COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

Declarative Programming

Answers to workshop exercises set 2.

QUESTION 1
Give a high level description (not programming language specific)
of at least five different possible representations of playing cards
from a standard 52 card deck. Describe the advantages and disadvantages
of each representation.

The standard 52 card deck has 13 cards in each of four suits.
The suits are clubs, diamonds, hearts and spades, and the 13 ranks in
each suit are the 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king and ace.
In this question, we ignore jokers.

ANSWER
The choice of how best to represent something depends on what you will
be
doing with it. It is good to be able to think "outside the box" and come
up with non-obvious representations of things (this is the basis of many
clever algorithms). Also, it is good to use abstract data types so you
can manipulate things without worrying about how they are represented
and change the representation without changing the rest of the program.

0) There is the representation mentioned in lectures (a pair of
enumerated types). It is pretty good for most purposes.

1) A pair containing a suit, S, H, D or C (an enumerated type) and a
number in the range 1 to 13, inclusive (or 0 to 12). Suits and ranks
can
be extracted simply. Aces could be at the bottom or top of the range
(the
top is better for most card games as aces are normally considered "high").

This is only slightly less convenient than the representation discussed
in lectures. It is less convenient because it is more error prone. For
example, it would allow the pair (S, 15), which does not correspond to
any
valid card.

2) A pair of strings, such as ("two", "spades"). Not bad for textual output and easier for comparison, determining the suit, etc than some representations (see below), but it is even more error prone than the previous representation.

3) A number between 0 and 51, inclusive. The 52 cards can be mapped onto this set of numbers in several different ways, but the obvious encoding is

- encode the suit with (club -> 0, diamonds -> 1, hearts -> 2, spades -> 3),
- encode the rank (2 -> 0, 3 -> 1, ... 10 -> 8, jack -> 9, ... ace -> 12)
- encode the card as (13 * encoded_suit + encoded_rank).

With this representation, if "card" holds a number encoded this way, you can compute the suit with card/13 and the rank with card mod 13. This is a very compact representation.

A version of this encoding could use the numbers 1-52, not 0-51. This requires encoding with (13 * encoded_suit + encoded_rank), and decoding with (card - 1) / 13 and (card - 1) mod 13, and is therefore a bit less convenient.

4) A bit string (or sequence of Booleans) of length 52 with a single one bit (True) and the rest zero (False), with the position of each card determined by encoding 1. This allows a very compact representation for SETS of cards: just "and" the representations of single cards together. It is also fairly compact for representing single cards, because on current computers, the standard size of integers is 64 bits anyway. However, operations on this representation require finding WHICH bit or bits are set, which is not trivial.

5) A JPEG image. Great for graphical display, but absolutely dreadful for pretty much every other purpose.

6) A URI (Universal Resource Identifier). This makes it easy to represent cards consistently across the web, but it does nothing to help implement any operations on cards.

7) A function which returns various information about the card depending on how it is called. For example, f "rank" could return a representation of the rank, f "suit" could return a representation of the suit, etc.

Since users cannot peer into the function, this representation is totally
abstract. How useful it is depends on the usefulness of the representations
it returns. That usefulness is greatly limited by the fact that f "suit" and
f "rank" have to return values of the same type.

QUESTION 2
Define a Haskell type for representing "font" tags in HTML. A font tag
can specify zero or more of the following: the size in points (e.g. 10),
the face (e.g. "courier") and the colour. The colour can be described
using a colour name (e.g., "red"), a six-digit hexidecimal number
(e.g. #02EA1F) or a RGB triple of numbers (e.g. rgb(255,100,0)).

Note: the font tag is the most widely misused of all HTML tags,
and in fact it is fundamentally misconceived. The font should be up to
the VIEWER of the web page, not the web page DESIGNER; if the designer
selects a small font, people with bad eyesight looking at the page
won't be able to read it. This is why the font tag is actually deprecated,
which means it is slated to disappear in a future version of the HTML
standard.

ANSWER
The colour can be specified by a string, a single hex integer or three
integers for the RGB components respectively:

```
>data Font_color
>   = Colour_name String
>   | Hex Int
>   | RGB Int Int Int
```

One possibility for the representation of font tags is as a list of specifiers,
each of which specifies one aspect of a font:

```
type Font_tag = [Font_specifier]
data Font_specifier
    = Font_size Int
    | Font_face String
    | Font_color Font_color
```

However, this is not a good idea, because it is not clear what the meaning

of such a list would be if it specified e.g. the font size twice: which would
actually set the font size used for display?

A better possibility is a data structure that allows each of these three aspects of fonts to be specified just once, with a maybe type used to allow
them to remain unspecified by the font tag (in which case whatever setting
was in effect outside the font tag continues to be in effect inside it). This could be done with a type like this.

```
>data Font_tag = Font_tag (Maybe Int) (Maybe String) (Maybe Font_color)
```

QUESTION 3
Implement a function 'factorial' that computes the factorial of a given integer. Include a type declaration.

ANSWER

```
>factorial :: Int -> Int
>factorial 0 = 1
>factorial n = n * (factorial (n-1))
```

QUESTION 4
Implement a function 'myElem' which returns True if a given item is present
in a given list. Include a type declaration.

ANSWER

```
>myElem :: Eq a => a -> [a] -> Bool
>myElem _ [] = False
>myElem x (y:ys)
>     | x==y      = True
>     | otherwise = myElem x ys
```

QUESTION 5
Implement a function 'longestPrefix' which returns the longest common prefix
of two lists. ie: When applied to "extras" and "extreme", the function should return "extr".

ANSWER

```
>longestPrefix :: Eq a => [a] -> [a] -> [a]
>longestPrefix [] _ = []
>longestPrefix _ [] = []
>longestPrefix (x:xs) (y:ys)
>     | x == y    = x:(longestPrefix xs ys)
>     | otherwise = []
```

QUESTION 6
Without necessarily understanding the code, translate the following
C function into Haskell.

```
int mccarthy_91(int n)
{
    int c = 1;

    while (c != 0) {
        if (n > 100) {
            n = n - 10;
            c--;
        } else {
            n = n + 11;
            c++;
        }
    }

    return n;
}
```

ANSWER
Note: this is an iterative coding of a famous function - see
http://en.wikipedia.org/wiki/McCarthy_91_function.

The main challenge is the while loop. For the sake of illustration we
will solve a more general problem. Consider the following generic while
loop in C which updates variable x in each iteration:

```
while (cond(x))
    x = next_version_of(x);
x = final_version_of(x);
```

It can be translated into Haskell as follows

```
>mywhile x =
```

```
>   if cond x then mywhile (next_version_of x)
>   else final_version_of x
```

In the C code there are two variables, so we can make x a pair and use the following definitions of cond, next_version_of and final_version_of:

```
>cond (c,n) = c /= 0
>next_version_of (c,n) =
>   if n > 100 then (c-1, n-10) else (c+1, n+11)
>final_version_of (c,n) = n
```

We simply have to call mywhile with the initial versions of c and n:

```
>mccarthy_91 :: Int -> Int
>mccarthy_91 n = mywhile (1, n)
```

QUESTION 7
Write a Haskell function which takes two integers, min and max, and returns a list of integers from min to max, inclusive. Note there are two different strategies to solve this problem: we can build up the list from min to max or backwards, from max to min. How does your Haskell code compare with a version in an imperative language such as C, and how
would you reason about the correctness of a C version?

ANSWER
Here is one simple solution.

```
>range_list min max =
>       if min > max then []
>       else min : range_list (min+1) max
```

It probably best to think of this as building up the list from min to max (thats also how its most likely evaluated, but don't get too tied up
with evaluation order).

A typical C version would use a while loop rather than recursion. Reasoning about correctness will be tricky in C. Typically you reason about "invariant" relationships between the values of different variables
at particular points in the execution. This reasoning can be even more messy if the list is built from min to max because the data structure being built will typically contain structs with uninitialised fields

(the last pointer in the linked list will only be set to NULL at the very end of the computation).