Problem-Based Intro. to Computer Science
Binary Arithmetic Strings Lab

09/27/2011

1 Problem

We understand a sequence of decimal digits to mean a particular number because of the position of each digit. For example, $24 = 2\times10^1+4\times10^0$ and $365 = 3\times10^2+6\times10^1+5\times10^0$. But what if we counted by fists rather than by fingers? Then there would be only two *bits*, or binary digits: 0 and 1.

Nevertheless, it is still possible to write numbers in binary notation. Below are some examples of decimal numbers and their binary equivalents in a mathematical notation.

$$5_{10} = 101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

and

$$24_{10} = 11000_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

We can represent binary numbers in Python using strings. For example, we *could* represent 11000_2 as `'11000'`. As with decimal numbers, the most significant digit is on the left.

Or we could represent 11000_2 as `'00011'`. Notice the bits are reversed in this example. Why would we represent a binary number as a reverse-order string? If we do so, then each time we 'peel off' the head of the string, we will get a digit with the next higher power of 2. In addition, trailing zero 'bits' will not be significant, and we may ignore them for valuation purposes.

In this lab, we will use the following binary number representation:

- The Python empty string `''` represents zero.
- The Python string '$b_0 \cdots b_n$' represents the binary number $b_n \cdots b_0$.

This representation of binary numbers does not produce unique strings; we can have as many trailing zeros as we wish without changing the meaning of a number. For example, `'00011'` and `'0001100'` both refer to 11000_2. This also means that we can write zero as `'0'`.

You will write Python functions to process binary number representations.

Constraints

You may **not** use any arithmetic operations that have numeric operands unless stated otherwise for a specific task.

Examples and Tasks

Here are examples of the functions you will design, implement and test.

```
>>> isZeroBinaryString('')
True
>>> isZeroBinaryString('0')
True
>>> isZeroBinaryString('000')
True
>>> isZeroBinaryString('010')
False
>>> addOne('0')    # add bit value 1 to binary representation of 0
'1'
>>> addOne('')
'1'
>>> addOne('1')
'01'
>>> addOne('01')
'11'
>>> addOne('11')    # add bit value 1 to binary representation of 3
'001'
>>> addBit('1', '101')
'011'
>>> addBit('0', '101')
'101'
>>> addBinary('01', '11')
'101'
>>> addBinary('1', '11')
'001'
>>> addBinary('', '01')
'01'
>>> binaryToInt( '' )
0
>>> binaryToInt( '01' )
2
>>> intToBinary( 2 )
'01'
>>> intToBinary( 6 )
'011'
```

Required Functions

1. `isZeroBinaryString` is a function that takes a binary number representation (i.e. a string of `1`s and `0`s), and returns a Boolean that indicates whether or not the representation represents zero. This function must be implemented using structural recursion.

2. `addOne` is a function that takes a binary number representation B, and returns a binary number representation that is the successor of B. This function must be implemented using structural recursion.

3. `addBit` is a function that takes a bit and binary number representation B, and returns a binary number representation that is the sum of the bit and B. This function must use `addOne`.

4. `addBinary` is a function that takes two binary number representations, and returns a binary number representation that represents their sum. To develop an algorithm that is almost structurally recursive, we must consider what happens if either of the two binary number representations is the empty string. We can determine what should happen if neither is empty from the following equations.
   Let **A** and **B** be reverse-order string representations of binary numbers of possibly different lengths, where $\mathbf{A} = a_0 \times 2^0 + a_1 \times 2^1 + \cdots + a_n \times 2^n$ and $\mathbf{B} = b_0 \times 2^0 + b_1 \times 2^1 + \cdots + b_m \times 2^m$. The a_i and b_i are the i^{th} bits of A and B respectively. Therefore we have:
   $\mathbf{A} + \mathbf{B} =$

   $$(a_0 \times 2^0 + a_1 \times 2^1 + \cdots + a_n \times 2^n) + (b_0 \times 2^0 + b_1 \times 2^1 + \cdots + b_m \times 2^m) =$$

   $$a_0 \times 2^0 + b_0 \times 2^0 + 2 \times ((a_1 \times 2^0 + \cdots + a_n \times 2^{n-1}) + (b_1 \times 2^0 + \cdots + b_m \times 2^{m-1}))$$

5. `binaryToInt` is a function that takes a binary number representation $bstr$, and returns its integer value. Design and implement this function using *iteration* and arithmetic operations to compute the result.

6. `intToBinary` is a function that takes a numeric, integer value and returns its reversed-string, binary number representation. Design and implement this function using *iteration* and arithmetic operations to construct the result.

Problem-Solving Session (20%)

You will work in a team of five to six students as determined by your instructor. Each team will work together to complete the following activities.

1. Write the following mathematical expressions and perform the operations:
   (a) Express the decimal value 22 as sequence of binary digits using the mathematical notation shown at the beginning of this assignment.
   Identify the N and b_i values in:
   $22_{10} = N_2 = b_i \times 2^i + b_{i-1} \times 2^{i-1} + \cdots + b_0 \times 2^0$

   (b) Express the binary value 1010_2 as a decimal digit sequence using the mathematical notation shown at the beginning of this assignment.
   Identify the N and d_i values in:
   $1010_2 = N_{10} = d_i \times 10^i + d_{i-1} \times 10^{i-1} + \cdots + d_0 \times 10^0$

   (c) Evaluate: $1011_2 + 1_2 =$? in base 2?

(d) Evaluate: $1011_2 + 1010_2 = ?$ in base 2?

2. Write pseudo-code for the function `isZeroBinaryString`.
3. Write pseudo-code for the structurally recursive function `addOne`.
4. Write a substitution trace of `addOne('1101')`.
5. Write pseudo-code for the recursive function `addBinary`.

Number each item, and hand in your team's sheet(s) at the end of problem-solving.

Implementation (80%)

Each student will *individually implement* and submit their own solution to the problem as a Python program named `binary.py`. This file should contain the following components:

- Python implementations of the *required functions* described in the *Required Functions* section above.
- Python functions that test the specified functions. For each required function, there should be one test function that performs at least 4 input/output checks.

Also you will submit a text document, `readme.txt`, that contains a description of your design and your test cases.

The program shall be runnable from the command line of a terminal window as follows:

```
python3 binary.py
```

When run, the program must execute your testing functions of each of the required binary arithmetic library functions and display the test results.

Your implementation grade will be based on these factors:

- 60%: The program implements the *Required Functions* correctly. Each function is worth 10%.
- 5% The code has one test function for each required function, and each test function executes at least 4 test cases for its subject of test.
- 5%: The code follows the style recommendations on the course web site.
- 10%: The `readme.txt` file contains appropriate design documentation, run, and test information. The file should have the following components:
  - A brief summary of your design.
  - A description of the test functions for each subject of test that includes a list of the test cases each test function executes. Each case should identify the inputs and expected outputs, plus an explanation of why that case is important to test.

Submission

Compress your `binary.py` and `readme.txt` into one file named **binary.zip**, and submit that to the **myCourses dropbox** for this lab. The `zip` compression command is:

```
zip binary.zip binary.py readme.txt
```