```
;; Matt Kaufmann

;; From a session with Shaun Cooper, 12/9/91.  Based on CLI Internal
;; Note 210 by Bill Young.
```

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

DEFINITION:
length (x)
$=$ **if** listp (x) **then** $1 +$ length (cdr (x))
else 0 **endif**

DEFINITION:
plistp (x)
$=$ **if** listp (x) **then** plistp (cdr (x))
else $x =$ **nil endif**

DEFINITION:
exp-p (exp)

$=$ **if** *exp* $\in \mathbf{N}$ **then t**
elseif $\neg$ plistp (*exp*) **then f**
elseif length (*exp*) $=$ `3`
then if car (*exp*) $=$ `'plus`
then exp-p (cadr (*exp*)) $\land$ exp-p (caddr (*exp*))
elseif car (*exp*) $=$ `'times`
then exp-p (cadr (*exp*)) $\land$ exp-p (caddr (*exp*))
elseif car (*exp*) $=$ `'subtract`
then exp-p (cadr (*exp*)) $\land$ exp-p (caddr (*exp*))
else f endif
else f endif

THEOREM: exp-p-opener
(*exp* $\notin \mathbf{N}$)
$\rightarrow$ (exp-p (*exp*)
$=$ **if** $\neg$ plistp (*exp*) **then f**
elseif length (*exp*) $=$ `3`
then if car (*exp*) $=$ `'plus`
then exp-p (cadr (*exp*)) $\land$ exp-p (caddr (*exp*))
elseif car (*exp*) $=$ `'times`
then exp-p (cadr (*exp*)) $\land$ exp-p (caddr (*exp*))
elseif car (*exp*) $=$ `'subtract`
then exp-p (cadr (*exp*)) $\land$ exp-p (caddr (*exp*))
else f endif
else f endif)

DEFINITION:
eval-s (*exp*)
$=$ **if** $\neg$ exp-p (*exp*) **then** `0`
elseif *exp* $\in \mathbf{N}$ **then** *exp*
elseif car (*exp*) $=$ `'plus`
then eval-s (cadr (*exp*)) $+$ eval-s (caddr (*exp*))
elseif car (*exp*) $=$ `'times`
then eval-s (cadr (*exp*)) $*$ eval-s (caddr (*exp*))
elseif car (*exp*) $=$ `'subtract`
then eval-s (cadr (*exp*)) $-$ eval-s (caddr (*exp*))
else `0` **endif**

EVENT: Disable exp-p-opener.

DEFINITION:
target-inst-p (*exp*)
$=$ **if** *exp* $\simeq$ **nil then** *exp* $\in$ `'(add mult sub)`
else plistp (*exp*)

$\wedge$ (length (exp) = `2`)
$\wedge$ (car (exp) = `'pushc`)
$\wedge$ (cadr (exp) $\in$ **N**) **endif**

Definition:
target-inst-list-p (exp)
= **if** listp (exp)
then target-inst-p (car (exp)) $\wedge$ target-inst-list-p (cdr (exp))
else exp = **nil endif**

Definition:
single-step ($inst$, s)
= **case on** $inst$:
case = add
then cons (cadr (s) + car (s), cddr (s))
case = $mult$
then cons (cadr (s) $*$ car (s), cddr (s))
case = sub
then cons (cadr (s) $-$ car (s), cddr (s))
otherwise cons (cadr ($inst$), s) **endcase**

Definition:
interpreter-target ($inst\text{-}list$, s)
= **if** listp ($inst\text{-}list$)
then interpreter-target (cdr ($inst\text{-}list$), single-step (car ($inst\text{-}list$), s))
else s **endif**

Event: Enable exp-p-opener.

Definition:
compile (exp)
= **if** $\neg$ exp-p (exp) **then nil**
elseif $exp \in$ **N** **then** list (list (`'pushc`, exp))
elseif car (exp) = `'plus`
then append (compile (cadr (exp)),
append (compile (caddr (exp)), list (`'add`)))
elseif car (exp) = `'times`
then append (compile (cadr (exp)),
append (compile (caddr (exp)), list (`'mult`)))
elseif car (exp) = `'subtract`
then append (compile (cadr (exp)),
append (compile (caddr (exp)), list (`'sub`)))
else nil endif

Event: Disable exp-p-opener.

THEOREM: compile-preserves-legality
exp-p $(exp) \rightarrow$ target-inst-list-p (compile (exp))

THEOREM: interpreter-target-append
interpreter-target (append (*inst-list1*, *inst-list2*), s)
= interpreter-target (*inst-list2*, interpreter-target (*inst-list1*, s))

```
#| first version:  provides too weak of an inductive hypothesis
(prove-lemma compiler-correctness (rewrite)
  (implies (exp-p exp)
           (equal (eval-s exp)
                  (car (interpreter-target (compile exp) s)))))
|#
```

DEFINITION:
compiler-correctness-induct (exp, s)
= **if** length $(exp) = 3$
then compiler-correctness-induct (cadr (exp), s)
$\wedge$ compiler-correctness-induct (caddr (exp),
cons (eval-s (cadr (exp)), s))
else t endif

THEOREM: compiler-correctness-plus
exp-p $(exp) \rightarrow$ (interpreter-target (compile (exp), s) = cons (eval-s (exp), s))

THEOREM: compiler-correctness
exp-p $(exp) \rightarrow$ (eval-s (exp) = car (interpreter-target (compile (exp), s)))

Index