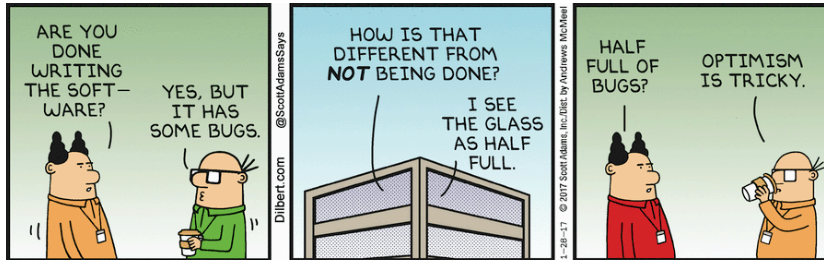
Computer Science 54

Assignment 4

Sunday, October 2 @ 11:59pm



`http://dilbert.com/strip/2017-01-28`

Partners On this assignment you may (and I'd encourage you to) work with a partner. If you choose to, you must both be there anytime you're working on the assignment.

Submission When you're all done, run the check file. If all is well, submit your assignment via Gradescope. If you worked with a partner, make sure that both names are at the top of the file. Only one person needs to submit, but make sure you add the other person when you submit.

I Substitution Ciphers, Concluded

On the last two assignments, we worked with substitution ciphers based on cycles and pangrams. Another strategy for creating substitutions is to shuffle the letters randomly. It has the advantage of being less predictable than pangrams, but it makes the substitution harder to remember. Here is an example of a random translation.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ␣
PLUMNZFDEWQOVASTYRCI␣JHKGXB
```

With this translation, `SNOW` becomes `CASH`, `ARISTOTLE` becomes `PRECISION`, and `SAGEHEN` becomes `CPFNDNA`. Figure 1 shows an old child's toy that combines an arbitrary permutation with a shift. Its encryption is not any better than a Caesar cipher because anyone can know the permutation. If you and your co-conspirator are to preserve secrecy, you must decide on a permutation and keep it secret.

Figure 1: The decoder pin from Radio Orphan Annie's Secret Society, 1935.

After our work on previous assignments, encoding with random permutations is not hard. We have only to find a way to generate the random permutations. We will use a random number generator to do it. See the appendix to this

assignment for instructions on how to create and use a random number generator.

1. [3 points] A good way to produce a random permutation of the elements in a list is with the *Knuth shuffle*. The idea is, given a list, to choose an element from it at random. That element will be the first element of the permutation. To get the rest of the permutation, take the chosen element out of the list and recursively shuffle the remaining elements. The implementation in SML is a natural list recursion, but it is a little different from our usual pattern. Instead of taking the first element and the rest of the list, `x` and `xs`, we take a random element and a list with the rest of the elements. The recursive call is still on a list that is shorter than the original list, so we have a correct recursive structure.

The function `knuthShuffle` has the following type signature. It takes a random number generator and a list and returns a random permutation of the list.

```
knuthShuffle : Random.rand -> 'a list -> 'a list
```

Note that you will have to first create the random number generator and then pass it as a parameter to the `knuthShuffle` function.

Like with our other encryption/decryption functions, use a `let` expression to help do the steps of the computation.

In this case it is easier to think of recursion on the length of the list. Here is a framework:

```
fun knuthShuffle gen lst =
        if length lst < 2
           then lst
           else random-elt :: (knuthShuffle gen other-elts);
```

To get the `random-elt` and the list of `other-elts` you will need to use the random number generator and some functions from the `List` package. Read through the documentation for the `List` package at `http://sml-family.org/Basis/list.html`. You can use the functions by typing the package's name and then the name of the function. For example, typing

```
List.nth ([#"A", #"B", #"C", #"D"], 2);
```

into SML results in a response of

```
val it = #"C" : char
```

II Representing Numbers in Other Bases

We normally write numbers in decimal notation, base 10. There is nothing special about 10 except that it is widely used today. The Aztec, Mayan, and Celtic civilizations used the *vigesimal* system, base 20. The Sumerians and Babylonians used the *sexagesimal* system, base 60. In computer science,

we sometimes use the binary system, base 2, and the hexadecimal system, base 16. The octal system. base 8, is occasionally used as well.

Let us consider systems of representing numbers using bases between 2 and 20, inclusive. For the digits beyond 9, we will use letters, starting at `A` and omitting `I` because it is too easily confused with the digit `1`. The value of the base is often called the *radix*. We will specify the radix, when necessary, as a decimal subscript. For example, the following are different representations for the same number:

101111_2 57_8 47_{10} $3B_{12}$ $2F_{16}$ 27_{20}

For problems 2 - 5 we will assume radix values between 2 and 20 and raise an exception for any of the functions that receive input that violates this assumption.

2. [1 point] Write a function called `digitToChar` that takes as input a number representing a single digit and returns the character that corresponds to that digit. For example,

`digitToChar 8` returns the character `#"8"`, and
`digitToChar 11` returns the character `#"B"`.

This function should only be called with digits for radix values up to 20, inclusive. Declare and raise an exception `RadixException` if you receive a digit that is not valid for that radix range. .

```
digitToChar : int -> char
```

Take a look at `digitToInt` and `intToDigit` from the last assignment. Note that A-Z have sequential underlying number representations (accessible via `ord`)

3. [1 point] Write the inverse function `charToDigit` that takes a character representing a digit and returns its numerical representation. For example, `charToDigit #"B"` returns 11. Raise the `RadixException` if the input character does not represent represent a valid digit given our radix constraint of at most 20, inclusive.

```
charToDigit : char -> int
```

4. [2 points] Now that we have digit-level functions, we can use them to write functions that process numbers. Write a function `toRadixString` that takes a radix and an integer and produces the string representation of the integer in the specified radix. For example,

`toRadixString (3,47)` returns the string `"1202"`, and
`toRadixString (18,71)` returns `"3H"`.

Raise the exception `RadixException` if the radix is not between 2 and 20, inclusive, or if the integer is negative. Non-zero values should be represented without leading zeroes. Zero should be represented by a string with a single

zero. Write your function from scratch without using any built-in conversion functions.

```
toRadixString : int * int -> string
```

5. [2 points] Write the inverse function `fromRadixString` that takes a radix and a string of "digits" and produces the corresponding integer. For example,

`fromRadixString (3,"1202")` returns the integer `47`, and
`fromRadixString (18,"3H")` returns `71`.

Raise `RadixException` if the radix is not between 2 and 20, inclusive; if the string is empty; or if any of the "digits" in the string are out of range for the given radix. Allow the SML system to raise the exception `Overflow` if the string represents an integer that is too large for the type `int`.

```
fromRadixString : int * string -> int
```

Last thoughts on number representation In SML, the largest value for an `int` is $2^{30} - 1$, or 1073741823. On current 64-bit processors, the largest integer is $2^{63} - 1$, a considerably larger number that is nearly 10^{19} but is still too small for many purposes. To get around the limitation, programmers use packages that represent very large integers as lists (or arrays) of "digits" with a very large radix. For example, we could take the radix to be 65, 536, and our numbers would be represented by lists of "digits" that range from 0 through 65, 535. In SML, there is a datatype `IntInf` that is implemented in that way, only with an even larger radix.

Appendix I: Random Number Generators

For Problem 1, you will need to generate random numbers. Informally speaking, a sequence of numbers is *random* if the next number is not predictable from its predecessors. True randomness is difficult to find. People who are serious about random numbers often use natural phenomena, like cosmic rays or quantum events, to generate sequences of bits. For many uses, however, *pseudo-random* numbers that are generated by an algorithm are adequate. Most modern programming languages have facilities for generating pseudo-random numbers using an algebraic formula that produces sequences that will eventually repeat, but have no apparent pattern in the short run. The generators start with a *seed* and generate numbers based on that value. If two people use the same seed, they will get identical sequences of numbers. To avoid such duplication, people often take a reading of the computer's clock and use the low order bits that represent tiny fractions of a second for the seed.

In SML, we create a random number generator with two integers as seeds.

```
val generator = Random.rand(47,42);
```

Don't use 47 and 42. Pick your own seeds.

The generator has type `Random.rand`. Using the generator, we can generate sequences of pseudo-random numbers.

```
Random.randInt : Random.rand -> int
Random.randNat : Random.rand -> int
Random.randReal : Random.rand -> real
Random.randRange : (int * int) -> Random.rand -> int
```

The function `Random.randRange` will be most useful for us here. The call

```
Random.randRange (0,6) generator
```

will produce a pseudo-random value between 0 and 6, inclusive.

Notice that producing random numbers is in conflict with one of the fundamental principles of functional programming—calling a function with the same arguments should always produce the same value. The random number functions have the side effect of changing some hidden value that affects the next result.

References and Credits

The image in Figure 1 is from the Scottish Rite Masonic Museum & Library—`http://nationalheritagemuseum.typepad.com/library_and_archives/2010/03/radio-orphan-annies-secret-society.html`.

The image in Figure **??** is licensed under CC BY-SA 2.0 via Commons—`https://commons.wikimedia.org/wiki/File:Mastermind.jpg#/media/File:Mastermind.jpg`.