McMaster University
Department of Computing and Software
Dr. W. Kahl

CAS 707 — Formal Specification Techniques

26 February 2018

Individual solutions to the assignment question here are due **electronically** via subversion **before 11:00 a.m. on Monday, March 5**.

Assignment Question 5.1 — Linked Lists

Use the following datatype (essentially as shown in class) for implemeting singly-linked lists in C:

```
typedef int value_type;

typedef struct _cons { value_type head;
                       struct _cons * tail;
                     } nonEmptyList;

typedef nonEmptyList * list; // NULL used as Nil

#define Nil NULL

/*@
  logic list Nil = \null;

  inductive hasSuffix{L} (list xs, list ys) {
    case hasSuffix_refl {L}:
      ∀ list xs; hasSuffix(xs,xs);
    case hasSuffix_next{L}:
      ∀ list xs, ys;
        \valid(xs) ⇒ hasSuffix(xs→tail, ys) ⇒ hasSuffix(xs,ys);
  }

  predicate finite{L}(list xs) = hasSuffix(xs,Nil);
*/

nonEmptyList * cons(value_type x, list xs) { // NULL used as error
  nonEmptyList * result = malloc(sizeof(struct _cons));
  if (result)
    { result→head = x;
      result→tail = xs;
    }
  return result;
}
```

Note:

- *_cons* is a struct name.
  This name starts with un underscore to document that it should not be used by users of this declaration.

- **struct** *_cons* is a type, the same type as:
- **struct** { **value_type** *head*; **struct** *_cons* ∗ *tail*; }.
- The **typedef** ... *nonEmptyList* makes "*nonEmptyList*" another name for that type.
- ***list* is the type of (possibly empty!) lists!**

(a) Separate the provided material into a header file list.h and an implementation file list.c. For each of the items below, update both as appropriate.

(b) Implement a function that calculates the length of a linked list.

(c) Implement a function that appends one list to any other list, including the empty list.

Test correct behaviour for empty lists, and document the test results.

Hint: C implements call-by-reference using pointers.

Hint: Keep the tests in a separate driver file main.c.

(d) Implement a function that prints the elements of the linear prefix of its argument list; if a node is encountered that was already previously traversed (because the argument list has a cycle), information about this should be printed, and the function should return after that.

(e) Produce interesting test cases for this using your append function.

(f) Start producing ACSL specifications and annotations for the functions so far — see in particular sections "2.7.2 Separation" and "2.8 Sets and lists" in the ACSL reference.

Document any issues you encounter!

(g) Implement the following C function for **insertion into ordered lists**:

bool *testAndInsert*(*list*∗ *p*, **value_type** *n*)

Assuming that *p* contains a *reference* to a list with *head* fields in ascending order, the function call "*testAndInsert*(*p*, *n*)" returns a **bool** result indicating whether the list referenced by *p* contained *n* as an *element*, and if it did not, it modifies that list by inserting a new list container with *n* as *element*, such that the resulting list is again in ascending order.

(h) Strive to produce an ACSL specification and annotations also for *testAndInsert*.

Assignment Question 5.2 — Reynolds: Linked Lists

(a) Read section 1.1 of the lecture notes of CS818A3-2011 by John Reynolds.

(b) Implement his in-place reverse function using the lists of Assignment Question 5.1.

(c) Produce an ACSL specification for this reverse function, and experiment with ACSL loop invariants.

(d) Read also the remainder of chapter 1 of Reynolds' lecture notes.

The code listings above have been produced by including, before `\begin{document}`, the following:

```
\usepackage{listings}
\usepackage{listingsACSL}
\lstset{%
  language=[ACSL]C,
  frame=single,
  identifierstyle=\slshape,
  columns=flexible}
```

I am using no other packages that might be interfering — if you run into trouble, perhaps comment out some `\usepackage{...}` lines you don't need?