Recursive-function problem

Exercise: all-fours?

Write a function that takes a natural number $n$ and returns true (1) if and only if all the digits in $n$’s numeral are 4’s.
Key design step: form of number

Choose inductive structure for natural numbers:
- Which case analysis do we want?

Step 1: Forms of DECNUMERAL proof system
(1st lesson in program design):
- Either a single digit $d$
- Or $10 \times m + d$, where $m \neq 0$
Example inputs

Step 2:

• Single digits: 4, 9
• Multi-digits: 44, 907, 48
Function’s name and contract

Steps 3 and 4:
Function (all-fours? n) returns nonzero if and only if the decimal representation of n can be written using only the digit 4.
Example results

Step 5: write expected results as unit tests:

(check-assert (all-fours? 4))
(check-assert (not (all-fours? 9)))
(check-assert (all-fours? 44))
(check-assert (not (all-fours? 48)))
(check-assert (not (all-fours? 907)))
Algebraic laws

Step 6: Generalize example results to arbitrary forms of data

\[(\text{all-fours? } d) \equiv (= d 4)\]

\[(\text{all-fours? } (+ (* 10 m) d)) \equiv
  (= d 4) \&\& (\text{all-fours? } m)\]
Left-hand sides turn into case analysis

Step 7:

; (all-fours? d) == ...
; (all-fours? (+ (* 10 m) d)) == ...

(define all-fours? (n)
  (if (< n 10)
      ... case for n = d ...
      ... case for n = (+ (* 10 m) d),
      so m = (/ n 10) and
      d = (mod n 10) ...))
Each right-hand side becomes a result

Step 8:

; (all-fours? d) == (= d 4)
; (all-fours? (+ (* 10 m) d)) ==
; (= d 4) && (all-fours? m)

(define all-fours? (n)
  (if (< n 10)
    (= n 4)
    (and (= 4 (mod n 10))
      (all-fours? (/ n 10))))))
Revisit tests:

Step 9:

(check-assert (all-fours? 4))
(check-assert (not (all-fours? 9)))
(check-assert (all-fours? 44))
(check-assert (not (all-fours? 907)))

(check-assert (not (all-fours? 48)))

Checklist:
• For each form of data, one true and one false
• One extra corner case *(partly fours)*
• Tests pass
Our common framework

Goal: eliminate superficial differences
  • Makes comparisons easy
  • Differences that remain must be important!

No new language ideas.

Imperative programming with an IMPerative CORE:
  • Has features found in most languages
    (loops and assignment)
  • Trivial syntax (from LISP)
Idea of LISP syntax

Parenthesized prefix syntax:

- Names and numerals are basic atoms
- Other constructs bracketed with (…) or […] (Possible keyword after opening bracket)

Examples:

(+ 2 2)
(if (isbound? x rho) (lookup rho x) (error 99))

(For now, we use just the round brackets)
Impcore structure

Two syntactic categories: expressions, definitions

No statements!—expression-oriented (compositional)

(if e1 e2 e3)
(while e1 e2)
(set x e)
(begin e1 ... en)
(f e1 ... en)

Evaluating e has value, may have side effects

Functions f named (e.g., + - * / = < > print)

The only type of data is “machine integer”
(deliberate oversimplification)
Syntactic structure of Impcore

An Impcore program is a sequence of definitions

\[
\text{(define mod (m n) (- m (* n (/ m n))) )}
\]

Compare

```c
int mod (int m, int n) {
    return m - n * (m / n);
}
```
Impcore variable definition

Example

(val n 99)

Compare

int n = 99;
Concrete syntax for Impcore

Definitions and expressions:

```plaintext
def ::= (define f (x1 ... xn) exp) ;; "true" defs
   |  (val x exp)
   |  exp
   |  (use filename) ;; "extended" defs
   |  (check-expect exp1 exp2)
   |  (check-assert exp)
   |  (check-error exp)

exp ::= integer-literal
   |  variable-name
   |  (set x exp)
   |  (if exp1 exp2 exp3)
   |  (while exp1 exp2)
   |  (begin exp1 ... expn)
   |  (function-name exp1 ... expn)
```
Example function shows every form

(define even? (n) (= (mod n 2) 0))

(define 3n+1-sequence (n) ; from Collatz
    (begin
        (while (!= n 1)
            (begin
                (println n)
                (if (even? n)
                    (set n (/ n 2))
                    (set n (+ (* 3 n) 1))))))
    n))