SE1-SMP Lab 4

4.1 Binary Addition & Subtraction

In the slides (PythonIntro) we have created a program for the addition of binary numbers. Extend that program to also implement subtraction (phase 1) of positive numbers and (phase 2) also of negative numbers.

- Phase 1: only handle subtraction of two positive numbers; the result is positive or negative (two's complement)
    - Package what we had in PythonIntro in a separate function (i.e., the second half of the program which returns S when given A and B)
    - Check if the input contains a + (addition) or a - (subtraction). Hint: use "in". How to know that? Google 'python check if string contains character'. Google is your friend
    - If a -, compute the two's complement of the second number. Hint: define auxiliary functions "Not" (using Nand), "Nots" (to Not all bits in a list) and "Inc" (using the addition function that we have already defined!)
    - Extend as before
- Phase 2: extend to handle negative inputs
    - Check if first character is -
    - Check if the remainder contains +- (add negative number), -- (subtract negative number) or just + (add positive number)
    - Process as above, by two's-complementing where necessary

4.2 Generic Converter bin, hex, dec

In this assignment we'll make a generic converter from bin, hex or decimal to the other two formats. Input is a string starting with 0b or 0x, or consisting only of decimal digits. If the string starts with 0b it should only contain bits (0 or 1) after the b. If it starts with 0x it should only contain hexadecimal digits after the x.

When designing your solution think carefully about how to convert hexadecimal digits to numbers. One way is to store the hex digits in a list and determine the index of each digit. A faster way is to use a Python "dict" (Goodle Python dict).

Hint: there are many ways to do this. One way uses generic functions which take a number and a list of digits to convert. For instance:

```
for i,d in enumerate(n):
    res = rdx*res+tbl[d]
```

calculates the value of digits n if **`rdx`** is the base (2 for bin, 16 for hex) and tbl hold the numeric value of each digt, say **`tbl['A']=10`**.

Converting using ['0','1'] converts to or from binary, whereas converting using ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'] converts to and/or from hex.

Note: Python has functions int()/bin()/hex() which do the job for you. Use them if you must, but using built-in functionality doesn't teach you to program.

4.3 Data Path

The data path is the heart of the core of a CPU. It is a bunch of very fast memory (the registers) connected to the ALU, continuously executing computations. In this assignment we mimic a data path.

- The set of registers is represented as a list of numbers. For example, it could be:

  R = [0, 0, 0, 0, 0, 0, 0, 0]

  This would represent eight registers, which you could access to read or write the value as:

  R[[0] = 1234 (to set the value in register 0)

  myVar = R[5] (to read the value of register 5
- The ALU is represented as a function **`ALU(A,B,F)`**, which takes two values (A and B) and a 'function' (F) as arguments. The function is a number which determines which operation the ALU will execute. For example:
  - 1 means PLUS (addition)
  - 2 means MULT (multiplication)
  - 3 means MINUS (subtraction)
  - 4 means XOR (eXclusive OR)
  - 5 means NOT (invert)
  - 6 means // (integer division)
  - Etcetera

  If the operation expects only one operand (e.g. NOT), B is ignored. The function returns the numeric result of the operation and a number representing status flags, represented as bits in a number. For instance, flag 'zero' is set when the result is zero; flag 'negative' is set when the result is negative. You may decide which flags you want to return and how to represent them in a number.

Define a function **`tick(a,b,c,f)`** which executes one cycle:

- Parameters a and b are the index of the registers that contain the values for A and B going to the ALU.
- Parameter c is the index of the register where the numeric output of the ALU (result of the computation) should be stored..
- Parameter f is the actual value of F going to the ALU
- Function ALU is called with the right arguments. The return is a number containing the calculated value, and a number encoding the flags.
- The flags are always stored in register 0

For example:

```
R = [0,0,5,6,0,0]
```

```
The registers have been loaded with the input values for the ALU
```

```
tick(2,3,4,2)
```

```
This tells the ALU that the input values are in registers 2 and 3,
the output value should be stored in register 4, and the operation
to perform is MULT, so it can get to work
```

```
print(R)
```

This will display **`[0,0,5,6,30,0],because 5 * 6 = 30`**

Note that executing sequences of 'ticks' is very much like executing an ISA program. For example:

```
R = [0,100,1,0,0,2]
```

```
Load the initial values into the registers
```

```
tick(1,2,3,6)
     Inputs in Registers 1 and 2, output to register 3, division. What
     will the contents of R look like after this step?
tick(2,3,4,1)
     Inputs in R 2 and 3, output to R 4, operation PLUS
tick(4,5,2,6)
     Inputs in R 4 and 5, output to R 2, operation division
...
tick(1,2,3,6)
tick(2,3,4,1)
tick(4,5,2,6)
```

Note that this repeats a sequence of three steps, storing the output from one step in the Registers and then using that Register as part of the input to the next step; it computes an integer approximation of the square root of the number which is initially in R[1] and at the end of each series of three steps the estimated value of the square root is in R[2] (using the Babylonian method). The following table shows the contents of R after each step, with the two input values in blue and the output value in green; you can see how the calculation progresses, and that the value in R[2] gets closer and closer to the actual value of the square root.

Note: each tick **cyan** (**blueish**) is an input register and **green** is an output register.

tick	R0	R1	R2	R3	R4	R5
0	0	100	1	0	0	2
1	0	100	1	100	0	2
2	0	100	1	100	101	2
3	0	100	50	100	101	2
4	0	100	50	2	101	2
5	0	100	50	2	52	2
6	0	100	26	2	52	2
7	0	100	26	3	52	2
8	0	100	26	3	29	2
9	0	100	14	3	29	2
10	0	100	14	7	29	2
11	0	100	14	7	21	2
12	0	100	10	7	21	2

For 100, five repetitions of the three instructions are sufficient; for larger numbers more repetitions are needed.

4.4 Binary Search Tree

Note: this is a fairly complex assignment, although the Python code is very straightforward. If you're stuck, ask a question in the forum (just added in VLO). Make the question clear and useful for other students with the same problem.

Lists, tuples and dicts are ways to combine multiple values into a single entity (and access that entity through one variable). So far, we have only put numbers and strings in a list, but it's easy to create a list which contains other lists.

A **data structure** is a logical entity which consists of a set of structures such as lists which contain base values or other structures. For instance `[['Pum', 'Walters'], 'h.r.walters@hva.nl', ['SE1', 'SMP', 'P3CS']]` is a list containing three items: a structure containing my name, my email address and a structure containing some of the classes I teach.

A data structure with lists containing other lists is often called a tree. Note that it's not always a tree. After the statements
`x=[]; y=[x]; x[0]=y`
`x` and `y` both refer to a node with one child: the other node. This is a circular non-tree called a graph.

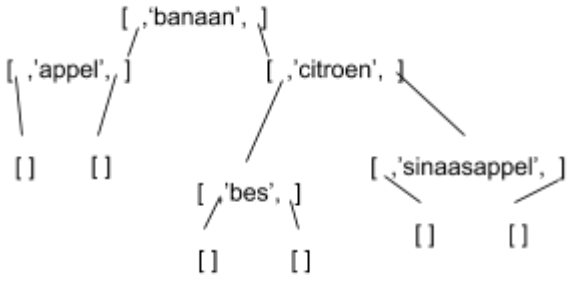
A **binary search tree** is a data structure that makes searching one value such as a keyword in a list of words very efficient.

Assignment

Our program should have

- A function `init` which gets a list of strings and builds a binary search tree (stored in a variable named `root`)
- A function `insert` which inserts one string into the tree
- A function `search` which checks if a string was contained in the list and accordingly returns its node (true value) or [] (false value); for more information refer to: https://docs.python.org/3/library/stdtypes.html#truth-value-testing).

Every node in the tree is either `[]` (the empty list) or `[A,str,B]` where str is one keyword, and `A` and `B` are the subtrees containing all keywords less than, or greater than `str`. Note that Python comparison operators (<, > etc.) when applied to strings check dictionary-order. For instance, the list `['banaan', 'appel', 'citroen', 'bes', 'sinaasappel']` might result in this tree:



- Phase 1: `insert(str,list)` as described above. This function can be made as a 'recursive' function, a function which calls itself. Something like this:

```
def insert(str,lst):
    if lst==[]:
        return [[],str,[]]
    else:
        left,word,right = lst
        if str==word:
            return lst
        if str<word:
            return [insert(str,left),word,right]
```

```
            return [left,word,insert(str,right)]
```

Test phase 1 before you continue!

- Phase 2: define function **`init(list_of_strings)`** as described above. Function **`init`** can be made by repeatedly inserting keywords into the tree being created.
- Phase 3: define function **`search(string)`** as described above. Function **`search`** can be made using the same pattern we used for insert.