Problem C: Programming the EDSAC

Greater New York Region

October 30, 2011

There are two different approaches we might try for this problem. One uses an algorithm that works in any programming language. The other only works properly on certain types of hardware. We program the first in C++, the second in Java.

First Version: C++

The first thing to do is to set up the boilerplate: read P (the number of data sets), then read and process the data sets themselves.

1 ⟨*c.cpp* 1⟩≡

```
⟨includes [[C++]] 2b⟩
using namespace std;

⟨constants [[C++]] 4b⟩
⟨subsidiary function declarations [[C++]] 3a⟩
⟨subsidiary function definitions [[C++]] 2e⟩

int main()
{
    int p;
    cin >> p;
    for (int i = 0; i < p; i++) {
        ⟨read and process a single data set [[C++]] 2a⟩;
    }
    return 0;
}
```

This code is written to file `c.cpp`.

Each data set consists of N (the data set number), and a fractional decimal number, D, which we read as a string

2a ⟨*read and process a single data set* ⟦**C++**⟧ 2a⟩≡ (1)

```
int n; string d;
cin >> n;
cin >> d;
```

⟨*process data set* #`n` *consisting of the value* `d` ⟦**C++**⟧ 2c⟩

We know now that we need the `<string>` header:

2b ⟨*includes* ⟦**C++**⟧ 2b⟩≡ (1) 2d ▷

```
#include <string>
```

Processing a data set consists of printing the data set number, `n`, followed by a space, followed by the data value, `d`, interpreted as an EDSAC instruction. We use a function, `e_convert`, to do the real work:

2c ⟨*process data set* #`n` *consisting of the value* `d` ⟦**C++**⟧ 2c⟩≡ (2a)

```
cout << n << "␣" << e_convert(d) << endl;
```

Uses `e_convert` 2e.

2d ⟨*includes* ⟦**C++**⟧ 2b⟩+≡ (1) ◁ 2b 3d ▷

```
#include <iostream>
```

The `e_convert` function determines the 17-bit pattern for the fraction (we use `long` for `bit_pattern`, as it is guaranteed to be at least 32-bits long, and we need one more bit than `int` guarantees). We break the input string into `prefix` and `suffix`, separated by a decimal point. We use `suffix` to generate the appropriate `bit_pattern`, using the `e_instruction` function to convert the pattern into the appropriate EDSAC instruction. Naturally, we take care to deal properly with negative input values. By ignoring the sign when we set `bit_pattern`, we automatically get a "round toward zero" effect when we negate it (assuming that we simply ignore bits after the 16th).

2e ⟨*subsidiary function definitions* ⟦**C++**⟧ 2e⟩≡ (1) 4c ▷

```
string e_convert(string d)
{
    long bit_pattern = 0;
    ⟨set is_negative iff our value is < 0 ⟦C++⟧ 3b⟩;
    ⟨use decimal point to determine integral prefix and suffix ⟦C++⟧ 3c⟩;
    ⟨return results for trivial cases ⟦C++⟧ 3e⟩;
    ⟨set bit_pattern, depending on the digits in suffix ⟦C++⟧ 4a⟩;
    if (is_negative) { bit_pattern = -bit_pattern; }
    return e_instruction(bit_pattern & 0x1ffff);
}
```

Defines:
`e_convert`, used in chunks 2c and 3a.
Uses `e_instruction` 4c.

3a ⟨*subsidiary function declarations* ⟦**C++**⟧ 3a⟩≡ (1) 5a▷

```
string e_convert(string);
```

Uses `e_convert` 2e.

An input value is negative if (and only if) the first character of its string representation is `'-'`.

3b ⟨*set* `is_negative` *iff our value is* < 0 ⟦**C++**⟧ 3b⟩≡ (2e)

```
bool is_negative = (d[0] == '-');
```

We convert the string to the left of the decimal point into an `int` (skipping the negative sign, of course, if there is one), and to the right of the point as a `long long`, as it we need 64 bits to hold all 16 digits (we must pad the suffix out to the full 16 digits for the algorithm to work). We subtract the string length from 18 (rather than 16) to get the number of zeros to pad with because the there are 2 characters (a single digit and a decimal point) before the suffix.

The standard C (as opposed to C++) string functions seem to provide for slightly cleaner code here, so we use `c_str` to get a `char *`.

3c ⟨*use decimal point to determine integral* `prefix` *and* `suffix` ⟦**C++**⟧ 3c⟩≡ (2e)

```
const char *s = d.c_str();
if (is_negative) { s++; }   // skip the sign char
int prefix = atoi(s);
long long suffix = atoll(strchr(s, '.') + 1);
suffix = pad(suffix, 18 - strlen(s));
```

Uses `pad` 5d.

The `cstdlib` header provides `atoi` and `atoll`, `cstring` provides `strlen` and `strchr`.

3d ⟨*includes* ⟦**C++**⟧ 2b⟩+≡ (1) ◁2d 5b▷

```
#include <cstdlib>
#include <cstring>
```

Negative 1.0 translates to an EDSAC bit pattern of `0x10000`, and any other valid values must have a prefix of zero, so we go ahead and deal with these cases immediately.

3e ⟨*return results for trivial cases* ⟦**C++**⟧ 3e⟩≡ (2e)

```
if (is_negative and prefix == 1 and suffix == 0) {
    return e_instruction(0x10000);
} else if (prefix > 0) {
    return "INVALID␣VALUE";
}
```

Uses `e_instruction` 4c.

The algorithm for turning 16 decimal digits into a binary fraction with the appropriate bits to the right of the binary point is simple: We treat our 16 digit "`suffix`" as a fraction with an imaginary decimal point to the left of the 16 digits. For each digit, we multiply the fraction by two. If the result of the multiplication gives us a result with a 1 to the left of our imaginary decimal point, we remove the 1 and set the next bit in our `bit_pattern`. When we've processed all 16 digits, `bit_pattern` contains the 16-bit fixed-point (truncated) binary fraction as nearly equivalent as possible to the specified decimal fraction. We ignore any bits after the 16th, truncating the result (which, as explained above, meets the "round toward zero" requirement).

4a ⟨*set* `bit_pattern`*, depending on the digits in* `suffix` ⟦**C++**⟧ 4a⟩≡ (2e)

```
for (int i = 0; i < 16; i++) {
    suffix *= 2;
    bit_pattern <<= 1;
    if (suffix >= ADJUST) {
        suffix -= ADJUST;
        bit_pattern += 1;
    }
}
```

To deal with our imaginary 16-digit decimal point, we need a constant equivalent to 10^{16}.

4b ⟨*constants* ⟦**C++**⟧ 4b⟩≡ (1) 5c ▷

```
#define ADJUST  10000000000000000LL
```

To convert an EDSAC bit pattern, `e`, into an EDSAC instruction, we simply concatenate the 5-bit character from the collating sequence with the decimal value of the next 11 bits and an `'F'` or `'D'`, depending on whether the last bit is 0 or 1, respectively. We separate the three parts of the instruction with spaces (for readability, and because that's what the problem requires).

4c ⟨*subsidiary function definitions* ⟦**C++**⟧ 2e⟩+≡ (1) ◁ 2e 5d ▷

```
string e_instruction(long e)
{
    stringstream result;
    char prefix = COLLATING[(e >> 12) & 0x1f];
    char suffix = ((e & 1) == 0) ? 'F' : 'D';
    result << prefix << "␣" << ((e>>1) & 0x7ff)
        << "␣" << suffix;
    return result.str();
}
```

Defines:
`e_instruction`, used in chunks 2e, 3e, and 5a.

5a ⟨*subsidiary function declarations* [[**C++**]] 3a⟩+≡ (1) ◁3a 5e▷

```
string e_instruction(long);
```

Uses `e_instruction` 4c.

5b ⟨*includes* [[**C++**]] 2b⟩+≡ (1) ◁3d

```
#include <sstream>
```

The 5-bit EDSAC teleprinter code gives a collating sequence of:

```
PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV
```

5c ⟨*constants* [[**C++**]] 4b⟩+≡ (1) ◁4b

```
#define COLLATING   "PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV"
```

To pad out a `long long` decimal number, `n`, by `d` decimal places on the right, we simply multiply it by $10^{\mathtt{d}}$. We could certainly do this more efficiently, but it's not really worth the trouble, since `d` will never be more than 15.

5d ⟨*subsidiary function definitions* [[**C++**]] 2e⟩+≡ (1) ◁4c

```
long long pad(long long n, int d)
{
    for (int i = 0; i < d; i++) {
        n *= 10;
    }
    return n;
}
```

Defines:
`pad`, used in chunks 3c and 5e.

5e ⟨*subsidiary function declarations* [[**C++**]] 3a⟩+≡ (1) ◁5a

```
long long pad(long long, int);
```

Uses `pad` 5d.

Second Version: Java

The second approach works only on floating point hardware with a binary mantissa and exponent. Of the allowable contest languages, only Java *guarantees* this to be true. Therefore, we code the second approach in Java.

The only real difference between the two versions (aside from contrastive naming conventions) is in the `eConvert` method, so we start with that. We rely on the fact that the high-order 17 bits of the binary mantissa of a floating point number is (for practical purposes) exactly the bit pattern we want, so we simply convert the input string into a floating point number, multiply it by 2^{16} (to shift the bits past the decimal point, and then convert it into a 17 bit integer. Since Java `float`'s have a 23-bit mantissa, it should be okay to use them instead of `double`'s, but we use a `double` anyway.

6a ⟨*subsidiary method definitions* **⟦Java⟧** 6a⟩≡ (7a) 6c▷

```
private static String eConvert(String d) {
    double x = Double.parseDouble(d);
    boolean isNegative = x < 0;
    ⟨return results for trivial cases ⟦Java⟧ 6b⟩
    int bitPattern = (int)(x * 0x10000);
    return eInstruction(bitPattern & 0x1ffff);
}
```

Defines:
`eConvert`, used in chunk 7d.
Uses `eInstruction` 6c.

The trivial cases are just as before, except that, because of precision issues, we need to check the `String` instead of the `double` representation for values $-2 < x < -1$.

6b ⟨*return results for trivial cases* **⟦Java⟧** 6b⟩≡ (6a)

```
if (d.matches("-1[.][0-9]*[1-9][0-9]*")) { return "INVALID␣VALUE"; }
else if (x == -1.0) { return eInstruction(0x10000); }
else if (Math.abs(x) >= 1.0) { return "INVALID␣VALUE"; }
```

Uses `eInstruction` 6c.

From here on, everything works exactly as it does in the C++ version. Since the code that follows is (basically) just a Java translation of the C++ code, we don't bother to repeat the explanations.

6c ⟨*subsidiary method definitions* **⟦Java⟧** 6a⟩+≡ (7a) ◁6a

```
private static String eInstruction(int e) {
    char prefix = COLLATING.charAt((e >> 12) & 0x1f);
    char suffix = ((e & 1) == 0) ? 'F' : 'D';
    return prefix + "␣" + ((e>>1) & 0x7ff) + "␣" + suffix;
}
```

Defines:
`eInstruction`, used in chunk 6.

6d ⟨*constants* **⟦Java⟧** 6d⟩≡ (7a)

```
private final static String
    COLLATING = "PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV";
```

7a ⟨*C.java* 7a⟩≡

```
⟨imports [[Java]] 7b⟩
public class C {
    ⟨constants [[Java]] 6d⟩
    ⟨subsidiary method definitions [[Java]] 6a⟩

    public static void main(String[] args) {
        int p;
        Scanner in = new Scanner(System.in);
        p = in.nextInt();
        for (int i = 0; i < p; i++) {
            ⟨read and process a single data set [[Java]] 7c⟩
        }
    }
}
```

This code is written to file `C.java`.

7b ⟨*imports* **[[Java]]** 7b⟩≡ (7a)

```
import java.util.Scanner;
```

7c ⟨*read and process a single data set* **[[Java]]** 7c⟩≡ (7a)

```
int n; String d;
n = in.nextInt();
d = in.nextLine().trim();
⟨process data set #n consisting of the value d [[Java]] 7d⟩
```

7d ⟨*process data set #*`n` *consisting of the value* `d` **[[Java]]** 7d⟩≡ (7c)

```
System.out.println(n + "␣" + eConvert(d));
```

Uses `eConvert` 6a.

A Chunk Index

B Identifier Index