```
; Copyright (C) 1995 by Ken Kunen.  All Rights Reserved.

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

; NO WARRANTY

; Ken Kunen 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 Ken Kunen 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.



; This script is described in Section 4.2 of our paper,
; Non-Constructive Computational Mathematics.
; It includes three items

; ITEM 1: We show that induction on omega^2 is provable in PRA,
; even though there are non-primitive-recursive functions definable
; by recursion on omega^2
; ITEM 2: We use eval$ and v&c$, but not ordinal induction, and
; define the Ackermann function (and prove that the definition is correct).
; ITEM 3: We define a function (next-twin-prime x) and show that either there
; are no twin primes above x or next-twin-prime produces one.
; This is a simple demonstration of the non-constructiveness of eval$ and v&C$
```

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                          ITEM 1                                ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Represent ordinals below omega^2 as omega X omega, ordered lexically
```

```
; Elements of omega X omega are represented as pairs (m . n)
```

DEFINITION:
$\mathrm{ordp}\,(p) = (\mathrm{listp}\,(p) \wedge (\mathrm{car}\,(p) \in \mathbf{N}) \wedge (\mathrm{cdr}\,(p) \in \mathbf{N}))$

DEFINITION:
$\mathrm{lexp}\,(p1,\, p2)$
$=\quad ((\mathrm{car}\,(p1) < \mathrm{car}\,(p2))$
$\quad\ \vee\quad ((\mathrm{car}\,(p1) = \mathrm{car}\,(p2)) \wedge (\mathrm{cdr}\,(p1) < \mathrm{cdr}\,(p2))))$

```
; now let Q be any property of pairs:
```

EVENT: Introduce the function symbol q of one argument.

```
; Assume Q is inductive -- that is:
; NOT Q(p)  --> EXISTS r < p NOT Q(r)
; Of course, to say this in the framework of PRA, we have to assume
; we have a function which gives us an r
```

EVENT: Introduce the function symbol g of one argument.

AXIOM: q-induction
$(\mathrm{ordp}\,(p) \wedge (\neg\ \mathrm{q}\,(p)))$
$\rightarrow\quad (\mathrm{ordp}\,(\mathrm{g}\,(p)) \wedge \mathrm{lexp}\,(\mathrm{g}\,(p),\, p) \wedge (\neg\ \mathrm{q}\,(\mathrm{g}\,(p))))$

```
; Now,  we show Q is true everywhere :   (implies (ordp p) (Q p))
; by ordinary induction

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; First,  by ordinary induction on y (holding x fixed):
; Let fy = (fy x y) <= y be least such that (not (Q (x . fy)))
; fy = F if there is no such y
```

DEFINITION:
$\mathrm{fy}\,(x,\, y)$
$=\quad$ **if** $y \notin \mathbf{N}$ **then f**
elseif $y = 0$
then if $\mathrm{q}\,(\mathrm{cons}\,(x,\, y))$ **then f**
else y **endif**
elseif $\mathrm{fy}\,(x,\, y-1)$ **then** $\mathrm{fy}\,(x,\, y-1)$
elseif $\mathrm{q}\,(\mathrm{cons}\,(x,\, y))$ **then f**
else y **endif**

THEOREM: q-fails-at-x-fy
$\mathrm{fy}\,(x,\ y) \rightarrow (\neg\ \mathrm{q}\,(\mathrm{cons}\,(x,\ \mathrm{fy}\,(x,\ y))))$

THEOREM: fy-is-a-number
$\mathrm{fy}\,(x,\ y) \rightarrow (\mathrm{fy}\,(x,\ y) \in \mathbf{N})$

THEOREM: fy-is-defined
$((\neg\ \mathrm{q}\,(\mathrm{cons}\,(x,\ y))) \wedge (y \in \mathbf{N})) \rightarrow \mathrm{fy}\,(x,\ y)$

THEOREM: fy-leq
$\mathrm{fy}\,(x,\ y) \rightarrow (\mathrm{fy}\,(x,\ y) \leq y)$

THEOREM: fy-lessp
$(\mathrm{fy}\,(x,\ y) \wedge \mathrm{q}\,(\mathrm{cons}\,(x,\ y))) \rightarrow (\mathrm{fy}\,(x,\ y) < y)$

THEOREM: q-below-undefined-fy
$((\neg\ \mathrm{fy}\,(x,\ y)) \wedge (y \in \mathbf{N}) \wedge (y1 \in \mathbf{N}) \wedge (y1 \leq y)) \rightarrow \mathrm{q}\,(\mathrm{cons}\,(x,\ y1))$

THEOREM: fy-is-first
$(\mathrm{fy}\,(x,\ y) \wedge (y1 \in \mathbf{N}) \wedge (y \in \mathbf{N}) \wedge (y1 < \mathrm{fy}\,(x,\ y))) \rightarrow \mathrm{q}\,(\mathrm{cons}\,(x,\ y1))$

```
; now, summarize what the Q-induction says for pairs:
```

THEOREM: form-of-g
$$\begin{array}{ll}
((x \in \mathbf{N}) & \\
\wedge & (y \in \mathbf{N}) \\
\wedge & (\neg\ \mathrm{q}\,(\mathrm{cons}\,(x,\ y))) \\
\wedge & (x1 = \mathrm{car}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\ y)))) \\
\wedge & (y1 = \mathrm{cdr}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\ y))))) \\
\rightarrow & ((x1 \in \mathbf{N}) \wedge (y1 \in \mathbf{N}) \wedge (\mathrm{g}\,(\mathrm{cons}\,(x,\ y)) = \mathrm{cons}\,(x1,\ y1)))
\end{array}$$

THEOREM: g-is-below
$$\begin{array}{ll}
((x \in \mathbf{N}) & \\
\wedge & (y \in \mathbf{N}) \\
\wedge & (\neg\ \mathrm{q}\,(\mathrm{cons}\,(x,\ y))) \\
\wedge & (x1 = \mathrm{car}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\ y)))) \\
\wedge & (y1 = \mathrm{cdr}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\ y))))) \\
\rightarrow & ((x1 < x) \vee ((x1 = x) \wedge (y1 < y)))
\end{array}$$

THEOREM: q-fails-at-g-aux1
$$\begin{array}{ll}
((\neg\ \mathrm{q}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\ y)))) \wedge (\mathrm{g}\,(\mathrm{cons}\,(x,\ y)) = \mathrm{cons}\,(x1,\ y1))) \\
\rightarrow & (\neg\ \mathrm{q}\,(\mathrm{cons}\,(x1,\ y1)))
\end{array}$$

THEOREM: q-fails-at-g
$((x \in \mathbf{N})$
$\wedge \quad (y \in \mathbf{N})$
$\wedge \quad (\neg\, \mathrm{q}\,(\mathrm{cons}\,(x,\, y)))$
$\wedge \quad (x1 = \mathrm{car}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, y))))$
$\wedge \quad (y1 = \mathrm{cdr}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, y)))))$
$\rightarrow \quad (\neg\, \mathrm{q}\,(\mathrm{cons}\,(x1,\, y1)))$

EVENT: Disable q-fails-at-g-aux1.

```
; Now, if Q fails at x,y then Q fails at x,fy.  If we
; obtain x1,y1 from x,fy as above, we can't have
; (and (equal x1 x) (lessp y1 fy)), so we must have x1 < x
```

THEOREM: smaller-x
$((x \in \mathbf{N})$
$\wedge \quad (y \in \mathbf{N})$
$\wedge \quad (\neg\, \mathrm{q}\,(\mathrm{cons}\,(x,\, y)))$
$\wedge \quad (x1 = \mathrm{car}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, \mathrm{fy}\,(x,\, y)))))$
$\wedge \quad (y1 = \mathrm{cdr}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, \mathrm{fy}\,(x,\, y))))))$
$\rightarrow \quad ((x1 \in \mathbf{N}) \wedge (y1 \in \mathbf{N}) \wedge (x1 < x) \wedge (\neg\, \mathrm{q}\,(\mathrm{cons}\,(x1,\, y1))))$

```
; It is immediate from this that Q can never fail
```

DEFINITION:
q-holds-kludge $(x,\, y)$
$=$ **if** $(x \in \mathbf{N})$
$\wedge \quad (y \in \mathbf{N})$
$\wedge \quad (\neg\, \mathrm{q}\,(\mathrm{cons}\,(x,\, y)))$
$\wedge \quad (\mathrm{car}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, \mathrm{fy}\,(x,\, y)))) \in \mathbf{N})$
$\wedge \quad (\mathrm{car}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, \mathrm{fy}\,(x,\, y)))) < x)$
then q-holds-kludge $(\mathrm{car}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, \mathrm{fy}\,(x,\, y)))),\, \mathrm{cdr}\,(\mathrm{g}\,(\mathrm{cons}\,(x,\, \mathrm{fy}\,(x,\, y)))))$
else `0` **endif**

THEOREM: q-holds
$((x \in \mathbf{N}) \wedge (y \in \mathbf{N})) \rightarrow \mathrm{q}\,(\mathrm{cons}\,(x,\, y))$

```
; Finally, re-expressing this in terms of pairs:
```

THEOREM: q-is-true
$\mathrm{ordp}\,(p) \rightarrow \mathrm{q}\,(p)$

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                          ITEM 2                                ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; 1. We define qval (quick valuation).  This is a version of v&c$ which
;    resets the cost to 0.  One problem with v&c$ is that it explicitly
;    keeps track of the cost.  That makes proving theorems about
;    v&c$ working correctly on specific terms needlessly complicated.
; 2. We re-state the recursive definition of v&c$ in terms of qval.
; 3. We show how to formalize double recursions, such as occur in the
;    definition of the Ackermann function.
```

```
; Since v&c$ uses the value of F to signify non-termination:
```

DEFINITION:
reset (x)
= **if** x **then** cons (car (x), 0)
else f endif

DEFINITION: qval $(term, va)$ = reset (v&c$ (**t**, $term$, va))

```
; va (a "variable assignment") is an association list which
; assigns values to variables.
; qval returns (val . 0) if the computation halts in the normal
; way to return val; otherwise, qval returns F
```

DEFINITION:
qval-list $(list\text{-}of\text{-}terms, va)$
= **if** $list\text{-}of\text{-}terms \simeq$ **nil then nil**
else cons (qval (car $(list\text{-}of\text{-}terms)$, va),
qval-list (cdr $(list\text{-}of\text{-}terms)$, va)) **endif**

```
; Now, unwind the v&c$ axioms so that they are statements about
; qval and qval-list, and then disable qval and qval-list
```

```
; first, the two cases for qval-list:
```

THEOREM: qval-list-empty
$(lst \simeq$ **nil**$) \rightarrow$ (qval-list (lst, va) = **nil**)

THEOREM: qval-list-non-empty
listp (lst)
$\rightarrow$ (qval-list (lst, va) = cons (qval (car (lst), va), qval-list (cdr (lst), va)))

; in proving the cases for qval, it will be useful to have:

THEOREM: same-f-membership
(**f** ∈ qval-list (*lst*, *va*)) ↔ (**f** ∈ v&c$ ('list, *lst*, *va*))

THEOREM: same-strip-cars
strip-cars (qval-list (*lst*, *va*)) = strip-cars (v&c$ ('list, *lst*, *va*))

EVENT: Disable qval-list.

; now, run down all the cases for qval:

; for a variable name, look up in assoc list

THEOREM: qval-litatom
litatom (*term*) → (qval (*term*, *va*) = cons (cdr (assoc (*term*, *va*)), 0))

; for a constant, return that constant value:

THEOREM: qval-constant
((¬ litatom (*term*)) ∧ (*term* ≃ **nil**)) → (qval (*term*, *va*) = cons (*term*, 0))

; for a quoted expression, return the expression

THEOREM: qval-quote
qval (list ('quote, *object*), *va*) = cons (*object*, 0)

; evaluating an (if ...):

THEOREM: qval-if
qval (list ('if, *test*, *thencase*, *elsecase*), *va*)
= **if** qval (*test*, *va*)
then if car (qval (*test*, *va*)) **then** qval (*thencase*, *va*)
else qval (*elsecase*, *va*) **endif**
else f endif

; except for (if ...) and (quote ...) : if evaluation of some
; arg doesn't halt, then qval doesn't halt.

THEOREM: qval-args-dont-halt-aux1
((*fn* ≠ 'quote) ∧ (*fn* ≠ 'if) ∧ (**f** ∈ v&c$ ('list, *args*, *va*)))
→ (v&c$ (**t**, cons (*fn*, *args*), *va*) = **f**)

THEOREM: qval-args-dont-halt
((*fn* ≠ 'quote) ∧ (*fn* ≠ 'if) ∧ (**f** ∈ qval-list (*args*, *va*)))
→ (qval (cons (*fn*, *args*), *va*) = **f**)

```
; for a SUBR (a basic built-in) other than "if": if evaluation of the args
; does halt, just use APPLY-SUBR on the value of args
; STRIP-CARS applied to ( (val1 . 0) (val2 . 0) ... ) returns (val1 val2 ...)
```

THEOREM: qval-subr-aux1
$((\mathbf{f} \notin \text{v\&c\$}\,(\texttt{'list},\ args,\ va)) \wedge (fn \neq \texttt{'if}) \wedge \text{subrp}\,(fn))$
$\rightarrow\ (\text{v\&c\$}\,(\mathbf{t},\ \text{cons}\,(fn,\ args),\ va)$
$=\ \text{cons}\,(\text{apply-subr}\,(fn,\ \text{strip-cars}\,(\text{v\&c\$}\,(\texttt{'list},\ args,\ va))),$
$1 + \text{sum-cdrs}\,(\text{v\&c\$}\,(\texttt{'list},\ args,\ va))))$

```
; the following auxiliary is trivial,
; but the prover hangs on it:
```

THEOREM: qval-subr-aux2
$((vvv = \text{cons}\,(uu,\ anything)) \wedge a \wedge b \wedge c \wedge d \wedge e)$
$\rightarrow\ (\text{car}\,(vvv) = uu)$

THEOREM: qval-subr
$((\mathbf{f} \notin \text{qval-list}\,(args,\ va)) \wedge (fn \neq \texttt{'if}) \wedge \text{subrp}\,(fn))$
$\rightarrow\ (\text{qval}\,(\text{cons}\,(fn,\ args),\ va)$
$=\ \text{cons}\,(\text{apply-subr}\,(fn,\ \text{strip-cars}\,(\text{qval-list}\,(args,\ va))),\ \texttt{0}))$

EVENT: Disable qval-subr-aux2.

```
; for a non-SUBR (a basic built-in) other than "quote":
; if evaluation of the args does halt, just evaluate
; the BODY of the function on the
; FORMALS of the function, paired with the value of the args
```

THEOREM: qval-non-subr-aux1
$((fn \neq \texttt{'quote}) \wedge (\mathbf{f} \notin \text{v\&c\$}\,(\texttt{'list},\ args,\ va)) \wedge (\neg\ \text{subrp}\,(fn)))$
$\rightarrow\ (\text{v\&c\$}\,(\mathbf{t},\ \text{cons}\,(fn,\ args),\ va)$
$=\ \text{fix-cost}\,(\text{v\&c\$}\,(\mathbf{t},$
$\text{body}\,(fn),$
$\text{pairlist}\,(\text{formals}\,(fn),$
$\text{strip-cars}\,(\text{v\&c\$}\,(\texttt{'list},\ args,\ va)))),$
$1 + \text{sum-cdrs}\,(\text{v\&c\$}\,(\texttt{'list},\ args,\ va))))$

```
; again, the prover hangs on a trivial point
```

THEOREM: qval-non-subr-aux2
$(x1 \wedge (v1 = \text{cons}\,(w,\ x2)) \wedge x3 \wedge x4 \wedge x5 \wedge x6 \wedge x7)$
$\rightarrow\ (\text{car}\,(v1) = w)$

THEOREM: qval-non-subr
$((\mathbf{f} \notin \text{qval-list}\,(\mathit{args},\ \mathit{va})) \wedge (\mathit{fn} \neq \texttt{'quote}) \wedge (\neg\ \text{subrp}\,(\mathit{fn})))$
$\rightarrow\quad (\text{qval}\,(\text{cons}\,(\mathit{fn},\ \mathit{args}),\ \mathit{va})$
$\quad = \quad \text{qval}\,(\text{body}\,(\mathit{fn}),$
$\quad\quad\quad\quad \text{pairlist}\,(\text{formals}\,(\mathit{fn}),\ \text{strip-cars}\,(\text{qval-list}\,(\mathit{args},\ \mathit{va})))))$

EVENT: Disable qval-non-subr-aux2.

```
; from now on, we need only the properties of qval and qval-list,
; not how they were defined:
```

EVENT: Disable qval.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; We illustrate the method by showing how to justify a double
; recursion, such as the definition of the Ackermann function:
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Values of Ackermann Function
;
;    --------------------
;  3| 1   2   4  16  2^16
;   |--------------------
;  2| 1   2   4   8   16
;   |--------------------
;  1| 1   2   4   6   8
;   |--------------------
;  0| 1   2   4   5   6
;   |--------------------
;     0   1   2   3   4
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; We can think of row 0 as given.  Row n+1 is obtained by
; iterating the function on row n, starting with a 1 in column 0.
```

DEFINITION:
row0 (x)

= **if** $x \simeq$ 0 **then** 1
elseif $x =$ 1 **then** 2
else $x +$ 2 **endif**

DEFINITION:
row1 (x)
= **if** $x \simeq$ 0 **then** 1
else row0 (row1 $(x - 1)$) **endif**

DEFINITION:
row2 (x)
= **if** $x \simeq$ 0 **then** 1
else row1 (row2 $(x - 1)$) **endif**

DEFINITION:
row3 (x)
= **if** $x \simeq$ 0 **then** 1
else row2 (row3 $(x - 1)$) **endif**

```
; GOAL: define a function, ACKERMANN, and prove
; (prove-lemma ackermann-works (rewrite) (equal (ackermann x y)
;  (if (zerop x) 1
;  (if (zerop y) (if (equal x 1) 2 (plus x 2))
;  (ackermann (ackermann (sub1 x) y) (sub1 y))  ))))

; Without using ordinals and ord-lessp, we can't just use this
; as a definition.  However, we can use EVAL$ to define
; function whose body looks like this.  To simplify the work, we
; phrase the defn of ack so that the body is as simple as possible
```

DEFINITION: base $(x, y) = ((x \simeq 0) \vee (y \simeq 0))$

DEFINITION:
base-fn (x, y)
= **if** $y \simeq$ 0 **then** row0 (x)
else 1 **endif**

```
; intended defn of ack is now:
;(defn ack (x y) (if (base x y) (base-fn x y)  (ack (ack (sub1 x) y) (sub1 y))))

; instead, use eval$ :
```

DEFINITION:
ack (x, y)

= eval$ (**t**,
'(if
(base x y)
(base-fn x y)
(ack (ack (sub1 x) y) (sub1 y))),
list (cons ('x, x), cons ('y, y)))

```
;;;;;;;;;;;;;;;;;;;;;;  begin r-loop

; *(body 'ack)
;  '(IF (BASE X Y)
;       (BASE-FN X Y)
;       (ACK (ACK (SUB1 X) Y) (SUB1 Y)))
; *(formals 'ack)
;  '(X Y)
; *(ack 3 1)
;  6
; *(ack 2 3)
;  4
; *(ack 4 2)
;  16
; *(qval '(ack x y) '( (x . 4) (y . 2) ))
;  '(16 . 0)
;;;;;;;;;;;;;;;;;;;;;;  end r-loop

; Now that we have arranged for ack to have the correct body and formals,
; we never use eval$ again.  We can define the official Ackermann function
; using qval, and then prove inductively that it works.
; Since the formals will always be '(x y), we can simplify
; some of the general notation:
```

DEFINITION:
assn ($valx$, $valy$) = list (cons ('x, $valx$), cons ('y, $valy$))

DEFINITION: a ($valx$, $valy$) = qval ('(ack x y), assn ($valx$, $valy$))

DEFINITION: ackermann ($valx$, $valy$) = car (a ($valx$, $valy$))

```
; Proving that ackermann works involves proving first that (A valx valy)
; holds (i.e., it is never F).

; First, since qval-list will always be operating on lists of terms of
; length two, let's concretely unwind that situation:
```

THEOREM: qval-list-1
qval-list (list (*term*), *va*) = list (qval (*term*, *va*))

THEOREM: qval-list-2
qval-list (list (*term1*, *term2*), *va*) = list (qval (*term1*, *va*), qval (*term2*, *va*))

EVENT: Disable qval-list-non-empty.

```
; Now, analyze how qval works on sub-expressions of the body:
;   (if (base x y) (base-fn x y) (ack (ack (sub1 x) y) (sub1 y)))
```

THEOREM: qval-on-x
qval ('x, assn (*valx*, *valy*)) = cons (*valx*, 0)

THEOREM: qval-on-y
qval ('y, assn (*valx*, *valy*)) = cons (*valy*, 0)

THEOREM: qval-on-sub1-x
qval ('(sub1 x), assn (*valx*, *valy*)) = cons (*valx* − 1, 0)

THEOREM: qval-on-sub1-y
qval ('(sub1 y), assn (*valx*, *valy*)) = cons (*valy* − 1, 0)

THEOREM: qval-on-base
qval ('(base x y), assn (*valx*, *valy*)) = cons (base (*valx*, *valy*), 0)

THEOREM: qval-on-base-fn
qval ('(base-fn x y), assn (*valx*, *valy*)) = cons (base-fn (*valx*, *valy*), 0)

```
; on the whole body
```

THEOREM: qval-on-body
qval ('(if
(base x y)
(base-fn x y)
(ack (ack (sub1 x) y) (sub1 y))),
assn (*valx*, *valy*))
= **if** base (*valx*, *valy*) **then** cons (base-fn (*valx*, *valy*), 0)
else qval ('(ack (ack (sub1 x) y) (sub1 y)),
assn (*valx*, *valy*)) **endif**

```
; Now on the ack expressions, we need two things:
; 1. Unwind the (qval '(ack x y) (assn valx valy)) in
;    the defn of (A valx valy) so we can prove thms about A
; 2. Reduce qval of the two ack expressions in the body:
```

```
;       (ack (sub1 x) y)       (ack (ack (sub1 x) y) (sub1 y))
;    to a qval of (ack x y)

; towards (2):
```

THEOREM: qval-of-ack-of-terms-aux1
(qval (*term1*, *va*) ∧ qval (*term2*, *va*))
→ (qval (list ('ack, *term1*, *term2*), *va*)
= qval ('(if
(base x y)
(base-fn x y)
(ack (ack (sub1 x) y) (sub1 y))),
assn (car (qval (*term1*, *va*)), car (qval (*term2*, *va*)))))

```
; as a special case, for (ack x y):
```

THEOREM: qval-of-ackxy-aux1
qval ('(ack x y), assn (*valx*, *valy*))
= qval ('(if
(base x y)
(base-fn x y)
(ack (ack (sub1 x) y) (sub1 y))),
assn (*valx*, *valy*))

```
; we can unwind the if expr later -- first, let's reduce the
; (ack term1 term2).
```

THEOREM: qval-of-ack-of-terms
(qval (*term1*, *va*) ∧ qval (*term2*, *va*))
→ (qval (list ('ack, *term1*, *term2*), *va*)
= qval ('(ack x y),
assn (car (qval (*term1*, *va*)), car (qval (*term2*, *va*)))))

EVENT: Disable qval-of-ack-of-terms-aux1.

```
; Now, unwinding the if expr:
```

THEOREM: qval-of-ackxy
qval ('(ack x y), assn (*valx*, *valy*))
= **if** base (*valx*, *valy*) **then** cons (base-fn (*valx*, *valy*), 0)
else qval ('(ack (ack (sub1 x) y) (sub1 y)),
assn (*valx*, *valy*)) **endif**

EVENT: Disable qval-of-ackxy-aux1.

```
; now, qval-of-ackxy would be simpler if we split it into
; the two cases base/non-base
```

THEOREM: qval-of-ackxy-base
base (*valx*, *valy*)
→ (qval (’(ack x y), assn (*valx*, *valy*)) = cons (base-fn (*valx*, *valy*), 0))

THEOREM: qval-of-ackxy-non-base-aux1
(¬ base (*valx*, *valy*))
→ (qval (’(ack x y), assn (*valx*, *valy*))
= qval (’(ack (ack (sub1 x) y) (sub1 y)),
assn (*valx*, *valy*)))

```
; we have to analyze this further
```

EVENT: Disable qval-of-ackxy.

```
; Now, we have to reduce the two ack expressions in the body:
;      (ack (sub1 x) y)       (ack (ack (sub1 x) y) (sub1 y))
;         small                         big
```

THEOREM: qval-of-ack-small
qval (’(ack (sub1 x) y), assn (*valx*, *valy*))
= qval (’(ack x y), assn (*valx* − 1, *valy*))

THEOREM: qval-of-ack-big
qval (’(ack x y), assn (*valx* − 1, *valy*))
→ (qval (’(ack (ack (sub1 x) y) (sub1 y)), assn (*valx*, *valy*))
= qval (’(ack x y),
assn (car (qval (’(ack (sub1 x) y), assn (*valx*, *valy*))),
valy − 1)))

EVENT: Disable qval-of-ack-of-terms.

```
; we only need it for "small" and "big"

; using these, we can analyze the non-base case
```

THEOREM: qval-of-ackxy-non-base
$((\neg\ \text{base}\,(valx,\ valy)) \wedge \text{qval}\,(\texttt{'(ack x y)},\ \text{assn}\,(valx - 1,\ valy)))$
$\rightarrow\quad (\text{qval}\,(\texttt{'(ack x y)},\ \text{assn}\,(valx,\ valy))$
$\quad = \quad \text{qval}\,(\texttt{'(ack x y)},$
$\text{assn}\,(\text{car}\,(\text{qval}\,(\texttt{'(ack x y)},\ \text{assn}\,(valx - 1,\ valy))),$
$valy - 1)))$

EVENT: Disable qval-of-ackxy-non-base-aux1.

EVENT: Disable assn.

```
; this is no longer needed, and it really slows things
; down when assn is unwound

; Now, let's prove : (A valx valy) by induction on (valx valy).
; Then -- we can remove the (qval '(ack x y) ...) hypotheses
```

THEOREM: a-is-true-aux1
$\text{base}\,(valx,\ valy) \rightarrow \text{a}\,(valx,\ valy)$

```
; specifically, there is only a problem if valx or valy are non-zero
```

THEOREM: a-is-true-aux2
$(valx \simeq \texttt{0}) \rightarrow \text{a}\,(valx,\ valy)$

THEOREM: a-is-true-aux3
$(valy \simeq \texttt{0}) \rightarrow \text{a}\,(valx,\ valy)$

```
; let's give a name to the x-value called in qval-of-ackxy-non-base
```

DEFINITION:
$\text{prev}\,(valx,\ valy) = \text{car}\,(\text{qval}\,(\texttt{'(ack x y)},\ \text{assn}\,(valx - 1,\ valy)))$

```
; this will make the induction simpler
```

THEOREM: a-is-true-aux4
$((\neg\ \text{base}\,(valx,\ valy)) \wedge \text{a}\,(valx - 1,\ valy) \wedge \text{a}\,(\text{prev}\,(valx,\ valy),\ valy - 1))$
$\rightarrow\quad \text{a}\,(valx,\ valy)$

EVENT: Disable a.

```
; temporarily
```

EVENT: Disable prev.

```
; permanently

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; We have enough now to prove that (A valx valy) is true by double induction,
; as in ITEM 1, above.
; If (A valx valy) fails, then (A valx' valy') fails, where
; (valy' valx') is lexically earlier than (valy valx).

; First,  by ordinary induction on valx (holding valy fixed):
; if (not (A valx valy)), there is a first-value,
; fx = (fx valx valy), such that: (not (A fx valy)) but (A (sub1 fx) valy)
```

DEFINITION:
fx (*valx*, *valy*)
= **if** *valx* ≃ 0 **then f**
elseif (¬ a (*valx*, *valy*)) ∧ a (*valx* − 1, *valy*) **then** *valx*
else fx (*valx* − 1, *valy*) **endif**

THEOREM: fx-is-first-1
(¬ a (*valx*, *valy*)) → (¬ a (fx (*valx*, *valy*), *valy*))

THEOREM: fx-is-first-2
(¬ a (*valx*, *valy*)) → a (fx (*valx*, *valy*) − 1, *valy*)

```
; now, reexamine what A-is-true-aux4 says
```

THEOREM: a-is-true-aux5
(¬ a (*valx*, *valy*)) → (¬ a (prev (fx (*valx*, *valy*), *valy*), *valy* − 1))

DEFINITION:
a-is-true-kludge (*valx*, *valy*)
= **if** *valy* ≃ 0 **then** 0
else a-is-true-kludge (prev (fx (*valx*, *valy*), *valy*), *valy* − 1) **endif**

THEOREM: a-is-true
a (*valx*, *valy*)

EVENT: Disable a-is-true-aux1.

EVENT: Disable a-is-true-aux2.

EVENT: Disable a-is-true-aux3.

EVENT: Disable a-is-true-aux4.

EVENT: Disable a-is-true-aux5.

EVENT: Disable fx.

EVENT: Disable fx-is-first-1.

EVENT: Disable fx-is-first-2.

EVENT: Disable a-is-true-kludge.

```
; now, we do care what A is
```

EVENT: Enable a.

```
; while we rephrase  qval-of-ackxy-base and  qval-of-ackxy-non-base
; in terms of A, using the fact that A is true:
```

THEOREM: a-base
base (*valx*, *valy*) → (a (*valx*, *valy*) = cons (base-fn (*valx*, *valy*), 0))

THEOREM: a-induction
(¬ base (*valx*, *valy*))
→ (a (*valx*, *valy*) = a (car (a (*valx* − 1, *valy*)), *valy* − 1))

```
; Now, we can forget how A was defined; its car satisfies
; the defn of the Ackermann function
```

EVENT: Disable a.

```
; GOAL:
```

THEOREM: ackermann-works
ackermann (*x*, *y*)
= **if** *x* ≃ 0 **then** 1
elseif *y* ≃ 0

then if $x = 1$ **then** 2
else $x + 2$ **endif**
else ackermann (ackermann $(x - 1,\ y),\ y - 1)$ **endif**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                         ITEM 3                              ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; GOAL:  define a function
; (next-twin-prime x) and prove, that for each x
; either:
;   1. (next-twin-prime x) is some y > x such that y, y+2 are both prime
; or
;   2. (next-twin-prime x) = F, and
;      for all y > x, it isn't true that y and y+2 are both prime

; First, define some primitive recursive functions

; a divisor of x between 2 and top (inclusively), or,  F if there is none :
```

Definition:
divisor $(x,\ \mathit{top})$
= **if** $\mathit{top} < 2$ **then f**
elseif divisor $(x,\ \mathit{top} - 1)$ **then** divisor $(x,\ \mathit{top} - 1)$
elseif $(x$ **mod** $\mathit{top}) \simeq 0$ **then** top
else f endif

Definition: primep $(x) = ((1 < x) \wedge (\neg$ divisor $(x,\ x - 1)))$

```
; y is the first part of a prime pair
```

Definition: fpp $(y) = ($primep $(y) \wedge$ primep $(1 + (1 + y)))$

```
; intended -- by upwards recursion
; (defn next-twin-prime(x) (if (fpp (add1 x))
;  (add1 x)
;  (next-twin-prime (add1 x))))

; obviously, this won't be accepted, but in Nqthm, one can do this with eval$ :
```

Definition:
ntp (x)
= eval\$ (**t**,
`'(if (fpp (add1 x)) (add1 x) (ntp (add1 x)))`,
list (cons (`'x`, x)))

```
;;;;;;;;;;;;;;;;;;;;;;  begin r-loop
;
; *(formals 'ntp)
;  '(X)
; *(body 'ntp)
;  '(IF (FPP (ADD1 X))
;       (ADD1 X)
;       (NTP (ADD1 X)))
; *(ntp 8)
;  11
; *(ntp 19)
;  29
; *(formals 'fpp)
;  '(Y)
; *(body 'fpp)
;  '(AND (PRIMEP Y)
;        (PRIMEP (ADD1 (ADD1 Y))))
;
;;;;;;;;;;;;;;;;;;;;;;  end r-loop

; Of course, this doesn't prove anything.
; We want to define next-twin-prime AND PROVE IT WORKS

; Using the same trick as with the Ackermann function:
```

DEFINITION: $\mathrm{n}(valx) = \mathrm{qval}(\texttt{'(ntp x)}, \mathrm{list}(\mathrm{cons}(\texttt{'x}, valx)))$

```
; Now, we have to analyze qval on the body of ntp
;  '(if (fpp (add1 x)) (add1 x) (ntp (add1 x)))
; Fortunately, for a pure primitive recursive function, such as fpp.
; the Nqthm built-ins REWRITE-V&C-APPLY$ and REWRITE-CAR-V&C-APPLY$
; easily prove that v&c$ does the right thing; so:
```

THEOREM: qval-on-fpp
$\mathrm{qval}(term, va)$
$\rightarrow$ $(\mathrm{qval}(\mathrm{list}(\texttt{'fpp}, term), va) = \mathrm{cons}(\mathrm{fpp}(\mathrm{car}(\mathrm{qval}(term, va))), \texttt{0}))$

EVENT: Disable fpp.

```
; as with ack, we have to compare qval on an expression
; (ntp term) with qval on the basic (ntp x)
```

THEOREM: qval-on-ntp-term

qval (*term*, *va*)
→ (qval (list (`'ntp`, *term*), *va*)
= qval (`'(ntp x)`, list (cons (`'x`, car (qval (*term*, *va*))))))

```
; we can rewrite this in terms of N
```

THEOREM: qval-on-ntp-term-n
qval (*term*, *va*) → (qval (list (`'ntp`, *term*), *va*) = n (car (qval (*term*, *va*))))

```
; Now, the relevant term in the body is just (add1 x), so:
```

THEOREM: qval-on-ntp-add1
qval (`'(ntp (add1 x))`, list (cons (`'x`, *valx*))) = n (1 + *valx*)

```
; now, we unwind the defn of N:
```

THEOREM: unwind-1
n (*valx*)
= qval (`'(if (fpp (add1 x)) (add1 x) (ntp (add1 x)))`,
list (cons (`'x`, *valx*)))

THEOREM: unwind-2
qval (`'(fpp (add1 x))`, *va*) = cons (fpp (1 + cdr (assoc (`'x`, *va*))), `0`)

THEOREM: unwind-3
qval (`'(if (fpp (add1 x)) (add1 x) (ntp (add1 x)))`, *va*)
= **if** fpp (1 + cdr (assoc (`'x`, *va*))) **then** cons (1 + cdr (assoc (`'x`, *va*)), `0`)
else qval (`'(ntp (add1 x))`, *va*) **endif**

THEOREM: unwind-4
n (*valx*)
= **if** fpp (1 + *valx*) **then** cons (1 + *valx*, `0`)
else qval (`'(ntp (add1 x))`, list (cons (`'x`, *valx*))) **endif**

THEOREM: unwind-5
n (*valx*)
= **if** fpp (1 + *valx*) **then** cons (1 + *valx*, `0`)
else n (1 + *valx*) **endif**

EVENT: Disable unwind-1.

EVENT: Disable unwind-2.

EVENT: Disable unwind-3.

EVENT: Disable unwind-4.

EVENT: Disable unwind-5.

```
; now, we can justify the upward recursion:
```

THEOREM: n-basis
$\mathrm{fpp}\,(1 + x) \rightarrow (\mathrm{n}\,(x) = \mathrm{cons}\,(1 + x,\ \mathtt{0}))$

THEOREM: n-induction
$(\neg\ \mathrm{fpp}\,(1 + x)) \rightarrow (\mathrm{n}\,(x) = \mathrm{n}\,(1 + x))$

```
; it might be useful to know the form of N
```

THEOREM: form-of-n
$\mathrm{n}\,(x) \rightarrow (\mathrm{n}\,(x) = \mathrm{cons}\,(\mathrm{car}\,(\mathrm{n}\,(x)),\ \mathtt{0}))$

THEOREM: n-of-zero
$(x \simeq \mathtt{0}) \rightarrow (\mathrm{n}\,(x) = \mathtt{'(3\ .\ 0)})$

DEFINITION: $\mathrm{good}\,(x) = (\mathrm{fpp}\,(\mathrm{car}\,(\mathrm{n}\,(x))) \wedge (x < \mathrm{car}\,(\mathrm{n}\,(x))))$

```
; so, the twin prime conjecture will imply that all x are good
```

```
; we don't need the defn of N any more:
```

EVENT: Disable n.

THEOREM: n-of-fpp
$\mathrm{fpp}\,(x) \rightarrow (\mathrm{n}\,(x - 1) = \mathrm{cons}\,(x,\ \mathtt{0}))$

THEOREM: n-of-non-fpp
$((\neg\ \mathrm{fpp}\,(x)) \wedge (x \not\simeq \mathtt{0})) \rightarrow (\mathrm{n}\,(x - 1) = \mathrm{n}\,(x))$

THEOREM: zero-is-good
$(x \simeq \mathtt{0}) \rightarrow \mathrm{good}\,(x)$

THEOREM: good-goes-down-aux1
$\mathrm{fpp}\,(x) \rightarrow \mathrm{good}\,(x - 1)$

THEOREM: good-goes-down-aux2
$((\neg\ \mathrm{fpp}\,(x)) \wedge (x \not\simeq \mathtt{0}) \wedge \mathrm{good}\,(x)) \rightarrow \mathrm{good}\,(x - 1)$

THEOREM: good-goes-down
$\mathrm{good}\,(x) \rightarrow \mathrm{good}\,(x - 1)$

```
; Next, we prove that (fpp y) and x < y implies (good x)
```

DEFINITION:
good-below-kludge $(y,\ z)$
$=$ **if** $z \simeq \mathtt{0}$ **then** $\mathtt{0}$
else good-below-kludge $(y,\ z - 1)$ **endif**

THEOREM: good-below-aux1
$((y - 1) - z) = ((y - z) - 1)$

THEOREM: good-below-aux2
$\mathrm{fpp}\,(y) \rightarrow \mathrm{good}\,(y - (1 + z))$

EVENT: Disable good-below-aux1.

EVENT: Disable good-below-aux2.

THEOREM: good-below-aux3
$((x < y) \wedge (x \in \mathbf{N})) \rightarrow ((y - (1 + ((y - x) - 1))) = x)$

THEOREM: good-below
$(\mathrm{fpp}\,(y) \wedge (x < y)) \rightarrow \mathrm{good}\,(x)$

EVENT: Disable good-below-aux3.

```
; now, every x is either good or not good.

; if x is not good, there are no twin primes above x:
```

THEOREM: properties-of-bad-aux1
$((\neg\ \mathrm{good}\,(x)) \wedge (x < y)) \rightarrow (\neg\ \mathrm{fpp}\,(y))$

```
; on the other hand, if x is good, then by the definition
```

THEOREM: properties-of-good-aux1
$\mathrm{good}\,(x) \rightarrow (\mathrm{fpp}\,(\mathrm{car}\,(\mathrm{n}\,(x))) \wedge (x < \mathrm{car}\,(\mathrm{n}\,(x))))$

```
; Now, we can define next-twin-prime as:
```

DEFINITION:
next-twin-prime (x)
$=$ **if** good (x) **then** car (n (x))
else f endif

EVENT: Disable good-below.

EVENT: Disable good.

```
; finally, GOAL:

; EITHER
;   1. (next-twin-prime x) is some y > x such that y, y+2 are both prime
```

THEOREM: properties-of-good
good $(x) \rightarrow$ (fpp (next-twin-prime (x)) $\wedge$ $(x <$ next-twin-prime (x)))

```
; OR
;   2. (next-twin-prime x) = F, and
;      for all y > x, it isn't true that y and y+2 are both prime
```

THEOREM: properties-of-bad-1
$(\neg$ good $(x)) \rightarrow$ (next-twin-prime $(x) =$ **f**)

THEOREM: properties-of-bad-2
$((\neg$ good $(x)) \wedge (x < y)) \rightarrow (\neg$ fpp $(y))$

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                          THE END                                ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Index