```
#|

 Copyright (C) 1994 by Computational Logic, Inc.  All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Computational Logic, Inc. PROVIDES ABSOLUTELY NO WARRANTY.  THE EVENT SCRIPT
 IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,
 INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE DEFECTIVE, YOU
 ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 IN NO EVENT WILL Computational Logic, Inc. BE LIABLE TO YOU FOR ANY DAMAGES,
 ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT
 NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
 SUSTAINED BY THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.
|#

; The Formalized Extended Syntax


; SECTION: Introduction

; This file, examples/basic/parser.events, is a formalization of
; the extended syntax as presented in Chapter 4.  We formalize the
; syntax in the logic itself.  Roughly speaking, for every occur-
; rence of ``Terminology'' in Chapter 4 there is an admissible
; function definition here that formalizes the concept defined in
; Chapter 4.  For example, we formally define what it is to be a
; ``numeric sequence'' by introducing a function which returns T or
; F according to whether its argument is such a sequence.

; The outline of our presentation is as follows.  We first develop
; the notion of a ``token tree.''  Token trees in the logic are re-
; presented by integers, literal atoms, and list structures.  The
; text in Chapter 4 then develops the idea of how to ``display'' a
; token tree.  Here we do the inverse: we formalize the notion of
; how to parse a token tree from a sequence of ASCII character
```

```
; codes.  This culminates in the definition of READ-TOKEN-TREE
; which takes a list of character codes and returns either F (in-
; dicating that the input sequence is unparsable) or a token tree.
; We then return to the text of Chapter 4 and formalize what it is
; for a token tree to be ``readable,'' what ``readmacro-expansion''
; is, and what an ``s-expression'' is.  This part of the formal-
; ization may illuminate backquote notation for those unfamiliar
; with it.  We finally define what it is for an s-expression to be
; ``well-formed'' and what the ``translation'' of such an
; s-expression is.

; This presentation is meant to clarify the extended syntax.  Thus,
; it would hardly do to express the formalization in terms of the
; extended syntax.  We therefore largely confine ourselves here to
; the formal syntax with the following exceptions:

; 1. We permit ourselves to write natural numbers, e.g., 2, in
;    place of their formal equivalents, e.g., (ADD1 (ADD1 (ZERO))).

; 2. We permit ourselves to use QUOTE notation to denote literal
;    atom constants.  Thus, we write 'ABC where
;      (PACK (CONS 65 (CONS 66 (CONS 67 0))))
;    would otherwise be needed.

; 3. We occasionally display list constants in QUOTE notation,
;    e.g., '(UPPER-B UPPER-O UPPER-X) is written in place of
;    (CONS 'UPPER-B (CONS 'UPPER-O (CONS 'UPPER-X 'NIL))).  Two
;    large association lists are displayed with QUOTE notation.
;    These displays are in conjunction with the definitions of
;    utility functions whose semantics is intuitively clear, namely
;    a function that maps from our name for an ASCII character to
;    its ASCII code and a function that maps from the name of a
;    function to its arity.

; 4. We permit ourselves to write comments.

; It should be stressed that these are ``exceptions'' only in the
; sense that they are not part of the formal syntax.  These con-
; ventions are entirely supported by the extended syntax and we
; note their use only because they are exceptions to our self-
; imposed restriction to the formal syntax in this file.

; We have followed the text's style of the definition very closely.
; This results in a somewhat ``inefficient'' formalization.  For
```

```
; example, it is possible to implement the parsing/readmacro-expan-
; sion process in a single pass, but we have not done so.  While we
; feel our use of the logic here is illustrative of ``good usage''
; the critical reader must keep in mind that we are trying to
; formalize the ideas as presented in the text and not merely code
; a parser in the logic.

; This file may be processed with PROVE-FILE and the resulting
; library may be noted and used in R-LOOP to experiment with the
; syntax.  We recommend that the library be compiled.  If the Nqthm
; installer has so processed the file, the library file may already
; exist as examples/basic/parser.  This source file also contains
; ``commented out'' Common Lisp code that permits more convenient
; experimentation.  Search for occurrences of ``defun'' to find the
; two regions in question.

; Finally, this file serves as a good example of a fairly substan-
; tial Nqthm formalization effort.  We recommend it even to readers
; who know the syntax but who wish to see how Nqthm is used to
; formalize ideas that are already precisely (but informally)
; understood.  We urge such readers to compare the formal defini-
; tions with the corresponding informal definitions of Chapter 4.

; We start by initializing Nqthm's data base to the GROUND-ZERO
; theory.
```

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

```
; SECTION: Conventions Concerning Characters

; In the text we imagine that we have integers, characters and
; sequences of characters as our atoms.  We then build token trees
; as sequences of these atoms and other token trees.  Thus, the
; sequence consisting of 65, 66, and 67 is uniquely recognized as
; being a sequence of integers, while the sequence consisting of
; the characters A, B, and C, is recognized as being a word.

; But in this formalization, we do not have characters and in fact
; we use their ASCII codes instead.  But the ASCII codes are them-
; selves integers.  Thus, the sequence containing 65, 66, and 67 is
; ambiguous as a token tree: is it a tree containing three integers
; or is it a tree containing the word ABC?  Fortunately, characters
; never enter token trees except as the elements of words and the
```

```
; various tokens.  Therefore, in this formalization we use LITATOMs
; to represent the words and the tokens.  That is, where the text
; represents the word ABC as a sequence consisting of the char-
; acters A, B, and C, the formalization will represent it as a
; LITATOM obtained by PACKing the ASCII codes for A, B, and C
; (around a 0).  Similarly, the backquote token in the text is the
; sequence consisting of the backquote character, but here it is
; the LITATOM obtained by packing the ASCII code for that character
; (around a 0).

; We first make it convenient to refer to the ASCII character code
; of all the printable characters.  We will invent a name for each
; character, e.g., UPPER-A, LOWER-B, OPEN-PAREN, and define (ASCII
; name) to be the ASCII integer code for the named character.  Thus
; (ASCII 'UPPER-A) will be 65.  We will also define ASCII so that
; when given a list of names it returns the list of ASCII codes for
; the characters named in the list.  Thus, (ASCII '(UPPER-A
; LOWER-A)) will be '(65 97).  This is just a clumsy way to circumvent
; the omission of character strings from Nqthm's universe of objects.

; The codes we assign to TAB, NEWLINE, PAGE, SPACE, and RUBOUT are
; those used in AKCL; typically those codes vary from one Common
; Lisp to another.  This variance is not important to our
; formalization, as we are just assigning unique integers to
; certain character names known to our function ASCII.  For
; example, even in a Lisp assigning #\Newline a code other than 10,
; our ASCII function will assign NEWLINE the code 10 and our parser
; will execute as intended on ``strings'' obtained by writing
; (ASCII '(... NEWLINE ...)).  However, the user wishing to
; experiment with the parser may wish to write a Lisp utility that
; converts user typein into lists of character codes, so as to
; avoid the use of our clumsy ASCII function.  Care must be taken
; in the definition of such a utility so that the first five
; characters listed below are assigned the codes shown.

; The names and their ASCII codes are given in the following
; association list:
```

DEFINITION:
ASCII-TABLE

```
=  '((tab . 9)
     (newline . 10)
     (page . 12)
```

```
(space . 32)
(rubout . 127)
(exclamation-point . 33)
(double-quote . 34)
(number-sign . 35)
(dollar-sign . 36)
(percent-sign . 37)
(ampersand . 38)
(single-quote . 39)
(open-paren . 40)
(close-paren . 41)
(asterisk . 42)
(plus-sign . 43)
(comma . 44)
(minus-sign . 45)
(dot . 46)
(slash . 47)
(digit-zero . 48)
(digit-one . 49)
(digit-two . 50)
(digit-three . 51)
(digit-four . 52)
(digit-five . 53)
(digit-six . 54)
(digit-seven . 55)
(digit-eight . 56)
(digit-nine . 57)
(colon . 58)
(semicolon . 59)
(less-than-sign . 60)
(equal-sign . 61)
(greater-than-sign . 62)
(question-mark . 63)
(at-sign . 64)
(upper-a . 65)
(upper-b . 66)
(upper-c . 67)
(upper-d . 68)
(upper-e . 69)
(upper-f . 70)
(upper-g . 71)
(upper-h . 72)
(upper-i . 73)
(upper-j . 74)
```

```
(upper-k . 75)
(upper-l . 76)
(upper-m . 77)
(upper-n . 78)
(upper-o . 79)
(upper-p . 80)
(upper-q . 81)
(upper-r . 82)
(upper-s . 83)
(upper-t . 84)
(upper-u . 85)
(upper-v . 86)
(upper-w . 87)
(upper-x . 88)
(upper-y . 89)
(upper-z . 90)
(open-bracket . 91)
(backslash . 92)
(close-bracket . 93)
(uparrow . 94)
(underscore . 95)
(backquote . 96)
(lower-a . 97)
(lower-b . 98)
(lower-c . 99)
(lower-d . 100)
(lower-e . 101)
(lower-f . 102)
(lower-g . 103)
(lower-h . 104)
(lower-i . 105)
(lower-j . 106)
(lower-k . 107)
(lower-l . 108)
(lower-m . 109)
(lower-n . 110)
(lower-o . 111)
(lower-p . 112)
(lower-q . 113)
(lower-r . 114)
(lower-s . 115)
(lower-t . 116)
(lower-u . 117)
(lower-v . 118)
```

```
      (lower-w . 119)
      (lower-x . 120)
      (lower-y . 121)
      (lower-z . 122)
      (open-brace . 123)
      (vertical-bar . 124)
      (close-brace . 125)
      (tilde . 126))

; ~
```

DEFINITION:
ascii-list (*lst*)
= **if** *lst* $\simeq$ **nil then nil**
else cons (cdr (assoc (car (*lst*), ASCII-TABLE)), ascii-list (cdr (*lst*))) **endif**

DEFINITION:
ascii (x)
= **if** litatom (x) **then** cdr (assoc (x, ASCII-TABLE))
else ascii-list (x) **endif**

```
; So now we can write (ASCII 'UPPER-A) for 65 and
; (ASCII '(UPPER-A LOWER-A)) for '(65 97).


; SECTION: Token Trees
```

DEFINITION:
UPPER-DIGITS
= ascii (`'(digit-zero digit-one digit-two digit-three digit-four digit-five digit-six digit-seven digit-eight digit-nine upper-a upper-b upper-c upper-d upper-e upper-f)`)

```
; We define (CADRN n LST) to be equivalent to (CADD...DR LST) where
; the number of D's is n.  Thus, (CADRN 2 LST) is (CADDR LST).  An-
; other way to think about CADRN is that it returns the Nth element
; of LST using 0-based enumeration, e.g., (CADRN 3 '(A B C D E F))
; is 'D.
```

DEFINITION:
cdrn (n, *lst*)
= **if** $n \simeq 0$ **then** *lst*
else cdrn ($n - 1$, cdr (*lst*)) **endif**

DEFINITION: $\text{cadrn}\,(n,\ \mathit{lst}) = \text{car}\,(\text{cdrn}\,(n,\ \mathit{lst}))$

```
; It is also convenient to have
```

DEFINITION: $\text{list1}\,(x) = \text{cons}\,(x,\ \mathbf{nil})$

DEFINITION: $\text{list2}\,(x,\ y) = \text{cons}\,(x,\ \text{list1}\,(y))$

DEFINITION: $\text{list3}\,(x,\ y,\ z) = \text{cons}\,(x,\ \text{list2}\,(y,\ z))$

```
; because we are eschewing the use of the abbreviation LIST.
```

DEFINITION:
first-n $(n,\ \mathit{lst})$
$=$ **if** $n \simeq 0$ **then nil**
else $\text{cons}\,(\text{car}\,(\mathit{lst}),\ \text{first-n}\,(n - 1,\ \text{cdr}\,(\mathit{lst})))$ **endif**

DEFINITION:
base-n-digit-character $(n,\ c)$
$= ((n \leq 16) \wedge (c \in \text{first-n}\,(n,\ \textsc{upper-digits})))$

DEFINITION:
position $(x,\ \mathit{lst})$
$=$ **if** $\mathit{lst} \simeq$ **nil then** 0
elseif $x = \text{car}\,(\mathit{lst})$ **then** 0
else $1 + \text{position}\,(x,\ \text{cdr}\,(\mathit{lst}))$ **endif**

DEFINITION: $\text{base-n-digit-value}\,(c) = \text{position}\,(c,\ \textsc{upper-digits})$

DEFINITION:
all-base-n-digit-characters $(n,\ \mathit{lst})$
$=$ **if** $\mathit{lst} \simeq$ **nil then t**
else base-n-digit-character $(n,\ \text{car}\,(\mathit{lst}))$
$\wedge$ all-base-n-digit-characters $(n,\ \text{cdr}\,(\mathit{lst}))$ **endif**

DEFINITION:
base-n-digit-sequence $(n,\ \mathit{lst})$
$= (\text{listp}\,(\mathit{lst}) \wedge \text{all-base-n-digit-characters}\,(n,\ \mathit{lst}))$

DEFINITION:
optionally-signed-base-n-digit-sequence $(n,\ \mathit{lst})$
$=$ (base-n-digit-sequence $(n,\ \mathit{lst})$
$\vee$ (listp (lst)
$\wedge$ (((car (lst) = ascii (`'plus-sign`))
$\vee$ (car (lst) = ascii (`'minus-sign`)))
$\wedge$ base-n-digit-sequence $(n,\ \text{cdr}\,(\mathit{lst}))))$)

DEFINITION:
length (lst)
= **if** $lst \simeq$ **nil** **then** `0`
else $1 +$ length (cdr (lst)) **endif**

DEFINITION:
exp (i, j)
= **if** $j \simeq$ `0` **then** `1`
else $i *$ exp (i, $j - 1$) **endif**

DEFINITION:
base-n-value (n, lst)
= **if** $lst \simeq$ **nil** **then** `0`
else (base-n-digit-value (car (lst)) $*$ exp (n, length (cdr (lst))))
$+$ base-n-value (n, cdr (lst)) **endif**

DEFINITION:
numerator-sequence (lst)
= **if** $lst \simeq$ **nil** **then nil**
elseif car (lst) = ascii (`'slash`) **then nil**
else cons (car (lst), numerator-sequence (cdr (lst))) **endif**

DEFINITION:
denominator-sequence (lst)
= **if** $lst \simeq$ **nil** **then nil**
elseif car (lst) = ascii (`'slash`) **then** cdr (lst)
else denominator-sequence (cdr (lst)) **endif**

DEFINITION:
base-n-signed-value (n, lst)
= **if** car (lst) = ascii (`'minus-sign`)
then $-$ base-n-value (n, cdr (lst))
elseif car (lst) = ascii (`'plus-sign`)
then base-n-value (n, cdr (lst))
else base-n-value (n, lst) **endif**

DEFINITION:
number-sign-sequence (lst)
= ((length (lst) $\geq$ `3`)
$\land$ (car (lst) = ascii (`'number-sign`))
$\land$ (cadrn (`1`, lst) $\in$ ascii (`'(upper-b upper-o upper-x)`))
$\land$ optionally-signed-base-n-digit-sequence (**if** cadrn (`1`, lst)
= ascii (`'upper-b`)
then `2`
elseif cadrn (`1`, lst)

$= \quad$ ascii ('upper-o)
then 8
else 16 **endif**,
cdr (cdr (*lst*))))

DEFINITION:
last (*lst*)
= **if** *lst* ≃ **nil** **then** *lst*
elseif cdr (*lst*) ≃ **nil** **then** *lst*
else last (cdr (*lst*)) **endif**

DEFINITION:
all-but-last (*lst*)
= **if** *lst* ≃ **nil** **then** **nil**
elseif cdr (*lst*) ≃ **nil** **then** **nil**
else cons (car (*lst*), all-but-last (cdr (*lst*))) **endif**

DEFINITION:
numeric-sequence (*lst*)
= (optionally-signed-base-n-digit-sequence (10, *lst*)
∨ (((car (last (*lst*)) = ascii ('dot))
∧ optionally-signed-base-n-digit-sequence (10,
all-but-last (*lst*)))
∨ number-sign-sequence (*lst*)))

DEFINITION:
numeric-value (*lst*)
= **if** optionally-signed-base-n-digit-sequence (10, *lst*)
then base-n-signed-value (10, *lst*)
elseif car (last (*lst*)) = ascii ('dot)
then base-n-signed-value (10, all-but-last (*lst*))
else base-n-signed-value (**if** cadrn (1, *lst*)
= ascii ('upper-b) **then** 2
elseif cadrn (1, *lst*)
= ascii ('upper-o)
then 8
else 16 **endif**,
cdr (cdr (*lst*))) **endif**

DEFINITION:
SINGLE-QUOTE-TOKEN = pack (cons (ascii ('single-quote), 0))

DEFINITION:
BACKQUOTE-TOKEN = pack (cons (ascii ('backquote), 0))

DEFINITION: DOT-TOKEN = pack (cons (ascii ('dot), 0))

DEFINITION: COMMA-TOKEN = pack (cons (ascii ('comma), 0))

DEFINITION:
COMMA-AT-SIGN-TOKEN
= pack (cons (ascii ('comma), cons (ascii ('at-sign), 0)))

DEFINITION:
COMMA-DOT-TOKEN = pack (cons (ascii ('comma), cons (ascii ('dot), 0)))

DEFINITION:
WORD-CHARACTERS
= ascii ('(upper-a upper-b upper-c upper-d upper-e upper-f
upper-g upper-h upper-i upper-j upper-k upper-l
upper-m upper-n upper-o upper-p upper-q upper-r
upper-s upper-t upper-u upper-v upper-w upper-x
upper-y upper-z digit-zero digit-one digit-two
digit-three digit-four digit-five digit-six
digit-seven digit-eight digit-nine dollar-sign
uparrow ampersand asterisk underscore minus-sign
plus-sign equal-sign tilde open-brace close-brace
question-mark less-than-sign greater-than-sign))

DEFINITION:
subsetp (x, y)
= **if** $x \simeq$ **nil then t**
elseif car $(x) \in y$ **then** subsetp (cdr (x), y)
else f endif

DEFINITION:
word (s)
= (litatom (s)
∧ (numeric-sequence (unpack (s))
∨ (listp (unpack (s))
∧ subsetp (unpack (s), WORD-CHARACTERS))))

```
; It is convenient to be able to recognize those words that are
; numeric sequences and to talk about their numeric values, without
; having to think about unpacking them.  So we define NUMERIC-WORD
; and NUMERIC-WORD-VALUE and use them below where the text would
; have us use ``numeric sequence'' and ``numeric value.''
```

DEFINITION:
numeric-word (s) = (litatom (s) ∧ numeric-sequence (unpack (s)))

DEFINITION: numeric-word-value (s) = numeric-value (unpack (s))

DEFINITION: $\text{integerp}(x) = ((x \in \mathbf{N}) \lor \text{negativep}(x))$

DEFINITION:
$\text{special-token}(x)$
$= ((x = \text{SINGLE-QUOTE-TOKEN})$
$\quad \lor \; ((x = \text{BACKQUOTE-TOKEN})$
$\quad\quad \lor \; ((x = \text{COMMA-TOKEN})$
$\quad\quad\quad \lor \; ((x = \text{COMMA-AT-SIGN-TOKEN})$
$\quad\quad\quad\quad \lor \; (x = \text{COMMA-DOT-TOKEN})))))$

```
; The following function can be used as follows.  If we test
; (EQLEN X 3) then we know that lst is of the form (x1 x2 x3).
```

DEFINITION:
$\text{eqlen}(lst, n)$
$=$ **if** $n \simeq 0$ **then** $lst =$ **nil**
elseif $lst \simeq$ **nil** **then f**
else $\text{eqlen}(\text{cdr}(lst), n - 1)$ **endif**

DEFINITION:
$\text{dotted-pair}(x) = (\text{eqlen}(x, 3) \land (\text{cadrn}(1, x) = \text{DOT-TOKEN}))$

DEFINITION:
$\text{dotted-s-expression}(x)$
$=$ **if** $x \simeq$ **nil** **then f**
elseif $\text{dotted-pair}(x)$ **then t**
else $\text{dotted-s-expression}(\text{cdr}(x))$ **endif**

DEFINITION: $\text{singleton}(x) = \text{eqlen}(x, 1)$

```
; We use the following lemma to make the CADRNs go away during the
; termination proof for TOKEN-TREE.
```

THEOREM: cdrn-expander
$\text{cdrn}(1 + n, x) = \text{cdrn}(n, \text{cdr}(x))$

```
; Now we define token trees formally.  We use the logic's LISTPs,
; together with the integers, words and tokens (the last two being
; LITATOMs), to represent token trees.  We are faithful to the
; text's style of representing dotted token trees by lists whose
; second-to-last elements are the dot token.  That is, we don't
; represent such trees by dotted pairs in the logic.  (This would
; not work because the token tree (A B . (C D E)) is legitimate and
; is different from (A B C D E).  The text permits such token trees
```

```
; since they may be typed.  They readmacro-expand to the same
; s-expression.)  However, we do not define ``token tree'' in quite
; the same style as the text, though the result is the same.  The
; text recognizes undotted and dotted token trees ``from the
; outside'' -- e.g., an undotted token tree is a nonempty sequence
; of token trees and a dotted one is similar but has a dot as its
; second-to-last element.  To define token trees formally this way
; we would need to use mutual recursion.  Rather than do that, we
; just recurse down dotted and undotted lists and catch the dot
; when we come to it.
```

DEFINITION:
token-tree (x)
$=$ **if** $x \simeq$ **nil** **then** integerp $(x) \lor$ word (x)
elseif special-token (car (x)) $\land$ (length $(x) = 2$)
then token-tree (cadrn (1, x))
elseif dotted-pair (x)
then token-tree (car (x)) $\land$ token-tree (cadrn (2, x))
elseif singleton (x) **then** token-tree (car (x))
else token-tree (car (x)) $\land$ token-tree (cdr (x)) **endif**

```
; We define a few functions to make token tree manipulation and
; recognition easier.
```

DEFINITION: special-token-tree (x) = special-token (car (x))

DEFINITION:
single-quote-token-tree (x) = (car (x) = SINGLE-QUOTE-TOKEN)

DEFINITION:
backquote-token-tree (x) = (car (x) = BACKQUOTE-TOKEN)

DEFINITION:
comma-escape-token-tree (x) = (car (x) = COMMA-TOKEN)

DEFINITION:
splice-escape-token-tree (x)
$=$ ((car (x) = COMMA-AT-SIGN-TOKEN) $\lor$ (car (x) = COMMA-DOT-TOKEN))

DEFINITION: constituent (x) = cadrn (1, x)

```
; SECTION:  Reading Token Trees

; At this point in the text, we define how to display a token tree.
```

```
; We will skip that here and instead formalize the process of pars-
; ing a token tree from a sequence of characters (i.e., from a dis-
; play of one).

; Our parser is reminiscent of Lisp's read routine with several im-
; portant exceptions.  We do not deal with readmacros in our parser
; -- that is the job of readmacro-expansion (defined later).  We
; also do not produce dotted pairs but rather lists containing the
; dot token.  Finally, we can produce ``unreadable'' token trees
; such as parsed from (PLUS ,X B).

; The parser is called READ-TOKEN-TREE and it is implemented in two
; passes.  The first pass transforms a list of ASCII character
; codes into a list of lexemes.  In our case, all the lexemes are
; LITATOMs, including the numeric sequences that look like inte-
; gers, i.e., we resolve the ambiguity in ``display'' by always
; using numeric sequences.  The second pass parses the lexemes into
; a tree.  In fact, the parser can produce trees that are not token
; trees, such as the one produced by parsing (PLUS # B).  This
; ``pseudo-token tree'' fails to be a token tree because one of the
; atoms in it that ``should'' be a word is not a word.  READ-TOKEN-
; TREE therefore calls the predicate TOKEN-TREE on the parsed tree
; to determine whether the parse was successful.


; SUBSECTION: Pass I of the Reader

; We give convenient names to the lexemes for open and close
; parentheses.  We could call these ``tokens'' but don't want to
; confuse them with the tokens of the text.  The lexemes for the
; tokens, e.g., the dot lexeme, will be the tokens themselves,
; e.g., the value of (DOT-TOKEN).
```

DEFINITION: $\text{OPEN-PAREN} = \text{pack}\,(\text{cons}\,(\text{ascii}\,(\texttt{'open-paren}), \texttt{0}))$

DEFINITION: $\text{CLOSE-PAREN} = \text{pack}\,(\text{cons}\,(\text{ascii}\,(\texttt{'close-paren}), \texttt{0}))$

```
; The following function scans past characters until it has passed
; the first newline character.  It is used to scan past semicolon
; comment.  It returns the stream starting just after that newline.
; If no newline is found, it returns the empty stream.  The ques-
; tion then arises: was the comment that started this scan well-
; formed or not?  It is missing its final newline.  It turns out
; that Common Lisp's answer to this is that it is ok.  That is, a
```

```
; file can end without there being a final newline on the end of a
; comment on the last line of the file.  The text of Chapter 4 does
; not deal with this issue.
```

DEFINITION:
skip-past-newline (*stream*)
= **if** *stream* $\simeq$ **nil then** *stream*
elseif car (*stream*) = ascii (`'newline`) **then** cdr (*stream*)
else skip-past-newline (cdr (*stream*)) **endif**

```
; The lexical analyzer will sometimes recurse using this function
; to ``decrement'' the input stream.  We must prove that the stream
; returned is weakly smaller (the semicolon that starts the
; comment will have already been stripped off by an explicit CDR).
; We measure size here with LENGTH instead of COUNT because COUNT
; doesn't decrease for all of the recursions (see the next function
; definition).
```

THEOREM: lessp-skip-past-newline
length (*stream*) $\not<$ length (skip-past-newline (*stream*))

```
; The next function scans for the end of a just-opened #-comment.
; The function returns the stream just past the closing sequence of
; the comment.  The variable I below counts the number of ``open''
; #| sequences we have seen.  We do not stop until we see a |#
; sequence that closes that one, i.e., decrements I to 0.  I is
; initially 1 because we call this function after reading the
; opening sequence.  Note that this function will scan the entire
; input stream if the balancing sequence is not present.  This is
; treated as an error by Common Lisp.  We therefore have the
; problem of signalling this error.  If we return the empty stream,
; it is indistinguishable from a well-formed comment that ends at
; the very end of the stream.  We therefore use a trick: we return
; the stream containing a single open parenthesis character.  This
; character will cause the lexical analyzer to put an unbalanced
; open parenthesis lexeme as the last lexeme in the stream fed to
; the parser.  That, in turn, will cause an error.  We also signal
; such an error if we find too many |# sequences.  A minor tech-
; nical problem arises: to insure that the lexical analyzer can
; recurse with this skipping function, we have to make sure the
; size of the stream we return is less than the one the lexical
; analyzer started with.  Since the analyzer will have CDRd past
; the opening #| our adding a single open parenthesis in the place
```

```
; of those two characters will not matter -- if we measure the size
; of the stream with LENGTH!  If we were to measure with COUNT, we
; would have to worry about the size of the various ASCII codes and
; about the object in the final CDR of the stream.
```

DEFINITION:
skip-past-balancing-vertical-bar-number-sign (*stream*, *i*)
= **if** *stream* $\simeq$ **nil** **then** cons (ascii (`'open-paren`), *stream*)
elseif (car (*stream*) = ascii (`'number-sign`))
$\wedge$ (cadrn (`1`, *stream*) = ascii (`'vertical-bar`))
then skip-past-balancing-vertical-bar-number-sign (cdrn (`2`, *stream*),
1 + *i*)
elseif (car (*stream*) = ascii (`'vertical-bar`))
$\wedge$ (cadrn (`1`, *stream*) = ascii (`'number-sign`))
then if i = `0` **then** cons (ascii (`'open-paren`), cdrn (`2`, *stream*))
elseif i = `1` **then** cdrn (`2`, *stream*)
else skip-past-balancing-vertical-bar-number-sign (cdrn (`2`,
stream),
$i - 1$) **endif**
else skip-past-balancing-vertical-bar-number-sign (cdr (*stream*), *i*) **endif**

```
; This is the key inductive fact about the relative LENGTHs of the
; input and output of the function above:
```

THEOREM: skip-past-balancing-vertical-bar-number-sign-lemma
(1 + length (*stream*))
$\not<$ length (skip-past-balancing-vertical-bar-number-sign (*stream*, *i*))

```
; However, this is the actual theorem we need to admit the lexical
; analyzer (eventually) below:
```

THEOREM: lessp-skip-past-balancing-vertical-bar-number-sign
(listp (*stream*)
$\wedge$ ((car (*stream*) = ascii (`'number-sign`))
$\wedge$ (cadr (*stream*) = ascii (`'vertical-bar`))))
$\rightarrow$ (length (skip-past-balancing-vertical-bar-number-sign (cddr (*stream*), `1`))
< length (*stream*))

```
; To recognize white space we need:
```

DEFINITION:
white-spacep (c) = ($c \in$ ascii (`'(space tab newline)`))

```
; We will accumulate the characters in a lexeme in a list (with a 0
; at the bottom) and when we have a completed lexeme we will add it
; to a growing list of lexemes.  The characters are accumulated in
; reverse order.  The empty lexeme is never added to the list.
```

DEFINITION:
rev1 (*lst*, *ans*)
= **if** *lst* $\simeq$ **nil** **then** *ans*
else rev1 (cdr (*lst*), cons (car (*lst*), *ans*)) **endif**

DEFINITION:
emit (*pname*, *lst*)
= **if** *pname* = `0` **then** *lst*
else cons (pack (rev1 (*pname*, `0`)), *lst*) **endif**

```
; The following function maps lower case alphabetic characters into
; their upper case counterparts.  Thus, (UPCASE (ASCII 'LOWER-A))
; is (ASCII 'UPPER-A), etc.  UPCASE is the identity function on
; characters other than LOWER-A through LOWER-Z.
```

DEFINITION:
upcase (*c*)
= **if** (ascii (`'lower-a`) $\leq$ *c*) $\land$ (*c* $\leq$ ascii (`'lower-z`))
then *c* $-$ (ascii (`'lower-a`) $-$ ascii (`'upper-a`))
else *c* **endif**

```
; Here then is the lexical analyzer, pass I of the parser.  The
; function returns a list of lexemes parsed from stream, which is a
; list of ASCII codes.  PNAME is the lexeme currently being
; assembled.  It is 0 when ``empty.''
```

DEFINITION:
lexemes (*stream*, *pname*)
= **if** *stream* $\simeq$ **nil** **then** emit (*pname*, **nil**)
elseif car (*stream*) = ascii (`'semicolon`)
then emit (*pname*, lexemes (skip-past-newline (cdr (*stream*)), `0`))
elseif car (*stream*)
$\in$ ascii (`'(backquote single-quote open-paren`
`close-paren)`)
then emit (*pname*, emit (cons (car (*stream*), `0`), lexemes (cdr (*stream*), `0`)))
elseif car (*stream*) = ascii (`'comma`)
then **if** (cadrn (`1`, *stream*) = ascii (`'at-sign`))

```
        ∨  (cadrn (1, stream) = ascii ('dot))
      then emit (pname,
                 emit (cons (cadrn (1, stream), cons (car (stream), 0)),
                       lexemes (cdr (cdr (stream)), 0)))
      else emit (pname,
                 emit (cons (car (stream), 0), lexemes (cdr (stream), 0))) endif
  elseif white-spacep (car (stream))
  then emit (pname, lexemes (cdr (stream), 0))
  elseif (pname = 0)
        ∧  ((car (stream) = ascii ('number-sign))
            ∧  (cadrn (1, stream) = ascii ('vertical-bar)))
  then lexemes (skip-past-balancing-vertical-bar-number-sign (cdrn (2,
                                                                    stream),
                                                              1),
                0)
  else lexemes (cdr (stream), cons (upcase (car (stream)), pname)) endif

; SUBSECTION:  Pass II of the Reader

; We now develop the parser, which maps a list of lexemes into a
; tree -- or returns F if the list does not parse.  One example of
; a list of lexemes that doesn't parse is one containing unbalanced
; parentheses.  Another example is one that contains more lexemes
; than necessary to describe exactly one tree.  That is, our parser
; balks if there are lexemes left over when one tree has been
; parsed.

; The formalization below is just a simple push-down stack auto-
; maton.  It scans the lexemes one at a time from the right all the
; way to the end of the list.  It has a stack of trees it is build-
; ing up.  When it sees an open parenthesis, it pushes an empty
; frame onto the stack.  When it sees a normal lexeme, like the
; word PLUS or the integer 123, it accumulates this onto the right
; end of the top-most frame.  When it sees a close parenthesis, it
; pops the top frame and accumulates it onto the right end of the
; frame below.  We code it so that if any of these stack operations
; is ill-formed, e.g., we pop an empty stack, the result is F.
; Furthermore, we arrange for an F stack to be propagated, i.e.,
; pushing something onto F produces F.  Thus, if a parsing error
; arises early in the scan of the lexemes, e.g., an unbalanced
; close parenthesis is seen, the stack becomes F and even though
; the scan continues until the last lexeme has been processed, the
; F is preserved as the signal that an error occurred.
```

```
; When all the lexemes have been processed we inspect the stack and
; verify that it contains a single frame.  If not, then we return
; F.  (If the stack contains less than one frame, then the stack is
; in fact F and an error was detected earlier.  If the stack
; contains more than one frame, we have unbalanced open paren-
; theses.)

; To make this work we have to start the stack as though we are
; accumulating a list, i.e., with one empty frame on the stack.
; Then when we are done we have to check that the list we accumu-
; lated has exactly one element.  If it has none, the input was
; empty, which is an error.  If it has more than one, the input
; contained more than one tree and that is an error.

; To handle the special tokens single-quote, backquote, comma,
; comma-at-sign, and comma-dot, we generalize the stack slightly so
; that when these tokens are read they push a new frame that con-
; tains the token rather than the empty list.  Then we continue
; parsing.  When the next tree is assembled it is ``accumulated
; onto the right end of the frame below,'' where we actually take
; into account the special tokens that might mark the frame below.
; For example, if x is to be accumulated onto the frame below, and
; the frame below contains the single-quote token, then we create
; the single-quote tree (' x) and recursively accumulate it onto
; the frame below that.

; Finally, the dot token is afforded no such special handling
; except that when a list is accumulated onto another (or returned
; as the answer) we verify that if it contains the dot token as an
; element then the token occurs as the next-to-last element.

; Of course, this whole process is much simpler than the Lisp
; reader because we are not implementing readmacros here, nor do we
; have to do the checks for ``readability.''  We just parse a tree
; and return it for the rest of this system to inspect.

; So here are the functions implementing our stacks
```

DEFINITION:
top-pstk (*stack*)
= **if** listp (*stack*) **then** car (*stack*)
else f endif

DEFINITION:

pop-pstk (*stack*)
= **if** listp (*stack*) **then** cdr (*stack*)
else f endif

DEFINITION:
push-pstk (x, *stack*)
= **if** *stack* = **f then f**
else cons (x, *stack*) **endif**

DEFINITION: empty-pstk (x) = ($x \simeq$ **nil**)

```
; Here is the predicate that verifies that a dotted tree is prop-
; erly formed.  We permit x to be the dot token, even though that
; is not a token tree, so that it can be accumulated just like
; other atoms.  We will filter out the isolated dot at the top.  So
; what we want to check here is that if X is a list and the dot
; token appears as an element then it appears in the next-to-last
; position and there is at least one element before it.  The vari-
; able I below just counts the number of elements we've scanned
; past.
```

DEFINITION:
dot-criterion (i, x)
= **if** $x \simeq$ **nil then t**
elseif car (x) = DOT-TOKEN
then ($i \not\simeq 0$) $\wedge$ (listp (cdr (x)) $\wedge$ (cdr (cdr (x)) $\simeq$ **nil**))
else dot-criterion ($1 + i$, cdr (x)) **endif**

```
; Here is how we accumulate X ``onto the frame below,'' where the
; ``frame below'' is the top of the stack passed into this func-
; tion.  We check first that X satisfies the dot token criterion
; and cause an error if it doesn't.  Then we handle the special
; tokens.
```

DEFINITION:
add-element-to-top (x, *stack*)
= **if** empty-pstk (*stack*) **then f**
elseif $\neg$ dot-criterion (0, x) **then f**
elseif special-token (top-pstk (*stack*))
then add-element-to-top (list2 (top-pstk (*stack*), x), pop-pstk (*stack*))
else push-pstk (append (top-pstk (*stack*), list1 (x)), pop-pstk (*stack*)) **endif**

```
; When we are all done, the following function is used to verify
```

```
; that exactly one whole object was constructed and that it is not
; an isolated dot-token.
```

DEFINITION:
stop (*stack*)
= **if** eqlen (*stack*, 1)
 ∧ (eqlen (top-pstk (*stack*), 1)
 ∧ (car (top-pstk (*stack*)) ≠ DOT-TOKEN))
then car (top-pstk (*stack*))
else f endif

```
; Finally, here is the parser.  The proper top-level call of this
; function is (PARSE lexemes (CONS NIL NIL)).  It returns either a
; tree or F.  F indicates that the lexemes either did not parse in-
; to a complete tree or parsed into more than one.  The tree may be
; ``unreadable.''
```

DEFINITION:
parse (*lexemes*, *stack*)
= **if** *lexemes* ≃ **nil** **then** stop (*stack*)
elseif car (*lexemes*) = OPEN-PAREN
then parse (cdr (*lexemes*), push-pstk (**nil**, *stack*))
elseif special-token (car (*lexemes*))
then parse (cdr (*lexemes*), push-pstk (car (*lexemes*), *stack*))
elseif car (*lexemes*) = CLOSE-PAREN
then parse (cdr (*lexemes*),
add-element-to-top (top-pstk (*stack*), pop-pstk (*stack*)))
else parse (cdr (*lexemes*), add-element-to-top (car (*lexemes*), *stack*)) **endif**

```
; SUBSECTION:  Putting the Two Passes Together

; As noted, the parser may not produce a token tree because the
; lexical analyzer can produce non-word, non-token lexemes.  For
; example, the stream (ASCII '(SPACE NUMBER-SIGN SPACE)) parses as
; the ``word'' we might display as # and which is logically repre-
; sented by the literal atom (PACK (CONS 35 0)).  But this is not a
; word, technically, because the number sign character is not among
; the word characters.  Therefore, after we have parsed the lexemes
; we check that the result is a token tree and return F if it is
; not.  Thus, this function may return F because
; (a) the lexical analyzer found fault with the character stream
;     (e.g., an unbalanced #-comment)
; (b) the parser found fault with the lexeme stream (e.g., an un-
```

```
;     balanced open parenthesis), or
; (c) the final tree failed to be a token tree.
```

DEFINITION:
read-token-tree (*stream*)
= **if** token-tree (parse (lexemes (*stream*, 0), cons (**nil**, **nil**)))
then parse (lexemes (*stream*, 0), cons (**nil**, **nil**))
else f endif

```
; SECTION:  Readable Token Trees and S-Expressions

; We now resume our formalization of the text.  The definition of
; ``readable from depth'' N assumes X is a token tree or F.  We
; allow F as a possible input only because that is the error signal
; presented by READ-TOKEN-TREE.  F is not a token tree -- the only
; atomic token trees are integers and words.  Our formalization of
; ``readable'' announces that F is not readable, passing the
; ``error'' up.
```

DEFINITION:
readable (x, n)
= **if** $x \simeq$ **nil** **then** $x \neq$ **f**
elseif single-quote-token-tree (x) **then** readable (constituent (x), n)
elseif backquote-token-tree (x)
then readable (constituent (x), $1 + n$)
$\wedge$ ($\neg$ splice-escape-token-tree (constituent (x)))
elseif comma-escape-token-tree (x) $\vee$ splice-escape-token-tree (x)
then ($0 < n$) $\wedge$ readable (constituent (x), $n - 1$)
elseif dotted-pair (x)
then readable (car (x), n)
$\wedge$ (readable (cadrn (2, x), n)
$\wedge$ ($\neg$ splice-escape-token-tree (cadrn (2, x))))
elseif singleton (x) **then** readable (car (x), n)
else readable (car (x), n) $\wedge$ readable (cdr (x), n) **endif**

```
; Like our formalization of ``readable,'' our formalization of ``s-
; expression'' allows its argument to be either a token tree or F
; and returns F on F.  Thus, if you apply S-EXPRESSION to the out-
; put of READ-TOKEN-TREE you will get F if the read ``caused an
; error.''  In actual use, we only apply S-EXPRESSION to the output
; of READMACRO-EXPANSION and we only apply that to READABLE token
; trees.  Thus, this aspect of our formalization of s-expressions
; is irrelevant.  We preserve it because it is sometimes nice,
```

```
; while testing these definitions, to run S-EXPRESSION on the
; output of the reader.
```

DEFINITION:
s-expression (x)
= **if** $x \simeq$ **nil**
  then if $x =$ **f** **then f**
    else $\neg$ numeric-word (x) **endif**
  elseif special-token-tree (x) **then f**
  elseif dotted-pair (x)
  then s-expression (car (x)) $\wedge$ s-expression (cadrn (2, x))
  elseif singleton (x) **then** s-expression (car (x))
  else s-expression (car (x)) $\wedge$ s-expression (cdr (x)) **endif**

```
; SECTION:  Backquote Expansion

; At this point in the text we give the definition of readmacro-
; expansion.  However, it is presented without first defining the
; notion of backquote expansion.  We therefore skip ahead in the
; text and formalize backquote expansion now so we can then present
; readmacro-expansion.

; Of course, backquote expansion and backquote-list expansion are
; mutually recursive.  If FLG below is T we are defining backquote
; expansion.  Otherwise, we are defining backquote-list expansion.
```

DEFINITION:
backquote-expansion (*flg*, x)
= **if** *flg*
  then if $x \simeq$ **nil** **then** list2 (`'quote`, x)
    elseif comma-escape-token-tree (x)
      $\vee$ splice-escape-token-tree (x) **then** constituent (x)
    else backquote-expansion (**f**, x) **endif**
  elseif $x \simeq$ **nil**
  then `'impossible-if-x-is-a-dotted-or-undotted-token-tree`
  else list3 (**if** splice-escape-token-tree (car (x)) **then** `'append`
      else `'cons` **endif**,
      backquote-expansion (**t**, car (x)),
      if singleton (x) **then** list2 (`'quote`, `'nil`)
      elseif dotted-pair (x)
      then backquote-expansion (**t**, cadrn (2, x))
      else backquote-expansion (**f**, cdr (x)) **endif**) **endif**

```
; SECTION:  Readmacro Expansion

; Our definition of readmacro-expansion differs from the text in
; that we do all four passes at once.  It is so easy in English to
; say ``replace every ...'' and its formalization requires a recur-
; sive sweep through the tree.  Even if we had higher order func-
; tions (or used V&C$) we would spend about as much space defining
; the generic sweep as we do below just doing it.
```

DEFINITION:
readmacro-expansion (x)
$=$ **if** $x \simeq$ **nil**
then if numeric-word (x) **then** numeric-word-value (x)
else x **endif**
elseif single-quote-token-tree (x)
then list2 (`'quote`, readmacro-expansion (constituent (x)))
elseif backquote-token-tree (x)
then backquote-expansion (**t**, readmacro-expansion (constituent (x)))
elseif dotted-pair (x)
then if ((readmacro-expansion (cadrn (2, x)) $\simeq$ **nil**)
$\wedge$ (readmacro-expansion (cadrn (2, x)) $\neq$ `'nil`))
$\vee$ special-token-tree (readmacro-expansion (cadrn (2, x)))
then list3 (readmacro-expansion (car (x)),
DOT-TOKEN,
readmacro-expansion (cadrn (2, x)))
else cons (readmacro-expansion (car (x)),
readmacro-expansion (cadrn (2, x))) **endif**
elseif singleton (x) **then** list1 (readmacro-expansion (car (x)))
else cons (readmacro-expansion (car (x)),
readmacro-expansion (cdr (x))) **endif**

```
; SECTION:  Some Common Lisp

#|

; This entire section is commented out.  It is here only as a con-
; venience for those wishing to test the token tree reader and
; readmacro-expansion.  The following function reads a token tree
; from a stream of ASCII character codes.  If it did not parse or
; is ``unreadable,'' suitable messages are returned.  Otherwise,
; the token tree is readmacro-expanded.  If an s-expression does
; not result, a suitable message is returned.  THIS ERROR MESSAGE
; SHOULD NEVER HAPPEN because the readmacro-expansion of a readable
```

```
; token tree supposedly always produces an s-expression!  If no
; errors are reported, the function returns the resulting s-
; expression.  Thus, this function is essentially a formalization
; of the the Lisp read routine.

(DEFN TEST-READER (STREAM)
  (LET ((X (READ-TOKEN-TREE STREAM)))
    (COND ((NOT X) 'DID-NOT-PARSE)
  ((NOT (READABLE X 0)) 'NOT-READABLE-FROM-0)
  (T (LET ((Y (READMACRO-EXPANSION X)))
       (COND
                ((S-EXPRESSION Y)
                 Y)
                (T (LIST
                    'READMACRO-EXPANSION-PRODUCED-NON-S-EXPRESSION
                    Y))))))))

; The following defines a Lisp routine, not a function in the
; logic.  The routine's name is test-reader and it is just a
; convenient interface to the logical function defined above.  You
; give it a Lisp string and it converts it into a list of ASCII
; codes.  This just lets us type things like (test-reader "(PLUS X
; Y)") to Common Lisp -- not to R-LOOP -- to execute the logic's
; TEST-READER.  This Lisp definition assigns the correct character
; codes in AKCL.  Recall the comment above about the codes for TAB,
; NEWLINE, PAGE, SPACE, and RUBOUT when implementing a utility to
; convert Common Lisp typein into lists of ASCII codes.

 (defun ascii (string)
   (mapcar (function char-code)
           (coerce string 'list)))

 (defun test-reader (string)
   (*1*test-reader (ascii string)))

; We now return to the main flow of this development of the
; extended syntax.

|#


; SECTION: Some Terminology Preliminary to Translation

; We now define the concepts used to describe the process of
```

```
; translation from an s-expression to a formal term.
```

DEFINITION:
corresponding-numberp (n)
$=$ **if** $n \simeq$ `0` **then** list1 (`'zero`)
else list2 (`'add1`, corresponding-numberp $(n - 1)$) **endif**

DEFINITION:
corresponding-negativep (n)
$=$ list2 (`'minus`, corresponding-numberp (negative-guts (n)))

DEFINITION:
a-d-sequencep (*lst*)
$=$ **if** *lst* $\simeq$ **nil then t**
elseif (car (*lst*) = ascii (`'upper-a`))
$\vee$ (car (*lst*) = ascii (`'upper-d`))
then a-d-sequencep (cdr (*lst*))
else f endif

DEFINITION:
car-cdr-symbolp (x)
$=$ (litatom (x)
$\wedge$ ((car (unpack (x)) = ascii (`'upper-c`))
$\wedge$ ((car (last (unpack (x))) = ascii (`'upper-r`))
$\wedge$ a-d-sequencep (all-but-last (cdr (unpack (x)))))))

```
; While the text defines the A/D sequence of a CAR/CDR symbol to be
; a sequence of A's and D's, we define it to be a sequence of the
; ASCII codes of A and of D.
```

DEFINITION: a-d-sequence (x) = all-but-last (cdr (unpack (x)))

DEFINITION:
car-cdr-nest (*lst*, x)
$=$ **if** *lst* $\simeq$ **nil then** x
elseif car (*lst*) = ascii (`'upper-a`)
then list2 (`'car`, car-cdr-nest (cdr (*lst*), x))
else list2 (`'cdr`, car-cdr-nest (cdr (*lst*), x)) **endif**

```
; Ultimately we will define the translation process.  This will
; involve us in consing together the translations of various com-
; ponents of an s-expression, after making sure those components
; are well-formed.  We signal ill-formed input by returning F in-
```

```
; stead of (the quotation of) the formal term.  To make the neces-
; sary well-formedness checks less distracting we define the fol-
; lowing function which we use to create the formal terms.  It is
; like CONS, but observes the convention that if either argument is
; F the result is F.
```

DEFINITION:
fcons $(x,\ y)$
$=$ **if** $x \land y$ **then** cons $(x,\ y)$
else f endif

```
; We also provide ourselves with several LIST-like functions that
; respect this convention.  'Tis a pity we do not have macros in
; the Nqthm logic because we really just want to define a ``func-
; tion'' symbol that takes an arbitrary number of arguments.
```

DEFINITION: flist1 (x) = fcons $(x,$ **nil**$)$

DEFINITION: flist2 $(x,\ y)$ = fcons $(x,$ flist1 $(y))$

DEFINITION: flist3 $(x,\ y,\ z)$ = fcons $(x,$ flist2 $(y,\ z))$

DEFINITION: flist4 $(x,\ y,\ z,\ w)$ = fcons $(x,$ flist3 $(y,\ z,\ w))$

DEFINITION: flist5 $(x,\ y,\ z,\ w,\ v)$ = fcons $(x,$ flist4 $(y,\ z,\ w,\ v))$

DEFINITION: flist6 $(x,\ y,\ z,\ w,\ v,\ u)$ = fcons $(x,$ flist5 $(y,\ z,\ w,\ v,\ u))$

DEFINITION:
flist7 $(x,\ y,\ z,\ w,\ v,\ u,\ r)$ = fcons $(x,$ flist6 $(y,\ z,\ w,\ v,\ u,\ r))$

```
; Here is the formalization of the ``fn nest around x for lst'',
; which is presented in the text before we get to the extended
; syntax.  Note that we use our FCONS convention so that if X or
; any element of LST is F, the result is F.
```

DEFINITION:
fn-nest $(fn,\ lst,\ x)$
$=$ **if** $lst \simeq$ **nil** **then** x
else flist3 $(fn,$ car $(lst),$ fn-nest $(fn,$ cdr $(lst),\ x))$ **endif**

DEFINITION:
corresponding-numberps (lst)
$=$ **if** $lst \simeq$ **nil** **then nil**
else cons (corresponding-numberp (car (lst)),
corresponding-numberps (cdr (lst))) **endif**

DEFINITION:
explosion (lst)
= fn-nest (`'cons`, corresponding-numberps (lst), list1 (`'zero`))

DEFINITION:
corresponding-litatom (x) = list2 (`'pack`, explosion (unpack (x)))

```
; The following function tacks the *1* prefix onto the front of
; LITATOMs.  Thus, (star-one-star 'true) is the LITATOM '*1*TRUE --
; EXCEPT we can't write that literal atom that way because it is
; not a symbolp.  It must be written as (PACK '(42 49 42 84 82 85
; 69 . 0)) or, equivalently, '(*1*QUOTE PACK (42 49 42 84 82 85 69
; . 0)).
```

DEFINITION:
star-one-star (x)
= pack (append (ascii (`'(asterisk digit-one asterisk)`), unpack (x)))

```
; We need to define what it is to be a symbol.  The text does this
; quite early, while introducing formal terms.  Here is the
; formalization.
```

DEFINITION:
ASCII-UPPER-ALPHABETICS
= ascii (`'(upper-a upper-b upper-c upper-d upper-e upper-f`
`upper-g upper-h upper-i upper-j upper-k upper-l`
`upper-m upper-n upper-o upper-p upper-q upper-r`
`upper-s upper-t upper-u upper-v upper-w upper-x`
`upper-y upper-z)`)

DEFINITION:
ASCII-DIGITS-AND-SIGNS
= ascii (`'(digit-one digit-two digit-three digit-four`
`digit-five digit-six digit-seven digit-eight`
`digit-nine dollar-sign uparrow ampersand asterisk`
`underscore minus-sign plus-sign equal-sign tilde`
`open-brace close-brace question-mark`
`less-than-sign greater-than-sign)`)

DEFINITION:
all-upper-alphabetics-digits-or-signs (l)
= **if** $l \simeq$ **nil then t**
else ((car (l) $\in$ ASCII-UPPER-ALPHABETICS)
$\lor$ (car (l) $\in$ ASCII-DIGITS-AND-SIGNS))
$\land$ all-upper-alphabetics-digits-or-signs (cdr (l)) **endif**

Definition:
legal-char-code-seq (l)
= (listp (l)
∧ ((cdr (last (l)) = 0)
∧ ((car (l) ∈ ascii-upper-alphabetics)
∧ all-upper-alphabetics-digits-or-signs (cdr (l)))))

Definition:
symbolp (x) = (litatom (x) ∧ legal-char-code-seq (unpack (x)))

```
; SECTION:  Formalizing Aspects of the History

; At this point in the text we define well-formedness and trans-
; lation.  But the text delays until later the definitions of
; several concepts used.  These delayed concepts are: QUOTE
; notation, the *1*QUOTE escape mechanism, and abbreviated FORs.
; We have to formalize these before we can define translation.
; Unfortunately, before we can formalize *1*QUOTE notation we must
; formalize certain concepts relating to the ``history'' in which
; the translation is occurring.  For example, (FN X Y) is well-
; formed and has a translation only if FN is a function symbol of
; arity 2.  We must formalize arity.  We must also formalize the
; notions of what it is to be a shell constructor, a base object,
; and to satisfy the ``type- restrictions'' of a shell.

; But functions that depend upon the ``current history'' cannot be
; defined unless we wish to make the ``current history'' an ex-
; plicit object in the formalization.  That is a reasonable thing
; to do but is beyond the scope of this book because the only DEFN
; events (for example) permitted in a history are those that are
; admissible, which involves the notion of proof.  Thus, to formal-
; ize histories accurately we would have to formalize the rules of
; inference (not just the syntax) and what it is to be a theorem.
; This has been done for other logics in the Nqthm logic.  See for
; example Shankar's work in examples/shankar or the bibliographic
; entries for Shankar.

; Rather than formalize histories in order to formalize the syntax,
; we will formalize the syntax for a particular history.  The his-
; tory we choose is the GROUND-ZERO history extended with one il-
; lustrative user-defined shell, the stacks as illustrated in the
; text, plus one illustrative user-defined function, CONCAT, which
; is just APPEND by another name.  We introduce these functions be-
; low into the theory.  They are nowhere used in this development
```

```
; and thus by searching for references to them you can see every-
; thing that must be known about a function symbol for it to be
; usable in the extended syntax.

; An irrelevant shell added to illustrate user-defined shells:
```

EVENT: Add the shell *push*, with bottom object function symbol *empty-stack*, with recognizer function symbol *stackp*, and 2 accessors: *top*, with type restriction (one-of numberp) and default value zero; *pop*, with type restriction (one-of stackp) and default value empty-stack.

```
; An irrelevant function added to illustrate user-defined
; functions:
```

DEFINITION:
concat $(x,\ y)$
$=$ **if** $x \simeq$ **nil then** y
else cons (car (x), concat (cdr (x), y)) **endif**

```
; So consider the GROUND-ZERO logic extended with the two logical
; acts above.  We will now define some functions that answer ques-
; tions about that history, e.g., what are the base function
; symbols?

; The following function returns non-F if its argument is (the quo-
; tation of) a shell base function symbol.  When it returns non-F
; it actually returns (the quotation of) the recognizer for the
; relevant shell.
```

DEFINITION:
shell-base-function (x)
$=$ **if** $x =$ `'true` **then** `'truep`
elseif $x =$ `'false` **then** `'falsep`
elseif $x =$ `'zero` **then** `'numberp`
elseif $x =$ `'empty-stack` **then** `'stackp`
else f endif

```
; The next function is the analogous one for shell constructor
; function symbols.
```

DEFINITION:
shell-constructor-function (x)
$=$ **if** $x =$ `'add1` **then** `'numberp`
elseif $x =$ `'cons` **then** `'listp`
elseif $x =$ `'pack` **then** `'litatom`
elseif $x =$ `'minus` **then** `'negativep`
elseif $x =$ `'push` **then** `'stackp`
else f endif

```
; The following function returns the list of type restrictions for
; a shell constructor.  The list is in 1:1 correspondence with the
; accessor names, i.e., the arguments of the constructor.  For ex-
; ample, for PUSH the result is '((ONE-OF NUMBERP) (ONE-OF STACKP))
; which tells us the first argument to PUSH must be a NUMBERP and
; the second must be a STACKP.
```

DEFINITION:
shell-type-restrictions (x)
$=$ **if** $(x =$ `'add1`$) \vee (x =$ `'minus`$)$
then list1 (list2 (`'one-of`, `'numberp`))
elseif $x =$ `'cons` **then** list2 (list1 (`'none-of`), list1 (`'none-of`))
elseif $x =$ `'pack` **then** list1 (list1 (`'none-of`))
elseif $x =$ `'push`
then list2 (list2 (`'one-of`, `'numberp`), list2 (`'one-of`, `'stackp`))
else f endif

```
; This function takes a shell constructor or base function symbol
; and determines whether it satisfies a given type-restriction.
; Thus, FN might be ADD1 and TYPE-RESTRICTION might be '(ONE-OF
; NUMBERP) -- in which case SATISFIES returns T.  If the type
; restriction were '(ONE-OF STACKP), SATISFIES would return F for
; FN 'ADD1.
```

DEFINITION:
satisfies (*fn*, *type-restriction*)
$=$ **if** car (*type-restriction*) $=$ `'one-of`
then if shell-base-function (*fn*) **then** shell-base-function (*fn*)
else shell-constructor-function (*fn*) **endif**
$\in$ cdr (*type-restriction*)
else if shell-base-function (*fn*) **then** shell-base-function (*fn*)
else shell-constructor-function (*fn*) **endif**
$\notin$ cdr (*type-restriction*) **endif**

```
; This function takes a list of function symbols and a list of type
; restrictions in 1:1 correspondence and determines whether all of
; the type restrictions are satisfied by the corresponding function
; symbols.
```

Definition:
all-satisfy (*fn-lst*, *type-restriction-lst*)
= **if** *fn-lst* $\simeq$ **nil** **then t**
else satisfies (car (*fn-lst*), car (*type-restriction-lst*))
$\wedge$ all-satisfy (cdr (*fn-lst*), cdr (*type-restriction-lst*)) **endif**

```
; The next two functions define the arity of each of the function
; symbols in the particular history we are considering.  We use the
; extended syntax here, namely QUOTE notation, to write down the
; alist.  Readers using this formalization as a means of mastering
; the notation should simply understand that as a result of the
; definition below, (ARITY 'IF) = 3, (ARITY 'STACKP) = 1 and (ARITY
; 'ANY-NEW-SYMBOL) = F.  That is, if (fn . n) appears in the defi-
; nition of ARITY-ALIST then (ARITY 'fn) = n.  Otherwise, (ARITY
; 'fn) = F.
```

Definition:
arity-alist
=
```
'((if . 3)
  (equal . 2)
  (count . 1)
  (false . 0)
  (falsep . 1)
  (true . 0)
  (truep . 1)
  (not . 1)
  (and . 2)
  (or . 2)
  (implies . 2)
  (add1 . 1)
  (numberp . 1)
  (sub1 . 1)
  (zero . 0)
  (lessp . 2)
  (greaterp . 2)
  (leq . 2)
  (geq . 2)
  (zerop . 1)
```

```
(fix . 1)
(plus . 2)
(pack . 1)
(litatom . 1)
(unpack . 1)
(cons . 2)
(listp . 1)
(car . 1)
(cdr . 1)
(nlistp . 1)
(minus . 1)
(negativep . 1)
(negative-guts . 1)
(difference . 2)
(times . 2)
(quotient . 2)
(remainder . 2)
(member . 2)
(iff . 2)
(ord-lessp . 2)
(ordinalp . 1)
(assoc . 2)
(pairlist . 2)
(subrp . 1)
(apply-subr . 2)
(formals . 1)
(body . 1)
(fix-cost . 2)
(strip-cars . 1)
(sum-cdrs . 1)
(v&c$ . 3)
(v&c-apply$ . 2)
(apply$ . 2)
(eval$ . 3)
(quantifier-initial-value . 1)
(add-to-set . 2)
(append . 2)
(max . 2)
(union . 2)
(quantifier-operation . 3)
(for . 6)
(push . 2)
(empty-stack . 0)
(stackp . 1)
```

```
    (top . 1)
    (pop . 2)
    (concat . 2))
```

DEFINITION:
arity (*fn*)
= **if** assoc (*fn*, ARITY-ALIST) **then** cdr (assoc (*fn*, ARITY-ALIST))
else f endif

```
; That concludes the definitions of history dependent concepts.  We
; now return to the text.

; SECTION:  QUOTE Notation and the *1*QUOTE Escape

; The essence of the QUOTE notation is the notion of ``explicit
; value descriptor.''  In the text, that notion is defined in terms
; of ``explicit value escape descriptor'' without the latter notion
; being defined.  We then define the latter notion and its defi-
; nition involves the former.  Thus, the two are mutually recur-
; sive.  We formalize the mutual recursion with our traditional FLG
; argument.  Normally, one would expect one value of the flag to
; indicate we were defining ``explicit value descriptor'' and the
; other value to indicate ``explicit value escape descriptor.''
; But it turns out that the basic form of mutual recursion here is
; ``explicit value descriptor'' versus ``list of explicit value
; descriptors'' and the notion of ``explicit value escape de-
; scriptor'' is written ``in-line.''

; So when FLG is T below, we check that X is an explicit value
; descriptor and if so we return the formal term it denotes.  We
; return F if X is not such a descriptor.  When FLG is F we check
; that X is a list of explicit value descriptors and we return
; either the list of denoted formal terms or F if X fails to be a
; list of descriptors.

; The following three events have no logical significance.  They
; do, however, permit Nqthm to process the next definition without
; inordinate delay.  We disable all function definitions except
; CADRN and CDRN.
```

EVENT: Set the status of all events. The status of each event is to be set as follows. disabled. Anything not otherwise mentioned is to be left “as-is”. Name this event ‘pre-explicit-value-descriptor’.

EVENT: Enable cadrn.

EVENT: Enable cdrn.

```
(DEFN EXPLICIT-VALUE-DESCRIPTOR (FLG X)
  (IF FLG

; Here we define what it is for X to be an explicit value de-
; scriptor and what formal term it denotes.

      (IF (NLISTP X)
          (IF (INTEGERP X)
              (IF (NUMBERP X)
                  (CORRESPONDING-NUMBERP X)
                  (CORRESPONDING-NEGATIVEP X))
          (IF (EQUAL X (STAR-ONE-STAR 'TRUE))
              (LIST1 'TRUE)
          (IF (EQUAL X (STAR-ONE-STAR 'FALSE))
              (LIST1 'FALSE)
          (IF (SYMBOLP X)
              (CORRESPONDING-LITATOM X)
              F))))
      (IF (EQUAL (CAR X) (STAR-ONE-STAR 'QUOTE))

; The test of the following IF contains the formalization of
; ``explicit value escape descriptor.''

          (IF (AND (OR (SHELL-CONSTRUCTOR-FUNCTION (CADRN 1 X))
                       (SHELL-BASE-FUNCTION (CADRN 1 X)))
               (AND (EQLEN (CDRN 2 X) (ARITY (CADRN 1 x)))
                (AND (EQUAL (CDR (LAST X)) NIL)
                 (AND (NOT (EQUAL (CADRN 1 X) 'ADD1))
                  (AND (NOT (EQUAL (CADRN 1 X) 'ZERO))
                   (AND (NOT (EQUAL (CADRN 1 X) 'CONS))
                    (AND (EXPLICIT-VALUE-DESCRIPTOR F (CDRN 2 X))
                     (AND
                      (IF (SHELL-CONSTRUCTOR-FUNCTION (CADRN 1 X))
                          (ALL-SATISFY
                           (STRIP-CARS
                            (EXPLICIT-VALUE-DESCRIPTOR F
                             (CDRN 2 X)))
```

```
                       (SHELL-TYPE-RESTRICTIONS (CADRN 1 X)))
                      T)
                  (IF (EQUAL (CADRN 1 X) 'PACK)
                      (NOT (LEGAL-CHAR-CODE-SEQ (CADRN 2 X)))
                      (IF (EQUAL (CADRN 1 X) 'MINUS)
                          (EQUAL (CADRN 2 X) (ZERO))
                          T))))))))))
      (CONS (CADRN 1 X)
    (EXPLICIT-VALUE-DESCRIPTOR F (CDRN 2 X)))
      F)
      (IF (DOTTED-PAIR X)
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (EXPLICIT-VALUE-DESCRIPTOR T (CADRN 2 X)))
      (IF (SINGLETON X)
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (CORRESPONDING-LITATOM NIL))
          (FLIST3 'CONS
                  (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                  (EXPLICIT-VALUE-DESCRIPTOR T (CDR X)))))))

; Here we define what it is for X to be a list of explicit value
; descriptors and the list of terms denoted by them.

      (IF (NLISTP X)
          NIL
          (FCONS (EXPLICIT-VALUE-DESCRIPTOR T (CAR X))
                 (EXPLICIT-VALUE-DESCRIPTOR F (CDR X))))))
```

EVENT: Set the status of all events. The status of each event is to be set as follows. enabled. Anything not otherwise mentioned is to be left "as-is". Name this event 'post-explicit-value-descriptor'.

```
; The function QT is just a convenient way to refer to the explicit
; value term denoted by an explicit value descriptor (or F if its
; argument is not such a descriptor).  Thus, (QT 'ABC) is '(PACK
; (CONS 65 (CONS 66 (CONS 67 0)))), except that the integers are
; actually ADD1 nests.  Read ``quotation'' for QT.
```

DEFINITION: $\mathrm{qt}(x) = \text{explicit-value-descriptor}(\mathbf{t}, x)$

```
; SECTION:  In Support of COND, CASE, and LET
```

```
; The next batch of functions are all involved in the translation
; of COND, CASE, and LET.  We are interested in recognizing lists
; of doublets, e.g., ((w1 v1) (w2 v2) ...), the absence of
; duplication among the wi, etc.
```

DEFINITION:
doublets (*lst*)
= **if** *lst* $\simeq$ **nil** **then** *lst* = **nil**
  else eqlen (car (*lst*), 2) $\wedge$ doublets (cdr (*lst*)) **endif**

DEFINITION:
duplicatesp (*lst*)
= **if** *lst* $\simeq$ **nil** **then** **f**
  elseif car (*lst*) $\in$ cdr (*lst*) **then** **t**
  else duplicatesp (cdr (*lst*)) **endif**

DEFINITION:
strip-cadrs (*lst*)
= **if** *lst* $\simeq$ **nil** **then** **nil**
  else cons (cadrn (1, car (*lst*)), strip-cadrs (cdr (*lst*))) **endif**

DEFINITION:
symbolps (*lst*)
= **if** *lst* $\simeq$ **nil** **then** **t**
  else symbolp (car (*lst*)) $\wedge$ symbolps (cdr (*lst*)) **endif**

```
; This function applies the substitution ALIST to TERM (FLG=T) or
; to a list of terms (FLG=F).
```

DEFINITION:
sublis-var (*flg*, *alist*, *term*)
= **if** *flg*
  then if *term* $\simeq$ **nil**
    then if assoc (*term*, *alist*) **then** cdr (assoc (*term*, *alist*))
      else *term* **endif**
    elseif car (*term*) = 'quote **then** *term*
    else cons (car (*term*), sublis-var (**f**, *alist*, cdr (*term*))) **endif**
  elseif *term* $\simeq$ **nil** **then** **nil**
  else cons (sublis-var (**t**, *alist*, car (*term*)),
      sublis-var (**f**, *alist*, cdr (*term*))) **endif**

```
; SECTION:  In Support of FOR
```

```
; The text delays the discussion of FOR statements until after V&C$
; has been presented.  We have to deal with them now.  The fol-
; lowing functions access or check certain parts of an abbreviated
; FOR.
```

DEFINITION: abbreviated-for-var (x) = cadrn (`1`, x)

DEFINITION: abbreviated-for-range (x) = cadrn (`3`, x)

DEFINITION:
abbreviated-for-when (x)
= **if** eqlen (x, `8`) **then** cadrn (`5`, x)
else `'t` **endif**

DEFINITION:
abbreviated-for-op (x)
= **if** eqlen (x, `8`) **then** cadrn (`6`, x)
else cadrn (`4`, x) **endif**

DEFINITION: abbreviated-for-body (x) = car (last (x))

DEFINITION:
for-operationp (x)
= ((x = `'add-to-set`)
∨ ((x = `'always`)
∨ ((x = `'append`)
∨ ((x = `'collect`)
∨ ((x = `'count`)
∨ ((x = `'do-return`)
∨ ((x = `'exists`)
∨ ((x = `'max`)
∨ ((x = `'sum`)
∨ ((x = `'multiply`)
∨ (x = `'union`))))))))))

```
; We now define the function that recognizes an abbreviated FOR.
```


; We now define the function that recognizes an abbreviated FOR.


```
(DEFN ABBREVIATED-FORP (X)
```

```
  (AND (LISTP X)
   (AND (EQUAL (CAR X) 'FOR)
    (AND (OR (EQLEN X 8)
     (EQLEN X 6))
     (AND (SYMBOLP (ABBREVIATED-FOR-VAR X))
      (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) NIL))
       (AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) 'T))
(AND (NOT (EQUAL (ABBREVIATED-FOR-VAR X) 'F))
 (AND (EQUAL (CADRN 2 X) 'IN)
  (AND (OR (EQLEN X 6)
   (EQUAL (CADRN 4 x) 'WHEN))
       (FOR-OPERATIONP
(ABBREVIATED-FOR-OP X))))))))))))

; To translate an abbreviated FOR we must sort the list of vari-
; ables used in the WHEN clause and the BODY.
```

Definition:
alphabetic-lessp1 ($l1$, $l2$)
= **if** $l1 \simeq$ **nil then t**
elseif $l2 \simeq$ **nil then f**
elseif car ($l1$) $<$ car ($l2$) **then t**
elseif car ($l1$) $=$ car ($l2$) **then** alphabetic-lessp1 (cdr ($l1$), cdr ($l2$))
else f endif

Definition:
alphabetic-lessp (x, y) = alphabetic-lessp1 (unpack (x), unpack (y))

Definition:
alphabetic-insert (x, l)
= **if** $l \simeq$ **nil then** list1 (x)
elseif alphabetic-lessp (x, car (l)) **then** cons (x, l)
else cons (car (l), alphabetic-insert (x, cdr (l))) **endif**

Definition:
alphabetize (l)
= **if** $l \simeq$ **nil then** l
else alphabetic-insert (car (l), alphabetize (cdr (l))) **endif**

```
; To collect the variable symbols that occur in a term (or list of
; terms) we use ALL-VARS.
```

Definition:

all-vars$(\mathit{flg},\ x)$
$= $ **if** flg
then if $x \simeq$ **nil** **then** cons$(x,$ **nil**$)$
else all-vars$($**f**$,$ cdr$(x))$ **endif**
elseif $x \simeq$ **nil** **then nil**
else all-vars$($**t**$,$ car$(x)) \cup$ all-vars$($**f**$,$ cdr$(x))$ **endif**

Definition:
standard-alist(vars)
$=$ **if** $\mathit{vars} \simeq$ **nil** **then** qt$($**nil**$)$
else list3$($`'cons`$,$
list3$($`'cons`$,$ qt$($car$(\mathit{vars})),$ car$(\mathit{vars})),$
standard-alist$($cdr$(\mathit{vars})))$ **endif**

Definition:
delete$(x,\ l)$
$=$ **if** $l \simeq$ **nil** **then** l
elseif $x =$ car(l) **then** cdr(l)
else cons$($car$(l),$ delete$(x,$ cdr$(l)))$ **endif**

Definition:
make-alist$(\mathit{var},\ \mathit{when},\ \mathit{body})$
$=$ standard-alist$($alphabetize$($delete$(\mathit{var},$
all-vars$($**t**$,\ \mathit{when})$
$\cup$ all-vars$($**t**$,\ \mathit{body}))))$

```
; The following lemmas are used in the justification of the defi-
; nition of TRANSLATE.
```

Theorem: lessp-abbreviated-for-range
$(\text{car}(x) = \texttt{'for}) \rightarrow (\text{count}(\text{abbreviated-for-range}(x)) < \text{count}(x))$

Theorem: lessp-abbreviated-for-when
$(\text{car}(x) = \texttt{'for}) \rightarrow (\text{count}(\text{abbreviated-for-when}(x)) < \text{count}(x))$

Theorem: lessp-last
$\text{count}(x) \not< \text{count}(\text{last}(x))$

Theorem: lessp-abbreviated-for-body
$(\text{car}(x) = \texttt{'for}) \rightarrow (\text{count}(\text{abbreviated-for-body}(x)) < \text{count}(x))$

Theorem: lessp-count-strip-cars
$\text{doublets}(\mathit{lst}) \rightarrow (\text{count}(\mathit{lst}) \not< \text{count}(\text{strip-cars}(\mathit{lst})))$

Theorem: lessp-count-strip-cadrs
$\text{doublets}(\mathit{lst}) \rightarrow (\text{count}(\mathit{lst}) \not< \text{count}(\text{strip-cadrs}(\mathit{lst})))$

THEOREM: listp-cddr-x-count-x
listp (cddr (x))
$\rightarrow$ (count (x)
= (1 + (1 + (count (car (x))
+ (count (cadr (x)) + count (cddr (x)))))))

THEOREM: listp-cdddr-x-count-x
listp (cdddr (x))
$\rightarrow$ (count (x)
= (1 + (1 + (1 + (count (car (x))
+ (count (cadr (x))
+ (count (caddr (x))
+ count (cdddr (x)))))))))

```
; SECTION:  Translation

; We disable all definitions except CADRN and CDRN.
```

EVENT: Set the status of all events. The status of each event is to be set as follows. disabled. Anything not otherwise mentioned is to be left "as-is". Name this event 'pre-translate'.

EVENT: Enable cadrn.

EVENT: Enable cdrn.

```
; Here, finally, is the formalization of what it is to be well-
; formed and what the translation of a well-formed term is.  If
; TRANSLATE (FLG=T) returns F, then X is not well-formed; other-
; wise, TRANSLATE (FLG=T) returns the (quotation of the) formal
; term denoted by X.  Because a formal term is either a variable
; symbol (i.e., LITATOM) or function application (i.e., LISTP), an
; answer of F unambiguously identifies the input as ill-formed.
; When FLG=F, TRANSLATE operates on a list of purported terms and
; returns either F, meaning at least one of the elements is
; ill-formed, or returns the list of their translations.
```

DEFINITION:
translate (*flg*, x)
= **if** *flg*
then if $x \simeq$ **nil**

```
then if integerp (x)  then qt (x)
     elseif symbolp (x)
     then if x = 't  then list1 ('true)
          elseif x = 'f  then list1 ('false)
          elseif x = nil  then qt (nil)
          else x endif
     else f endif
elseif dotted-s-expression (x)  then f
elseif car (x) = 'quote
then if eqlen (x, 2)  then qt (cadrn (1, x))
     else f endif
elseif car (x) = 'cond
then if eqlen (x, 2)
        ∧  (eqlen (cadrn (1, x), 2)
            ∧  (car (cadrn (1, x)) = 't))
     then translate (t, cadrn (1, cadrn (1, x)))
     elseif eqlen (cadrn (1, x), 2)
            ∧  ((car (cadrn (1, x)) ≠ 't)
                ∧  listp (cdrn (2, x)))
     then flist4 ('if,
                  translate (t, car (cadrn (1, x))),
                  translate (t, cadrn (1, cadrn (1, x))),
                  translate (t, cons ('cond, cdrn (2, x))))
     else f endif
elseif car (x) = 'case
then if eqlen (x, 3)
        ∧  (eqlen (cadrn (2, x), 2)
            ∧  ((car (cadrn (2, x)) = 'otherwise)
                ∧  translate (t, cadrn (1, x))))
     then translate (t, cadrn (1, cadrn (2, x)))
     elseif eqlen (cadrn (2, x), 2)
            ∧  (listp (cdrn (3, x))
                ∧  (car (cadrn (2, x))
                    ∉  strip-cars (cdrn (3, x))))
     then flist4 ('if,
                  flist3 ('equal,
                          translate (t, cadrn (1, x)),
                          qt (car (cadrn (2, x)))),
                  translate (t, cadrn (1, cadrn (2, x))),
                  translate (t,
                             cons ('case,
                                   cons (cadrn (1, x), cdrn (3, x)))))
     else f endif
elseif car (x) = 'let
```

```
then if doublets (cadrn (1, x))
        then if eqlen (x, 3)
                   ∧   (translate (f, strip-cars (cadrn (1, x)))
                         ∧   (translate (f,
                                              strip-cadrs (cadrn (1, x)))
                               ∧   (translate (t, cadrn (2, x))
                                     ∧   (symbolps (translate (f,
                                                                strip-cars (cadrn (1,
                                                                                   x))))
                                           ∧   (¬ duplicatesp (translate (f,
                                                                          strip-cars (cadrn (1,
                                                                                             x)))))))))
                then sublis-var (t,
                                 pairlist (translate (f,
                                                      strip-cars (cadrn (1,
                                                                         x))),
                                           translate (f,
                                                      strip-cadrs (cadrn (1,
                                                                          x)))),
                                 translate (t, cadrn (2, x)))
                else f endif
        else f endif
elseif ¬ translate (f, cdr (x))  then f
elseif (car (x) = nil)
         ∨   ((car (x) = 't) ∨ (car (x) = 'f))  then f
elseif car (x) = 'list
then fn-nest ('cons, translate (f, cdr (x)), qt (nil))
elseif car (x) = 'list*
then if eqlen (x, 1)  then f
        else fn-nest ('cons,
                      all-but-last (translate (f, cdr (x))),
                      car (last (translate (f, cdr (x))))) endif
elseif car-cdr-symbolp (car (x))
then if eqlen (x, 2)
        then car-cdr-nest (a-d-sequence (car (x)),
                           translate (t, cadrn (1, x)))
        else f endif
elseif eqlen (cdr (x), arity (car (x)))
then fcons (car (x), translate (f, cdr (x)))
elseif car (x) = 'for
then if abbreviated-forp (x)
        then flist7 ('for,
                     qt (abbreviated-for-var (x)),
                     translate (t, abbreviated-for-range (x)),
```

qt (translate (**t**, abbreviated-for-when (x))),
qt (abbreviated-for-op (x)),
qt (translate (**t**, abbreviated-for-body (x))),
make-alist (abbreviated-for-var (x),
translate (**t**, abbreviated-for-when (x)),
translate (**t**, abbreviated-for-body (x))))
else f endif
elseif (2 < length (cdr (x)))
∧ ((car (x) = `'and`)
∨ ((car (x) = `'or`)
∨ ((car (x) = `'plus`)
∨ (car (x) = `'times`))))
then fn-nest (car (x),
all-but-last (translate (**f**, cdr (x))),
car (last (translate (**f**, cdr (x)))))
else f endif
elseif $x \simeq$ **nil then nil**
else fcons (translate (**t**, car (x)), translate (**f**, cdr (x))) **endif**

EVENT: Set the status of all events. The status of each event is to be set as follows. enabled. Anything not otherwise mentioned is to be left "as-is". Name this event 'post-translate'.

```
; SECTION:  The Extended Syntax

; Finally, here is EXSYN, which reads an s-expression from a stream
; of ASCII character codes and translates it into a formal term or
; returns F if the stream is not the display of a term in the ex-
; tended syntax.
```

DEFINITION:
exsyn (*stream*)
= **if** readable (read-token-tree (*stream*), `0`)
then translate (**t**, readmacro-expansion (read-token-tree (*stream*)))
else f endif

```
; EXSYN returns F if the stream cannot be parsed.  The explanation
; of this remark is that an unparsable stream causes READ-TOKEN-
; TREE to return F and READABLE returns F on that input.

; SECTION:  Slightly Abbreviated Formal Terms

; It is exceedingly difficult to read the output of TRANSLATE and
```

```
; EXSYN because quoted literal atoms and numbers are exploded.
; Thus 'ABC translates to (PACK (CONS (ADD1 ...)  ...))  where the
; ellipses are very large nests of CONSes and ADD1s.  Below, we
; develop a function that can be used to massage the output of
; TRANSLATE to introduce QUOTE notation for LITATOMs and to intro-
; duce the normal decimal representation for ADD1 nests.  While
; this convention is employed in Chapter 4, e.g., when we exhibit
; two token trees with the same translation, the concepts
; formalized below are not defined in the text.

; If X is an ADD1-nest n deep with a (ZERO) at the bottom, we
; return n; otherwise F.  The variable I is used as an accumulator
; and should be 0 at the top-level.
```

DEFINITION:
add1-nestp $(x,\ i)$
= **if** $x \simeq$ **nil then f**
elseif (car (x) = `'zero`) $\wedge$ eqlen $(x,$ `1`$)$ **then** i
elseif (car (x) = `'add1`) $\wedge$ eqlen $(x,$ `2`$)$
then add1-nestp (cadrn (`1`, x), $1 + i$)
else f endif

DEFINITION:
cons-add1-nestp (x)
= **if** $x \simeq$ **nil then f**
elseif (car (x) = `'zero`) $\wedge$ eqlen $(x,$ `1`$)$ **then** `0`
elseif (car (x) = `'cons`) $\wedge$ eqlen $(x,$ `3`$)$
then fcons (add1-nestp (cadrn (`1`, x), `0`), cons-add1-nestp (cadrn (`2`, x)))
else f endif

DEFINITION:
exploded-litatom (x)
= **if** eqlen $(x,$ `2`$)$
$\wedge$ ((car (x) = `'pack`) $\wedge$ cons-add1-nestp (cadrn (`1`, x)))
then list2 (`'quote`, pack (cons-add1-nestp (cadrn (`1`, x))))
else f endif

DEFINITION:
abbrev (*flg*, x)
= **if** *flg*
then if $x \simeq$ **nil then** x
elseif add1-nestp $(x,$ `0`$)$ **then** add1-nestp $(x,$ `0`$)$
elseif exploded-litatom (x) **then** exploded-litatom (x)
else cons (car (x), abbrev (**f**, cdr (x))) **endif**

elseif $x \simeq$ **nil then nil**
else cons (abbrev (**t**, car (x)), abbrev (**f**, cdr (x))) **endif**

```
; Here is a version of EXSYN that uses abbreviations.
```

DEFINITION:
aexsyn (*stream*)
= **if** readable (read-token-tree (*stream*), 0)
then abbrev (**t**,
translate (**t**, readmacro-expansion (read-token-tree (*stream*))))
else f endif

```
#|

Here are Common Lisp interfaces to EXSYN and AEXSYN.

 (defun exsyn (string)
  (*1*exsyn (ascii string)))

 (defun aexsyn (string)
  (*1*aexsyn (ascii string)))

|#

; We conclude by making a compiled library containing the current
; data base.  If one executes (NOTE-LIB "../examples/basic/parser"
; T) in Nqthm, where the ellipsis is meant to be the local
; directory containing our examples subdirectory, then one can use
; R-LOOP to execute these function.  For example, one can type to
; R-LOOP:

;(AEXSYN
; (ASCII
; '(OPEN-PAREN
;   LOWER-A LOWER-D LOWER-D DIGIT-ONE
;   SPACE
;   NUMBER-SIGN VERTICAL-BAR
;   UPPER-C LOWER-O LOWER-M LOWER-M LOWER-E LOWER-N LOWER-T
;   VERTICAL-BAR NUMBER-SIGN
;   LOWER-X
;   CLOSE-PAREN)))

; and get the result '(ADD X).
```

```
; This is sufficiently cumbersome that we find the Lisp interface
; functions much more convenient.  If the ``defuns'' in this file
; are executed in an acceptable Common Lisp, then it is possible to
; type to Common Lisp (rather than R-LOOP):

; (aexsyn "(add1 #|Comment|#x)")

; and get the result (ADD1 X).  The Lisp routine aexsyn actually
; executes our logical function AEXSYN but it first converts the
; string argument into a list of ASCII characters.
```

EVENT: Make the library "parser" and compile it.

Index