

Re: Change for a Dollar

Source: <http://coding.derkeiler.com/Archive/General/comp.programming/2007-04/msg00413.html>

- *From:* Simon Richard Clarkstone <s.r.clarkstone@xxxxxxxxxxxxx>
 - *Date:* Sat, 28 Apr 2007 13:28:20 +0100
-

Charles Richmond wrote:

I heard one night on David Letterman that there are 293 ways to make change for a dollar. So I started thinking about writing a program to list out **all** the ways a dollar could be broken into change. I just can **not** seem to get a hold on this problem.

I wrote a program with a recursive function attempting to make the change, but it had excessive runtime and does **not** seem to be working in any case...

Does anyone have an idea how I can proceed???

Idea: if you want to make change for an amount A , and you have a coins of value $[V_1, V_2, V_3, \dots, V_n]$ available, and V_i is less than A , then you can make change by:
(the coin V_i)+(change you can make for $(A - V_i)$ from $[V_i, \dots, V_n]$)

This gives all the combinations in one ordering, with the coins in the same order as the original list.

I sat down and bashed out a program based on that idea in Haskell:

We have a module declaration:

```
module Change (mkChange, Coin) where
import Data.List (tails)
```

The the definition of a Coin (there is a string to distinguish multiple types with the same value, as someone suggested elsewhere in this thread):

```
data Coin = Coin { cName :: String, val :: Integer }
```

How to print a coin (just print the String part)

```
instance Show Coin where show = cName
```

Re: Change for a Dollar

The `mkChange` function takes a list of available coin types and the value to make up, then returns the list of ways to make the change:

```
mkChange :: [Coin] -> Integer -> [[Coin]]
mkChange coins total
```

The only way for a total of zero is no coins:

```
| total == 0 = [[]]
```

Otherwise, we have a list comprehension. The coin `c` followed by the list `ch` is a possible way to make change...

```
| otherwise = [ c : ch
```

For each `c` being a coin and `ccs` being the list of that coin and all later coins in the available list...

```
| ccs@(c:cs) <- tails coins
```

Check that the value of `c` is not bigger than the total...

```
, val c <= total
```

and `ch` is one of the ways to make change for the remaining amount from the coins `ccs`.

```
, ch <- mkChange ccs (total - val c)
]
```

To use it, we can define:

```
> usaCoins = [Coin "penny" 1, Coin "nickle" 5, Coin "dime" 10, Coin "quarter" 25, Coin "half" 50, Coin "dollar" 100]
```

Then call:

```
> mkChange usaCoins 100
```

to get the full list or:

```
> length $ mkChange usaCoins 100
```

to find out how many ways there are.

A good compiler will deforest that expression into recursive code very similar to the typical imperative solution. My approach can be easily adapted for any language that has list comprehensions (e.g. Python), though linked lists work better for it than arrays do.

--

Simon Richard Clarkstone:

s.r.cl?rkst?n?@durham.ac.uk/s?m?n.cl?rkst?n?@hotmail.com

Re: Change for a Dollar

Re: Change for a Dollar

Scheme guy says: (> Scheme Haskell)

Haskell guy says: (> Scheme) Haskell

.