
;;  2) if nfsa accepts then dfsa accepts, and
;;  3) if dfsa accepts then nfsa accepts.

;; 1) generate-dfsa gives a deterministic automaton

(prove-lemma deterministic-table-append (rewrite)
 (equal (deterministic-table (append a b) alphabet states)
        (and (deterministic-table a alphabet states)
             (deterministic-table b alphabet states)))
 ((induct (length a))))

(prove-lemma deterministic-table-dfsa-table-for-symbol1 (rewrite)
   (implies (and (wf-table m alphabet nstates)
                 (subsetp dstates (all-subbags nstates))
                 (member symbol alphabet))
            (deterministic-table
             (dfsa-table-for-symbol symbol dstates m nstates)
             alphabet
             (all-subbags nstates))))

;; This needs subsetp-reflexive and the wierd previous lemma.

(prove-lemma deterministic-table-dfsa-table-for-symbol (rewrite)
 (let ((dstates (all-subbags nstates)))
   (implies (and (member symbol alphabet)
                 (wf-table m alphabet nstates))
            (deterministic-table
             (dfsa-table-for-symbol symbol dstates m nstates)
             alphabet
             dstates))))

;; The dfsa-table generated is deterministic.

(prove-lemma deterministic-table-dfsa-table (rewrite)
 (implies (and (wf-table m alphabet nstates)
               (subsetp x alphabet))
          (deterministic-table (dfsa-table x (all-subbags nstates) m nstates)
                               alphabet
                               (all-subbags nstates))))

;; ------------------------------------------------------------------------
;;      Theorem 1 : A deterministic automaton is obtained.
;; ------------------------------------------------------------------------

(prove-lemma dfsap-generate-dfsa ()
 (implies (ndfsap a)
          (dfsap (generate-dfsa a))))

;; -------------------------------------------------------------------------
;; 2) dfsa accepts if nfsa accepts
```

```
;; This function collects up all the next states for a list of states and a
;; symbol.

(defn next-states-list (table states a)
  (if (nlistp states)
      nil
      (union (next-states table (car states) a)
             (next-states-list table (cdr states) a))))

(prove-lemma next-states-list-same-as-dfsa-next-state (rewrite)
 (equal (next-states-list m nstates symbol)
        (dfsa-next-state nstates symbol m)))

(prove-lemma next-states-dfsa-table-for-symbol (rewrite)
 (implies (member dstate dstates)
          (equal (next-states
                  (dfsa-table-for-symbol c dstates ntab d) dstate c)
                 (nexts (dfsa-next-transition dstate c ntab d)))))

(prove-lemma dfsa-next-state-union (rewrite)
          (equal (dfsa-next-state (cons a b) c d)
                 (union (next-states d a c)
                        (dfsa-next-state b c d))))

;; This is the fifth different function I have used to define the notion
;; of acceptance in an automaton. I use the witness function some-member
;; to construct a member from the intersection of the final states reached
;; and the finals of the automaton. accept1 runs the automaton on the tape,
;; collecting up all reachable states.

(defn accept1 (table states finals tape)
 (if (nlistp tape)
     (some-member states finals)
     (accept1 table
              (next-states-list table states (car tape))
              finals
              (cdr tape))))

(defn accept (fsa tape)
 (accept1 (table fsa) (starts fsa) (finals fsa) tape))

(prove-lemma member-dstate-dfsa-final-states (rewrite)
             (implies (and (some-member dstate nfinals)
                           (member dstate dstates))
                      (member dstate (dfsa-final-states dstates nfinals))))

(prove-lemma order-final-states (rewrite)
  (implies (and (subsetp nfsa-finals nfsa-states)
                (some-member w nfsa-finals))
           (member (order w nfsa-states)
                   (dfsa-final-states (all-subbags nfsa-states)
                                      nfsa-finals))))

(disable member-all-subbags)

(prove-lemma subsetp-order (rewrite)
 (subsetp (order a b) a)
 ((disable ordered-member-all-subbags)))

(prove-lemma subsetp-next-states-2 (rewrite)
 (implies (member z b)
          (subsetp (next-states v z c)
                   (dfsa-next-state b c v))))

(prove-lemma dfsa-next-state-distrib (rewrite)
```

```
 (implies (subsetp a b)
          (subsetp (dfsa-next-state a c v)
                   (dfsa-next-state b c v)))
  ((do-not-generalize t)))

(prove-lemma subsetp-dfsa-next-state-1 (rewrite)
  (subsetp (dfsa-next-state (order w d) c v)
           (dfsa-next-state w c v)))

;; This is the schema I will use for proving order-dfsa-next-state-order.

(prove-lemma equal-order-subsetp (rewrite)
 (implies (and (subsetp a b)
               (subsetp b a))
          (equal (order a c)
                 (order b c))))

;; This is one half of the conjunct. It is always true.

(prove-lemma subsetp-dfsa-next-state-2-helper (rewrite)
 (subsetp (dfsa-next-state (order x y) c v)
          (dfsa-next-state (order (cons z x) y) c v)))

;; This is the second half. It is only true when w is a subset of d.

(prove-lemma subsetp-dfsa-next-state-2 (rewrite)
 (implies (subsetp w d)
          (subsetp (dfsa-next-state w c v)
                   (dfsa-next-state (order w d) c v))))

;; This one I got in the proof checker.

(prove-lemma order-dfsa-next-state-order
             (rewrite)
             (implies (and (wf-table ntab alphabet nstates)
                           (subsetp dstate nstates))
                      (equal (order (dfsa-next-state
                                      (order dstate nstates) symbol ntab)
                                     nstates)
                             (order
                              (dfsa-next-state dstate symbol ntab) nstates)))
             ((use (equal-order-subsetp
                    (b (dfsa-next-state dstate symbol ntab))
                    (a (dfsa-next-state (order dstate nstates) symbol ntab))
                    (c nstates)))))

(prove-lemma do-not-push (rewrite)
 (implies (and (subsetp dstate nstates)
               (subsetp nfinals nstates)
               (wf-table ntab alphabet nstates)
               (all-member tape alphabet)
               (accept1 ntab dstate nfinals tape))
          (accept1 (dfsa-table alphabet (all-subbags nstates) ntab nstates)
                   (list (order dstate nstates))
                   (dfsa-final-states (all-subbags nstates) nfinals)
                   tape))
 ((do-not-generalize t)))

;; ----------------------------------------------------------------------
;;      Theorem 2 : If a tape is recognized by the nondeterministic
;;            automaton, then it is recognized by the deterministic one.
;; ----------------------------------------------------------------------

(prove-lemma nfsa-accepts=>dfsa-accepts (rewrite)
 (implies (and (ndfsap nfsa)
               (all-member tape (ALPHABET NFSA))
```

```
               (accept nfsa tape))
          (accept (generate-dfsa nfsa) tape)))

;; ------------------------------------------------------------------------
;; 3) And now for the grand finale... nfsa accepts when the dfsa does

(prove-lemma member-dfsa-final-states-some-member (rewrite)
 (implies (member x (dfsa-final-states foo bar))
          (some-member x bar)))

;; This theorem was suggested by the proof checker.

(prove-lemma not-some-member-not-member-dfsa-final-states (rewrite)
 (implies (not (some-member w z))
          (not (member (order w d)
                       (dfsa-final-states (all-subbags d) z))))
 ((use (some-member-order (d d) (w w) (z z))
       (member-dfsa-final-states-some-member))))

(prove-lemma member-order-dfsa-final=>some-member (rewrite)
 (implies (and (subsetp w d)
               (subsetp z d)
               (member (order w d)
                       (dfsa-final-states (all-subbags d) z)))
          (some-member w z)))

(prove-lemma do-not-push-theorem-3 (rewrite)
  (implies (and (subsetp w d)
                (subsetp z d)
                (wf-table v x d)
                (all-member tape x)
                (accept1 (dfsa-table x (all-subbags d) v d)
                         (list (order w d))
                         (dfsa-final-states (all-subbags d) z)
                         tape))
           (accept1 v w z tape)))

;; ------------------------------------------------------------------------
;;            Theorem 3 : If a deterministic automaton accepts
;;                        then the non-deterministic one does too
;; ------------------------------------------------------------------------

;; And it proves!

(prove-lemma dfsa-accepts=>nfsa-accepts (rewrite)
 (implies (and (ndfsap nfsa)
               (all-member tape (ALPHABET NFSA))
               (accept (generate-dfsa nfsa) tape))
          (accept nfsa tape)))

;; Main Theorem: nfsa accepts iff dfsa accepts.

(prove-lemma NFSA=DFSA ()
 (implies (and (ndfsap nfsa)
               (all-member tape (alphabet nfsa)))
          (iff (accept (generate-dfsa nfsa) tape)
               (accept nfsa tape))))

#|
; As a test I have the example given in Aho/Ullman page 117 that
; is converted correctly to a deterministic automaton. Thus one
; knows that the proof is not just vacuously true: it does compute
; something other than nil.

(setq trans (list
                (mk-transition ´q0 1 ´(q0 q1))
```

```
                (mk-transition 'q0 2 '(q0 q2))
                (mk-transition 'q0 3 '(q0 q3))
                (mk-transition 'q1 1 '(q1 qf))
                (mk-transition 'q1 2 '(q1))
                (mk-transition 'q1 3 '(q1))
                (mk-transition 'q2 1 '(q2))
                (mk-transition 'q2 2 '(q2 qf))
                (mk-transition 'q2 3 '(q2))
                (mk-transition 'q3 1 '(q3))
                (mk-transition 'q3 2 '(q3))
                (mk-transition 'q3 3 '(q3 qf))))

(setq tape-bad '(1 2 1 2 1 2 3))
(setq tape-good  '(1 2 1 2 1 2 3 3))

(setq alphabet        '(1 2 3))
(setq states '(q0 q1 q2 q3 qf))
(setq starts '(q0))
(setq finals '(qf))

(setq nfsa (fsa* alphabet states starts trans finals))
(setq dfsa (generate-dfsa nfsa))

(accept nfsa tape-good)
(accept dfsa tape-good)
(accept nfsa tape-bad)
(accept dfsa tape-bad)
|#
```

A.2 With ϵ-transitions

This is the proof script for the ϵ-closure. Start PC-NQTHM (there are proofs that only PC-NQTHM can do) and submit the following script.

```
;; ---------------------------------------------------------------------------
;; ~/events/epsilon.events
;; ---------------------------------------------------------------------------

(boot-strap nqthm)

;; ---------------------------------------------------------------------------
;;      Library functions
;; ---------------------------------------------------------------------------
;; These functions were copied from the basis.events, bags-and-sets.events,
;; and lists.events for nqthm-1992 that I obtained in Austin 1993.
;; Only these were needed for the proof, and it speeds up the proof
;; considerably to keep unnecessary stuff out - they do not need to be
;; considered at every step

(defn setp (l)
  (if (not (listp l))
      t
      (and (not (member (car l) (cdr l)))
           (setp (cdr l)))))

(prove-lemma setp-cons (rewrite)
  (equal (setp (cons x l))
         (and (not (member x l))
              (setp l)))
  ((enable setp)))

(defn consl (x l)
   (if (listp l)
       (cons (cons x (car l))
             (consl x (cdr l)))
       nil))
```

```
(prove-lemma member-consl (rewrite)
  (equal (member x (consl v lst))
         (and (listp x)
              (equal (car x) v)
              (member (cdr x) lst))))

(prove-lemma member-non-list (rewrite)
  (implies (not (listp l))
           (not (member x l))))

(prove-lemma member-union (rewrite)
  (equal (member x (union a b))
         (or (member x a)
             (member x b))))

(defn length (l)
  (if (not (listp l))
      0
      (add1 (length (cdr l)))))

(prove-lemma length-cons (rewrite)
  (equal (length (cons a x))
         (add1 (length x)))
  ((enable length)))

(defn plistp (l)
  (if (not (listp l))
      (equal l nil)
      (plistp (cdr l))))

(prove-lemma plistp-cons (rewrite)
  (equal (plistp (cons a l))
         (plistp l))
  ((enable plistp)))

(defn all-subbags (l)
   (if (listp l)
       (let ((x (all-subbags (cdr l))))
         (union x (consl (car l) x)))
       (list nil)))
;; ------------------------------------------------------------------------
;;      Basic functions
;; ------------------------------------------------------------------------

;; some-member
;; This is a witness for the existence of an element in the intersection

(defn some-member (l1 l2)
 (if (nlistp l1)
     F
     (if (member (car l1) l2)
         T
         (some-member (cdr l1) l2))))

(prove-lemma some-member-member (rewrite)
  (implies (and (member n y)
                (member n x))
           (some-member x y)))

;; ------------------------------------------------------------------------
;;      subsetp
;; ------------------------------------------------------------------------
;; I will use a weak definition of subset: all elements of a are in b
;; (some may be double).
```

```
(defn subsetp (a b)
  (if (nlistp a)
      T
      (and (member (car a) b)
           (subsetp (cdr a) b))))

(prove-lemma subsetp-union (rewrite)
 (equal (subsetp (union a b) c)
        (and (subsetp a c)
             (subsetp b c))))

(prove-lemma subsetp-union2 (rewrite)
 (subsetp y (union y x)))

(prove-lemma subsetp-cons (rewrite)
 (implies (subsetp x y)
          (subsetp x (cons z y))))

(prove-lemma subsetp-reflexive (rewrite)
 (subsetp x x))

(prove-lemma member-subsetp (Rewrite)
 (implies (and (member x a)
               (subsetp a b))
          (member x b)))

(prove-lemma some-member-and-subsetp (rewrite)
 (implies (and (some-member a b)
               (subsetp b c))
          (some-member a c)))

(prove-lemma another-some-member-rule (rewrite)
 (implies (and (subsetp a b)
               (some-member a c))
          (some-member b c)))

(prove-lemma subsetp-union-anything (rewrite)
 (implies (subsetp x y)
          (subsetp x (union anything y))))

;; ------------------------------------------------------------------------
;; ----- theorems about setp -----

; consl preserves setp.

(prove-lemma setp-consl (Rewrite)
 (implies (setp x)
          (setp (consl y x))))

(prove-lemma setp-union (rewrite)
 (implies (and (setp x)
               (setp y))
          (setp (union x y))))

;; ------------------------------------------------------------------------
;; ----- theorems about consl -----

(prove-lemma setp-union-consl (REWRITE)
   (IMPLIES (SETP Y)
            (SETP (UNION Y (CONSL V Y)))))

;; ------------------------------------------------------------------------
;; ----- theorems about all-subbags -----

(prove-lemma listp-all-subbags (rewrite)
 (listp (all-subbags d)))
```

```
;; nil is a member of all power sets.

(prove-lemma nil-member-all-subbags (rewrite)
 (member nil (all-subbags x)))

;; Singleton lists of members of a list are members of the power set.

(prove-lemma member-list-all-subbags (rewrite)
  (implies (member x y)
           (member (list x) (all-subbags y))))

;; If x is a set, the ´power-set´ is a set.

(prove-lemma setp-all-subbags (rewrite)
 (implies (setp x)
          (setp (all-subbags x))))

(defn all-subbags-hint1 (x y)
  (if (nlistp y)
      T
      (and (all-subbags-hint1 (cdr x) (cdr y))
           (all-subbags-hint1 x (cdr y)))))

(prove-lemma member-all-subbags (rewrite)
 (implies (member x (all-subbags y))
          (subsetp x y))
 ((induct (all-subbags-hint1 x y))))

;; ------------------------------------------------------------------------
;; ----- Theorems about union

(prove-lemma nothing-member-nil (rewrite)
             (not (member x nil)))

(prove-lemma union-right-id (rewrite)
 (implies (plistp foo)
          (equal (union foo nil) foo)))

(prove-lemma plistp-union (rewrite)
 (implies (plistp y)
          (plistp (union anything y))))

;; ------------------------------------------------------------------------
;;      order
;; ------------------------------------------------------------------------
;; To simulate set equality, I order the list representation of
;; a set (x) along a basis (lst).

;; Order the elements of x according to lst, a finite total order for x.

(defn order (x lst)
  (if (nlistp lst)
      nil
      (if (member (car lst) x)
          (cons (car lst) (order x (cdr lst)))
          (order x (cdr lst)))))

(prove-lemma order-preserves-member (rewrite)
 (implies (member x (order y z))
          (and (member x y)
               (member x z))))

(prove-lemma member-order (rewrite)
 (implies (and (member z x)
               (member z v))
```

```
          (member z (order x v))))

(prove-lemma order-cons (rewrite)
  (implies (member z x)
           (equal (order (cons z x) v)
                  (order x v))))

(prove-lemma order-cons2 (rewrite)
  (implies (not (member z v))
           (equal (order (cons z x) v)
                  (order x v))))

(prove-lemma order-cons4 (rewrite)
  (implies (member z x)
           (equal (order (cons z (order x v)) v)
                  (order (order x v) v)))
  ((use (member-order))
   (disable member-order)))

(prove-lemma order-idempotent (rewrite)
  (equal (order (order x y) y) (order x y)))

(prove-lemma subsetp-order-2 (rewrite)
 (subsetp (order a b)
          (order (cons c a) b)))

(prove-lemma subsetp-order-4 (rewrite)
 (implies (and (subsetp a b)
               (member x a))
          (member x (order a b))))

; ---- order and all-subbags

(prove-lemma helper1 (rewrite)
 (implies (equal (order x z) y)
          (member y (all-subbags z))))

(prove-lemma ordered-member-all-subbags (rewrite)
 (implies (equal (order x z) x)
          (member x (all-subbags z))))

(disable helper1)
(disable ordered-member-all-subbags)

(prove-lemma member-order-all-subbags (rewrite)
 (member (order w d) (all-subbags d)))

(prove-lemma some-member-order1 (rewrite)
 (implies (some-member (order a b) c)
          (some-member (order (cons x a) b) c))
 ((disable member-subsetp)))

(prove-lemma some-member-order (rewrite)
 (implies (some-member (order w d) z)
          (some-member w z)))

(prove-lemma some-member-order-2 (rewrite)
 (implies (and (subsetp b c)
               (some-member a b))
          (some-member (order a c) b))
 ((induct (length a))))

(prove-lemma subsetp-order (rewrite)
 (subsetp (order a b) a)
 ((disable ordered-member-all-subbags)))
```

```
;; ----------------------------------------------------------------------
;;      all-member
;; ----------------------------------------------------------------------

;; This is a key definition for the proof - it turns out to be the same
;; as my definition of subsetp.

(defn all-member
      (a b)
      (if (nlistp a)
          t
          (and (member (car a) b)
               (all-member (cdr a) b))))

;; ----------------------------------------------------------------------
;; ----- Definedp
;; ----------------------------------------------------------------------
;; Needed to explain the workings of next-state

(defn definedp (x table)
  (if (nlistp table)
      F
      (or (equal x (caar table))
          (definedp x (cdr table)))))

;; ----------------------------------------------------------------------
;; This is the basic automaton recognizer for NQTHM.

(add-shell fsa* nil fsap*
  ((alphabet (none-of) zero)
   (states   (none-of) zero)
   (starts   (none-of) zero)
   (table    (none-of) zero)
   (finals   (none-of) zero)))

;; A "real" finite state automaton is an NQTHM-automaton with the following
;; properties:

(defn fsap (auto)
  (let ((al (alphabet auto))
        (st (states auto))
        (s0 (starts auto))
        (tr (table auto))
        (fi (finals auto)))
    (and (fsap* auto)
         (listp al)
         (listp st)
         (listp s0)
         (subsetp s0 st)
         (setp st)
         (setp fi)
         (subsetp fi st))))

;; A transition is a list ((state . input) nexts) where nexts is a list
;; of following states, for example (mk-transition ´q ´a ´(q r s)).

(defn mk-transition (state input nexts)
  (cons (cons state input) nexts))

;; Selector functions, will open up immediately.

(defn state (trans) (caar trans))
(defn input (trans) (cdar trans))
(defn nexts (trans) (cdr trans))

;; Not being able to determine if the epsilon representation is a member
```

```
;; of the alphabet is not a GOOD THING. Therefore an explicit
;; representation will be used.

(defn epsilon () ´epsilon)

;; A good transition is one where the input is a member of the alphabet,
;; and the states and all the elements of nexts are members of states.
;; nexts must also be a proper list.

(defn nfsa-transitionp (trans alphabet states)
 (and (or (member (input trans) alphabet)
          (equal (input trans) (epsilon)))
      (member (state trans) states)
      (subsetp (nexts trans) states)
      (plistp (nexts trans))))

;; A table is well-formed if all the entries are transitions
;; with respect to an alphabet and a table.

(defn all-nfsa-transitions (table alphabet states)
  (if (nlistp table)
      (equal table nil)
      (and (nfsa-transitionp (car table) alphabet states)
           (all-nfsa-transitions (cdr table) alphabet states))))

(defn wf-nfsa-table (table alphabet states)
  (and (all-nfsa-transitions table alphabet states)
       (not (member (epsilon) alphabet))))

;; A nice nondeterministic automaton is a well-formed one.

(defn ndfsap (a)
  (and (fsap a)
       (wf-nfsa-table (table a) (alphabet a) (states a))))

;; ----- Construct the new automaton

;; Looking up the next states is done like this. nil is returned if
;; there is no next state.

(defn next-states (table st a)
  (if (nlistp table)
      nil
      (if (equal (cons st a) (caar table))
          (nexts (car table))
          (next-states (cdr table) st a))))

(prove-lemma subsetp-next-states (rewrite)
 (implies (all-nfsa-transitions M alphabet nstates)
          (subsetp (next-states M state symbol)
                   nstates)))

;; ------------------------------------------------------------------------
;; Epsilon closure

;; For all states in the list, take an epsilon step

(defn one-epsilon-step-all (states table)
 (if (nlistp states)
     nil
     (union
      (next-states table (car states) (epsilon))
      (one-epsilon-step-all (cdr states) table))))

;; This will only prove on (length b), not on (length a)! It seems to depend
;; on the structure of the function ´union!
```

```
(prove-lemma not-lessp-length-union (rewrite)
          (not (lessp (length (union a b))
                      (length b))))

;; This is the epsilon closure without needing a clock.
;; The termination argument: if they are the same length, they are the same.

(defn epsilon-closure (table states all-states)
    (let ((next-set (union (one-epsilon-step-all states table) states)))
      (if (or (equal (length states) (length next-set))
              (not (lessp (length next-set) (length all-states))))
          states
        (epsilon-closure table next-set all-states)))
 ((lessp (difference (length all-states) (length states)))))

;; Take a step from state on sym, and then take the epsilon closure.

(defn next-with-epsilon-closure (table state sym all-states)
  (epsilon-closure table  (next-states table state sym) all-states))

;; This lemma needed three levels of induction.

(prove-lemma subsetp-one-epsilon-step-all1 (rewrite)
 (implies (and (all-nfsa-transitions M alphabet nstates)
               (subsetp states nstates))
          (subsetp (one-epsilon-step-all states M)
                   nstates)))

(prove-lemma subsetp-epsilon-closure (rewrite)
 (implies (and (all-nfsa-transitions m alphabet nstates)
               (subsetp states nstates))
          (subsetp (epsilon-closure m states nstates) nstates)))

(prove-lemma subsetp-epsilon-closure-2 (Rewrite)
 (subsetp states
          (epsilon-closure m states all-states)))

;; If (cons st a) is not defined, the next-state is nil.

(prove-lemma non-definedp-next-state (rewrite)
 (implies (not (definedp (cons st a) table))
          (equal (next-states table st a) nil)))

;; If the next state is not in the first part, look in the second.

(prove-lemma next-states-append (rewrite)
 (equal (next-states (append a b) s x)
        (if (definedp (cons s x) a)
            (next-states a s x)
            (next-states b s x))))

;; The deterministic start states is a list containing the
;; epsilon-closure of all the nondeterministic start states.

(defn dfsa-starts (l table nstates)
  (list (order (epsilon-closure table l nstates) nstates)))

;; The function next-states-list collects up all the next states for a list
;; of states and a symbol. I have to take the epsilon closure for the
;; nondeterministic automaton. It won´t hurt the deterministic one if it is
;; shown that there are no epsilon productions in the generated automaton !

(defn next-states-list (states a table all-states)
  (if (nlistp states)
      nil
```

```
      (union
       (next-with-epsilon-closure table (car states) a all-states)
       (next-states-list (cdr states) a table all-states))))

(prove-lemma plistp-next-states-list (rewrite)
 (implies (all-nfsa-transitions table alphabet nstates)
  (plistp  (next-states-list dstates symbol table nstates))))

(prove-lemma subsetp-next-states-list (rewrite)
 (implies (and (subsetp dstates nstates)
               (all-nfsa-transitions table alphabet nstates))
          (subsetp (next-states-list dstates symbol table nstates) nstates)))

;; The deterministic next state is the closure of dstate in nfsa over symbol.

(defn dfsa-next-state (dstate symbol M all-states)
  (if (nlistp dstate)
      nil
      (next-states-list (epsilon-closure M dstate all-states)
                        symbol M all-states)))

;; The calculated next state for the deterministic machine is a subset
;; of the nstates.

(prove-lemma subsetp-dfsa-next-state (rewrite)
 (implies (and (all-nfsa-transitions  M alphabet nstates)
               (subsetp dstate nstates))
          (subsetp (dfsa-next-state dstate symbol M nstates) nstates)))

;; I construct the deterministic transition by calculating the
;; deterministic next state and ordering it according to nfsa-states,
;; which is the basis for constructing the power-set, so that I
;; have a real member of the power-set as the next step.

(defn dfsa-next-transition (dstate symbol nfsa-table nfsa-states)
  (mk-transition dstate
                 symbol
                 (list (order
                        (dfsa-next-state
                         dstate symbol nfsa-table nfsa-states)
                        nfsa-states))))

;; cdring down states, which is the powerset of all nfsa-states.

(defn dfsa-table-for-symbol (symbol states nfsa-table nfsa-states)
  (if (nlistp states)
      nil
      (cons (dfsa-next-transition (car states) symbol nfsa-table nfsa-states)
            (dfsa-table-for-symbol
             symbol (cdr states) nfsa-table nfsa-states))))

;; cdring down the alphabet...

(defn dfsa-table (alphabet states nfsa-table nfsa-states)
  (if (nlistp alphabet)
      nil
      (append (dfsa-table-for-symbol
               (car alphabet)
               states
               nfsa-table
               nfsa-states)
              (dfsa-table (cdr alphabet) states nfsa-table nfsa-states))))

;; This lemma, too, needs three levels of induction, be patient.

(prove-lemma not-defined-next-states-nil (rewrite)
```

```
  (implies (not (definedp (cons s x)
                  (dfsa-table-for-symbol x b c d)))
           (equal (next-states (dfsa-table z b c d) s x) nil))
 ((do-not-generalize t)))

;; This is needed for some following lemma, but it is a bad rewrite rule,
;; as it is considered in every test for equality. It is disabled.

(prove-lemma definedp-means-equal (rewrite)
  (implies (definedp (cons s a) (dfsa-table-for-symbol x b c d))
           (equal (equal a x) t)))

(disable definedp-means-equal)

; proves with next-states append, just give it time!
(prove-lemma next-states-dfsa-table (rewrite)
 (implies (member a alphabet)
          (equal (next-states (dfsa-table alphabet b c d) s a)
                 (next-states (dfsa-table-for-symbol a b c d) s a)))
 ((enable definedp-means-equal)))

;; All final states with at least one nondeterministic final are
;; deterministic final states.

(defn dfsa-final-states (dstates nfsa-finals)
  (if (nlistp dstates)
      nil
       (if (some-member (car dstates) nfsa-finals)
          (cons (car dstates)
                (dfsa-final-states (cdr dstates) nfsa-finals))
          (dfsa-final-states (cdr dstates) nfsa-finals))))

; ********************************************
; This is the function that generates the DFSA
; ********************************************

(defn generate-dfsa (nfsa)
  (let ((nstates (states nfsa)))
    (let ((dstates  (all-subbags nstates))
          (table    (table nfsa))
          (alphabet (alphabet nfsa)))
      (fsa* alphabet
            dstates
            (dfsa-starts (starts nfsa) (table nfsa) nstates)
            (dfsa-table alphabet dstates table nstates)
            (dfsa-final-states
             dstates
             (finals nfsa))))))

;; ----- Some theorems about the DFSA -----

;; Need to show that the deterministic starts are members of the powerset.

(prove-lemma dfsa-final-states-subsetp (rewrite)
 (subsetp (dfsa-final-states x y) x))

;; This rewrite rule must be disabled!
(prove-lemma dfsa-final-states-member (rewrite)
 (implies (member z (dfsa-final-states x y))
          (member z x)))

(disable dfsa-final-states-member)

;; Need to show that the deterministic finals are going to be a set.

(prove-lemma setp-dfsa-final-states (rewrite)
```

```
  (implies (setp dstates)
           (setp (dfsa-final-states dstates nfinals)))
 ((enable dfsa-final-states-member)))

;; When is an automaton a deterministic automaton?

;; A transition is deterministic if it is a transition over the alphabet
;; and the states, and there is at most one state in the list of nexts.

(defn dfsa-transitionp (trans alphabet states)
 (and (member (input trans) alphabet)
      (member (state trans) states)
      (subsetp (nexts trans) states)
      (plistp (nexts trans))
      (leq (length (nexts trans)) 1)))

; A table is deterministic is all the transitions are.

(defn all-dfsa-transitions (table alphabet states)
  (if (nlistp table)
      T
      (and (dfsa-transitionp (car table) alphabet states)
           (all-dfsa-transitions (cdr table) alphabet states))))

(defn wf-dfsa-table
      (table alphabet states)
      (and (all-dfsa-transitions table alphabet states)
           (not (member (epsilon) alphabet))))

(defn dfsap (d)
  (and (fsap d)
       (wf-dfsa-table (table d) (alphabet d) (states d))))

;; -----------------------------------------------------------------------
;; Many proofs are the same as without epsilon-closure.
;; ------------- The theorems ---------------------

;;  If nfsa is a nondeterministic fsa and dfsa the one constructed
;;  from nfsa, then
;;  1) dfsa is deterministic,
;;  2) if nfsa accepts then dfsa accepts, and
;;  3) if dfsa accepts then nfsa accepts.

;; 1) generate-dfsa gives a deterministic automaton.

(prove-lemma all-dfsa-transitions-append (rewrite)
 (equal (all-dfsa-transitions (append a b) alphabet states)
        (and (all-dfsa-transitions a alphabet states)
             (all-dfsa-transitions b alphabet states)))
 ((induct (length a))))

;; This is STRANGE! It is necessary for the following proof to go through.
;; It seems to be just a method to force the prover to induct on dstates!
;; If I use EQUAL instead of SUBSETP (which is "truer"), then the prover
;; selects the extremely stupid induction on M, and cannot be
;; convinced, even by hint, to try an induction on dstates.

(prove-lemma all-dfsa-transitions-dfsa-table-for-symbol1 (rewrite)
   (implies (and (wf-nfsa-table m alphabet nstates)
                 (subsetp dstates (all-subbags nstates))
                 (member symbol alphabet))
            (all-dfsa-transitions
             (dfsa-table-for-symbol symbol dstates m nstates)
             alphabet
             (all-subbags nstates))))
```

```
;; Needed subsetp-reflexive and the wierd previous lemma.

(prove-lemma all-dfsa-transitions-dfsa-table-for-symbol (rewrite)
 (let ((dstates (all-subbags nstates)))
   (implies (and (member symbol alphabet)
                 (wf-nfsa-table m alphabet nstates))
            (all-dfsa-transitions
             (dfsa-table-for-symbol symbol dstates m nstates)
             alphabet
             dstates))))

;; The dfsa-table generated is deterministic.

(prove-lemma all-dfsa-transitions-dfsa-table (rewrite)
 (implies (and (wf-nfsa-table m alphabet nstates)
               (subsetp x alphabet))
          (all-dfsa-transitions
               (dfsa-table x (all-subbags nstates) m nstates)
               alphabet
               (all-subbags nstates))))

;; ------------------------------------------------------------------------
;;      Theorem 1 : A deterministic automaton is obtained
;; ------------------------------------------------------------------------

(prove-lemma dfsap-generate-dfsa ()
 (implies (ndfsap a)
          (dfsap (generate-dfsa a))))

;; -------------------------------------------------------------------------
;; 2) dfsa accepts if nfsa accepts

(prove-lemma next-states-dfsa-table-for-symbol (rewrite)
 (implies (member dstate dstates)
          (equal (next-states
                   (dfsa-table-for-symbol c dstates ntab d) dstate c)
                  (nexts (dfsa-next-transition dstate c ntab d)))))

;; -------------------------------------------------------------------------
;; The proof in the Hopcroft/Ullman book is not acceptance, but involves
;; the set of states reached on a tape (input).

(defn reached (table states tape all-states)
 (if (nlistp tape)
     states
     (reached table
              (next-states-list states (car tape) table all-states)
              (cdr tape)
              all-states)))

(prove-lemma reached-append (rewrite)
 (equal (reached table states (append a b) all-states)
        (reached table (reached table states a all-states) b all-states)))

(prove-lemma subsetp-reached (rewrite)
 (implies (and (subsetp starts nstates)
               (all-nfsa-transitions table alphabet nstates))
          (subsetp (reached table starts tape nstates) nstates)))

(prove-lemma plistp-reached (rewrite)
 (implies (and (all-nfsa-transitions ntab alphabet nstates)
               (plistp starts)
               (subsetp starts nstates))
          (plistp (reached ntab starts tape nstates))))

(prove-lemma plistp-epsilon-closure (rewrite)
```

```
 (implies (plistp states)
          (plistp (epsilon-closure table states all-states))))

(prove-lemma next-states-list-nil (rewrite)
 (equal (next-states-list nil symbol table states)
        nil))

;; ------------------------------------------------------------------------
;; Here are the axioms that I strongly believe.
;; ------------------------------------------------------------------------

(axiom next-states-list-order-equal (rewrite)
 (implies (subsetp a b)
          (equal (next-states-list (order a b) symbol table states)
                 (order (next-states-list a symbol table states) b))))

(axiom epsilon-closure-dfsa-identity (rewrite)
          (equal (epsilon-closure (DFSA-TABLE ALPHABET dstates NTAB NSTATES)
                                  x
                                  dstates)
                 x))

;; If we reached a state, it has just been epsilon-closed, so another
;; epsilon-closure won´t change anything.

(axiom next-states-list-epsilon-closure-reached (rewrite)
  (equal (next-states-list
           (epsilon-closure ntab
             (order (reached ntab starts tape nstates) nstates)
             nstates)
           symbol ntab nstates)
         (next-states-list (order (reached ntab starts tape nstates) nstates)
                           symbol ntab nstates)))

;; This should be the main result helper, but it uses a "wrong"
;; induction structure.

(prove-lemma reaches-nfsa-reaches-dfsa (rewrite)
 (implies (and (subsetp starts nstates)
               (listp tape)
               (all-nfsa-transitions ntab alphabet nstates)
               (member symbol alphabet)
               (listp (order (reached ntab starts tape nstates) nstates))
               (equal (reached (dfsa-table alphabet
                                           (all-subbags nstates)
                                           ntab nstates)
                               (dfsa-starts starts nfsa nstates)
                               tape
                               (all-subbags nstates))
                      (list
                       (order (reached ntab starts tape nstates) nstates))))
          (equal (reached (dfsa-table alphabet
                                      (all-subbags nstates)
                                      ntab nstates)
                          (dfsa-starts starts nfsa nstates)
                          (append tape (list symbol))
                          (all-subbags nstates))
                 (list (order (reached ntab starts
                                       (append tape (list symbol))
                                       nstates)
                              nstates))))
           ((INSTRUCTIONS PROMOTE
                          (DIVE 1)
                          (REWRITE REACHED-APPEND)
                          (DIVE 2)
                          = TOP
```

```
                        (DIVE 2 1 1)
                        (REWRITE REACHED-APPEND)
                        TOP
                        (DIVE 1)
                        X X
                        (REWRITE UNION-RIGHT-ID)
                        (DIVE 2)
                        (REWRITE NEXT-STATES-DFSA-TABLE)
                        (REWRITE NEXT-STATES-DFSA-TABLE-FOR-SYMBOL)
                        S UP
                        (REWRITE EPSILON-CLOSURE-DFSA-IDENTITY)
                        TOP S
                        (DIVE 2 1)
                        X TOP
                        (DIVE 1 1)
                        (REWRITE NEXT-STATES-LIST-EPSILON-CLOSURE-REACHED)
                        (REWRITE NEXT-STATES-LIST-ORDER-EQUAL)
                        UP
                        (REWRITE ORDER-IDEMPOTENT)
                        TOP PROVE PROVE PROVE PROVE )))

;; This is the fifth different acceptance function.

(defn accept (fsa tape)
 (some-member
  (reached (table fsa) (starts fsa) tape (states fsa))
  (finals fsa)))

;; ------------------------------------------------------------------------
;; The following are the formulations of the next theorems. There were,
;; however, not completed as the axioms could not be removed from the
;; previous proof.

;; ------------------------------------------------------------------------
;;      Theorem 2 : If a tape is recognized by the non-deterministic
;;            automaton, then it is recognized by the deterministic one
;; ------------------------------------------------------------------------

;; need to do the induction "backwards", from the back.

;(prove-lemma nfsa-accepts=>dfsa-accepts (rewrite)
; (implies (and (ndfsap nfsa)
;               (all-member tape (ALPHABET NFSA))
;               (accept nfsa tape))
;          (accept (generate-dfsa nfsa) tape))
; ((disable  wf-nfsa-table reached dfsa-starts)))

;; -------------------------------------------------------------------------
;;           Theorem 3 : If a deterministic automaton accepts
;;                       then the non-deterministic one does too
;; -------------------------------------------------------------------------

;(prove-lemma dfsa-accepts=>nfsa-accepts (rewrite)
; (implies (and (ndfsap nfsa)
;               (all-member tape (alphabet nfsa))
;               (listp tape) ; only valid for non-empty tapes!
;               (accept (generate-dfsa nfsa) tape))
;          (accept nfsa tape)))

;; Main Theorem: iff statt equal

;(prove-lemma NFSA=DFSA ()
; (implies (and (ndfsap nfsa)
;               (all-member tape (alphabet nfsa)))
;          (iff (accept (generate-dfsa nfsa) tape)
;               (accept nfsa tape))))
```

Appendix B

Basic Data Type Modules

B.1 Script Load Order

This is the init.lsp file needed to load and prove the scanner and the parser scripts.

```
; Contributions to mechanically proven correct compiler front-ends
;
; Debora Weber-Wulff
;
; This is the init.lsp file needed to load everything up

(boot-strap nqthm)
(load "/usr1/name/weberwu/new/events/numbers.events")
(load "/usr1/name/weberwu/new/events/sets.events")
(load "/usr1/name/weberwu/new/events/lists.events")
(load "/usr1/name/weberwu/new/events/grammar.events")
(load "/usr1/name/weberwu/new/events/stack.events")
(load "/usr1/name/weberwu/new/events/token.events")
(load "/usr1/name/weberwu/new/events/tree.events")
(load "/usr1/name/weberwu/new/events/configuration.events")
(load "/usr1/name/weberwu/new/events/actions.events")
(load "/usr1/name/weberwu/new/events/scanning.events")
(load "/usr1/name/weberwu/new/events/toktrans-1.events")
(load "/usr1/name/weberwu/new/events/toktrans-2.events")
(load "/usr1/name/weberwu/new/events/toktrans-3.events")
(load "/usr1/name/weberwu/new/events/toktrans-4.events")
(load "/usr1/name/weberwu/new/events/toktrans-5.events")
(load "/usr1/name/weberwu/new/events/toktrans-6.events")
(load "/usr1/name/weberwu/new/events/toktrans-7.events")
(load "/usr1/name/weberwu/new/events/derivation.events")
(load "/usr1/name/weberwu/new/events/follow.events")
(load "/usr1/name/weberwu/new/events/table-generator.events")
(load "/usr1/name/weberwu/new/events/parser.events")
(load "/usr1/name/weberwu/new/events/parsing-inv.events")
```

These are the scripts for the basic data types used in the scanner and the parser.

B.2 Numbers

```
;; ---------------------------------------------------------------------------
;; ~/events/numbers.events
;; ---------------------------------------------------------------------------
;;
;; This is the collection of theorems from the naturals library
;; needed for toktrans-4.
;;
;; ---------------------------------------------------------------------------
;; pick one of the following: Either the library or the easy lemmata and
;;                            the axioms
;; ---------------------------------------------------------------------------
#|
```

```
; ------------------------ The naturals library ---------------------------
;;   from the nqthm-1992 examples/numbers directory
(note-lib "/usr1/name/weberwu/bm/nqthm-1992/examples/numbers/naturals")
;(enable-theory addition)
:(enable-theory multiplication)
:(enable-theory remainders)
:(enable-theory quotients)
(enable-theory naturals)

|#

;#|
;; -----------------------------------------------------------------------
;; These are lemmata extracted from the naturals library.
;; -----------------------------------------------------------------------

(prove-lemma commutativity-of-plus (rewrite)
       (equal (plus x y) (plus y x)))

(prove-lemma equal-plus-0 (rewrite)
       (equal (equal (plus a b) 0)
              (and (zerop a)
                   (zerop b))))

(prove-lemma plus-zero-arg2 (rewrite)
       (implies (zerop y)
                (equal (plus x y)
                       (fix x)))
       ((induct (plus x y))))

(prove-lemma times-zero (rewrite)
       (implies (zerop y)
                (equal (times x y) 0)))

(prove-lemma equal-times-0 (rewrite)
       (equal (equal (times x y) 0)
              (or (zerop x)
                  (zerop y)))
       ((induct (times x y))))

;; Fix coerces non-numbers to zero.

(prove-lemma times-add1 (rewrite)
       (equal (times x (add1 y))
              (if (numberp y)
                  (plus x (times x y))
                  (fix x))))

(prove-lemma plus-remainder-times-quotient (rewrite)
       (equal (plus (remainder x y) (times y (quotient x y)))
              (fix x)))

(prove-lemma lessp-quotient (rewrite)
       (equal (lessp (quotient i j) i)
              (and (not (zerop i))
                   (not (equal j 1)))))

(prove-lemma commutativity-of-times (rewrite)
       (equal (times y x)
              (times x y)))

(prove-lemma quotient-lessp-arg1 (rewrite)
       (implies (lessp a b)
                (equal (quotient a b) 0)))

;; -----------------------------------------------------------------------
;; These are lemmata from the naturals library expressed as axioms.
;; -----------------------------------------------------------------------

(add-axiom remainder-plus (rewrite)
 (implies (equal (remainder a c) 0)
          (equal (remainder (plus b a) c)
```

```
                          (remainder b c))))

(add-axiom quotient-plus (rewrite)
           (implies (equal (remainder a c) 0)
                    (equal (quotient (plus b a) c)
                           (plus (quotient a c) (quotient b c)))))

(add-axiom quotient-times-instance (rewrite)
           (equal (quotient (times y x) y)
                  (if (zerop y)
                      0
                      (fix x)))))

(add-axiom remainder-times1-instance (rewrite)
       (and (equal (remainder (times x y) y) 0)
            (equal (remainder (times x y) x) 0)))

;; -----------------------------------------------------------------
;|#
```

B.3 Sets

```
;; -----------------------------------------------------------------------------
;;  ~/events/sets.events
;; -----------------------------------------------------------------------------
;; This is the set theory needed for the proof.
;; Major portions of it are taken from Matt Kaufmann´s set library.
;; -----------------------------------------------------------------------------

;; The function subset should probably be named subbag.

(defn subsetp (a b)
  (if (nlistp a)
      T
      (and (member (car a) b)
           (subsetp (cdr a) b))))

;; A proper set has no duplicates.

(defn setp (l)
  (if (not (listp l))
      t
      (and (not (member (car l) (cdr l)))
           (setp (cdr l)))))

;; The cardinality of a set is just the length of the list representing it.

(defn card (l)
  (if (listp l)
      (add1 (card (cdr l)))
      0))

;; The intersection of two sets is all elements that are members in both.

(defn intersection (x y)
  (if (listp x)
      (if (member (car x) y)
          (cons (car x) (intersection (cdr x) y))
          (intersection (cdr x) y))
      nil))

;; This function will make a proper set out of the elements of a list.

(defn mk-unique-set (set)
 (if (nlistp set)
     nil
     (union (car set) (mk-unique-set (cdr set)))))

(deftheory sets (subsetp setp card intersection mk-unique-set))

(disable-theory sets)
```

B.4 Lists

```
;; ---------------------------------------------------------------------------
;; ~/events/lists.events
;; ---------------------------------------------------------------------------
;; These are some functions on lists, adapted from the CLInc list library.

(defn last (x)
  (if (nlistp x)
      x
      (if (nlistp (cdr x))
          x
          (last (cdr x)))))

;; ------------- length and lemmata on length -----------------

(defn length (l)
  (if (not (listp l))
      0
      (add1 (length (cdr l)))))

(prove-lemma equal-length-0 (rewrite)
  (equal (equal (length l) 0)
         (not (listp l)))
  ((enable length)))

(prove-lemma length-nlistp (rewrite)
  (implies (nlistp x)
           (equal (length x) 0))
  ((enable length)))

(prove-lemma length-cons (rewrite)
  (equal (length (cons a x))
         (add1 (length x)))
  ((enable length)))

(prove-lemma lessp-length-cons (rewrite)
  (equal (lessp (length z) (length (cons v z)))
         t)
 ((enable length-cons)))

(prove-lemma lessp-length-cdr (rewrite)
 (implies (listp x)
          (lessp (length (cdr x))
                 (length x)))
 ((do-not-induct t)))

;; The function plist constructs a proper (i.e. nil as last cdr) list out of l.

(defn plist (l)
  (if (not (listp l))
      nil
      (cons (car l) (plist (cdr l)))))

(defn plistp (l)
  (if (not (listp l))
      (equal l nil)
      (plistp (cdr l))))

(prove-lemma plistp-nlistp (rewrite)
  (implies (nlistp l)
           (equal (plistp l)
                  (equal l nil)))
  ((enable plistp)))

(prove-lemma equal-plist (rewrite)
  (implies (plistp l)
           (equal (plist l) l))
  ((enable plistp plist)))

(prove-lemma plistp-cons (rewrite)
  (equal (plistp (cons a l))
```

```
        (plistp l))
  ((enable plistp)))

; ---------- lemmata on append ----------

(prove-lemma plistp-append (rewrite)
  (equal (plistp (append a b))
         (plistp b))
  ((enable plistp append)))

(prove-lemma append-left-id (rewrite)
  (implies (not (listp a))
           (equal (append a b)
                  b))
  ((enable append)))

(prove-lemma append-nil (rewrite)
  (equal (append a nil)
         (plist a))
  ((enable append plist)))

(prove-lemma append-append (rewrite)
 (equal (append (append a b) c)
        (append a (append b c))))

;; These are some definitions from the association list library.

(defn domain (map)
  (if (listp map)
      (if (listp (car map))
          (cons (car (car map)) (domain (cdr map)))
        (domain (cdr map)))
    nil))

(disable domain)

(defn alistp (x)
  (if (listp x)
      (and (listp (car x))
           (alistp (cdr x)))
    (equal x nil)))

(disable alistp)


(defn value (x map)
  (if (listp map)
      (if (and (listp (car map))
               (equal x (caar map)))
          (cdar map)
        (value x (cdr map)))
    0))

;; ---------- reverse and lemmata on reverse ----------

(defn reverse (l)
 (if (nlistp l)
     nil
     (append (reverse (cdr l)) (list (car l)))))

(prove-lemma plistp-reverse (rewrite)
  (plistp (reverse a)))

(prove-lemma reverse-append (rewrite)
  (equal (reverse (append a b))
         (append (reverse b) (reverse a))))

(prove-lemma reverse-reverse (rewrite)
  (implies (plistp l)
           (equal (reverse (reverse l)) l)))

;; This predicate returns true if all elements of the string are in vocab.
```

```
(defn is-string-in (string vocab)
  (if (nlistp string)
      T
      (and (member (car string) vocab)
           (is-string-in (cdr string) vocab))))

; These two functions are needed for generating the tables

(defn position (x l)
  (if (listp l)
      (if (equal x (car l))
          0
          (add1 (position x (cdr l))))
      0))

(defn nth (n l)
  (if (listp l)
      (if (zerop n)
          (car l)
          (nth (sub1 n) (cdr l)))
      0))

(deftheory lists
 (REVERSE-REVERSE REVERSE-APPEND PLISTP-REVERSE REVERSE ALISTP
  DOMAIN APPEND-NIL APPEND-LEFT-ID APPEND-APPEND PLISTP-APPEND
  PLISTP-CONS EQUAL-PLIST PLISTP-NLISTP PLISTP PLIST LENGTH-CONS
  LENGTH-NLISTP EQUAL-LENGTH-0 IS-STRING-IN LENGTH LAST VALUE
  NTH POSITION))

(disable-theory lists)
```

B.5 Grammars

```
;; ---------------------------------------------------------------------------
;; ~/events/grammar.events
;; ---------------------------------------------------------------------------

;; This the implementation of the grammar data type.

(enable-theory sets)
(enable-theory lists)

(add-shell mk-grammar Empty-Grammar is-Grammar
  ((sel-Nonterminals (none-of) zero)
   (sel-Terminals    (none-of) zero)
   (sel-Productions  (none-of) zero)
   (sel-Axiom        (none-of) zero)))

;; Using a shell for mk-prod keeps it from opening up.

(add-shell mk-prod Empty-Prod is-production
  ((sel-label (one-of numberp) Zero)
   (sel-lhs   (none-of)        Zero)
   (sel-rhs   (none-of)        Zero)))

(defn vocab (grammar)
 (union (sel-nonterminals grammar) (sel-terminals grammar)))

;; One needs to be able to access the production labelled label.

(defn prod-nr (prods label)
  (if (nlistp prods)
      nil
      (if (equal (sel-label (car prods)) label)
          (car prods)
          (prod-nr (cdr prods) label))))

;; These are the explicit fresh tokens.

(defn end-of-file () 'ef)
```

```
(defn dot () 'dot)

(defn left-hand-sides (prods)
 (if (nlistp prods)
     nil
     (union (sel-lhs (car prods)) (left-hand-sides (cdr prods)))))

(defn right-hand-sides (prods)
 (if (nlistp prods)
     nil
     (union (sel-rhs (car prods)) (right-hand-sides (cdr prods)))))

(defn all-but-axiom (prods axiom)
 (if (nlistp prods)
      prods
      (if (equal (sel-label (car prods)) axiom)
          (cdr prods)
          (cons (car prods) (all-but-axiom (cdr prods) axiom)))))

(defn no-unused-productions (prods axiom)
    (subsetp (left-hand-sides (all-but-axiom prods axiom))
             (right-hand-sides prods)))

(defn labels (prods)
  (if (nlistp prods)
      nil
      (append (list (sel-label (car prods))) (labels (cdr prods)))))

;; This function finds the label number for a production mk-prod (_,lhs, rhs).

(defn find-label (lhs rhs prods)
 (if (nlistp prods)
     'no-such-label
     (if (and (equal lhs (sel-lhs (car prods)))
              (equal rhs (sel-rhs (car prods))))
         (sel-label (car prods))
         (find-label lhs rhs (cdr prods)))))

(defn is-wf-grammar (grammar)
 (let ((prods (sel-productions grammar))
       (nonts (sel-nonterminals grammar))
       (terms (sel-terminals grammar))
       (axiom (sel-axiom grammar)))
   (let ((vocab (union nonts terms))
         (labs  (labels prods)))
     (and (is-grammar grammar)
          (no-unused-productions prods axiom)
          (equal (card labs)
                 (length prods))
          (member axiom labs)
          (equal (intersection  nonts terms) nil )
          (not (member (end-of-file) vocab))
          (not (member (dot) vocab))
          (subsetp (right-hand-sides prods) vocab)))))

#|
(r-loop)
;; The functions can be tested with the expression grammar.

(setq grammar-expr
 (mk-grammar
  '(e t f s)
  '(plus times open close a)
    (list (mk-prod 0 's '(e))
          (mk-prod 1 'e '(e plus t))
          (mk-prod 2 'e '(t))
          (mk-prod 3 't '(t times f))
          (mk-prod 4 't '(f))
          (mk-prod 5 'f '(open e close))
          (mk-prod 6 'f '(a)))
  0 ))
```

```
(is-wf-grammar grammar)
|#

(deftheory grammar
 (IS-WF-GRAMMAR LABELS NO-UNUSED-PRODUCTIONS ALL-BUT-AXIOM
  RIGHT-HAND-SIDES LEFT-HAND-SIDES DOT END-OF-FILE PROD-NR find-label VOCAB
  MK-PROD MK-GRAMMAR))
(disable-theory grammar)
(disable-theory lists)
(disable-theory sets)
```

B.6 Stacks

```
;; ---------------------------------------------------------------------------
;; ~/events/stack.events
;; ---------------------------------------------------------------------------

;; This file implements a generic data type stack.

(enable-theory lists)

;; I will use a shell for implementing the basic stack. The bottom element is
;; EmptyStack, stackp is the recognizer, push the constructor and
;; top and pop the destructors.

(add-shell push EmptyStack is-stack
     ((top (none-of) zero)
      (pop (one-of is-stack) EmptyStack)))

;; The function to test for bottom element equality

(defn is-empty (s)
  (if (is-stack s)
      (equal s (EmptyStack))
      F))

;; The functions for popping or pushing more than one element at a time.

(defn pop-n (n s)
 (if (not (is-stack s))
     s
     (if (zerop n)
         s
         (pop-n (sub1 n) (pop s)))))

(prove-lemma pop-n-emptystack (rewrite)
 (implies (numberp size)
          (equal (pop-n size (emptystack))
                 (emptystack))))

;; The function top-n returns the top n elements of the stack in reverse order.

(defn top-n (n s)
 (if (or (not (is-stack s))
         (equal s (emptystack)))
     nil
     (if (zerop n)
         nil
         (append (top-n (sub1 n) (pop s))
                 (list (top s))))))

;; This function returns the number of elements on a stack.

(defn stack-length (s)
  (if (or (not (is-stack s))
          (is-empty s))
      0
      (add1 (stack-length (pop s)))))

;; This is a minor lemma to prove that a non-empty stack has non-zero length.
```

```
(prove-lemma not-empty-not-zero (rewrite)
 (implies (and (is-stack stack)
               (not (is-empty stack)))
          (lessp 0 (stack-length stack))))

;; Reading the stack from the bottom, i.e. the top element is last and
;; the bottom element is first.  Note that (not (is-stack s)) was necessary
;; for the proof of termination of this function. And I had to use append
;; and put the second parameter in a list in order to obtain the function I
;; expected to have specified.

(defn from-bottom (s)
   (if (or (is-empty s)
           (not (is-stack s)))
       nil
       (append (from-bottom (pop s)) (list (top s)))))

;; One needs to know that pop is smaller than the original for a measure.

(prove-lemma lessp-pop-stack (rewrite)
  (implies (and (is-stack s)
                (not (is-empty s)))
           (lessp (stack-length (pop s)) (stack-length s))))

;; This theorem is needed for the parser proof.

(prove-lemma append-from-bottom-pop-n (rewrite)
 (implies (is-stack s)
          (equal (append (from-bottom (pop-n n s))
                         (top-n n s))
                 (from-bottom s))))

(prove-lemma plistp-from-bottom (rewrite)
 (implies (is-stack s)
          (plistp (from-bottom s))))

(prove-lemma from-bottom-push (rewrite)
 (implies (is-stack b)
          (equal (from-bottom (push a b))
                 (append (from-bottom b) (list a)))))

(deftheory stacks
 (LESSP-POP-STACK FROM-BOTTOM NOT-EMPTY-NOT-ZERO STACK-LENGTH
  TOP-N POP-N pop-n-emptystack IS-EMPTY PUSH APPEND-FROM-BOTTOM-POP-N
  PLISTP-FROM-BOTTOM FROM-BOTTOM-PUSH))
(disable-theory stacks)
(disable-theory lists)
```

B.7 Tokens

```
;; --------------------------------------------------------------------------
;; ~/events/token.events
;; --------------------------------------------------------------------------

;; This is the representation for a (pre-)token, which is what the scanner is
;; to deliver.

(add-shell mk-token nil tokenp
           ((token-name (none-of) zero)
            (token-value (none-of) zero)))

(defn token-listp (l)
  (if (nlistp l)
      (equal l nil)
      (and (tokenp (car l))
           (token-listp (cdr l)))))

;; This function filters out the token names from a token list.

(defn pick-token-names (l)
```

```
 (if (nlistp l)
     nil
     (if (tokenp (car l))
         (cons (token-name (car l)) (pick-token-names (cdr l)))
         (cons (car l) (pick-token-names (cdr l))))))

(deftheory tokens (MK-TOKEN PICK-TOKEN-NAMES TOKEN-LISTP))
(disable-theory tokens)
```

B.8 Trees

```
;; -----------------------------------------------------------------------------
;; ~/events/tree.events
;; -----------------------------------------------------------------------------

;; These events construct trees and prove theorems about them

(enable-theory lists)
(enable-theory stacks)

;; A tree is a root and a sequence of branches.

(add-shell mk-Tree EmptyTree is-tree
  ((sel-Root     (none-of) Zero)
   (sel-Branches (none-of) Zero)))

;; A tree is a leaf if the branches are empty.

(defn is-leaf (tree)
 (and (is-tree tree)
      (equal (sel-branches tree)
             Nil)))

(defn is-subtree (tag sub Tree)
 (if (equal tag 'tree)
     (if (or (equal (EmptyTree) Tree)
             (not (is-tree Tree)))
         F
         (if (equal sub Tree)
             T
             (is-subtree 'branches sub (sel-branches Tree))))
     ; list of branches
     (if (nlistp Tree)
         F
         (or (is-subtree 'tree sub (car Tree))
             (is-subtree 'branches sub (cdr Tree))))))

(prove-lemma subtree-reflexive (rewrite)
 (implies (and (is-tree tree)
               (not (equal tree (EmptyTree))))
          (is-subtree 'tree tree tree)))

(defn is-leaf-in (node tree)
  (and (is-leaf node)
       (is-subtree 'tree node tree)))

;; The function frontier collects the leaves of one tree into a list.

(defn frontier (tag Item)
 (if (equal tag 'tree)
     (if (or (not (is-tree Item))
             (equal Item (Emptytree)))
         nil
       (if (is-leaf Item)
           (list (sel-root Item))
           (frontier 'branches  (sel-Branches Item))))
     ; Branches
     (if (nlistp Item)
         nil
         (append (frontier 'tree (car Item))
                 (frontier 'branches (cdr Item))))))
```

```
(prove-lemma plistp-frontier (rewrite)
 (plistp (frontier tag anything)))

(prove-lemma leaf-frontier (rewrite)
 (equal (frontier 'tree (mk-tree v nil))
        (list v)))

(prove-lemma member-append (rewrite)
 (equal (member x (append a b))
        (or (member x a)
            (member x b))))

;; This is the key lemma, which needs member-append.

(prove-lemma is-subtree-leaf-is-member-frontier (rewrite)
 (equal (is-subtree tag (mk-tree z nil) tree)
        (member z (frontier tag tree))))

;; This is theorem 1, all leaves in the tree are in the frontier.

(prove-lemma all-leaves-in-frontier (rewrite)
 (implies (and (is-leaf-in subtree tree)
               (is-tree tree))
          (member (sel-root subtree) (frontier 'tree tree))))

;; This is the nearest I come to demonstrating that only leaves
;; are in the frontier.

(prove-lemma only-leaves-in-frontier (rewrite)
 (implies (and (member x (frontier 'tree tree))
               (is-tree tree))
          (is-leaf-in (mk-tree x nil) tree)))

;; I want to show that the leaves are kept in order.

;; I need to define an ordering on the nodes of the tree so that I can
;; determine when a node (or leaf) preceeds another. I will print the
;; entire collection of nodes in preorder (although postorder would do as
;; well) and then prove that the frontier is a subsequence of the preorder.
;; See below for the definition of subsequence.

(defn preorder-print (tag tree)
 (if (equal tag 'tree)
     (if (or (not (is-tree tree))
             (equal tree (Emptytree)))
         nil
       (append (list (sel-root tree))
               (preorder-print 'branches (sel-branches tree))))
   ; branches
   (if (nlistp tree)
       nil
       (append (preorder-print 'tree (car tree))
               (preorder-print 'branches  (cdr tree))))))

;; The theorem is; (subseq (frontier 'tree tree) (preorder-print 'tree tree))

;; x is a subsequence of y if I can remove some elements of y (not
;; necessarily in order) and obtain x. That is, the elements of x are found
;; in the same order in y.

(defn subseq (x y)
  (if (nlistp x)
      t
      (if (nlistp y)
          f
          (or (and (equal (car x) (car y))
                   (subseq (cdr x) (cdr y)))
              (and (not (equal (car x) (car y)))
                   (subseq x (cdr y)))))))

(prove-lemma subseq-cons-lemma (rewrite)
```

```
  (and (implies (subseq x y)
                (subseq (cdr x) y))
       (implies (subseq x (cdr y))
                (subseq x y))))

(prove-lemma subseq-cons-1 (rewrite)
  (implies (subseq (cons a x) y)
           (subseq x y)))

(prove-lemma subseq-cons-2 (rewrite)
  (implies (subseq x y)
           (subseq x (cons b y))))

(prove-lemma subseq-cons-append (rewrite)
            (implies (subseq (cons x z) u)
                     (subseq z (append v u))))

(prove-lemma subseq-append-append (rewrite)
        (implies (and (subseq b u) (subseq a y))
                 (subseq (append a b) (append y u))))

(prove-lemma subseq-frontier-preorder (rewrite)
          (subseq (frontier tag tree)
                  (preorder-print tag tree)))

;; ---------------------------------------------------------------------------
;; Other functions using trees

;; The function roots collects up the roots of all the trees in a sequence,
;; It is often used on a collection of branches.

(defn roots (trees)
  (if (nlistp trees)
      nil
      (if (is-tree (car trees))
          (append (list (sel-Root (car trees)))
                  (roots (cdr trees)))
        (roots (cdr trees)))))

(prove-lemma plistp-roots (rewrite)
 (plistp (roots r)))

;; The function leaves collects up the leaves of a list of trees from the
;; bottom.

(defn leaves (Trees)
  (if (nlistp trees)
      NIL
      (append (frontier 'tree (car Trees))
              (leaves (cdr trees)))))

(prove-lemma plistp-leaves (rewrite)
  (plistp (leaves s))
 ((enable-theory lists)))

(deftheory trees
 (ROOTS SUBSEQ-FRONTIER-PREORDER SUBSEQ-APPEND-APPEND SUBSEQ-CONS-APPEND
        SUBSEQ-CONS-2 SUBSEQ-CONS-1 SUBSEQ-CONS-LEMMA SUBSEQ PREORDER-PRINT
        ONLY-LEAVES-IN-FRONTIER ALL-LEAVES-IN-FRONTIER
        IS-SUBTREE-LEAF-IS-MEMBER-FRONTIER MEMBER-APPEND LEAF-FRONTIER FRONTIER
        IS-LEAF-IN SUBTREE-REFLEXIVE IS-SUBTREE IS-LEAF MK-TREE
        PLISTP-FRONTIER
        LEAVES PLISTP-LEAVES))
(disable-theory trees)
(disable-theory lists)
(disable-theory stacks)
```

B.9 Configurations

```
;; ---------------------------------------------------------------------------
;; ~/events/configuration.events
```

```
;; -------------------------------------------------------------------------

;; These events are the constructors and destructors for configurations.

(enable-theory stacks)
(enable-theory grammar)
(enable-theory lists)
(enable-theory trees)

(add-shell mk-configuration Undefined-Configuration is-configuration
  ((sel-input   (none-of) zero)
   (sel-states  (none-of) zero)
   (sel-symbols (none-of) zero)
   (sel-Trees   (none-of) zero)
   (sel-Parse   (none-of) zero)
   (sel-Deriv   (none-of) zero)
   (sel-Error   (none-of) Zero)))

(defn stack-of-trees (trs)
  (if (not (is-stack trs))
      F
      (if (equal (EmptyStack) trs)
          T
          (and (is-stack trs)
               (is-tree (top trs))
               (stack-of-trees (pop trs)))))
  ((lessp (stack-length trs))))

(defn stack-of-numbers (trs)
  (if (not (is-stack trs))
      F
      (if (equal (EmptyStack) trs)
          T
          (and (is-stack trs)
               (numberp (top trs))
               (stack-of-numbers (pop trs)))))
  ((lessp (stack-length trs))))

;; A well-formed configuration has as its states a stack of numbers, its
;; symbols a stack of litatoms, its trees a stack consisting of individual
;; trees, etc. Probably the derivation is a well-formed derivation and so on.
;; The input can only be a list of terminals so the terminal list is needed.

(defn is-wf-configuration (c terms)
  (and (stack-of-trees (sel-trees c))
        (not (equal (sel-trees c) (EmptyStack)))
        (is-string-in (sel-input c) terms)
        (stack-of-numbers (sel-states c))))

#| This lemma MUST fail to prove!
(prove-lemma sanity-check ()
 (equal (is-wf-configuration c terms)
        F))
|#

;; This function creates the set of productions used in a tree.

(defn get-prods (flag Param)
      (if (equal flag 'tree)
          (if (or (is-leaf Param) ; only non-terminal nodes are collected
                  (not (is-tree Param))
                  (equal Param (EmptyTree)))
              nil
            (union (mk-Prod nil (sel-Root Param) (roots (sel-branches Param)))
                   (get-prods 'branches (sel-branches Param))))
        (if (equal flag 'sequence)
            (if (nlistp Param)
                nil
                (union (get-prods 'tree (car Param))
                       (get-prods 'sequence (cdr Param))))
          nil)))
```

```
;; This function collects the productions for a stack of trees.

(defn nodes (tree-stack terms)
  (if (or (is-empty tree-stack)
          (not (is-stack tree-stack)))
      nil
      (union
        (get-prods 'tree (top tree-stack))
        (nodes (pop tree-stack) terms)))
  ((lessp (stack-length tree-stack))))

(deftheory configurations
 (NODES GET-PRODS IS-WF-CONFIGURATION STACK-OF-NUMBERS STACK-OF-TREES
  MK-CONFIGURATION ))

(disable-theory configurations)
(disable-theory stacks)
(disable-theory grammar)
(disable-theory lists)
(disable-theory trees)
```

B.10 Actions

```
;; ---------------------------------------------------------------------------
;; ~/events/actions.events
;; ---------------------------------------------------------------------------

;; These events describe the parsing actions.
;; A parsing table consists of an action table and a goto table

(defn mk-Tables (Actiontab Gototab)
  (cons Actiontab Gototab))

(defn sel-Actiontab (tables)
  (car tables))

(defn sel-Gototab (tables)
  (cdr tables))

;; An action is a tag, which is either 'shift, 'reduce or 'error, the state to
;; shift to, the label of a reduction, the left hand side of a reduction and
;; the size of the reduction. It is a union type of the different components.

(add-shell mk-Action Empty-Action is-Action
           ((sel-action-tag   (none-of)        zero)
            (sel-state-shift  (one-of numberp) zero)
            (sel-label-reduce (one-of numberp) zero)
            (sel-lhs-reduce   (none-of)        zero)
            (sel-size-reduce  (one-of numberp) zero)))

;; I construct actions with the following functions.

(defn mk-shift-action (state)
  (mk-action 'SHIFT state 0 0 0))

(defn mk-reduce-action (label lhs size)
  (mk-action 'REDUCE 0 label lhs size))

(defn mk-error-action ()
  (mk-action 'ERROR 0 0 0 0))

;; For look-up a selector is needed, which is a symbol and a state.

(defn mk-Selector (State Symbol)
  (list State Symbol))

(defn action-lookup (terminal state actiontab)
  (let ((key (mk-selector state terminal)))
    (cdr (assoc key actiontab))))

(defn goto-lookup (lhs state gototab)
```

```
  (let ((key (mk-selector state lhs)))
    (cdr (assoc key gototab))))
```

B.11 Derivations

```
;; ----------------------------------------------------------------------------
;;  ~/events/derivation.events
;; ----------------------------------------------------------------------------

;; These are the functions defining a derivation.

(enable-theory sets)
(enable-theory lists)
(enable-theory grammar)
(enable-theory tokens)

;; A Derivation is a list of Derivation Steps.
;; A Derivation Step consists of a left part, a prod and a right part.
;; A Derivation Rule is a list of Labels.

;; There will be some lists with tokens and nonterminals mixed. One needs to
;; pick the token-name out for the tokens and make a list.

(add-shell mk-Derivation-Step Undefined-DS is-derivation-step
  ((sel-left-Derivation-Step (none-of) zero)
   (sel-prod-Derivation-Step (none-of) zero)
   (sel-right-Derivation-Step (none-of) zero)))

;; The lhs is an atom, so it must be put in a list, or append will ignore it!

(defn step-source (DS)
 (append (sel-left-Derivation-Step DS)
         (append (list (sel-lhs (sel-prod-Derivation-Step DS)))
                 (sel-right-Derivation-Step DS))))

(defn step-target (DS)
 (append (sel-left-Derivation-Step DS)
         (append (sel-rhs (sel-prod-Derivation-Step DS))
                 (sel-right-Derivation-Step DS))))

(defn source (Derivation)
 (if (nlistp Derivation)
     nil
     (step-source (car Derivation))))

(defn target (Derivation)
 (if (nlistp Derivation)
     nil
     (step-target (last Derivation))))

(defn deriv-rule (Derivation)
 (if (nlistp Derivation)
     nil
     (append
      (sel-label (sel-prod-Derivation-Step (car Derivation)))
      (deriv-rule (cdr Derivation)))))

(defn productions (Derivation)
 (if (nlistp Derivation)
     nil
     (union (list (sel-prod-Derivation-Step (car Derivation)))
            (productions (cdr Derivation)))))

; This function would be a witness for a forall construct.

(defn lockstep (Derivation)
  (if (or (nlistp Derivation)
          (nlistp (cdr Derivation)))
      T
      (and (equal (step-source (cadr Derivation))
                  (step-target (car Derivation)))
```

```
          (lockstep (cdr Derivation)))))

;; This function, too, is for the forall. Since the lockstep checks that
;; source and target are equal, it should suffice to test just the source here.

(defn all-in-vocab (Derivation v)
  (if (nlistp Derivation)
      T
      (and (is-string-in (step-source (car Derivation)) v)
           (all-in-vocab (cdr Derivation) v))))

(defn is-Derivation-in (Derivation Grammar)
   (and (subsetp (productions Derivation) (sel-productions Grammar))
        (all-in-vocab Derivation (append (vocab Grammar) (list (end-of-file))))
        (lockstep Derivation)))

(defn all-rights-terminal (Derivation Terminals)
 (if (nlistp Derivation)
     T
     (and (is-string-in
           (sel-right-Derivation-Step (car Derivation)) Terminals)
          (all-rights-terminal (cdr Derivation) Terminals))))

(defn all-lefts-terminal (Derivation Terminals)
 (if (nlistp Derivation)
     T
     (and (is-string-in (sel-left-Derivation-Step (car Derivation)) Terminals)
          (all-rights-terminal (cdr Derivation) Terminals))))


(defn is-right-derivation-in (Derivation Grammar)
 (and (is-Derivation-in Derivation Grammar)
      (all-rights-terminal
       Derivation
       (append (sel-terminals Grammar) (list (end-of-file))))))

(defn is-left-derivation-in (Derivation Grammar)
 (and (is-Derivation-in Derivation Grammar)
      (all-lefts-terminal
       Derivation
       (append (sel-terminals Grammar) (list (end-of-file))))))
(deftheory derivations
 (IS-LEFT-DERIVATION-IN IS-RIGHT-DERIVATION-IN
  ALL-LEFTS-TERMINAL ALL-RIGHTS-TERMINAL IS-DERIVATION-IN ALL-IN-VOCAB
  LOCKSTEP PRODUCTIONS DERIV-RULE TARGET SOURCE STEP-TARGET STEP-SOURCE
  MK-DERIVATION-STEP ))

(disable-theory derivations)
```

Appendix C

Scanning Appendices

C.1 Character Class Specification for PL_0^R

It did not seem necessary to introduce an explicit shell constructor for character classes, so they are represented as a list of cons-pairs, commonly referred to as a map. The first element of each pair is the literal atom giving the name of the character class, the second one is the list of ASCII-codes for the participating characters. Only explicit listing is available, so for example all the letters have to be explicitly stated instead of giving an interval for now. For PL_0^R we have the following eleven classes. All of the operators have been grouped together in one class.

```
(setq cc
 (list
      ;; #\A #\B #\C ... #\Z #\a #\b #\c ... #\z
      (cons ´le
        (list ´(65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
                81 82 83 84 85 86 87 88 89 90
                97 98 99 100 101 102 103 104 105 106 107 108 109
                110 111 112 113 114 115 116 117 118 119 120 121 122)))
      ;; #\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9
      (cons ´di (list ´(48 49 50 51 52 53 54 55 56 57)))
      (cons ´pe (list ´(46)))      ;; #\.
      (cons ´bl (list ´(32)))      ;; #\Space
      (cons ´co (list ´(58)))      ;; #\:
      (cons ´eq (list ´(61)))      ;; #\=
      (cons ´mi (list ´(45)))      ;; #\-
      (cons ´lt (list ´(60)))      ;; #\<
      (cons ´gt (list ´(62)))      ;; #\>
      (cons ´nl (list ´(10)))      ;; #\Newline
      ;; #\+ #\* #\/ #\\ #\? #\! #\[ #\] #\( #\)
      (cons ´op (list ´(43 42 47 92 63 33 91 93 40 41)))))
```

C.2 DFSA for PL_0^R

This is a collection of setqs for a DFSA that recognizes the token classes for PL_0^R.

```
(setq trans
      (list
(mk-transition ´A ´le ´(B))
(mk-transition ´A ´di ´(C))
(mk-transition ´A ´bl ´(D))
(mk-transition ´A ´co ´(E))
(mk-transition ´A ´eq ´(F))
(mk-transition ´A ´mi ´(G))
(mk-transition ´A ´lt ´(H))
(mk-transition ´A ´gt ´(I))
(mk-transition ´A ´op ´(J))
(mk-transition ´A ´nl ´(K))
(mk-transition ´A ´bf ´(K))
(mk-transition ´A ´nf ´(L))
(mk-transition ´B ´le ´(B))
(mk-transition ´B ´di ´(B))
(mk-transition ´B ´pe ´(B))
(mk-transition ´C ´di ´(C))
```

```
(mk-transition ´D ´bl ´(D))
(mk-transition ´E ´eq ´(M))
(mk-transition ´G ´mi ´(N))
(mk-transition ´H ´eq ´(O))
(mk-transition ´H ´gt ´(P))
(mk-transition ´I ´eq ´(Q))
(mk-transition ´K ´bl ´(R))
(mk-transition ´N ´le ´(N))
(mk-transition ´N ´di ´(N))
(mk-transition ´N ´pe ´(N))
(mk-transition ´N ´bl ´(N))
(mk-transition ´N ´co ´(N))
(mk-transition ´N ´eq ´(N))
(mk-transition ´N ´mi ´(N))
(mk-transition ´N ´lt ´(N))
(mk-transition ´N ´gt ´(N))
(mk-transition ´N ´op ´(N))
(mk-transition ´R ´bl ´(K))))

(setq alphabet ´(li di pe bl co eq mi lt gt op nl bf nf))
(setq states ´(A B C D E F G H I J K L M N O P Q R))
(setq starts ´(A))
(setq finals ´((B . name)    (C . integer) (D . ws) (E . colon)
               (F . eq)      (G . op)      (H . lt) (I . gt)
               (J . op)      (K . indent)  (L . ef) (M . coloneq)
               (N . comment) (O . le)   (P . ne) (Q . ge) ))

(setq nfsa (fsa* alphabet states starts trans finals))
```

C.3 A Mechanically Proven-Correct Scanner

```
;; ---------------------------------------------------------------------------
;; ~/events/scanning.events
;; ---------------------------------------------------------------------------

;; The scanner will split a sequence of bytes into a sequence of tokens by
;; checking all prefixes with an automaton that recognizes tokens,
;; and selecting the last one found, which is the longest such prefix.

;; I use an automaton similar to the one in the fsa proof, but with
;; (state, re) pairs in the set of finals instead of just states. This
;; enables me to determine which regular expression was responsible for
;; the acceptance of the prefix.

(enable-theory tokens)
(enable-theory sets)
(enable-theory lists)

(add-shell fsa* nil fsap*
  ((alphabet (none-of) zero)
   (states   (none-of) zero)
   (starts   (none-of) zero)
   (table    (none-of) zero)
   (finals   (none-of) zero)))

;; A "real" finite state automaton is an NQTHM-automaton with the following
;; properties. Note that finals is an association list. The domain is a
;; subset of the set of states.

(defn fsap (auto)
  (let ((al (alphabet auto))
        (st (states auto))
        (s0 (starts auto))
        (tr (table auto))
        (fi (finals auto)))
    (and (fsap* auto)
         (listp al)
         (listp st)
         (listp s0)
         (alistp fi)
         (subsetp s0 st)
```

```
         (setp st)
         (setp fi)
         (subsetp (domain fi) st))))

(defn mk-transition (state input nexts)
  (cons (cons state input) nexts))

;; These are the selector functions for the transitions in the table.

(defn state (trans) (caar trans))
(defn input (trans) (cdar trans))
(defn nexts (trans) (cdr trans))

(defn transitionp (trans alphabet states)
 (and (member (input trans) alphabet)
      (member (state trans) states)
      (subsetp (nexts trans) states)
      (plistp (nexts trans))))

(defn wf-table (table alphabet states)
  (if (nlistp table)
      (equal table nil)
      (and (transitionp (car table) alphabet states)
           (wf-table (cdr table) alphabet states))))

(defn ndfsap (a)
  (and (fsap a)
       (wf-table (table a) (alphabet a) (states a))))

;; The automaton must be run against a tape.
;; (cons st a) is the state and symbol look-up element.

(defn next-states (table st a)
  (if (nlistp table)
      nil
      (if (equal (cons st a) (caar table))
          (nexts (car table))
          (next-states (cdr table) st a))))

(defn next-states-list (table states a)
  (if (nlistp states)
      nil
      (union
       (next-states table (car states) a)
       (next-states-list table (cdr states) a))))

(defn cc-name (char cc)
 (if (nlistp cc)
     'character-class-not-defined
     (if (member char (cadar cc))
         (caar cc)
         (cc-name char (cdr cc)))))

(defn all-regular-expressions-for-state (state finals)
 (if (nlistp finals)
     nil
     (if (equal state (caar finals))
         (append (list (cdar finals))
                 (all-regular-expressions-for-state state (cdr finals)))
         (all-regular-expressions-for-state state (cdr finals)))))

(defn all-regular-expressions (states finals)
 (if (nlistp states)
     nil
     (append (all-regular-expressions-for-state (car states) finals)
             (all-regular-expressions (cdr states) finals))))

;; The function accept1 returns a list of all regular expressions matching
;; the states in finals. If there are no such expressions, nil is returned.

(defn accept1 (table states finals tape cc)
 (if (nlistp tape)
```

```
     (all-regular-expressions states finals)
     (accept1 table
              (next-states-list table states
                                (cc-name (car tape) cc))
              finals
              (cdr tape)
              cc)))

(defn accepting-regular-expressions (fsa cc tape)
 (accept1 (table fsa) (starts fsa) (finals fsa) tape cc))

(defn accept (fsa cc tape)
 (listp (accepting-regular-expressions fsa cc tape)))

;; -----------------------------------------------------------------------
;; Proof
;; -----------------------------------------------------------------------

;; I need the concept of "what does it mean to be a prefix?" prefixp
;; returns T if a is a prefix of b. Note that nlisps are prefixes of anything.

(defn prefixp (a b)
    (if (nlistp a)
        T
        (if (listp b)
            (if (equal (car a) (car b))
                (prefixp (cdr a) (cdr b))
                F)
            F)))

;; prefixp is transitive.

(prove-lemma prefixp-transitive (rewrite)
               (implies (and (prefixp a b)
                             (prefixp b c))
                        (prefixp a c)))

;; The function consl conses x onto each element in l. I need this in
;; order to construct all the prefixes for a list.

(defn consl (x l)
   (if (listp l)
       (cons (cons x (car l))
             (consl x (cdr l)))
       (list (list x)))); changed this case

;; This is all prefixes without the nil prefix. They happen to be sorted
;; longest first, but I will not use that fact.

(defn all-prefixes (tape)
  (if (nlistp tape)
      nil
      (consl (car tape) (all-prefixes (cdr tape)))))

;; When does a list l contain only prefixes of full?

(defn all-prefixp (l full)
  (if (nlistp l)
      T
      (and (prefixp (car l) full)
           (all-prefixp (cdr l) full))))

(prove-lemma all-prefixp-all-prefixes (rewrite)
 (all-prefixp (all-prefixes tape) tape))

;; When are all the elements of a list accepting tapes for the
;; automaton nfsa/cc?

(defn all-accepting (prefixes nfsa cc)
  (if (nlistp prefixes)
      nil
      (if (accept nfsa cc (car prefixes))
```

```
          (cons (car prefixes) (all-accepting (cdr prefixes) nfsa cc))
          (all-accepting (cdr prefixes) nfsa cc))))

(prove-lemma all-prefixp-all-accepting (rewrite)
 (implies (all-prefixp x y)
          (all-prefixp (all-accepting x nfsa cc) y)))

;; This function expresses the concept of longest element of a list  -  the
;; longest-to-date is kept around until the entire list has been seen. If I
;; find something longer, that is then the longest-to-date.

(defn longest1 (rest longest-to-date)
 (if (nlistp rest)
     longest-to-date
     (if (lessp (length longest-to-date)
                (length (car rest)))
         (longest1 (cdr rest) (car rest))
         (longest1 (cdr rest) longest-to-date))))

(prove-lemma prefixp-longest1 (rewrite)
 (implies (and (all-prefixp rest tape)
               (prefixp longest-to-date tape))
          (prefixp (longest1 rest longest-to-date) tape)))

(defn longest (l)
 (if (nlistp l)
     nil
     (longest1 l (car l))))

(prove-lemma prefixp-longest (rewrite)
 (implies (all-prefixp l tape)
          (prefixp (longest l) tape)))

;; This function delivers the longest of all the accepting prefixes of tape.

(defn lop (nfsa cc tape)
 (longest (all-accepting (all-prefixes tape) nfsa cc )))

;; ---------------------------------------------------------------------------
;; Theorem 1: Prefixp longest accepting prefix
;; ---------------------------------------------------------------------------

(prove-lemma all-prefixp-all-accepting-all-prefixes (rewrite)
 (all-prefixp (all-accepting (all-prefixes tape) nfsa cc) tape))

(prove-lemma prefixp-lop (rewrite)
 (prefixp (lop nfsa cc tape) tape)
  ((use (prefixp-longest (l (all-accepting (all-prefixes tape) nfsa cc))
                         (tape tape)))))

;; ---------------------------------------------------------------------------
;; Theorem 2: longest accepting prefix really accepts
;; ---------------------------------------------------------------------------

(defn accept-all (l nfsa cc)
 (if (nlistp l)
     T
     (and (accept nfsa cc (car l))
          (accept-all (cdr l) nfsa cc))))

(prove-lemma member-accept-all-accepts (rewrite)
 (implies (and (accept-all l nfsa cc)
               (member p l))
          (accept nfsa cc p)))

(prove-lemma accept-all-all-accepting (rewrite)
 (accept-all (all-accepting x nfsa cc) nfsa cc))

(prove-lemma member-longest1 (rewrite)
      (implies (not (equal (longest1 x z) z))
               (member (longest1 x z) x)))
```

```
(prove-lemma member-longest (rewrite)
 (implies (listp l)
          (member (longest l) l)))

;; This lemma never seemed to prove on a SPARC, but on a Pentium-90 it
;; zipped through in less time than you need to drink a cup of coffee!

(prove-lemma helper (rewrite)
 (implies (and (accept-all l nfsa cc)
               (listp l))
          (accept nfsa cc (longest l))))

;; The theorem should have as its hypothesis something like
;; (or (accepts-longest-prefix ...)
;;     (accepts-no-prefix ...)
;; So I restate this to be: if the longest prefix is not nil, then it
;; accepts. If the longest prefix IS nil (for epsilon productions) then we
;; have the problem of deciding their acceptance.

(prove-lemma accepts-lop ()
 (implies (and (listp tape)
               (not (equal (lop nfsa cc tape) nil)))
          (accept nfsa cc (lop nfsa cc tape)))
          ((use (helper (l (all-accepting (all-prefixes tape) nfsa cc))
               (tape tape)))))

;; ------------------------------------------------------------------------
;; Theorem 3: longest accepting prefix is really the longest one
;; ------------------------------------------------------------------------

;; The definition of longest is twofold, it has to be a member and there can
;; be none larger.

(defn none-larger (x l)
 (if (nlistp l)
     T
     (if (lessp (length x) (length (car l)))
         F
         (none-larger x (cdr l)))))

(defn longestp (x l)
  (and (member x l)
       (none-larger x l)))

(prove-lemma not-lessp-length-longest1-other (rewrite)
 (not (lessp (length (longest1 v z))
                     (length z))))

(prove-lemma none-larger-longest1 (rewrite)
 (none-larger (longest1 v z) v))

;; This gets accepted using generalization THREE times!

(prove-lemma accepting-prefix-is-longest ()
 (implies (and (listp tape)
               (not (equal (lop nfsa cc tape) nil)))
          (longestp (lop nfsa cc tape)
                    (all-accepting (all-prefixes tape) nfsa cc))))

;; ------------------------------------------------------------------------
;; Now we have to make something useful out of the prefix: a token
;; ------------------------------------------------------------------------

(defn longest-prefix-token (nfsa cc prefix)
 (mk-token (car (accepting-regular-expressions nfsa cc prefix))
             prefix))

;; Once a token has been made out of the longest prefix, I have to continue.

(defn remove-common-prefix (a b)
  (if (nlistp a)
      b
```

```
      (if (nlistp b)
          nil
          (if (equal (car a) (car b))
              (remove-common-prefix (cdr a) (cdr b))
              nil))))

;; This lemma is needed for the termination argument in split.

(prove-lemma remove-common-prefix-lessp (rewrite)
 (implies (and (prefixp a b)
               (listp a)
               (listp b))
          (equal (lessp (length (remove-common-prefix a b)) (length b))
                 T)))
(disable lop)

;; Now one must split off the longest prefix token. Note that Ie
;; have to exclude epsilon prefixes or I have no termination argument!

(defn split (nfsa cc tape)
 (if (nlistp tape)
     tape
     (let ((prefix (lop nfsa cc tape)))
       (if (equal (length prefix) 0)
           (cons (mk-token 'lexicographic-error tape) nil)
           (cons (longest-prefix-token nfsa cc prefix)
                 (split nfsa cc (remove-common-prefix prefix tape))))))
 ((lessp (length tape))))

;; And the theorem for split is that it splits the tape, i.e. collecting up
;; the token-values gives us the tape again.

(defn collect-values (toklist)
 (if (nlistp toklist)
     toklist
     (if (not (tokenp (car toklist)))
         'not-a-token-list
         (append (token-value (car toklist))
                 (collect-values (cdr toklist))))))

(prove-lemma plistp-remove-common-prefix (rewrite)
 (implies (plistp tape)
          (plistp (remove-common-prefix a tape))))

(prove-lemma collect-values-cons (rewrite)
 (implies (tokenp a)
          (equal (collect-values (cons a b))
                 (append (token-value a) (collect-values b)))))

(prove-lemma append-remove-common-prefix (rewrite)
 (implies (prefixp a b)
          (equal (append a (remove-common-prefix a b))
                 b)))

; The nasty induction structure must be spoon-fed to the prover.

(defn split-splits-hint (nfsa cc tape)
  (if (or (nlistp tape)
          (nlistp (lop nfsa cc tape)))
      T
      (cons (lop nfsa cc tape)
            (split-splits-hint
             nfsa cc (remove-common-prefix (lop nfsa cc tape) tape))))
 ((lessp (length tape))))

(prove-lemma split-splits (rewrite)
 (implies (plistp tape)
          (equal (collect-values (split nfsa cc tape))
                 tape))
  ((do-not-generalize T)
   (induct (split-splits-hint nfsa cc tape ))))
```

```
(deftheory scanning-defs
 (SPLIT-SPLITS-HINT COLLECT-VALUES REMOVE-COMMON-PREFIX LONGEST-PREFIX-TOKEN
  SPLIT LONGESTP NONE-LARGER ACCEPT-ALL LOP LONGEST LONGEST1 ALL-ACCEPTING
  ALL-PREFIXP ALL-PREFIXES CONSL PREFIXP ACCEPT ACCEPTING-REGULAR-EXPRESSIONS
  ACCEPT1 ALL-REGULAR-EXPRESSIONS ALL-REGULAR-EXPRESSIONS-FOR-STATE CC-NAME
  NEXT-STATES-LIST NEXT-STATES NDFSAP WF-TABLE TRANSITIONP NEXTS INPUT STATE
  MK-TRANSITION FSAP FSA* ))

(disable-theory scanning-defs)

(deftheory scanning-proofs
 (SPLIT-SPLITS APPEND-REMOVE-COMMON-PREFIX
  COLLECT-VALUES-CONS PLISTP-REMOVE-COMMON-PREFIX REMOVE-COMMON-PREFIX-LESSP
  ACCEPTING-PREFIX-IS-LONGEST NONE-LARGER-LONGEST1
  NOT-LESSP-LENGTH-LONGEST1-OTHER ACCEPTS-LOP HELPER MEMBER-LONGEST
  MEMBER-LONGEST1 ACCEPT-ALL-ALL-ACCEPTING MEMBER-ACCEPT-ALL-ACCEPTS
  PREFIXP-LOP ALL-PREFIXP-ALL-ACCEPTING-ALL-PREFIXES PREFIXP-LONGEST
  PREFIXP-LONGEST1 ALL-PREFIXP-ALL-ACCEPTING ALL-PREFIXP-ALL-PREFIXES
  PREFIXP-TRANSITIVE ))

(disable-theory scanning-proofs)
```

C.4 Token Transformations

```
;; ---------------------------------------------------------------------------
;;      ~/events/toktrans-1.events
;; ---------------------------------------------------------------------------;
(enable-theory tokens)

;; This is the token transformation function that discards tokens.
;; "discard" is the name of the token transformation function. It takes
;; as parameters the token sequence and a list of token names to be removed.

(defn discard (toks discard-list)
 (if (nlistp toks)
     toks
     (if (member (token-name (car toks)) discard-list)
         (discard (cdr toks) discard-list)
         (cons (car toks)
               (discard (cdr toks) discard-list)))))

;; The first correctness predicate states that any tokens not on the discard
;; list remain unchanged and are in the same order as before the application
;; of discard.

(defn non-discards-undisturbed (toks1 toks2 discard-list)
  (if (nlistp toks1)
      (nlistp toks2)
      (if (not (member (token-name (car toks1)) discard-list))
          (and (equal (car toks1) (car toks2))
               (non-discards-undisturbed (cdr toks1) (cdr toks2) discard-list))
          (non-discards-undisturbed (cdr toks1) toks2 discard-list))))

(prove-lemma discard-does-not-disturb-non-discards (rewrite)
 (non-discards-undisturbed toks (discard toks discard-list) discard-list))

;; The second predicate states that after application of discard, no
;; tokens with a name on the discard list remain.

(defn no-discards-left (toks discard-list)
  (if (nlistp toks)
      T
      (if (member (token-name (car toks)) discard-list)
          F
          (no-discards-left (cdr toks) discard-list))))

;; The main theorem for toktrans-1 is trivially proven.

(prove-lemma toktrans-1-main-theorem (rewrite)
 (no-discards-left (discard toks discard-list) discard-list))

;; ---------------------------------------------------------------------------
```

```
;;   ~/events/toktrans-2.events
;; ------------------------------------------------------------------------

;; This token transformation function looks through the token list for tokens
;; that are in the domain of the replace alist, and replaces them with the
;; range component. This is in contrast to toktrans-3, which has a default
;; value to be used when the value is not found. Perhaps these two could be
;; combined, with the default value being 'error!

(enable-theory tokens)
(enable-theory lists)

(defn make-replace (toks name dom replace-list)
  (if (nlistp toks)
      toks
      (if (and (equal (token-name (car toks)) name)
               (member (car (token-value (car toks))) dom))
          (cons (mk-token (value (car (token-value (car toks))) replace-list)
                          (token-value (car toks)))
                (make-replace (cdr toks) name dom replace-list))
          (cons (car toks)
                (make-replace (cdr toks) name dom replace-list)))))


;; I am using this as an optimization, selecting the domain once for the
;; whole token list instead of constructing it for every token.
;; This is the token transformation function toktrans-2.

(defn replace (toks name replace-list)
 (if (listp replace-list)
     (let ((dom (domain replace-list)))
       (make-replace toks name dom replace-list))
     toks))

;; I need a stepper that will demonstrate that everything else stays the
;; same and that the domain tokens get replaceed.

(defn replace-step (source target name replace-list)
 (if (nlistp source)
     (nlistp target)
      (if (and (equal (token-name (car source)) name)
               (member (car (token-value (car source))) (domain replace-list)))
         (and (equal (car target)
                     (mk-token (value (car (token-value (car source)))
                                      replace-list)
                          (token-value (car source))))
              (replace-step (cdr source) (cdr target) name replace-list))
         (and (equal (car source) (car target))
              (replace-step (cdr source) (cdr target) name replace-list)))))

;; When is replace correct?
;; When all tokens not in the domain of replace-list remain the same
;; and the tokens in the domain are replaced by the values from the alist.

(prove-lemma toktrans-2-main-theorem (rewrite)
 (implies (and (token-listp toks)
               (listp replace-list))
          (replace-step toks (replace toks name replace-list)
                        name replace-list)))

;; ------------------------------------------------------------------------
;;   ~/events/toktrans-3.events
;; ------------------------------------------------------------------------

;; This token transformation function looks through the token list for
;; tokens with  token-name equal to name. The value is looked up in a table,
;; as in toktrans-2. If there is no entry in the table, a default value
;; is substituted.

(enable-theory tokens)

(defn make-key-words (toks name dom key-words-list default)
  (if (nlistp toks)
```

```
      toks
      (if (equal (token-name (car toks)) name)
          (if (member (token-value (car toks)) dom)
              (cons (mk-token (value (token-value (car toks))
                                     key-words-list)
                              (token-value (car toks)))
                    (make-key-words (cdr toks)
                                    name dom key-words-list default))
              (cons (mk-token default
                              (token-value (car toks)))
                    (make-key-words (cdr toks)
                                    name dom key-words-list default)))
          (cons (car toks)
                (make-key-words (cdr toks)
                                name dom key-words-list default)))))

(defn determine-key-words (toks name key-words-list default)
 (if (listp key-words-list)
     (let ((dom (domain key-words-list)))
       (make-key-words toks name dom key-words-list default))
     toks))

(defn key-words-step (source target name key-words-list default)
 (if (nlistp source)
     (nlistp target)
      (if (equal (token-name (car source)) name)
          (if (member (token-value (car source)) (domain key-words-list))
              (and (equal (car target)
                          (mk-token (value (token-value (car source))
                                           key-words-list)
                                    (token-value (car source))))
                   (key-words-step (cdr source) (cdr target)
                                   name key-words-list default))
              (and (equal (car target)
                          (mk-token default
                                    (token-value (car target))))
                   (key-words-step (cdr source) (cdr target)
                                   name key-words-list default)))
          (and (equal (car source) (car target))
               (key-words-step (cdr source) (cdr target)
                               name key-words-list default)))))

(prove-lemma toktrans-3-main-theorem (rewrite)
 (implies (and (token-listp toks)
               (listp key-words-list))
          (key-words-step
               toks
               (determine-key-words toks name key-words-list default)
               name key-words-list default)))

;; ---------------------------------------------------------------------------
;; ~/events/toktrans-4.events
;; ---------------------------------------------------------------------------

(enable-theory tokens)
(enable-theory lists)

;; This is a token transformation function for converting lists of digit
;; characters which are contained in token values to decimal integers. The
;; function is called integer-convert.
;; It only works on the INTEGER tokens, and leaves all other tokens intact.
;; It is modelled on Bill Bevier's positional number etude.
;; The digits must first be converted from ASCII values to digits.

(defn ascii-zero () 48)
(defn ascii-nine () 57)

(defn digit-zero () 0)
(defn digit-nine () 9)

(defn base () 10)

;; This is the predicate for finding the tokens that toktrans-4 will work on.
```

```
(defn is-integer-token (tok)
 (equal (token-name tok) 'INTEGER))

;; This function checks that the list consists of digits less than b.

(defn just-digits-less-than-b (l b)
 (if (listp l)
     (and (numberp (car l))
          (lessp (car l) b) ; Each digit must be less than the base
          (just-digits-less-than-b (cdr l) b))
     (equal l nil)))

;; This is the Horner method for computing a natural number from
;; a positional number representation l and a base b.
;; This means the list l must be in the reverse digit form - but that
;; saves worrying about exponents and such.

(defn pn-to-nat (l b)
  (if (listp l)
      (plus (car l)
            (times (pn-to-nat (cdr l) b) b))
      (digit-zero)))

;; This is the conversion function that returns the digit for an ASCII code.
;; This function coerces all non-digital digits to 0,

(defn ascii-to-digit (ascii)
 (if (or (lessp ascii (ascii-zero))
         (lessp (ascii-nine) ascii))
     (digit-zero)
     (difference ascii (ascii-zero))))

(defn valid-ascii-digit-p (ascii)
 (and (numberp ascii)
      (not (or (lessp ascii (ascii-zero))
               (lessp (ascii-nine) ascii)))))

(defn ascii-to-digits (l)
 (if (nlistp l)
     l
     (cons (ascii-to-digit (car l))
           (ascii-to-digits (cdr l)))))

(prove-lemma plistp-ascii-to-digits (rewrite)
 (equal (plistp (ascii-to-digits w))
        (plistp w)))

;; This is the token transformation function toktrans-4.

(defn integer-convert (toks)
 (if (nlistp toks)
     toks
     (if (is-integer-token (car toks))
         (let ((value (ascii-to-digits (token-value (car toks)))))
           (if (just-digits-less-than-b value (base))
               (cons (mk-token (token-name (car toks))
                               (pn-to-nat (reverse value) (base)))
                     (integer-convert (cdr toks)))
               'token-error))
         (cons (car toks) (integer-convert (cdr toks))))))

;; This is the retrieve function for one number, it converts a natural
;; number to a positional number with digits of base b.

(defn nat-to-pn (n b)
 (if (lessp 1 b)
     (if (zerop n)
         nil
         (cons (remainder n b)
               (nat-to-pn (quotient n b) b )))
     nil))
```

```
;; There have to be inverse functions for ascii-to-digits as well.

(defn digit-to-ascii (digit)
 (plus digit (ascii-zero))))

(defn digits-to-ascii (digits)
 (if (nlistp digits)
     digits
     (cons (digit-to-ascii (car digits))
           (digits-to-ascii (cdr digits)))))

;; When does a list consist of valid ascii digits?

(defn valid-ascii-digits-p (l)
 (if (nlistp l)
     T
     (and (valid-ascii-digit-p (car l))
          (valid-ascii-digits-p (cdr l)))))

(prove-lemma d-a-inverse-1 (rewrite)
 (implies (and (listp value)
               (valid-ascii-digits-p value))
          (equal (digits-to-ascii (ascii-to-digits value))
                 (cons (digit-to-ascii (ascii-to-digit (car value)))
                       (digits-to-ascii (ascii-to-digits (cdr value)))))))

(prove-lemma d-a-inverse-2 (rewrite)
 (implies (and (listp value)
               (valid-ascii-digits-p value))
          (equal (digits-to-ascii (ascii-to-digits value))
                 value))
 ((induct (length value))))

;; This is the retrieve function for a list. Note that I have to reverse
;; the result before constructing the token.

(defn convert-back (toks)
 (if (nlistp toks)
     toks
     (if (is-integer-token (car toks))
           (cons (mk-token
                  (token-name (car toks))
                  (reverse (digits-to-ascii
                            (nat-to-pn (token-value (car toks))
                                       (base)))))
                 (convert-back (cdr toks)))
         (cons (car toks) (convert-back (cdr toks))))))


;; This lemma is to show that this direction is an inverse.

(prove-lemma inverse1 (rewrite)
 (implies (and (lessp 1 b)
               (numberp n)
               (numberp b))
          (equal (pn-to-nat (nat-to-pn n b) b)
                 n)))

;; One has to know if the positional notation lists are well-formed.
;; Leading zeros are not acceptable.

(defn lastdigit (l)
     (if (listp l)
         (if (not (equal (cdr l) nil))
             (lastdigit (cdr l))
             (car l))
         F))

(defn no-leading-zeros (l)
     (not (equal (lastdigit l) (digit-zero))))
```

```
(defn well-formed-pn (l b)
     (and (lessp 1 b)
              (plistp l)
              (numberp b)
              (no-leading-zeros l)
              (just-digits-less-than-b l b)))

;; This is the interesting inverse function.

(prove-lemma inverse2 (rewrite)
 (implies (well-formed-pn l b)
          (equal (nat-to-pn (pn-to-nat l b) b)
                 l)))

;; These are the lemmata needed for the proof of the main theorem.

;; If x is a just-digits-less-than-b, reversing it preserves
;; just-digits-less-than-b-ness

(prove-lemma toktrans-4-help1 (rewrite)
  (implies (just-digits-less-than-b x b)
          (just-digits-less-than-b (reverse x) b)))

(prove-lemma toktrans-4-help2 (Rewrite)
 (implies (and (plistp x)
               (just-digits-less-than-b (reverse x) b))
          (just-digits-less-than-b x b))
 ((use (toktrans-4-help1 (x (reverse x)) (b b)))))

;; I need to know that all the integer token values are well formed.

(defn integer-tokens-well-formed (toks)
 (if (nlistp toks)
     T
     (if (is-integer-token (car toks))
         (and (no-leading-zeros
               (reverse (ascii-to-digits (token-value (car toks)))))
              (valid-ascii-digits-p (token-value (car toks)))
              (just-digits-less-than-b
               (reverse (ascii-to-digits (token-value (car toks)))) (base))
              (plistp (ascii-to-digits (token-value (car toks))))
              (integer-tokens-well-formed (cdr toks)))
         (integer-tokens-well-formed (cdr toks)))))

(prove-lemma toktrans-4-help3 (rewrite)
 (implies (and (not (just-digits-less-than-b
                     (ascii-to-digits (token-value (car z))) (base)))
               (is-integer-token (car z))
               (listp z))
          (not (integer-tokens-well-formed z))))

(prove-lemma ascii-to-digits-append (rewrite)
 (equal (ascii-to-digits (append a b))
        (append (ascii-to-digits a) (ascii-to-digits b))))

(prove-lemma reverse-ascii-to-digits (rewrite)
 (implies (valid-ascii-digits-p l)
          (equal (reverse (ascii-to-digits l))
                 (ascii-to-digits (reverse l)))))

(prove-lemma valid-ascii-digits-p-append (rewrite)
 (equal (valid-ascii-digits-p (append a b))
        (and (valid-ascii-digits-p a) (valid-ascii-digits-p b))))

(prove-lemma valid-ascii-digits-p-reverse (rewrite)
 (equal (valid-ascii-digits-p (reverse w))
        (valid-ascii-digits-p w)))

(prove-lemma reverse-d-a-reverse (rewrite)
 (implies (and (not (equal (lastdigit (ascii-to-digits (reverse w))) 0))
                    (valid-ascii-digits-p w)
                    (just-digits-less-than-b (ascii-to-digits (reverse w))
```

```
                                        10)
               (plistp w))
          (equal (reverse (digits-to-ascii (ascii-to-digits (reverse w))))
                 w))
 ((use (d-a-inverse-2 (value (reverse w)))
       (reverse-reverse (l w)))))

; -----------------------------------------------------
;      Main theorem for toktrans-4
; -----------------------------------------------------

(prove-lemma toktrans-4-main-theorem (rewrite)
 (implies (and (token-listp toks)
               (integer-tokens-well-formed toks)
               (listp toks))
          (equal (convert-back (integer-convert toks))
                 toks))
 ((do-not-generalize T)))

;; ---------------------------------------------------------------------------
;;   ~/events/toktrans-5.events
;; ---------------------------------------------------------------------------

(enable-theory tokens)

;; This token transformation function finds and removes continuations from a
;; token list.
;; From the occam Reference Manual:
;; "A long statement may be broken immediately after one of the following:
;; an operator, a comma, a semi-colon, an assignment, or the keywords
;; IS, FROM or FOR. A statement can be broken over several lines, providing the
;; continuation is indentated at least as much as the first line of the
;; statement." In this token transformation function, I remove any
;; indentation that immediately follows the elements from the list
;; that are contained in PLOR.

(defn is-kw-indentation (x)
  (equal (token-name x) 'indent))

(defn is-indentation (x)
  (and (is-kw-indentation x)
       (numberp (token-value x))))

(prove-lemma tokenp-is-kw-indentation (rewrite)
 (implies (is-kw-indentation x)
          (tokenp x)))

;; Since empty lines disturb the proof as well, they are removed.

(defn remove-empty-lines (l)
  (if (nlistp l)
      l
      (if (and (is-kw-indentation (car l))
               (or (nlistp (cdr l))
                   (is-kw-indentation (cadr l))))
          (remove-empty-lines (cdr l))
          (cons (car l) (remove-empty-lines (cdr l))))))

(defn no-empty-lines (l)
 (if (nlistp l)
     t
     (if (and (is-kw-indentation (car l))
              (or (nlistp (cdr l))
                  (is-kw-indentation (cadr l))))
         f
         (no-empty-lines (cdr l)))))

(prove-lemma main-theorem-toktrans-5a (rewrite)
  (no-empty-lines (remove-empty-lines l)))

(prove-lemma no-empty-lines-meaning (rewrite)
 (implies (and (no-empty-lines toks)
               (is-kw-indentation (car toks)))
```

```
          (not (is-kw-indentation (cadr toks)))))

;; This function removes the continuations.

(defn discontinue (toks continue-list)
 (if (nlistp toks)
     toks
     (if (member (token-name (car toks)) continue-list)
         (if (is-kw-indentation (cadr toks))
             (cons (car toks) (discontinue (cddr toks) continue-list))
             (cons (car toks) (discontinue (cdr toks) continue-list)))
         (cons (car toks) (discontinue (cdr toks) continue-list)))))

(defn no-continuations-p (toks continue-list)
  (if (nlistp toks)
      T
      (if (member (token-name (car toks)) continue-list)
          (if (is-kw-indentation (cadr toks))
              F
              (no-continuations-p (cdr toks) continue-list))
          (no-continuations-p (cdr toks) continue-list))))

(prove-lemma discontinue-car (rewrite)
 (implies (listp toks)
          (equal (car (discontinue toks list))
                 (car toks))))

;; The main theorem for toktrans-5b insists on no empty lines.

(prove-lemma main-theorem-toktrans-5b (rewrite)
 (implies (no-empty-lines toks)
          (no-continuations-p (discontinue toks continue-list) continue-list))
          ((disable is-kw-indentation)))

(defn toktrans-5 (toks continue-list)
 (remove-empty-lines
   (discontinue toks continue-list)))

;; -------------------------------------------------------------------------
;;     ~/events/toktrans-6.events
;; -------------------------------------------------------------------------

(enable-theory tokens)
(enable-theory lists)

;; This token transformation function prepares for the indentation work to be
;; done in toktrans-7. It looks for the indentation tokens and replaces the
;; value, which is a carriage return followed by 0 or more blanks, by the
;; number of blanks found.
;; The halving of the indentation (2 blanks = 1 level) is done here as well.
;; The function toktrans-6a removes the CR and puts in the number of blanks,
;; the function toktrans-6b halves.

;; ------------ replace chars by length --------------
;;              toktrans 6a

(defn prepare-indentations (toks)
 (if (nlistp toks)
     toks
     (if (is-kw-indentation (car toks))
         (cons
          (mk-token
           (token-name (car toks))
           (sub1 (length (token-value (car toks)))))
          (prepare-indentations (cdr toks)))
         (cons (car toks) (prepare-indentations (cdr toks))))))

(defn ok-indentation-value (l1 l2)
 (if (nlistp l1)
     (nlistp l2)
     (and (if (is-kw-indentation (car l1))
              (and (equal (token-name (car l1))
                          (token-name (car l2)))
```

```
                   (equal (token-value (car l2))
                          (sub1 (length (token-value (car l1))))))
              (equal (car l1) (car l2)))
          (ok-indentation-value (cdr l1) (cdr l2)))))

(prove-lemma toktrans-6a-main-theorem (rewrite)
 (implies (and (token-listp toks)
               (listp toks))
          (ok-indentation-value toks (prepare-indentations toks))))

;; ------------ divide length by 2 --------------
;;              toktrans6b

;; The function half returns half of a number (rounded down).

(defn half (n) (quotient n 2))

;; The function halve is applied to a list, not a token!

(defn halve (l)
  (if (nlistp l)
      l
      (if (is-indentation (car l))
          (cons (mk-token (token-name (car l))
                          (half (token-value (car l))))
                (halve (cdr l)))
          (cons (car l)
                (halve (cdr l))))))

(defn indent-positions-preserved (l1 l2)
  (if (nlistp l1)
      (equal l1 l2)
      (if (is-indentation (car l1))
          (and (is-indentation (car l2))
               (indent-positions-preserved (cdr l1) (cdr l2)))
          (and (equal (car l1) (car l2))
               (indent-positions-preserved (cdr l1) (cdr l2))))))

(prove-lemma indent-positions-preserved-halve (rewrite)
  (indent-positions-preserved l (halve l)))

(defn collect-indents (l)
  (if (nlistp l)
      nil
      (if (is-indentation (car l))
          (cons (fix (token-value (car l)))
                (collect-indents (cdr l)))
          (collect-indents (cdr l)))))

(prove-lemma collect-indents-nlistp (rewrite)
 (implies (nlistp l)
          (equal (collect-indents l) nil)))

(defn halved-listp (l1 l2)
  (if (nlistp l1)
      (nlistp l2)
      (and (equal (car l2) (half (car l1)))
           (halved-listp (cdr l1) (cdr l2)))))

(prove-lemma indents-halved (rewrite)
  (halved-listp (collect-indents l)
                (collect-indents (halve l))))

;; ----- combine both passes -----

(defn toktrans-6 (toks)
 (halve (prepare-indentations toks)))

(defn ok-toktrans-6 (l1 l2)
 (if (nlistp l1)
     (nlistp l2)
     (and (if (is-kw-indentation (car l1))
```

```
              (and (equal (token-name (car l1))
                          (token-name (car l2)))
                   (equal (token-value (car l2))
                          (half (sub1 (length (token-value (car l1)))) )))
              (equal (car l1) (car l2)))
          (ok-toktrans-6 (cdr l1) (cdr l2)))))

(prove-lemma main-theorem-toktrans-6 (rewrite)
  (ok-toktrans-6 toks (toktrans-6 toks)))

(disable-theory tokens)
(disable-theory lists)

;; ---------------------------------------------------------------------------
;; ~/events/toktrans-7.events
;; ---------------------------------------------------------------------------

(enable-theory tokens)
(enable-theory lists)

;; These definitions and theorems are from an old lists library.

(defn firstn (n l)
  (if (not (listp l))
      nil
      (if (zerop n)
          nil
          (cons (car l) (firstn (sub1 n) (cdr l))))))

(defn nthcdr (n l)
  (if (not (listp l))
      l
      (if (zerop n)
          l
          (nthcdr (sub1 n) (cdr l)))))

(prove-lemma listp-append (rewrite)
  (equal (listp (append a b))
         (or (listp a)
             (listp b))))

(prove-lemma nthcdr-append (rewrite)
  (equal (nthcdr n (append a b))
         (if (leq n (length a))
             (append (nthcdr n a) b)
             (nthcdr (difference n (length a)) b))))

(prove-lemma append-firstn-nthcdr (rewrite)
  (equal (append (firstn i l) (nthcdr i l))
         l))

;; The specification function diff-cycle states the relationship between
;; absolute and relative indentations using a proper difference function.

(defn pdiff (i j)
  (if (lessp (fix i) (fix j))
      (minus (difference j i))
      (if (equal (fix i) (fix j))
          0
          (difference i j))))

(defn diff-cycle1 (l orig prev)
  (if (nlistp l)
      (list (pdiff orig prev))
      (cons (pdiff (car l) prev)
            (diff-cycle1 (cdr l) orig (car l)))))

(prove-lemma diff-cycle1-nlistp (rewrite)
             (implies (nlistp l)
                      (equal (diff-cycle1 l orig prev)
                             (list (pdiff orig prev)))))

(prove-lemma diff-cycle1-listp (rewrite)
```

```
             (implies (listp l)
                      (equal (diff-cycle1 l orig prev)
                             (cons (pdiff (car l) prev)
                                   (diff-cycle1 (cdr l) orig (car l))))))

(defn diff-cycle (l old)
  (diff-cycle1 l old old))


; First determine the magnitude and sign of the indentation. Use a
; fresh token name, ´relative, for this kind of token.
; This means that the resulting list remains a token-listp, which will
; go easy on the induction proof.

(defn is-relative (tok)
  (and (tokenp tok)
       (equal (token-name tok) ´relative)
       (or (numberp (token-value tok))
           (and (negativep (token-value tok))
                (not (equal (negative-guts (token-value tok)) 0))))))

(defn emit-relative (i j)
  (if (lessp (fix i) (fix j))
      (mk-token ´relative (minus (difference j i)))
      (if (equal (fix i) (fix j))
          (mk-token ´relative 0)
          (mk-token ´relative (difference i j)))))

;; Unfortunately, emit1 has to watch out for indentations which are not.
;; well formed. I will ignore such indentations.

(defn emit1 (l orig prev)
  (if (nlistp l)
      (list (emit-relative orig prev))
      (if (is-kw-indentation (car l))
          (if (is-indentation (car l))
              (cons (emit-relative (token-value (car l)) prev)
                    (emit1 (cdr l) orig (token-value (car l))))
              ; should be an error
              (emit1 (cdr l) orig prev))
          (cons (car l) (emit1 (cdr l) orig prev)))))

(defn emit (l old)
  (emit1 l old old))

;; This is the meaning function for the relative indentations.

(defn relative-meaning (l)
  (if (nlistp l)
      nil
      (if (is-relative (car l))
          (cons (token-value (car l))
                (relative-meaning (cdr l)))
          (relative-meaning (cdr l)))))

;; There are no ´relative tokens left in the list.

(defn relative-free (l)
  (if (nlistp l)
      T
      (if (tokenp (car l))
          (and (not (equal (token-name (car l)) ´relative))
               (relative-free (cdr l)))
          F)))

(prove-lemma diff-cycle1-not-numberp (rewrite)
             (implies  (not (numberp v))
                       (equal (diff-cycle1 z orig v)
                              (diff-cycle1 z orig 0))))

(prove-lemma emit1-theorem (rewrite)
             (implies (and (relative-free l)
```

```
                          (token-listp l))
                     (equal (relative-meaning (emit1 l orig prev))
                            (diff-cycle1 (collect-indents l) orig prev))))

(prove-lemma emit-theorem (rewrite)
             (implies (and (relative-free l)
                           (token-listp l))
                      (equal (relative-meaning (emit l old))
                             (diff-cycle (collect-indents l) old)))
             ((do-not-induct T)))

;; This makes a list which contains n copies of v.

(defn my-make-list (v n)
  (if (or (zerop n)
          (not (numberp n)))
      nil
      (cons v (my-make-list v (sub1 n)))))

(prove-lemma length-my-make-list (rewrite)
             (equal (length (my-make-list v n))
                    (fix n)))

(prove-lemma not-numberp-my-make-list (rewrite)
             (implies (not (numberp n))
                      (equal (my-make-list x n) nil)))

(prove-lemma listp-my-make-list (rewrite)
             (implies (lessp 0 (fix n))
                      (listp (my-make-list x n))))

(prove-lemma my-make-list-zero (rewrite)
             (implies (equal n 0)
                      (equal (my-make-list x n) nil)))

(prove-lemma plist-my-make-list (rewrite)
             (equal (plist (my-make-list v n))
                    (my-make-list v n)))

;; Now all I have to do is go through and expand the ´relative tokens
;; to the ni, si, and bi stuff the grammar needs.

(defn ni-si-bis (n)
  (if (equal n 0)
      (my-make-list (mk-token ´si nil) 1)
      (if (lessp 0 n)
          (my-make-list (mk-token ´ni nil) n)
          (my-make-list (mk-token ´bi nil) (negative-guts n)))))

(defn relative-to-ni-si-bi (toks)
  (if (nlistp toks)
      toks
      (if (is-relative (car toks))
          (append (ni-si-bis (token-value (car toks)))
                  (relative-to-ni-si-bi (cdr toks)))
          (cons (car toks) (relative-to-ni-si-bi (cdr toks))))))

;; The specification is more complex than that of the implementation. The
;; function matches cdrs down the toks, seeing if it matches the
;; relative value.

(defn matches (n toks)
  (if (nlistp toks)
      F
      (if (equal n 0)
          (equal (car toks) (mk-token ´si nil))
          (if (lessp 0 n)
              (equal (firstn n toks) (my-make-list (mk-token ´ni nil) n))
              (equal (firstn (negative-guts n) toks)
                     (my-make-list (mk-token ´bi nil) (negative-guts n)))))))

(defn how-much (n)
```

```
  (if (equal n 0)
      1
      (if (lessp 0 n)
          (fix n)
          (negative-guts n))))

(defn relative-conversion-ok (pre post)
  (if (nlistp pre)
      ;; There should be just a number of bi tokens on the end.
      (if (nlistp post)
          T
          (and (equal (token-name (car post)) 'bi)
               (relative-conversion-ok pre (cdr post))))
      (if (is-relative (car pre))
          (and (matches (token-value (car pre)) post)
               (relative-conversion-ok
                (cdr pre)
                (nthcdr (how-much (token-value (car pre))) post)))
          (and (equal (car pre) (car post))
               (relative-conversion-ok (cdr pre) (cdr post)))))
  ((ord-lessp (cons (add1 (length pre)) (length post)))))

;; If it has the name 'relative, it had better be well formed.

(defn all-relative-tokens-good (toks)
  (if (nlistp toks)
      T
      (if (equal (token-name (car toks)) 'relative)
          (and (is-relative (car toks))
               (all-relative-tokens-good (cdr toks)))
          (all-relative-tokens-good (cdr toks)))))

(prove-lemma firstn-my-make-list (rewrite)
 (implies (numberp x)
          (equal (firstn x (my-make-list l x))
                 (my-make-list l  x))))

(prove-lemma nthcdr-my-make-list (rewrite)
 (implies (numberp x)
          (equal (nthcdr x (my-make-list l x))
                 nil)))

(prove-lemma firstn-append-my-make-list (rewrite)
 (implies (numberp x)
          (equal (firstn x (append (my-make-list z x) foo))
                 (my-make-list z x))))


(prove-lemma relative-theorem (rewrite)
 (implies (and (token-listp toks)
               (all-relative-tokens-good toks))
          (relative-conversion-ok toks
                                  (relative-to-ni-si-bi toks)))
 ((do-not-generalize t)))


;; Now I try proving some glue so I can feel good about combining the
;; passes. If toks is a token list and free of relative tokens, emit1 will
;; have the properties so that the relative conversion will go okay.

(prove-lemma glue1 (rewrite)
             (implies (and (token-listp toks)
                           (relative-free toks))
                      (all-relative-tokens-good (emit1 toks n m))))

(prove-lemma glue2 (rewrite)
 (implies (token-listp toks)
          (token-listp (emit1 toks n m))))

;; This is the indentator function.

(defn indentator (l)
```

```
 (relative-to-ni-si-bi (emit1 l 0 0)))

;; The result is free of indentation tokens

(defn indent-free (l)
 (if (nlistp l)
     (not (is-kw-indentation l))
     (if (is-kw-indentation (car l))
         F
         (indent-free (cdr l)))))

(prove-lemma indent-free-cons (rewrite)
 (implies (and (indent-free a)
               (not (is-kw-indentation b)))
          (indent-free (cons b a))))

(prove-lemma indent-free-append (rewrite)
 (implies (and (indent-free a)
               (indent-free b))
          (indent-free (append a b)))
 ((disable is-kw-indentation)))

(prove-lemma indent-free-my-make-list (rewrite)
 (implies (indent-free a)
          (indent-free (my-make-list a n)))))

(prove-lemma indent-free-realtive-to-ni-si-bi (rewrite)
 (implies (and (token-listp x)
               (indent-free x))
          (indent-free (relative-to-ni-si-bi x))))


(prove-lemma non-wf-indentations-removed (rewrite)
 (implies (and (not (nlistp z))
               (is-kw-indentation (car z))
               (not (is-indentation (car z)))
               (numberp n))
         (equal (emit1 z m n)
                (emit1 (cdr z) m n))))

(prove-lemma indent-free-emit1 (rewrite)
 (implies (numberp n)
          (indent-free (emit1 z 0 n))))

(prove-lemma indent-free-indentator (rewrite)
 (implies (token-listp l)
          (indent-free (indentator l))))

(deftheory toktrans
 (INDENT-FREE INDENTATOR ALL-RELATIVE-TOKENS-GOOD
  RELATIVE-CONVERSION-OK HOW-MUCH MATCHES RELATIVE-TO-NI-SI-BI NI-SI-BIS
  MY-MAKE-LIST RELATIVE-FREE RELATIVE-MEANING EMIT EMIT1 EMIT-RELATIVE
  IS-RELATIVE DIFF-CYCLE DIFF-CYCLE1 PDIFF NTHCDR FIRSTN OK-TOKTRANS-6
  TOKTRANS-6 HALVED-LISTP COLLECT-INDENTS INDENT-POSITIONS-PRESERVED HALVE
  HALF OK-INDENTATION-VALUE PREPARE-INDENTATIONS TOKTRANS-5 NO-CONTINUATIONS-P
  DISCONTINUE NO-EMPTY-LINES REMOVE-EMPTY-LINES IS-INDENTATION
  IS-KW-INDENTATION INTEGER-TOKENS-WELL-FORMED WELL-FORMED-PN NO-LEADING-ZEROS
  LASTDIGIT CONVERT-BACK VALID-ASCII-DIGITS-P DIGITS-TO-ASCII DIGIT-TO-ASCII
  NAT-TO-PN INTEGER-CONVERT ASCII-TO-DIGITS VALID-ASCII-DIGIT-P ASCII-TO-DIGIT
  PN-TO-NAT JUST-DIGITS-LESS-THAN-B IS-INTEGER-TOKEN BASE DIGIT-NINE
  DIGIT-ZERO ASCII-NINE ASCII-ZERO KEY-WORDS-STEP DETERMINE-KEY-WORDS
  MAKE-KEY-WORDS REPLACE-STEP REPLACE MAKE-REPLACE TOKEN-LISTP VALUE
  NO-DISCARDS-LEFT NON-DISCARDS-UNDISTURBED DISCARD ))

(disable-theory toktrans)

(deftheory toktrans-proofs
 (INDENT-FREE-INDENTATOR INDENT-FREE-EMIT1 NON-WF-INDENTATIONS-REMOVED
  INDENT-FREE-REALTIVE-TO-NI-SI-BI INDENT-FREE-MY-MAKE-LIST INDENT-FREE-APPEND
  INDENT-FREE-CONS
  GLUE2 GLUE1 RELATIVE-THEOREM
  NTHCDR-MY-MAKE-LIST FIRSTN-MY-MAKE-LIST PLIST-MY-MAKE-LIST MY-MAKE-LIST-ZERO
```

```
  LISTP-MY-MAKE-LIST NOT-NUMBERP-MY-MAKE-LIST LENGTH-MY-MAKE-LIST EMIT-THEOREM
  EMIT1-THEOREM DIFF-CYCLE1-NOT-NUMBERP DIFF-CYCLE1-LISTP DIFF-CYCLE1-NLISTP
  APPEND-FIRSTN-NTHCDR NTHCDR-APPEND LISTP-APPEND
  MAIN-THEOREM-TOKTRANS-6 INDENTS-HALVED COLLECT-INDENTS-NLISTP
  INDENT-POSITIONS-PRESERVED-HALVE TOKTRANS-6A-MAIN-THEOREM
  MAIN-THEOREM-TOKTRANS-5B DISCONTINUE-CAR NO-EMPTY-LINES-MEANING
  MAIN-THEOREM-TOKTRANS-5A TOKENP-IS-KW-INDENTATION TOKTRANS-4-MAIN-THEOREM
  VALID-ASCII-DIGITS-P-REVERSE VALID-ASCII-DIGITS-P-APPEND
  REVERSE-ASCII-TO-DIGITS ASCII-TO-DIGITS-APPEND TOKTRANS-4-HELP3
  TOKTRANS-4-HELP2 TOKTRANS-4-HELP1 INVERSE2 INVERSE1 D-A-INVERSE-2
  PLISTP-ASCII-TO-DIGITS REMAINDER-TIMES1-INSTANCE QUOTIENT-TIMES-INSTANCE
  QUOTIENT-PLUS REMAINDER-PLUS QUOTIENT-LESSP-ARG1 COMMUTATIVITY-OF-TIMES
  LESSP-QUOTIENT PLUS-REMAINDER-TIMES-QUOTIENT TIMES-ADD1 EQUAL-TIMES-0
  TIMES-ZERO PLUS-ZERO-ARG2 EQUAL-PLUS-0 COMMUTATIVITY-OF-PLUS
  FIRSTN-APPEND-MY-MAKE-LIST
  TOKTRANS-3-MAIN-THEOREM TOKTRANS-2-MAIN-THEOREM TOKTRANS-1-MAIN-THEOREM
  DISCARD-DOES-NOT-DISTURB-NON-DISCARDS ))

(disable-theory tokens)
(disable-theory lists)
```

C.5 Token Transformation Implementation for PL_0^R

The full token transformation specification from a character sequence to the corresponding token sequence for PL_0^R is given in S-expression notation below.

```
(setq discard-list ´(comment ws))

(setq replace-words
  (list (cons 43 ´plus)
    (cons 42 ´times)
    (cons 47 ´div)
    (cons 92 ´mod)
    (cons 63 ´quest)
    (cons 33 ´exclaim)
    (cons 91 ´arrayopen)
    (cons 93 ´arrayclose)
    (cons 40 ´parenopen)
    (cons 41 ´parenclose)))

(setq key-words
      (LIST
       (CONS ´(65 78 68)          ´AND)
       (CONS ´(67 65 76 76)       ´CALL)
       (CONS ´(70 65 76 83 69)    ´FALSE)
       (CONS ´(73 70)             ´IF)
       (CONS ´(73 78 80 85 84)    ´INPUT)
       (CONS ´(73 78 84)          ´INT)
       (CONS ´(78 79 84)          ´NOT)
       (CONS ´(79 82)             ´OR)
       (CONS ´(79 85 84 80 85 84) ´OUTPUT)
       (CONS ´(80 82 79 67)       ´PROCKW)
       (CONS ´(82 69 67)          ´REC)
       (CONS ´(83 69 81)          ´SEQ)
       (CONS ´(83 75 73 80)       ´SKIP)
       (CONS ´(83 84 79 80)       ´STOP)
       (CONS ´(84 82 85 69)       ´TRUE)
       (CONS ´(87 72 73 76 69)    ´WHILE )))

(setq continue-list
      (cons ´(plus times div mod quest
              exclaim minus coloneq) nil))

(defn token-transformations (toks discard-list replace-words key-words
  continue-list discard-name
  determine-name determine-default)
  (indentator
   (halve
    (prepare-indentations
     (remove-empty-lines
      (discontinue
```

```
       (integer-convert
(determine-key-words
 (replace
  (discard toks discard-list) discard-name replace-words)
 determine-name key-words determine-default))
       continue-list))))))
```

A scanner for PL_0^R is the following function:

```
(defn scan (nfsa cc tape discard-list
 replace-words key-words continue-list
 discard-name determine-name determine-default)
  (token-transformations
   (split nfsa cc tape)
   discard-list replace-words key-words continue-list
   discard-name determine-name determine-default))
```

called as

```
(scan nfsa cc pl0r discard-list replace-words key-words continue-list
  ´op ´name ´ident))
```

The parameter `pl0r` needs to be a list of bytes, not a file. The following Lisp forms can be used to create such a list.

```
(defparameter a-very-rare-cons ´eof)

(defun current-byte (stream)
  ;; peek at the first character/byte in the stream
  (let ((char
 (peek-char nil          ;; don´t ignore whitespace
    stream
    nil
    a-very-rare-cons)))
    (progn ;; (princ char)
      char)))

(defun rest-bytes (stream)
;; remove the first character from the stream, return the rest
  (let ((char (read-char stream nil a-very-rare-cons )))
    (if (eq char a-very-rare-cons)
nil
stream)))

(defun convert (stream)
 (if (eq (current-byte stream) a-very-rare-cons)
     nil
     (cons (current-byte stream)
   (convert (rest-bytes stream)))))

(defun doit (prog)
  (with-input-from-string (stream prog) (convert stream)))

(defun text-to-ascii (l)
 (if (equal l nil)
     nil
     (cons (char-code (car l))
   (text-to-ascii (cdr l)))))

(defun make-bytes (text) (text-to-ascii (doit text)))

;; This is the program pl0r/tiny.pl0r
(setq text1
"INT x :
INT y :
SEQ
  INPUT ? x
  y := x * x
  OUTPUT ! y
")
```

```
;; We make bytes out of it
(setq bytes1 (make-bytes text1))

;; or are lazy and use this for the R-LOOP
(setq bytes1
 '(73 78 84 32 120 32 58 10 73 78 84 32 121 32 58 10 83 69 81 10 32 32 73
 78 80 85 84 32 63 32 120 10 32 32 121 32 58 61 32 120 32 42 32 120 10
 32 32 79 85 84 80 85 84 32 33 32 121 10))
```

C.6 Retrieval of PL_0^R Characters

This is the outer retrieval function for obtaining a character sequence representation that will scan back to the same token sequence.

```
(defn retrieve (toks replace-words discard-name determine-default
                key-words determine-name)
  (spacing
   (compact
    (squash
     (convert-back
      (retrieve-blanks
       (retrieve-indents toks)))
     determine-name key-words determine-default)
    discard-name replace-words)))
```

called as

```
(retrieve toks replace-words 'op 'ident key-words 'name)
```

Appendix D

Parsing Appendices

These are the events for the parsing skeleton, for the proof of the invariants, for the parsing table generator, and for the table generated for PL_0^R.

D.1 Parsing Skeleton

These are the function definitions for the parsing skeleton.

```
;; ---------------------------------------------------------------------------
;; ~/events/parser.events
;; ---------------------------------------------------------------------------
;;
;; This is the skeleton parser
;;

(enable-theory stacks)
(enable-theory lists)
(enable-theory grammar)
(enable-theory configurations)
(enable-theory derivations)

;; This is what happens to trees during reduction. It should keep the
;; frontier invariant.

(defn reduce-trees (lhs size trees)
 (if (or (lessp (stack-length trees) size)
         (not (is-stack trees))
         (zerop size))
     (emptystack)
     (push (mk-tree
            lhs
            (top-n size trees))
           (pop-n size trees))))

;; The list l matches s if l corresponds to s in stack order.

(defn matches-stack (l s)
 (if (not (is-stack s))
     F
     (if (is-empty s)
         (nlistp l)
         (if (nlistp l)
             T
             (and (equal (car l) (top s))
                  (matches-stack (cdr l) (pop s)))))))

;; This is the basic parsing step.

(defn parsing-step (conf tables grammar)
 (let ((input   (sel-input conf))
       (states  (sel-states conf))
       (symbols (sel-symbols conf))
       (trees   (sel-trees conf))
       (parse   (sel-parse conf))
```

```
       (deriv   (sel-deriv conf))
       (error   (sel-error conf))
       (prods   (sel-productions grammar)))

   (if (nlistp input)
       (mk-configuration
         input states symbols trees parse deriv T)
       (let ((act  (action-lookup (token-name (car input))
                                  (top states)
                                  (sel-Actiontab tables))))
         (case (sel-action-tag act)
           (error
            ;; Set the error flag and terminate.
            (mk-configuration
             input states symbols trees parse deriv T))
           (shift
            (let ((s (sel-state-shift act)))
              (mk-configuration
               (cdr input)
               (push s states)
               ;; I only push the token-name, not the whole token.
               (push (token-name (car input)) symbols)
               (push (mk-tree (car input) nil) trees)
               parse
               deriv
               F)))
           (reduce
              (let ((label (sel-Label-Reduce act))
                    (lhs   (car (sel-Lhs-Reduce act)))
                    (size  (sel-Size-Reduce act)))
                (let ((rhs (sel-rhs (prod-nr label prods))))
                (if (or (lessp (stack-length trees) size)
                        (not (matches-stack (reverse rhs) symbols))
                        (zerop size))
                    ;; I cannot do a reduction if there are less than
                    ;; size things on the stack or if the rhs does not match,
                    ;; so terminate with error.
                    (mk-configuration
                     input states symbols trees parse deriv T)
                  (let ((goto (goto-lookup lhs
                                           (top (pop-n size states))
                                           (sel-Gototab tables))))
                    (mk-configuration
                     input
                     (push goto (pop-n size states))
                     (push lhs (pop-n size symbols))
                     (reduce-trees lhs size trees)
                     (append parse (list label))
                     (append (list (mk-Derivation-Step
                                    (from-bottom (pop-n size symbols))
                                    (mk-prod label lhs (top-n size symbols))
                                    ;; I only use the token names.
                                    (pick-token-names input)))
                             deriv)
                     F))))))
           (otherwise
                (mk-configuration
                 input states symbols trees parse deriv T)))))))

;; I define an explicit acceptance predicate. This is the difference
;; to the standard method of parsing, which has an accept action which
;; is a special reduction. Special cases are not especially conductive
;; to proofs!

(defn accepting (conf axiom)
 (and (equal (car (sel-Input conf))
             (mk-token (end-of-file) nil))  ; input consumed
      (equal (last (sel-Parse conf))        ; Last reduction was the goal
             (list Axiom ))))

(defn error (conf)
  (sel-Error conf))
```

```
;; This is the skeleton parser. I parse until either the clock runs out,
;; or a configuration is reached which is either in error or accepting.

(defn parse-it (conf tables grammar clock)
   (if (zerop clock) ; go buy some more time
       conf
     (if (or (error conf)
             (accepting conf (sel-axiom grammar)))
         conf
         (parse-it (parsing-step conf tables grammar)
                   tables
                   grammar
                   (sub1 clock))))
 ((lessp (count clock))))

;; The input string needs an end-of-file on the end.
;; I use the twice the length of the input (once for each read and once
;; for the maximum stack size) times the number of productions, which
;; is the same as the maximum reduction sequence possible without a cycle,
;; as an approximation for the length of the parsing process.

(defn parser (string tables grammar)
  (parse-it
   (mk-configuration
    (append string (list (mk-token (end-of-file) nil)))
    (push 0 (emptystack))
    (emptystack)
    (emptystack)
    nil
    nil
    F)
    tables
    grammar
    (times (times (length string) 2)
           (length (sel-productions grammar)))))

(disable-theory stacks)
(disable-theory lists)
(disable-theory grammar)
(disable-theory configurations)
(disable-theory derivations)
```

D.2 Invariants

These lemmata are for the proof of the invariants.

```
;; ---------------------------------------------------------------------------
;; ~/events/parsing-inv.events
;; ---------------------------------------------------------------------------

;; This is an attempt to prove the invariants of parsing correct.
;; A *massive* use of PC-NQTHM was necessary to "see" the proofs.

(enable-theory lists)
(enable-theory stacks)
(enable-theory trees)

;; ---------------------------------------------------------------------------
;;      Leaves invariant
;; ---------------------------------------------------------------------------

(prove-lemma leaves-base (rewrite)
 (equal (append (leaves (from-bottom (emptystack))) input)
        input))

(prove-lemma leaves-append (rewrite)
 (implies (listp b)
          (equal (leaves (append a b))
                 (append (leaves a) (leaves b))))
 ((enable-theory lists trees)))
```

```
(prove-lemma better-leaves (rewrite)
 (implies (listp l)
   (equal (leaves (list (mk-tree x l)))
          (leaves l)))
 ((enable-theory trees)))

(prove-lemma configuration-induction-step-shift (rewrite)
 (implies (and (token-listp input)
               (listp input)
               (is-stack trees)
               (equal next (mk-configuration
                            (cdr input)
                            (push s states)
                            (push (car input) symbols)
                            (push (mk-tree (car input) nil) trees)
                            parse
                            deriv
                            F)))
            (equal (append (leaves (from-bottom (sel-trees next)))
                           (sel-input next))
                   (append (leaves (from-bottom trees)) input)))
 ((enable-theory stacks lists trees)))

(prove-lemma frontier-leaf-rewrite (rewrite)
 (equal (frontier 'tree (mk-tree a b))
        (if (is-leaf (mk-tree a b))
            (list a)
            (frontier 'branches b))))

(prove-lemma frontier-tree-is-leaves (rewrite)
 (implies (listp l)
          (equal (frontier 'tree (mk-tree x l))
                 (leaves l))))

(prove-lemma append-leaves (rewrite)
 (implies (listp b)
          (equal (append (leaves a) (leaves b))
                 (leaves (append a b)))))

;; The lemma needs the help of PC-NQTHM to prove, it is just a rewriting
;; proof. NQTHM alone cannot be coaxed to do the necessary rewriting.

(prove-lemma leaves-from-bottom-reduce-leaves (rewrite)
 (implies (and (is-stack trees)
               (not (zerop size))
               (not (lessp (stack-length trees) size)))
          (equal (leaves (from-bottom (reduce-trees lhs size trees)))
                 (leaves (from-bottom trees))))
           ((instructions promote
                          (dive 1 1 1)
                          x up
                          (rewrite from-bottom-push)
                          up
                          (rewrite leaves-append)
                          (dive 2)
                          (rewrite better-leaves)
                          up
                          (rewrite append-leaves)
                          (dive 1)
                          (rewrite append-from-bottom-pop-n)
                          top prove prove prove)))

;; Disable this lemma because it is an incredibly bad rewrite rule!

(disable append-leaves)

(prove-lemma configuration-induction-step-reduce (rewrite)
 (implies (and (token-listp input)
               (listp input)
               (is-stack trees)
               (not (zerop size))
```

```
               (not (lessp (stack-length trees) size))
               (equal next (mk-configuration
                            input
                            (push s states)
                            (push (car input) symbols)
                            (reduce-trees lhs size trees)
                            parse
                            deriv
                            F)))
            (equal (append (leaves (from-bottom (sel-trees next)))
                           (sel-input next))
                   (append (leaves (from-bottom trees)) input)))
 ((disable reduce-trees)))

(prove-lemma frontier-branches-is-leaves (rewrite)
 (equal (frontier 'branches q)
        (leaves q)))

(prove-lemma leaves-from-bottom-pop-n-trees (rewrite)
 (implies (and (is-stack trees)
               (not (zerop size))
               (not (lessp (stack-length trees) size)))
          (equal (append (leaves (from-bottom (pop-n size trees)))
                         (frontier 'branches
                                   (top-n size trees)))
                 (leaves (from-bottom trees))))
             ((instructions promote
                            (dive 1 2)
                            top
                            (dive 2)
                            top
                            (dive 1 2)
                            (rewrite frontier-branches-is-leaves)
                            up
                            (claim (listp (top-n size trees)))
                            (rewrite append-leaves)
                            (dive 1)
                            (rewrite append-from-bottom-pop-n)
                            top prove)))

(prove-lemma append-elimination (rewrite)
 (implies (equal (append x y) a)
          (equal (append x (append y z))
                 (append a z))))

(prove-lemma great-parsing-step-invariant (rewrite)
 (implies (equal next
                 (parsing-step (mk-configuration input states
                                                 symbols
                                                 trees parse
                                                 deriv f)
                               tables grammar))
          (equal (append (leaves (from-bottom (sel-trees next)))
                         (sel-input next))
                 (append (leaves (from-bottom trees))
                         input)))
             ((instructions promote
                            (dive 1 1 1 1 1)
                            = top bash prove promote
                            (dive 1)
                            (rewrite append-elimination
                                     (($a (leaves (from-bottom trees)))))
                            top prove
                            (dive 1)
                            (rewrite append-leaves)
                            (dive 1)
                            (rewrite append-from-bottom-pop-n)
                            top prove
                            prove)))

;; ------------------------------------------------------------------------
;;      Right Sentential form
```

```
;; ---------------------------------------------------------------------------

(enable-theory derivations)
(enable-theory grammar)
(disable sel-action-tag)
(disable action-lookup)

;; A problem with this invariant is that it is not valid until I have made
;; a first reduction, as the derivation is empty at the beginning of parsing.
;; I cannot put a "dummy" derivation on, as there would be nothing to
;; put in the production component, and thus the derivation would not be
;; in lockstep.

(prove-lemma car-append-list (rewrite)
 (equal (car (append (list x) y)) x))

(prove-lemma inv-rt-sent-1-shift (rewrite)
  (implies (and (equal next
                       (mk-configuration
                        (cdr input)
                        (push s states)
                        (push (token-name (car input)) symbols)
                        (push (mk-tree (car input) nil) trees)
                        parse
                        deriv
                        f))
                (listp input)
                (token-listp input)
                (equal (append (from-bottom symbols) (pick-token-names input))
                       (step-source (car deriv))))
           (equal (append (from-bottom (sel-symbols next))
                          (pick-token-names (sel-input next)))
                  (step-source (car (sel-deriv next))))))

(prove-lemma append-from-bottom-push (rewrite)
      (implies (and (is-stack symbols)
                    (equal (append (from-bottom symbols)
                                   (pick-token-names input))
                           (append d (cons x w))))
               (equal (append (from-bottom (push lhs (pop-n size symbols)))
                              (pick-token-names input))
                      (append (from-bottom (pop-n size symbols))
                              (cons lhs (pick-token-names input))))))

(prove-lemma inv-rt-sent-1-reduce (rewrite)
  (implies (and (equal next
                       (mk-configuration
                        input
                        (push goto (pop-n size states))
                        (push lhs (pop-n size symbols))
                        (reduce-trees lhs size trees)
                        (append parse (list label))
                        (append (list (mk-derivation-step
                                       (from-bottom (pop-n size symbols))
                                       (mk-prod label lhs (top-n size symbols))
                                       (pick-token-names input)))
                                deriv)
                        f))
                (is-stack symbols)
                (equal (append (from-bottom symbols) (pick-token-names input))
                       (step-source (car deriv))))
           (equal (append (from-bottom (sel-symbols next))
                          (pick-token-names (sel-input next)))
                   (step-source (car (sel-deriv next))))))

(prove-lemma inv-rt-sent-1 (rewrite)
  (implies (and (equal next
                      (parsing-step (mk-configuration input states
                                                      symbols
                                                      trees parse
                                                      deriv f)
                                    tables grammar))
```

```
                (is-stack symbols)
                (token-listp input)
                (listp input)
                (equal (append (from-bottom symbols)
                               (pick-token-names input))
                       (step-source (car deriv))))
           (equal (append (from-bottom (sel-symbols next))
                          (pick-token-names (sel-input next)))
                  (step-source (car (sel-deriv next))))))

;; ---------------------------------------------------------------------------
;;      Stack size
;; ---------------------------------------------------------------------------

(prove-lemma inv-stack-size-shift (rewrite)
 (implies (and (equal next (mk-configuration
                             (cdr input)
                             (push s states)
                             (push (token-name (car input)) symbols)
                             (push (mk-tree (car input) nil) trees)
                             parse
                             deriv
                             f))
                (is-stack symbols)
                (is-stack states)
                (is-stack trees)
                (and (equal (stack-length trees)
                            (stack-length symbols))
                     (equal (add1 (stack-length symbols))
                            (stack-length states))))
           (and (equal (stack-length (sel-trees next))
                       (stack-length (sel-symbols next)))
                (equal (add1 (stack-length (sel-symbols next)))
                       (stack-length (sel-states next))))))

(prove-lemma stack-length-push (rewrite)
 (equal (stack-length (push a b))
        (add1 (stack-length b))))

(prove-lemma stack-length-pop-n (rewrite)
 (implies (not (lessp (stack-length a) size))
          (equal (stack-length (pop-n size a))
                 (difference (stack-length a) size))))

(prove-lemma inv-stack-size-reduce (rewrite)
 (implies (and (equal next
                      (mk-configuration
                       input
                       (push goto (pop-n size states))
                       (push lhs (pop-n size symbols))
                       (reduce-trees lhs size trees)
                       (append parse (list label))
                       (append (list (mk-derivation-step
                                      (from-bottom (pop-n size symbols))
                                      (mk-prod label lhs (top-n size symbols))
                                      (pick-token-names input)))
                               deriv)
                       f))
                (is-stack symbols)
                (is-stack states)
                (is-stack trees)
                (not (zerop size))
                (not (lessp (stack-length trees) size))
                (and (equal (stack-length trees)
                            (stack-length symbols))
                     (equal (add1 (stack-length symbols))
                            (stack-length states))))
           (and (equal (stack-length (sel-trees next))
                       (stack-length (sel-symbols next)))
                (equal (add1 (stack-length (sel-symbols next)))
                       (stack-length (sel-states next))))))
```

```
(prove-lemma inv-stack-size (rewrite)
 (implies (and (equal next
                      (parsing-step (mk-configuration input states
                                                      symbols
                                                      trees parse
                                                      deriv f)
                                    tables grammar))
               (is-stack symbols)
               (is-stack states)
               (is-stack trees)
               (and (equal (stack-length trees)
                           (stack-length symbols))
                    (equal (add1 (stack-length symbols))
                           (stack-length states))))
          (and (equal (stack-length (sel-trees next))
                      (stack-length (sel-symbols next)))
               (equal (add1 (stack-length (sel-symbols next)))
                      (stack-length (sel-states next))))))

;; ----------------------------------------------------------------------------
;;      Number of reductions
;; ----------------------------------------------------------------------------

;; I only count the inner nodes.

(defn node-count (tag tree)
  (if (equal tag 'tree)
      (if (or (not (is-tree tree))
              (equal tree (emptytree))
              (is-leaf tree))
          0
          (add1 (node-count 'branches (sel-branches tree))))
      (if (nlistp tree)
          0
          (plus (node-count 'tree (car tree))
                (node-count 'branches (cdr tree))))))

(prove-lemma length-append (rewrite)
 (equal (length (append a b))
        (plus (length a)
              (length b))))

(prove-lemma node-count-append (rewrite)
 (equal (node-count 'branches (append a b))
        (plus (node-count 'branches a)
              (node-count 'branches b))))

(prove-lemma node-count-top-n-pop-n (rewrite)
 (equal (plus (node-count 'branches (top-n size trees))
              (node-count 'branches (from-bottom (pop-n size trees))))
        (node-count 'branches (from-bottom trees))))

(prove-lemma inv-reductions-shift (rewrite)
 (implies (and (equal next (mk-configuration
                            (cdr input)
                            (push s states)
                            (push (token-name (car input)) symbols)
                            (push (mk-tree (car input) nil) trees)
                            parse
                            deriv
                            f))
               (equal (node-count 'branches (from-bottom trees))
                      (length parse)))
          (equal (node-count 'branches (from-bottom (sel-trees next)))
                 (length (sel-parse next)))))

;; I need to know that trees is not empty, as the node-count
;; of an empty tree is 0, as is the node-count of a leaf.

(prove-lemma node-count-reduce-trees (rewrite)
 (implies (and (not (zerop size))
               (not (lessp (stack-length trees) size))
```

```
              (not (equal trees (emptystack))))
         (equal (node-count 'branches
                            (from-bottom (reduce-trees lhs size trees)))
                (add1 (node-count 'branches (from-bottom trees)))))
 ((enable-theory trees)))

(prove-lemma inv-reductions-reduce (rewrite)
 (implies (and (equal next
                      (mk-configuration
                       input
                       (push goto (pop-n size states))
                       (push lhs (pop-n size symbols))
                       (reduce-trees lhs size trees)
                       (append parse (list label))
                       (append (list (mk-Derivation-Step
                                      (from-bottom (pop-n size symbols))
                                      (mk-prod label lhs (top-n size symbols))
                                      (pick-token-names input)))
                               deriv)
                       f))
              (not (zerop size))
              (not (lessp (stack-length trees) size))
              (equal (node-count 'branches (from-bottom trees))
                     (length parse))
                     )
          (equal (node-count 'branches (from-bottom (sel-trees next)))
                 (length (sel-parse next))))
 ((disable reduce-trees)))

(prove-lemma inv-reductions (rewrite)
  (implies (and (equal next
                      (parsing-step (mk-configuration input states
                                                      symbols
                                                      trees parse
                                                      deriv f)
                                    tables grammar))
               (not  (equal trees (emptytree)))
               (equal (node-count 'branches (from-bottom trees))
                      (length parse)))
          (equal (node-count 'branches (from-bottom (sel-trees next)))
                 (length (sel-parse next))))
 ((disable reduce-trees )
  (induct (stack-length trees))))

;; ----------------------------------------------------------------------------
;;      Roots
;; ----------------------------------------------------------------------------

(prove-lemma roots-mk-tree (rewrite)
 (equal (roots (list (mk-tree a b))) (list a)))

(prove-lemma roots-append (rewrite)
 (implies (listp b)
         (equal (roots (append a b))
                (append (roots a) (roots b)))))

(prove-lemma pick-token-names-append (rewrite)
 (equal (pick-token-names (append a b))
        (append (pick-token-names a) (pick-token-names b))))

(prove-lemma pick-token-names-list (rewrite)
 (implies (tokenp a)
         (equal (pick-token-names (list a))
                (list (token-name a)))))

(prove-lemma inv-roots-shift (rewrite)
  (implies (and (equal next (mk-configuration
                            (cdr input)
                            (push s states)
                            (push (token-name (car input)) symbols)
                            (push (mk-tree (car input) nil) trees)
                            parse
```

```
                              deriv
                              f))
                (is-stack symbols)
                (token-listp input)
                (listp input)
                (is-stack trees)
                (equal (pick-token-names (roots (from-bottom trees)))
                       (from-bottom symbols)))
           (equal (pick-token-names (roots (from-bottom (sel-trees next))))
                  (from-bottom (sel-symbols next)))))

#|
;; -----------------------------------------------------------------------
;; This is the statement of the roots reduction theorem
;; -----------------------------------------------------------------------

;; The problem is: Sometimes the nodes are tokens (when they are leaves)
;; and sometimes they are just the names of the lhs.
;; Even when I ensure that trees is a stack of trees - what if
;; a left hand side happens to be a token? Then it is not valid!
;; Proof abandonded at this point, as I would need to go back to the
;; definition of grammar and introduce an explicit shell for left hand
;; sides so that I can know that a left hand side can never be a token.

(prove-lemma inv-roots-reduce (rewrite)
  (implies (and (equal next
                       (mk-configuration
                        input
                        (push goto (pop-n size states))
                        (push lhs (pop-n size symbols))
                        (reduce-trees lhs size trees)
                        (append parse (list label))
                        (append (list (mk-derivation-step
                                       (from-bottom (pop-n size symbols))
                                       (mk-prod label lhs (top-n size symbols))
                                       (pick-token-names input)))
                                deriv)
                        f))
                (is-stack symbols)
                (token-listp input)
                (not (lessp (stack-length trees) size))
                (not (zerop size))
                (listp input)
                (is-stack trees)
                (stack-of-trees trees)
                (equal (pick-token-names (roots (from-bottom trees)))
                       (from-bottom symbols)))
           (equal (pick-token-names (roots (from-bottom (sel-trees next))))
                  (from-bottom (sel-symbols next)))))

(prove-lemma inv-roots (rewrite)
  (implies (and (equal next
                      (parsing-step (mk-configuration input states
                                                      symbols
                                                      trees parse
                                                      deriv f)
                                    tables grammar))
                (is-stack symbols)
                (token-listp input)
                (listp input)
                (is-stack trees)
                (equal (pick-token-names (roots (from-bottom trees)))
                       (from-bottom symbols)))
           (equal (pick-token-names (roots (from-bottom (sel-trees next))))
                  (from-bottom (sel-symbols next)))))

|#
```

D.3 Parsing Table Generator

D.3.1 Events

These are the NQTHM events for calculating the FOLLOW sets

```
;; -------------------------------------------------------------------------
;; ~/events/follows.events
;; -------------------------------------------------------------------------

;; This is to show that it is possible to compute the first
;; and follow in Boyer-Moore. However, I will calculate the tables
;; separately so that the run-time for creating the parsing tables does
;; not go up exponentially!

(enable-theory toktrans)

(defn epsilon () 'epsilon)

;; This function is from nqthm-1992/examples/numbers/bags.events.

(defn delete (x l)
      (if (listp l)
          (if (equal x (car l))
              (cdr l)
              (cons (car l) (delete x (cdr l))))
          l))

(defn exists-prod? (prods lhs rhs)
 (if (nlistp prods)
     F
     (if (and (equal lhs (sel-lhs (car prods)))
              (equal rhs (sel-rhs (car prods))))
         T
         (exists-prod? (cdr prods) lhs rhs))))

;; There are three recursive functions involved in calculating first
;; that are mutually recursive. I use a tag to distinguish them:
;; all-rhs   : cdr down the rhs
;; all-prods : cdr down the productions
;; first     : the "real" first function
;; and a clock, as not even an ord-lessp measure is evident -
;; first-all-rhs calls first again, and I know nothing about the length
;; of the car of the rhs.

(defn first (tag x rhs prods terms gram-prods clock)
 (if (zerop clock)
     'time-ran-out

     (if (equal tag 'all-rhs)
         ;; first-all-rhs
         (if (nlistp rhs)
             nil
             (let ((handle (first 'first (car rhs) nil  nil
                                  terms gram-prods (sub1 clock))))
               (if (member (epsilon) handle)
                   (union handle
                          (first 'all-rhs x (cdr rhs) nil
                                 terms gram-prods (sub1 clock)))
                   handle)))

         (if (equal tag 'all-prods)
             ;; first-all-prods
             (if (nlistp prods)
                 nil
                 (if (equal x (car (sel-lhs (car prods))))
                     (union (first 'all-rhs x (sel-rhs (car prods)) nil
                                   terms gram-prods (sub1 clock))
                            (first 'all-prods x nil (cdr prods)
                                   terms gram-prods (sub1 clock)))
                     (first 'all-prods x nil (cdr prods)
                            terms gram-prods (sub1 clock))))
```

```
             ;; The tag is 'first or something else.
             (if (member x terms)
                 (list x) ;; x is a terminal.
                 ;; x is a non-terminal.
                 (union ;; If this is an epsilon production, include epsilon.
                  (if (exists-prod? gram-prods x nil)
                      (list (epsilon))
                      nil)
                  (first 'all-prods x nil gram-prods
                         terms gram-prods (sub1 clock))))))))

;; The first of a list is the first of the car, and if it includes epsilon,
;; the first of the rest. I use the product of the number of productions
;; and the size of the terminals set as an upper bound on the clock.

(defn first-list (l terms prods)
  (if (nlistp l)
      nil
      (let ((first-car (first 'first (car l) nil nil
                              terms prods
                              (times (length terms) (length prods)))))
        (if (member (epsilon) first-car)
            (union first-car (first-list (cdr l) terms prods))
            first-car))))

;; The follow is also a mutually recursive function. Note that it is
;; only defined for nonterminals, and I must terminate if a cyclic
;; production is encountered.
;; A clock is needed, since the prover cannot see that I need to call
;; follow for, at most, all the non-terminals and all productions

(defn follow1 (B prods terms all-prods axiom-lhs cycle-killer clock)
  (if (zerop clock)
      'time-ran-out
      (if (or (member B cycle-killer)
              (member B terms))
          ;; I am trying to add the follow of B to the follow of B.
          nil
          (if (nlistp prods)
              nil
              (let ((rhs  (sel-rhs (car prods)))
                    (lhs  (car (sel-lhs (car prods)))))
                (if (not (member B rhs))
                    (follow1 B (cdr prods)
                             terms all-prods axiom-lhs
                             cycle-killer (sub1 clock))
                    (let ((beta (nthcdr (add1 (position B rhs)) rhs)))
                      (let ((f-1  (first-list beta terms all-prods)))
                        (let
                          ((follow-lhs
                            (if (or (equal beta nil)
                                    (member (epsilon) f-1))
                                (if (equal lhs axiom-lhs)
                                    (union
                                     (list (end-of-file))
                                     (follow1 lhs all-prods
                                              terms all-prods axiom-lhs
                                              (cons B cycle-killer)
                                              (sub1 clock)))
                                    (follow1 lhs all-prods
                                             terms all-prods axiom-lhs
                                             (cons B cycle-killer)
                                             (sub1 clock)))
                                  nil)))
                          (union follow-lhs
                                 (union (delete (epsilon) f-1)
                                        (follow1 B (cdr prods)
                                                 terms all-prods axiom-lhs
                                                 cycle-killer (sub1 clock))))
                          )))))))))

(defn follow (A gram)
```

```
 (let ((prods (sel-productions gram))
       (axiom (sel-axiom gram))
       (terms (sel-terminals gram))
       (nonts (sel-nonterminals gram)))
   (let ((axiom-lhs (car (sel-lhs (prod-nr (sel-productions gram) axiom))))
         (clock (times (length prods) (length nonts))))
     (if (equal A axiom-lhs)
         (union (list (end-of-file))
                (follow1 A prods terms prods axiom-lhs nil clock))
         (follow1 A prods terms prods axiom-lhs nil clock)))))

;; The function all-follows makes the follow table used in the table generator.
;; I only construct the follow for nonterminals.

(defn all-follows1 (nts gram)
 (if (nlistp nts)
     nil
     (cons (cons (car nts) (follow (car nts) gram))
           (all-follows1 (cdr nts) gram))))

(defn all-follows (gram)
  (all-follows1 (sel-nonterminals gram) gram))
```

These are NQTHM events for generating a parsing table.

```
;; ---------------------------------------------------------------------------
;; ~/events/table-generator.events
;; ---------------------------------------------------------------------------

;; This script is for actually constructing the goto and action tables for
;; a grammar.

(enable-theory grammar)
(enable-theory lists)

;; I have defined a transition to be a three-element list consisting of a
;; state, an input symbol, and an action. I use a similar definition for
;; transition as the one used in the NFSA=DFSA proof. There, the third element
;; was the list of following states. Here I have just one action, which is
;; in itself a shell determining what is to be done.
;; Since a name conflict with fsap-transition in the scanner arises,
;; I prefix mk-transition with an a-.

(defn a-mk-transition (state input action)
  (cons (cons state input) action))

;; ---------------------------------------------------------------------------
;;      I T E M   S E T
;; ---------------------------------------------------------------------------
;; Now we want to start constructing an SLR parsing table. The first task is
;; the construction of the item set for each production in the production set.
:: This means that a special token DOT is inserted at every possible position
;; on the rhs, and the resulting productions are collected into the item set
;; for the production. The union (they ought to be disjoint) of the item sets
;; for each production yields the item set for the production set.

;; This function puts the dot between lrhs and rrhs and recurses with the
;; car of rrhs appended to lrhs, remembering the production number.

(defn shift-dots-through (lhs lrhs-in rrhs label)
  (let ((lrhs (if (nlistp lrhs-in)
                  nil
                  lrhs-in)))
        (if (nlistp rrhs)
            (list (mk-prod label lhs (append lrhs (list (dot)))))
            (append
             (list (mk-prod label lhs
                        (append lrhs (append (list (dot)) rrhs))))
             (shift-dots-through
              lhs
              (append lrhs (list (car rrhs)))
              (cdr rrhs)
              label)))))
```

```
;; I split the rhs into 2 parts, the part to the left of the dot and the
;; part to the right of the dot, and insert the dot at that point.

(defn insert-dots (prod)
  (let ((lhs  (sel-lhs prod))
        (lrhs nil)
        (rrhs (sel-rhs prod))
        (label (sel-label prod)))
    (shift-dots-through lhs lrhs rrhs label)))

(defn construct-item-set (prods)
 (if (nlistp prods)
     nil
     (union (insert-dots (car prods))
            (construct-item-set (cdr prods)))))

;; This function constructs the LR(0) items for the grammar

(defn LR-0-items (grammar)
 (construct-item-set (sel-productions grammar)))

;; I use an explicit shell to obtain a tagged representation
;; and a recognizer for item-sets.

(add-shell item-set empty-item-set item-setp
 ((sel-items (none-of) zero))) ; should be listp and nil

(defn first-item (is)
 (if (nlistp (sel-items is))
     nil
     (car (sel-items is))))

(defn rest-items (is)
 (if (nlistp (sel-items is))
     nil
     (item-set (cdr (sel-items is)))))

;; I must construct my own union for shell types.

(defn item-set-union (is1 is2)
 (if (or (equal is1 (empty-item-set))
         (nlistp (sel-items is1))
         (not (item-setp is2))
         (not (item-setp is1)))
     is2
     (if (member (first-item is1) (sel-items is2))
         (item-set-union (rest-items is1) is2)
         (item-set-union  (rest-items is1)
                          (item-set (append (list (first-item is1))
                                            (sel-items is2))))))
 ((lessp (length (sel-items is1)))))

(defn equal-item-set (is1 is2)
 (let ((guts-is1 (sel-items is1))
       (guts-is2 (sel-items is2)))
   (and (item-setp is1)
        (item-setp is2)
        (equal guts-is1 guts-is2))))

(defn item-set-size (item-set)
 (if (or (not (item-setp item-set))
         (nlistp (sel-items item-set))
         (equal item-set (empty-item-set)))
     0
     (add1 (item-set-size (rest-items item-set))))
 ((lessp (length (sel-items item-set)))))

; This is part of the epsilon closure. That is something that is
; reached in an epsilon step from anything in the set. This means that
; the lhs is equal to the sym, and the dot is in the first position.
```

```
(defn next-items (sym all-items)
 (if (nlistp all-items)
     nil
     (if (and
          ;; Sym is the lhs non-terminal.
          (equal sym (car (sel-lhs (car all-items))))
          ;; Dot is in the first rhs position.
          (equal (car (sel-rhs (car all-items))) (dot)))
         (append (list (car all-items))
               (next-items sym (cdr all-items)))
         (next-items sym (cdr all-items)))))

(defn symbol-after-dot (item-rhs)
  (if (nlistp item-rhs)
      nil
      (if (equal (car item-rhs) (dot))
          (if (nlistp (cdr item-rhs))
              nil
              (cadr item-rhs))
          (symbol-after-dot (cdr item-rhs)))))

;; For all items in sis, fis is the full item set.

(defn epsilon-step-all (sis fis)
 (if (or (nlistp (sel-items sis))
         (equal sis (empty-item-set))
         (not (item-setp sis)))
     sis
     (item-set-union
      (item-set (next-items (symbol-after-dot (sel-rhs (first-item sis)))
                            fis))
      (epsilon-step-all (rest-items sis) fis)))
 ((lessp (item-set-size sis))))

;; Note: this closure was implemented before I figured out how to do
;; the termination. Since it works, I am NOT TOUCHING it now!

(defn closure-step (set1 set2 fis clock)
 (if (zerop clock)
     'timed-out
     (if (equal-item-set set1 set2)
         set1
         (item-set-union
          set1
          (closure-step set2
                        (epsilon-step-all set2 fis)
                        fis
                        (sub1 clock))))))

;; The closure adds reachable productions from the grammar until no
;; new ones can be added.

;; I need to find a clock for the closure. Since I cannot have more than
;; the entire item set, I'll use its length.

(defn closure (seed-item-set fis)
 (let ((first  (epsilon-step-all seed-item-set fis)))
      (closure-step seed-item-set first fis (length fis))))

(defn jump-dot (item sym fis)
  (if (nlistp fis)
      nil
      (if (and (equal (sel-label item)
                      (sel-label (car fis)))
               (equal (position (dot) (sel-rhs (car fis)))
                      (add1 (position (dot) (sel-rhs item))))
               (equal (nth (position (dot) (sel-rhs item)) (sel-rhs (car fis)))
                      sym))
          (car fis)
          (jump-dot item sym (cdr fis)))))

(defn dot-sym-in-item-set (sym items fis)
```

```
 (if (nlistp items)
     nil
     (if (equal sym (nth (add1 (position (dot) (sel-rhs (car items))))
                         (sel-rhs (car items))))
         (let ((new (list (jump-dot (car items) sym fis))))
           (union new
                  (dot-sym-in-item-set sym (cdr items)  fis)))
         (dot-sym-in-item-set sym (cdr items) fis))))

;; The goto-function takes the closure of the set of items for which the
;; dot "jumps the symbol".

(defn goto-function (is symbol fis)
  (let ((jump (dot-sym-in-item-set symbol (sel-items is) fis)))
    (if (nlistp jump)
        (empty-item-set)
        (closure (item-set jump) fis))))

;; The collection for an item-set cdrs down the vocabulary (nts and terms),
;; checking the goto-function for the item-set and each vocabulary symbol.
;; If the goto-function is not empty, it is added to the collection.

(defn collection (is symbol-list fis)
 (if (nlistp symbol-list)
     nil
     (let ((goto (goto-function is (car symbol-list) fis)))
       (if (equal goto (empty-item-set))
           (collection is (cdr symbol-list) fis)
           (append
            (list goto)
            (collection is (cdr symbol-list) fis))))))

;; The function items1 cdrs down the current collection (set of item sets)
;; adding the collection for each item set.

(defn items1 (set-of-item-sets v fis)
  (if (nlistp set-of-item-sets)
      set-of-item-sets
      (union
       (collection (car set-of-item-sets) v fis)
       (items1 (cdr set-of-item-sets) v fis))))

;; The function items expands the collection by repeatedly calling
;; items one. When no new item-sets have been added, it terminates.

;; I need a clock for termination. The true measure is the difference
;; between the full-item-set and the sis, but since the latter
;; is defined in an inner let-clause, it is not visible to the measure
;; hint. I can use the length of the full-item-set for the measure
;; and split the work between two functions.

(defn items (set-of-item-sets v fis clock)
 (if (zerop clock)
     (cons set-of-item-sets 'items-times-out)
     (let ((cprime (items1 set-of-item-sets v fis)))
       (if (subsetp cprime set-of-item-sets)
           set-of-item-sets
           (let ((sis (union set-of-item-sets cprime) ))
             (items sis v fis (sub1 clock)))))))

;; Construct the canonical collection for a grammar by setting the collection
;; to the closure for the start production for the first iteration. Then call
;; items to iterate until no more item sets can be added to the collection.
;; Fis is passed to keep from constructing this every time it is needed.
;; An additional speed-up could be obtained by pre-calculating the set of
;; productions that belong to each non-terminal.

(defn start-item (start fis)
 (if (nlistp fis)
     nil
     (if (and (equal start (sel-label (car fis)))
              (equal (car (sel-rhs (car fis))) (dot)))
```

```
        (item-set (list (car fis)))
        (start-item start (cdr fis)))))

(defn canonical-collection (grammar)
 (let ((start (sel-axiom      grammar))
       (voc   (vocab          grammar))
       (fis   (lr-0-items grammar)))
   (let ((c   (list (closure (start-item start fis) fis))))
     (items c voc fis (length fis)))))

;; ----------------------------------------------------------------------------
;;     Extracting the action table
;; ----------------------------------------------------------------------------

(defn is-completed-item (item) (equal (car (last (sel-rhs item))) (dot)))

(defn completed-items (item-set)
 (if (or (not (item-setp item-set))
         (equal item-set (empty-item-set))
         (nlistp (sel-items item-set)))
     nil
     (if (is-completed-item (first-item item-set))
         (cons (first-item item-set) (completed-items (rest-items item-set)))
         (completed-items (rest-items item-set))))
 ((lessp (item-set-size item-set)))))

(defn is-in-follow (after before follows)
  (if (nlistp follows)
      f
      (if (equal before (caar follows))
          (member after (cdar follows))
          (is-in-follow after before (cdr follows)))))

(defn valid-item (item-set sym)
 (if (or (not (item-setp item-set))
         (equal item-set (empty-item-set))
         (nlistp (sel-items item-set)))
     f
     (let ((item (first-item item-set)))
       (let ((rhs (sel-rhs item)))
         (let ((dot-pos (position (dot) rhs)))
           (if (equal sym (nth (add1 dot-pos) rhs))
               t
               (valid-item (rest-items item-set) sym))))))
 ((lessp (item-set-size item-set))))

(defn lookup-follow (sym follows)
 (if (nlistp follows)
     nil
     (if (equal sym (caar follows))
         (cdar follows)
         (lookup-follow sym (cdr follows)))))

;; The following function computes the action for one state and one symbol.

(defn state-action (item-set cc sym fis follows)
 (let ((shift (if (valid-item item-set sym)
                  (mk-shift-action
                   (position (goto-function item-set sym fis) cc))
                  (empty-action)))
       (reduce (let ((comps (item-set (completed-items item-set))))
                 (if (equal (item-set-size comps) 1)
                     (if (member
                          sym
                          (lookup-follow
                           (car (sel-lhs (first-item comps))) follows))
                         (mk-reduce-action
                          (sel-label (first-item comps))
                          (sel-lhs   (first-item comps))
                          ;; Ignore the dot.
                          (sub1 (length (sel-rhs (first-item comps)))))
                         (empty-action))
```

```
                    (if (zerop (item-set-size comps))
                        (empty-action)
                        ´reduce-reduce-conflict)))))
   (if (is-action reduce)
       (if (equal shift (empty-action))
           (if (equal reduce (empty-action))
               (mk-error-action)
               reduce)
           (if (equal reduce (empty-action))
               shift
               ´shift-reduce-conflict))
       reduce)))

(defn one-state (state item-set cc terms fis follows)
  (if (nlistp terms)
      nil
      (append (list (a-mk-transition
                     state
                     (car terms)
                     (state-action item-set cc (car terms) fis follows)))
              (one-state state item-set cc (cdr terms) fis follows))))

(defn mk-actiontab (state cc fullcc terms fis follows)
  (if (nlistp cc)
      nil
      (append (one-state state (car cc) fullcc terms fis follows)
              (mk-actiontab (add1 state) (cdr cc) fullcc terms fis follows))))

;; The set of terminals must have the ´eof symbol added:
;; (mk-actiontab 0 cc cc (append terms (list (end-of-file))) fis follow)


;; ---------------------------------------------------------------------------
;;      Extracting the goto table
;; ---------------------------------------------------------------------------

;; For all nonterminals A if goto (Ii, A) = Ij then goto [i, A] = j.

;; For all states, note that the number which will represent the state
;; is one more than the "real" state, i.e. the position in the list.

(defn mk-goto-1-nt (sis nt state fis)
  (if (zerop state)
      nil
      (let ((I-i  (nth (sub1 state) sis)))
        (let ((goto (goto-function I-i nt fis)))
          (if (equal goto (empty-item-set))
              (mk-goto-1-nt sis nt (sub1 state) fis)
              (union (list (list (cons (sub1 state) nt)
                                 (list ´goto (position goto sis))))
                     (mk-goto-1-nt sis nt (sub1 state) fis)))))))

;; I cdr down the list of nonterminals.

(defn mk-gototab (sis nts fis)
  (if (nlistp nts)
      nil
      (union (mk-goto-1-nt sis (car nts) (length sis) fis)
             (mk-gototab sis (cdr nts) fis))))


;; First I have to construct the canonical collection, then we make the
;; tables and cons them together.

(defn construct-tables (grammar)
 (let ((cc      (canonical-collection grammar))
       (nts     (sel-nonterminals grammar))
       (terms   (append (sel-terminals grammar) (end-of-file)))
       (fis     (LR-0-items grammar))
       (follows (all-follows grammar)))
   (mk-Tables
    (mk-actiontab 0 cc cc terms fis follows)
```

```
    (mk-gototab cc nts fis)))))

#|
; In order to save time, I can use this (and not have to
; calculate the canonical collection three times...

(defn construct-tables1 (cc nts terms fis follows)
   (mk-Tables
    (mk-actiontab 0 cc cc terms fis follows)
    (mk-gototab cc nts fis)))))
|#
```

D.3.2 Generation Instructions

In order to generate a table for the parsing skeleton one must go to a bit of trouble, as the first and follow calculation could not be expressed in NQTHM. A non-left-recursive context-free grammar is needed as input to the table generator. The following is a list of the instructions in the order they need to be done.

1. Bootstrap NQTHM and make sure that all files in the init.lsp are loaded.
2. Start (R-LOOP)
3. Submit the grammar in this form: `(mk-grammar nonterms terms prods axiom)`

```
(setq grammar
      (mk-grammar
       ´(PROG BLK PROC PDECLLIST PDECL DECL SPROCLIST
      PDECLREST SPROCREST GCREST
      SPROC GCLIST GC EXP LITERAL SIMPLE
      DOP MOP VAR) ; non-terminals
       ´(MINUS NOT PLUS TIMES DIV REM EQ LT GT NE LE GE AND OR
       QUEST EXCLAIM
       INT TRUE FALSE SKIP STOP COLONEQ INPUT OUTPUT SEQ IF WHILE CALL
       IDENT COLON LP RP LB RB REC INTEGER ni si bi PROCKW) ; terminals
(list
 (mk-prod 0 ´(PROG) ´(BLK))
 (mk-prod 1 ´(BLK) ´(DECL COLON si BLK))
 (mk-prod 2 ´(BLK) ´(PROC))
 (mk-prod 3 ´(DECL) ´(INT IDENT))
 (mk-prod 4 ´(DECL) ´(LB INTEGER RB INT IDENT))
 (mk-prod 5 ´(PDECL) ´(PROCKW IDENT LP RP ni SPROC bi COLON))
 (mk-prod 6 ´(PDECLLIST) ´(PDECL si PDECLREST))
 (mk-prod 7 ´(PDECLLIST) ´(PDECL))
 (mk-prod 8 ´(PDECLLIST) ´())
 (mk-prod 9 ´(PDECLREST) ´(PDECL))
 (mk-prod 10 ´(PDECLREST) ´(PDECL si PDECLREST))
 (mk-prod 11 ´(PROC) ´(REC ni PDECLLIST bi COLON si PROC))
 (mk-prod 12 ´(PROC) ´(SPROC))
 (mk-prod 13 ´(SPROC) ´(SKIP))
 (mk-prod 14 ´(SPROC) ´(STOP))
 (mk-prod 15 ´(SPROC) ´(VAR COLONEQ EXP))
 (mk-prod 16 ´(SPROC) ´(INPUT QUEST IDENT))
 (mk-prod 17 ´(SPROC) ´(OUTPUT EXCLAIM EXP))
 (mk-prod 18 ´(SPROC) ´(CALL IDENT LP RP))
 (mk-prod 19 ´(SPROC) ´(SEQ ni SPROCLIST bi))
 (mk-prod 20 ´(SPROC) ´(IF ni GCLIST bi))
 (mk-prod 21 ´(SPROC) ´(WHILE EXP ni SPROC bi))
 (mk-prod 22 ´(SPROCLIST) ´(SPROC si SPROCREST))
 (mk-prod 23 ´(SPROCLIST) ´(SPROC))
 (mk-prod 24 ´(SPROCLIST) ´())
 (mk-prod 25 ´(SPROCREST) ´(SPROC))
 (mk-prod 26 ´(SPROCREST) ´(SPROC si SPROCREST))
 (mk-prod 27 ´(GCLIST) ´(GC si GCREST))
 (mk-prod 28 ´(GCLIST) ´(GC))
 (mk-prod 29 ´(GCLIST) ´())
 (mk-prod 30 ´(GCREST) ´(GC si GCREST))
 (mk-prod 31 ´(GCREST) ´(GC))
 (mk-prod 32 ´(GC) ´(EXP ni SPROC bi))
 (mk-prod 33 ´(EXP)  ´(SIMPLE))
 (mk-prod 34 ´(EXP)    ´(MOP SIMPLE))
 (mk-prod 35 ´(EXP)  ´(SIMPLE DOP SIMPLE))
 (mk-prod 36 ´(SIMPLE) ´(VAR))
 (mk-prod 37 ´(SIMPLE) ´(LITERAL))
 (mk-prod 38 ´(SIMPLE) ´(LP EXP RP))
 (mk-prod 39 ´(LITERAL) ´(INTEGER))
 (mk-prod 40 ´(LITERAL) ´(TRUE))
 (mk-prod 41 ´(LITERAL) ´(FALSE))
 (mk-prod 42 ´(VAR)     ´(IDENT))
 (mk-prod 43 ´(VAR)     ´(IDENT LB EXP RB))
 (mk-prod 44 ´(DOP)   ´(PLUS))
 (mk-prod 45 ´(DOP) ´(MINUS))
 (mk-prod 46 ´(DOP) ´(TIMES))
 (mk-prod 47 ´(DOP) ´(DIV))
 (mk-prod 48 ´(DOP) ´(REM))
 (mk-prod 49 ´(DOP) ´(EQ))
 (mk-prod 50 ´(DOP) ´(LT))
 (mk-prod 51 ´(DOP) ´(GT))
 (mk-prod 52 ´(DOP) ´(NE))
 (mk-prod 53 ´(DOP) ´(LE))
 (mk-prod 54 ´(DOP) ´(GE))
 (mk-prod 55 ´(DOP) ´(AND))
 (mk-prod 56 ´(DOP) ´(OR))
 (mk-prod 57 ´(MOP)   ´(MINUS))
 (mk-prod 58 ´(MOP) ´(NOT))
 )
       0))

; Pull out the non-terminals
(setq nts     (sel-nonterminals grammar))

; The terminals need the end-of-file marker
(setq terms   (append (sel-terminals grammar) (list (end-of-file))))
```

4. Calculate the set of LR(0) items for PL_0^R in (R-LOOP). The result, with 178 items, can be found at the WWW-site given in Section 1.2, where the other large results are also kept.

```
* (setq fis (LR-0-items grammar))
 (LIST (MK-PROD 0 ´(PROG) ´(DOT BLK))
       (MK-PROD 0 ´(PROG) ´(BLK DOT))
       (MK-PROD 1 ´(BLK) ´(DOT DECL COLON SI BLK))
       (MK-PROD 1 ´(BLK) ´(DECL DOT COLON SI BLK))
...)
```

5. Create the follow set

```
(setq follows (all-follows grammar))
ok
```

6. Start the LISP-Timer with

```
(get-decoded-time)
```

7. Reenter (R-LOOP) and calculate the canonical collection. For PL_0^R it consists of 112 sets of items, each determining a state in the deterministic automaton.

```
* (setq cc  (canonical-collection grammar))
  (LIST
   (ITEM-SET
    (LIST (MK-PROD 0 ´(PROG) ´(DOT BLK))
          (MK-PROD 1 ´(BLK) ´(DOT DECL COLON SI BLK))
          (MK-PROD 2 ´(BLK) ´(DOT PROC))
          (MK-PROD 3 ´(DECL) ´(DOT INT IDENT))
          (MK-PROD 4 ´(DECL) ´(DOT LB INTEGER RB INT IDENT))
...)))
```

8. Construct the action and goto tables.

   There are 511 entries in the action table and 87 in the goto table for PL_0^R.

```
*  (setq tables (construct-tables1 cc nts terms fis follows))
(LIST
 (LIST (CONS ´(0 . INT) (MK-ACTION ´SHIFT 18 0 0 0))
       (CONS ´(0 . SKIP) (MK-ACTION ´SHIFT 70 0 0 0))
       (CONS ´(0 . STOP) (MK-ACTION ´SHIFT 71 0 0 0))
...)
 ´((0 . BLK) (GOTO 1))
 ´((0 . PROC) (GOTO 16))
 ´((8 . PDECLLIST) (GOTO 12))
 ´((9 . BLK) (GOTO 15))
 ´((9 . PROC) (GOTO 16))
...)
```

9. How long does the calculation take?

```
(get-decoded-time)
```

10. Save a copy of the tables for future reference!

Bibliography

[AL92] Mark Aagaard and Miriam Leeser. Verifying a Logic Synthesis Tool in Nuprl: A Case Study in Software Verification. In *Proceedings of the 4th Workshop on Computer Aided Verification*, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *COMPILERS : Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume I : Parsing. Prentice-Hall, 1972.

[BBMS89] Bettina Buth, Karl-Heinz Buth, Ursula Martin, and Victoria Stavridou. Experiments with program verification systems. Technical Report BB 2, ProCoS[1], Kiel, London, 1989.

[BE76] F. L. Bauer and J. Eickel, editors. *Compiler construction. An Advanced Course*, Berlin, Heidelberg, 1976. Springer Verlag.

[Bev88] William R. Bevier. KIT: A Study in Operating System Verification. Technical Report 28, CLInc, 1988.

[Bev89] William R. Bevier. Kit and the Short Stack. *Journal of Automated Reasoning*, 5(4), Dec 1989.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 41, 1989.

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, New York, 1979.

[BM84a] Robert S. Boyer and J Strother Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In W. W. Bledsoe and D. L. Loveland, editors, *Automated Theorem Proving: After 25 years*, pages 133–167. American Mathematical Society, Providence, R.I., 1984.

[BM84b] Robert S. Boyer and J Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 31(3):441–458, July 1984.

[BM84c] Robert S. Boyer and J Strother Moore. Proof Checking the RSA Public Key Encryption Algorithm. *American Mathematical Monthly*, 91(3):133–167, 1984.

[1]ProCos reports reflect work which was partially funded by the Commission of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Action project no. 3104: "ProCoS: Provably Correct Systems" and are available from the authors or from Dines Bjørner, Department of Computer Science, Technical University of Denmark, Building 344Ø, DK-2800 Lyngby, Denmark

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

[Bro89] A. Bronstein. *MLP: String-functional semantics and Boyer-Moore mechanization for the formal verification of synchronous circuits.* PhD thesis, Stanford University, 1989.

[Brz64] J. A. Brzozowski. Derivations of Regular Expressions. *JACM*, 11(4):481–494, Oct. 1964.

[BWW91] Karl-Heinz Buth and Debora Weber-Wulff. The "Automated Proving and Term Rewriting" Praktikum. Technical Report KHB 3, ProCoS, Kiel, February 1991.

[BY91] R. S. Boyer and Y. Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. Technical Report TR-91-33, Computer Science Dept., University of Texas, Austin, November 1991.

[BY92] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. In *Proceedings of the 11th International Conference on Automated Deduction*, 1992.

[CM82] Avra Cohn and Robin Milner. On using Edinburgh LCF to prove the correctness of a parsing algorithm. Technical Report CSR-112-82, University of Edinburgh, 1982.

[CO90] Rachel Cardell-Oliver. Formal verification of real time protocols using higher order logic. Technical Report 206, University of Cambridge, Computer Laboratory, August 1990.

[Coh82] Avra Cohn. The correctness of a precedence parsing algorithm in LCF. Technical Report 21, University of Cambridge, April 1982.

[Coh88] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, chapter 1, pages 1–91. Kluwer Academic Publishers, 1988.

[Coh89a] Avra Cohn. Correctness properties of the Viper block model: The second level. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 2, pages 27–72. Springer-Verlag, 1989.

[Coh89b] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning 5*, (5):127–138, 1989.

[DB91] ProCoS - ESPRIT BRA 3104 Final report : Provably Correct Systems. Technical report, ProCoS ID/DTH, October 1991.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *CACM*, 14(7):453–460, 1971.

[Fet88] James H. Fetzer. Program verification: The very idea. *CACM*, 31(9):1048–1063, September 1988.

[Fou94] Free Software Foundation. Gnu software archives. Walnut Creek CD-ROM, 1994.

[Frä90] Martin Fränzle. Spezifikation und Verifikation eines übersetzers für eine rekursive occam-artige Programmiersprache. Master's thesis, Institut für Informatik und Prakt. Mathematik der Universität Kiel, Oktober 1990.

[GAS89] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. Report on Gypsy 2.05. Technical Report 1c, CLInc, 1989. Classified.

[Glo80] Paul Gloess. An experiment with the Boyer-Moore theorem prover: A proof of the correctness of a simple parser of expressions. In *LNCS 87 : Proceedings of the CADE-5*, pages 154–169, Berlin, 1980. Springer Verlag.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF*. Springer Verlag, New York, 1979.

[Gor85] Michael J. C. Gordon. HOL: a machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[Gou88] Kevin John Gough. *Syntax Analysis and Software Tools*. Addison-Wesley, Sydney, 1988.

[Gro79] Stanford Verification Group. Stanford Pascal Verifier, User Manual. Technical Report STAN-CS-79-731, Stanford University, Dept. Comp. Sci., March 1979.

[HLS+96] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Deduction in the Verification Support Environment (VSE). In *Proceedings of the Formal Methods in Europe 1996, Oxford*, 1996. To appear.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, 1979.

[Hun87] Warren A. Hunt, Jr. The mechanical verification of a microprocessor design. Technical Report 6, CLInc, 1987.

[Hun89] Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 48, 1989.

[HW90] R. Nigel Horspool and Michael Whitney. Even faster LR parsing. *Software – Practice & Experience*, 20(6):515–535, June 1990.

[il88] inmos ltd. `occam` *2 Reference Manual*. Series in Computer Science. Prentice-Hall International, 1988.

[Jon80] Cliff B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall, 1980.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall International, London, 1990.

[Kau89] Matt Kaufmann. DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers. Technical Report 43, CLInc, 1989.

[Kau91] Matt Kaufmann. Generalization in the presence of free variables: A mechanically-checked correctness proof for one algorithm. *Journal of Automated Reasoning*, 7:109–159, 1991.

[KLW94] Kolyang, Junbo Liu, and Burkhart Wolff. Transformational development of lex. Technical Report Draft version 2 July 94, Universität Bremen, 1994.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1981.

[KW95] Kolyang and Burkhart Wolff. Development by Refinement Revisited: Lessons learnt from an example. In *Proceedings of the Softwaretechnik'95, Braunschweig*, 10 1995. also in "Mitteilung der GI-Fachgruppe Software-Engineering und Requirements-Engineering, Band 15, Heft 3, Okt. 1995.

[Lan66] Peter Landin. The next 700 programming languages. *CACM*, 9, March 1966.

[Lan71] Hans Langmaack. Application of regular canonical systems to grammars translatable from left to right. *Acta Informatica*, 1(1):111–114, 1971.

[Les75] M. E. Lesk. LEX - a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[May78] Otto Mayer. *Syntaxanalyse.* Reihe Informatik 27. BI Wissenschaftsverlag, 1978.

[MO90] Markus Müller-Olm. Korrektheit einer übersetzung der Sprache rekursiver Funktionsdefinitionen erster Ordnung in eine einfache imperative Sprache. Master's thesis, Institut für Informatik und Prakt. Mathematik der Universität Kiel, November 1990.

[Moo88] J Strother Moore. Piton: A verified assembly-level language. Technical Report 22, CLInc, 1988.

[Moo89] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 30, 1988.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science. Proc. Symp. Appl. Math*, volume XIX, pages 33–41. American Mathematical Society, 1967.

[Myh57] J.R. Myhill. Finite automata and representation of events. Technical Report Tech Rep. 57-624, Wright Air Development Center, 1957.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings of the CADE 11, Saratoga, NY, June 1992*, number 607 in LNAI, pages 748–752. Springer Verlag, 1992.

[ORS93] S. Owre, J. Rushby, and N. Shankar. A tutorial on specification and verification using PVS. In *Tutorial Material for FME'93: Industrial-Strength Formal Methods. Proceedings of the First International Symposium of Formal Methods Europe, Odense, Denmark*, pages 357–406, April 1993.

[Pag77] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7:249–268, 1977.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau93] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in LNCS. Springer-Verlag, New York, 1994.

[Pen83] Volker Penner. Entwicklung und Verifikation eines Scanner Generators mit dem Gypsy Verification Environment. Technical Report 86, RWTH Aachen, Schriften zur Informatik und Angewandten Mathematik, 1983.

[Pie90] Laurence Pierre. The formal proof of sequential circuits described in CASCADE using the Boyer-Moore theorem prover. In L. Claesen, editor, *Formal VLSI Correctness Verification*. Elsevier, 1990.

[Pie93] Laurence Pierre. VHDL description and formal verification of systolic multipliers. In *IFIP Conference on Hardware Description Languages and their applications*, Ottawa, Canada, April 1993.

[Pie94] Laurence Pierre. An automatic generalization method for the inductive proof of replicated and parallel architectures. In *Theorem Provers in Circuit Design*, Bad Herrenalb (Blackforest), Germany, September 1994.

[Pol81] Wolfgang Polak. *Compiler Specification and Verification*. LNCS 124. Springer Verlag, New York, 1981.

[Pro88] GNU Project. *Bison - Manual Page*. Public Domain Software, 1988.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal*, pages 114–125, April 1959.

[Rus85] David M. Russinoff. An experiment with the Boyer-Moore theorem prover: A proof of Wilson's theorem. *Journal of Automated Reasoning*, 1(2):121–139, 1985.

[Rus92] David M. Russinoff. A mechanical proof of quadratic reciprocity. *Journal of Automated Reasoning*, 8(1), 1992.

[Sha85] N. Shankar. A mechanical proof of the Church-Rosser theorem. Technical Report 45, University of Texas, Institute for Computer Science, Austin, Texas, March 1985.

[Sha86] N. Shankar. *Proof Checking Metamathematics*. PhD thesis, Univ. of Texas, Austin, 1986.

[SSS88] S. Sippu and E. Soisalon-Soininen. *Parsing Theory. Vol.1: Languages and Parsing*, volume 15 of *EATCS Monograph on Theoretical Computer Science*. Springer Verlag, Berlin, 1988.

[VCDM90] D. Verkest, L. Claesen, and H. De Man. Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment. In *Proceedings of EDAC-90*, pages 62 – 66, March 1990.

[VVCDM92] D. Verkest, J. Vandenbergh, L. Claesen, and H. De Man. A description methodology for parameterized modules in the Boyer-Moore logic. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *IFIP Transactions A-10: Theorem Provers in Circuit Design*, pages 37 – 57. Elsevier, 1992.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung*. Springer Verlag, Berlin, Heidelberg, 1992.

[WW90] Debora Weber-Wulff. Trip report : Visit to Computational Logic, Inc., Austin, Texas. Technical Report DWW 1, ProCoS, Kiel, February 1990.

[WW91] Debora Weber-Wulff. Pass collapsing : An optimization method for compiler proofs. Technical Report DWW 7, ProCoS Kiel, September 1991.

[WW92] Debora Weber-Wulff. When whitespace conveys meaning. Technical Report DWW 10, TFH Berlin, Berlin, October 1992.

[WW93a] Debora Weber-Wulff. Proof movie : A Proof with the Boyer-Moore prover. *Formal Aspects of Computing*, 5:121–151, 1993.

[WW93b] Debora Weber-Wulff. Selling formal methods to industry. In *FME'93 Symposium Industrial Strength Formal Methods. Proceedings. April 19-23, 1993, Odense, Denmark*, number 670 in LNCS, pages 671–678, 1993.

[You88] William D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, CLInc, 1988.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4), Dec 1989. also available as CLInc Technical Report 37 , 1989.

[You93] William D. Young. A mechanically checked proof of the equivalence of deterministic and non-deterministic finite state machines, October 19, 1993. CLInc Internal Note #290.

[Yu90] Yuan Yu. Computer proofs in group theory. *Journal of Automated Reasoning*, 6:251–286, 1990.