Corso di Programmazione

II Accertamento del 13 Giugno 2017

cognome e nome

2. Procedure con argomenti procedurali

Data una sequenza `s` di interi positivi diversi fra loro, rappresentata in Scheme da una lista, la procedura `lis` risolve il problema della *sottosequenza crescente più lunga* (LIS) restituendo la lista che corrisponde a *una* possibile soluzione:

```
(define lis
  (lambda (s)
    (lis-rec s 0)
    ))

(define longer
  (lambda (u v)
    (if (< (length u) (length v))
        v
        u)
    ))
```

```
(define lis-rec
  (lambda (s t)
    (cond ((null? s)       null)
          ((<= (car s) t)  (lis-rec (cdr s) t))
          (else
           (longer
            (cons (car s) (lis-rec (cdr s) (car s)))
            (lis-rec (cdr s) t)
            )))
    ))
```

La procedura `all-lis`, componente del programma riportato nel riquadro, è invece intesa a determinare la lista di *tutte* le soluzioni. Completa il programma introducendo opportune espressioni negli appositi spazi.

```
(define all-lis (lambda (s)  (all-lis-rec s 0) ))

(define all-lis-rec
  (lambda (s t)

    (cond ((null? s)   (list null)                                                    )
          ((<= (car s) t) (all-lis-rec (cdr s) t))
          (else
           (all-longer

            (map   (lambda (x) (cons (car s) x))

                 (all-lis-rec (cdr s) (car s))
                 )
            (all-lis-rec (cdr s) t)
            ))
          )))

(define all-longer
  (lambda (u v)

    (let ((m (length   (car u)                                                        ))

          (n (length   (car v)                                                        ))
          )
      (cond ((< m n) v)
            ((> m n) u)
            (else (append u v))
            ))
    ))
```

3. Classi in Java

La classe `IntList` consente di modellare in Java strutture dati corrispondenti a *liste di interi* in Scheme. Gli oggetti *immutabili* di tipo `IntList` rappresentano liste vuote o coppie "car/cdr", in modo sostanzialmente analogo a quanto avviene in Scheme. Più specificamente, il protocollo è definito da due costruttori e da cinque metodi così caratterizzati:

`new IntList()`	*// costruisce una lista vuota — corrispondente in Scheme a:*	`null`
`new IntList( n, u )`	*// costruisce la lista con primo elemento n e resto della lista u:*	`(cons n u)`
`s.isNull()`	*// verifica se la lista s è vuota:*	`(null? s)`
`s.car()`	*// restituisce il primo elemento della lista s:*	`(car s)`
`s.cdr()`	*// restituisce il resto della lista s, tolto il primo elemento:*	`(cdr s)`
`s.length()`	*// restituisce la lunghezza della lista s:*	`(length s)`
`s.append( v )`	*// restituisce la lista che si ottiene giustapponendo s e v:*	`(append s v)`

Completa la definizione della classe `IntList` introducendo opportune variabili d'istanza (rappresentazione interna nascosta) e realizzando i costruttori e i metodi coerentemente con le scelte implementative fatte.

```
public class IntList {


  private final boolean noItem;
  private final int n;
  private final IntList u;
  private final int k;


  public IntList() {                    // null
    noItem = true;
    n = 0;
    u = null;
    k = 0;
  }

  public IntList( int n, IntList u ) {  // cons
    noItem = false;
    this.n = n;
    this.u = u;
    k = u.length() + 1;
  }

  public boolean isNull() {             // null?

    return noItem;

  }

  public int car() {                    // car

    return n;

  }

  public IntList cdr() {                // cdr

    return u;

  }

  public int length() {                 // length

    return k;

  }

  public IntList append( IntList v ) {  // append
    if ( isNull() ) {
      return v;
    } else {
      return new IntList( car(), cdr().append(v) );
    }
  }

}  // class IntList
```

Corso di Programmazione

II Accertamento del 13 Giugno 2017

cognome e nome

4. Memoization

Il seguente programma è una traduzione in Java del codice Scheme riportato all'inizio dell'esercizio **2**, dove le liste sono realizzate applicando la classe `IntList` il cui protocollo è specificato nell'esercizio **3**:

```
public static IntList lis( IntList s ) {
  return lisRec( s, 0 );
}

public static IntList lisRec( IntList s, int t ) {
  if ( s.isNull() ) {
    return new IntList();
  } else if ( s.car() <= t ) {
    return lisRec( s.cdr(), t );
  } else {
    return longer( new IntList( s.car(), lisRec( s.cdr(), s.car() ) ),
                   lisRec( s.cdr(), t )
                   );
  }
}

public static IntList longer( IntList u, IntList v ) {
  if ( u.length() < v.length() ) {
    return v;
  } else {
    return u;
  }
}
```

Applica la tecnica *top-down* di *memoization* per realizzare una versione più efficiente del programma. A tal fine, assumi che le sequenze passate come argomento al metodo statico `lis` soddisfino sempre questa proprietà: una sequenza *s* di lunghezza *k* è costituita da una permutazione degli interi *1, 2, ..., k*. (Tale proprietà limita l'intervallo di valori che possono essere assunti dal parametro `t` del metodo ricorsivo.)

```
public static IntList lisMem( IntList s ) {
  int n = s.length();
  IntList[][] h = new IntList[ n+1 ][ n+1 ];
  for ( int i=0; i<=n; i=i+1 ) {
    for ( int j=0; j<=n; j=j+1 ) {
      h[i][j] = UNKNOWN;
  }}
  return lisMRec( s, 0, h );
}
```

```
public static IntList lisMRec( IntList s, int t, IntList[][] h ) {

  int k = s.length();

  if ( h[k][t] == UNKNOWN ) {

    if ( s.isNull() ) {
      h[k][t] = new IntList();
    } else if ( s.car() <= t ) {
      h[k][t] = lisMRec( s.cdr(), t, h );
    } else {
      h[k][t] = longer(
                  new IntList( s.car(), lisMRec(s.cdr(),s.car(),h) ),
                  lisMRec( s.cdr(), t, h )
                  );
  }}
  return h[k][t];
}


private static final IntList UNKNOWN = null;
```