Condense the List of Numbers

Prabhaker Mateti

Consider the informal description, given in the next paragraph, of a program. Give (a) requirements analysis, (b) precise specifications (using logic/discrete mathematics, or in carefully worded English), (c) a design in pseudo-code

You are given a comma separated list of items in a text file. The list is terminated with a period. Each item is either a number, or a hyphen separated pair of numbers. Your program should condense this list as much as possible, as can be seen from the following examples.

`1,3,4,2,6,5,8,5,7.` becomes
`1-8.`

`9,100-4870,4993,4871-4875,5016,5118,7-250,5100-5123.` becomes
`7-4875,4993,5016,5100-5123.`

`1103,1023-1100,87654321,1050-1150,1110,1250-1344.` becomes
`1023-1150,1250-1344,87654321.`

The rest of this handout is a 'solution' to the above problem. My comments are enclosed in /* */

1 Requirements Analysis

/* Certain 'debatable' resolutions of incomplete or ambiguous requirements are shown *slanted* for emphasis. */

A range is written as $m - n$. This is three tokens: a number, a hyphen, and another number. The first number m is expected to be $\leq$ n; *we will ignore it otherwise.*

A single item appears as one number token, x. This is equivalent to x-x.

We allow the numbers to be negative.

We are not permitting an empty list. We do not expect the ranges or single items to appear in any particular order.

The output produced should represent the same set of numbers represented by the input. The output is required to be the shortest possible. We take this to mean the length of the output viewed as a sequence of tokens is to be minimized, and not, e.g., that all blanks are to be eliminated.

The real need is clearly for a condensation of the list given in some internal data structure format, rather than a textually presented list of numbers.

2 Specification

Some notation is inevitably needed to do precise specs. A 'list' of, say three items `a, b, c`, is written as `<a, b, c>`. If `p` is pair of things, `p = (x, y)`, then `p.1 = x`, and `p.2 = y`. We abbreviate the words `such that` as `st`.

List of Input Items

A range `m-n` denotes the set `range(m, n) = {z st m <= z <= n}`. A single-item `x` denotes the singleton set `{ x }`.

`setofoneitem(ars) = range(m, n)`, if `ars = <number m, hyphen, number n>`
`setofoneitem(ars) = { x }` , if `ars = <number x>`

The list is a comma-separated sequence of ranges and single items:

```
single-item ::= number
range ::= number hyphen number
rs ::= range | single-item
list ::= rs | list comma rs
external-list ::= list period
```

A list denotes the union of the sets denoted by the rs items it contains.

`setoflist(<ars>) = setofoneitem(ars)`. Note that `<ars>` stands for a list of just one thing, the `ars`.

`setoflist(q | <comma, ars>) = setoflist(q) + setofoneitem(ars)`. Here `q` is a `list` as defined above. In grammar productions, the vertical-bar `|` denotes 'alternate'; when used for sequences it denotes sequence catenation.

The input 'file' is expected to contain `well-formed` sequences that conform to the above `external-list`.

List of Output Items

The list `ol` that the program generates should be equal, as a set, to the input list `il`: `setoflist(ol) = setoflist(il)`.

The list `ol` is the shortest of all such lists: if a list `ls` exists such that `setoflist(ls) = setoflist(ol)` then `length(ls) >= length(ol)`.

The items in the list `ol` are in the increasing order: if r_i and r_j are two ranges appearing in ol, $i < j$, then the second number of r_i is less than the first number of r_j ($r_i.2 < r_j.1$). For the purpose of this rule, a single item `x` is to be taken as a range `x-x`.

Fine Print

The lists are sequences of parsed entities: `rs`, `comma`, `period`. The `rs` itself is a sequence; it is a 3-long sequence if it is a range, a 1-long sequence if it is a single item.

Unless we define `length`(.) carefully, we will permit a single item `x` to appear as a range `x-x` in the output list.

`rslen(rs) = 3`, if `rs` is a range
`rslen(rs) = 1`, if `rs` is a single item

```
length(<rs>) = rslen(rs)
length(q | <comma, rs>) = length(q) + 1 + rslen(rs)
```

3 A First Design

/* This first design appears inefficient, but it can be refined and implemented in such a sophisticated way that it is faster than the second design shown later. */

Psuedo Code

```
var nums: set of numbers := {};

input-bgn(input-fnm);
repeat
  nums := nums + set-of(input-one-rs());
until input-next-token() = period;
input-end();

ol-initialize();
while nums != {} do
  var m: number := min-of(nums);
  var n: number := m;
  repeat
    nums := nums - {n};
    n := n + 1;
  until n not-in nums;
  ol-append(m, n-1);
od;
ol-period();
```

Operations Assumed

We imagine the file content to be a stream of tokens: `rsf` is an imaginary stream variable conceptually denoting the tokens read so far; `ytr` is a similar var denoting the file content yet to be read.

/* A 'filter' that accomplishes the above is, strictly speaking, a part of this and the next design. To save space, and to avoid discussing the well-known, we skip the design of such a lexical analyzer. */

```
proc input-bgn(fnm: string)
  -- open/initialize input stream
  pre : file-exists(fnm)
  post: rsf = <>, ytr = file-content(fnm)

proc input-end()
  -- close/finalize input stream
```

```
  pre : is-defined(ytr)
  post: rsf =  file-content(fnm), ytr = <>

func input-next-token(): token
  -- gives the next token from the input stream.
  pre : ytr != <>
  post: result = ytr0[1], rsf = rsf0 | result, ytr = ytr0[2..],

func input-one-rs(): rs
  -- gives a representation of the next rs from the input.
  -- The rs may be a range or a single item.
  pre : well-formed(ytr)
  post: exists i: {0, 1, 3} (
    result = ytr0[1..i], rsf = rsf0 | result, ytr = ytr0[i+1..],
    i = 0 -> ytr[1] = period,
    i = 1 -> ytr[1] = comma,
    i = 3 -> result[2] = hyphen
    )

func set-of(r): set of numbers
  -- yields a set of numbers represented by the r.
  pre : is-rs(r)
  post: result = setofoneitem(r)

func min-of(A: set of number): number
  pre : A != {}
  post: result in A, for-all e: A (result <= e)

infix + (A: set of T, B: set of T) == A union B.
infix - (A: set of T, B: set of T) == A difference B.
infix not-in (e: T, A: set of T) == not (e member-of A)

proc ol-initialize() -- for you to finish!
proc ol-append() ...
proc ol-period() ...
```

4 Second Design

Design Idea

/* This concentrates on how to represent nums. You will agree that it is a lower-level (i.e., closer to the machine) design than the first design. */

The set is stored as an array of ranges. Each range is stored as a pair of, its lowest and highest, numbers. The array may are not be kept sorted.

Psuedo Code

This code is not as obvious as that of the previous design. Also, even to understand it vaguely requires the detailed meaning of the primitive operations.

```
var numa: grow-shrink array of pairs of numbers := create-empty();

input-bgn(input-fnm);
repeat
  numa-append(numa, input-one-rs());
until input-next-token() = period;
input-end();

out-bgn()
while not numa-empty() do
  var x: number := numa-index-of-min();
  var m: number := numa[x].1;  -- .1 means first of the pair
  var n: number := numa[x].2;  -- .2 means second
  repeat
    n := n + 1;
  until numa-ix-delete(n-1) = 0
  out-append(m, n-1);
od;
out-end();
```

Operations Assumed

```
proc out-bgn()
  pre : true
  post: out = ""

proc out-append(x, y: number)
  pre : x <= y
  post: exists s, t: string (
    out = out0 | s | t,
    x = y -> t = itoa(x),
    x < y -> t = itoa(x) | "-" | itoa(y),
    s = if out0 = "" then "" else "," fi
  )

proc numa-append(m, n: number)
  pre : true
  post: numa = if m <= n then  numa0 | <m, n> else numa0 fi

proc numa-ix-delete(b: number): number
  -- function with side-effect on numa[]
```

```
  -- if numa has a range that includes b delete that range, and
  -- return the index of the deletion; otherwise return 0
  pre : true
  post: (result = 0) && not-in-numa(b) && (numa = numa0)
    or (
      (numa0[result].1 <= b) && (b <= numa0[result])
      && (numa = numa0[1..result-1] | numa0[result+1..]))

aux func not-in-numa(b: number): boolean is
  not exists i: 1..len(numa) st
    (numa[i].1 <= b) && (b <= numa[i].2)

func numa-index-of-min(): number
  -- deliver the index of the range with the least first number
  pre : numa non empty
  post: numa = numa0 &&
    numa[result].1 = min { numa[i].1 st i in 1..len(numa)}
```

5 Implementation

/* Your exercise! */

6 Pedagogical Confessions

/* I began writing the specifications precisely. In the process I discovered the ambiguities in the problem statement, and resolved them as explained in requirements analysis.

I wrote the psuedo code design first, and then defined more precisely the operations that I used in the design.

Of course, there are numerous other designs that are equally good. I deliberately chose this one because every student in my classes so far constructs the linked list of input ranges so that they are in non-decreasing order without duplicates or overlapped ranges. Do I know why they instinctively do that? No.

Unless one gets into representing the set in sophisticated ways, the second design is pretty efficient. Asymptotic time bounds are rarely relevant in software engineering; in any case, that was not the point of this exercise. */

7 Discussion

What does our spec require the program to produce for `1, 2.` as input? Is it consistent with the requirements?

What about input, such as `6, 7, 8`?

How do we check if the specs are specifying what the requirements are describing?

Same question between design(s) and specs.