Principles of Programming Languages

2015.07.24

Notes

- NAME: _____
- Did you present a small project? YES / NO
- Total available time: 2h.
- You may use any written material you need.
- You cannot use computers or phones during the exam.

1 Scheme

1.1 Urmax (6 points)

Define a tail recursive procedure, called `urmax`, that takes a list of lists $((a_1^1 a_2^1 a_3^1 ...)(a_1^2 a_2^2 a_3^2 ...)(a_1^3 a_2^3 a_3^3 ...)...)$ and returns $(\max a_1^1\ a_2^2\ a_3^3\ ...)$.

E.g. `(urmax '((-1)(1 2)(1 2 3)(10 2 3 -4)))` is 3.

1.2 Higher order functions (5 points)

Define a variant of urmax based on higher order functions like `map` (you cannot use iterative loops or recursion in it).

2 Haskell

2.1 Lists as instances of Num (6 points)

Make lists of numbers a instances of the class Num. If the two lists have different length, you must assume that the missing elements of the shorter are all 0.

E.g. `[1,2,3] * [2,-1]` should be `[2,-2,0]`.

(Remember that you need to define methods for +, -, *, abs, signum, and fromInteger.)

2.2 List of lists of lists... (3 points)

Define a recursive data structure of type `TT` which can be used to represent lists of Int of any depth (e.g. in Scheme `'(1 2 (3 9) ((1) -7))`).

2.3 Lile (3 points)

Define a predicate `lile`, which, given a `TT` value, check if it contains its own length.

E.g., using a Scheme-like notation `(lile '(2 1))` holds, while `(lile '(1 2 1))` does not.

2.4 Lileg (5 points)

Define a version of the predicate `lile`, called `lileg`, which takes a value of type `TT`, and check if all the lists in it contain their length.

E.g., using a Scheme-like notation: `(lileg '(2 (2 (1)) 3))` must hold.

3 Prolog (5 points)

Define a non-deterministic Finite State Automata simulator in Prolog.

Solutions

Scheme

```
(define (urmax LL)
  (define (urr LL k m)
    (if (null? LL)
        m
        (let* ((x  (list-ref (car LL) k))
               (mm (max m x)))
          (urr (cdr LL) (+ k 1) mm))))
  (urr LL 0 (caar LL)))


(define count
  (let ((start -1))
    (lambda ()
      (set! start (+ start 1))
      start)))

(define (hurmax LL)
  (apply max (map (lambda (x)
                    (list-ref x (count))) LL)))
```

Haskell

```
instance (Num f) => (Num [f]) where
  a + [] = a
  [] + b = b
  (x:xs) + (y:ys) = (x+y):(xs + ys)
  a - [] = a
  [] - b = -b
  (x:xs) - (y:ys) = (x-y):(xs - ys)
  a * [] = map (\x -> 0) a
  [] * b = map (\x -> 0) b
  (x:xs) * (y:ys) = (x*y):(xs * ys)
  abs = map abs
  signum = map signum
  fromInteger a = [fromInteger a]


data TT = VV Int | LL [TT] deriving (Show, Eq)

len (VV _) = 0
len (LL x) = length x

member x (LL y) = (VV x) `elem` y

lile ll = member (len ll) ll

iflc (LL []) = True
iflc (LL ((VV x):xs)) = iflc (LL xs)
```

```
iflc (LL (x:xs)) = lileg x && iflc (LL xs)

lileg t = lile t && iflc t
```

Prolog

```
% acceptance
config(State, []) :- final(State).

% move
config(State, [C|String]) :-
    delta(State, C, NewState), config(NewState, String).

run(Input) :- initial(Q), config(Q, Input).
```