UNIVERSITY
OF SKÖVDE

ASSIGNMENT 1

ALEXANDER CROLL

Advanced Programming, IT712A

Data Science, University of Skövde

Lecturer: Vicenç Torra

October 12, 2016

Table of Contents

1. Introduction

1.1. Purpose of this paper

This paper is meant to accompany the practical programming assignment (“Assignment 1”) in the course Advanced Programming.

While the exercises were implemented in Eclipse and are handed in as text file (“[Advanced Programming] Alexander Croll – Assignment 1”), this document will do the following:

- Discuss the implemented solution
- How the solution was build
- Provide references, where required

2. Exercise 1

Consider the defnition of a list of all possible combinations of digits {0, 1} with length n >= 1. That is, given n = 2 we need a function to return the following elements:

List[List[Int]] = List(List(0, 0), List(0, 1), List(1, 0), List(1, 1))

and for n = 3 we need a function to return :

List(List(0, 0, 0), List(0, 0, 1), List(0, 1, 0), List(0, 1, 1), List(1, 0, 0), List(1, 0, 1), List(1, 1, 0), List(1, 1, 1))

Consider a more general defnition where all possible combinations are computed using digits from a list. For example, the following call

combinationList(2,List(0,1,2))

will return

List(List(0, 0), List(0, 1), List(0, 2), List(1, 0), List(1, 1), List(1, 2), List(2, 0), List(2, 1), List(2, 2))

More specifically, implement a recursive function with the name combination that return this list. We should be able to call it as follows:

combination(2)
combination(3)

Define the most general case where all possible combinations from a list of digits is considered. Show examples of their use. We should be able to call it as follows:

combinationList(2,List(0,1,2))

2.1. Function 1 – “combination”

In order to find n possible combinations of {0, 1}, there are two possibilities of implementation.

- Using a tree-like approach where {0, 1} are recursively added to first the {0} and then the {1}.

n = 3	**n = 2**	**n = 1**	**(Binary) Result**
		0	0
		1	1
	0	0	00
		1	01
	1	0	10
		1	11
0	0	0	000
		1	001
	1	0	010
		1	011
1	0	0	100
		1	101
	1	0	110
		1	111

  It is also worth mentioning that the result represents binary numbers. From this, the second approach can be derived.

- Using binary numbers to retrieve all possible combinations of {0, 1}. The amount of possible combinations is given by

  $$\textit{Binary Combinations} = 2^n-1$$

  So for example, n=3 will give 2^3-1 which is 7. As can be seen in the above table, for n = 3, there are 8 combinations. Thus, the binary combinations always range from 0 to 2^n-1. In this case, the range from 0 to 7 in binary represents all possible combinations of {0, 1} which we were looking for.[1]

[1] Math is Fun (2015)

2.1.1 Implementation

The function *combination* was built according to the first alternative, using recursion to append n {0, 1] to first 0 and then do the same to 1.

- The function receives one input parameter, n, which represents the depth of each combination, so for example n = 2 should return combinations like {0, 0} while n = 4 should return combinations like {0, 0, 1, 1}.
- Before the function starts computing any combinations, it will check if the given n is negative, because we cannot return any negative combinations. If n is negative, a appropriate exception message is thrown.
- The function *combination* is driven by the *match* statement at the very end of the function. This match statement checks n. Its base case is *n = 0* in which case the input suggests that no (0) combinations of {0, 1} are wanted. Hence, the function returns an empty List (i.e. no lists of combinations available).

  The recursive case takes any number n and calls another function *combine* with the parameters n and List(List(0), List(1)). Now, the process of computing combinations of {0, 1} (which are provided in the parameter) will begin.
- The function *combine* receives two input parameters, n and a List[List[Int]], which will always be List(List(0), List(1)). It returns another List[List[Int]].
- As opposed to the function *combination*, this function's match statement checks the list. It has two cases, the base case Nil and the case for a list. The base case Nil (for an empty list) is necessary because *combine* calls itself until there is nothing of the list left. In this specific case of List(List(0), List(1)), it will always be two calls, one for the head List(0), and then one recursive call for the new head, List(1). The recursive case calls another function, *add*, with the parameters of n-1 and the head of the List (i.e. List(0)) and then concatenates this result to a recursive call to itself (as mentioned above) with the parameters n and the tail of the list (i.e. List(1)). Subtracting 1 from n is necessary to achieve the correct length of the combinations, because for a combination length of n = 2, the function really only needs to append {0, 1} once.

  The method of appending n times {0, 1} to first 0 and then doing the same thing to 1 already becomes apparent here, because List(0) is passed to *add* first and then List(1) is passed to *add*.
- The function *add* also receives two input parameters, n and List[Int]. It is only a List[Int] here because from List(List(0), List(1)) only the head (= List(0)) is passed. The final output will be a List[List[Int]].
- The match statement checks the input parameter n. It has a base case of 0 and a recursive case. When the base case is reached (n = 0), it indicates that the desired

length of a combination of {0, 1} has been reached. Thus, this case returns the list which was given as input parameter.

The recursive case is responsible for adding 0 and 1 to the input as long as n is not reached. Therefore, in this case, *add* is called recursively with parameters n-1 and the concatenation of the input List with 0. All of this is concatenated to a recursive call of *add* with parameters n-1 and the concatenation of the input list with 1. As can be seen here, both sides are basically doing the same thing, adding an element to the input list – the difference is that the first part add 0 while the second part adds 1.

2.1.2 Alternative Solution

As described in section 2.1, there is also the possibility of implementing the function *combination* based on binary numbers. In order to provide a dense alternative to the above solution, this has also been done using built-in Scala functions.

The basic idea of this solution is that based on the input n, the function will calculate 2^n-1 (max. binary number) and then recursively create all binary numbers until zero while wrapping them into lists at the same time.

- The function altC*ombination* receives one input parameter *limit* which describes the maximum depth of each combination. The output will be a List[List[Any]].
- Before starting the computation of any combination, the function will check if the input parameter is negative, because negative number of combinations cannot be built.
- Then, the function is driven by a match statement, which will match the input parameter *limit*. Its base case is 0, where an empty list will be returned. For any other positive case, a function *createCombinations* is called with the input parameter *math.pow(2, limit).toInt -1*. This input parameter is essentially passing the maximum binary number to the function *createCombinations* to tell the function at which number to start converting the binary numbers into lists.
- So the *createCombinations* function receives this input parameter n which will then be matched using pattern matching. The base case is 0, in which event a list will be returned. This list is composed by
  - Taking the number n and converting it into a binary string, using the Scala function .toBinaryString
  - Reversing this string by using the Scala function .reverse
  - Padding zeros to the end of the string in order to make sure that all strings (i.e. combinations) have the same length. This is necessary because the Scala

function .toBinaryString will covert without any leading zeros, so e.g. 3_{10} is 11_2 in binary. But if combinations of limit = 3 are required, the function should return 011_2. Additionally, the function .padTo(limit, 0) always appends numbers to the end of a string, which is why the string first needs to be reversed to "append to the beginning of the string".

- Reversing the string again to bring it back into the correct order
- Applying the function .toList which will take the string "xxx" and convert it into List(x, x, x)

While this implementation makes use of in-built Scala functions[2], one could also implement the conversion to binary numbers itself by adding another function to the solution which will do this conversion. (which would basically be a function as required to solve Exercise 2)

[2] Scala (w.y.)

2.2. Function 2 – “combinationList”

The function *combinationList* is implemented based on the same logic as above alternative solution – using the range of binary numbers to form all possible combinations of a given list with any given depth of the combinations.

2.2.1 Implementation

The function *combinationList*

- Has two input parameters, the *depth* of the combinations and *inputList*. So for instance, depth = 2 with an inputList = List(0, 1, 2) would except combinations such as List(0, 0), List(0, 1), List(0, 2), etc. All these lists will be returned inside another list, so the output is List[List[Any]]. The type Any is chosen because the function is designed for a general case, where the input cannot only be integers, but also other characters and symbols.
- Before any computations begin, an if-statement checks if the *depth* input is negative, because there are no negative numbers of combinations possible. If a negative input is given, an appropriate exception message is thrown.
  The function then starts off by calling another function, *makeOutput*. This function has two parameters, depth and limit, and is originally called with the parameters *depth* and 0. The parameter limit represents the maximum number of possible combinations, which for the specific case of {0, 1} were 2^n-1, but for the general case where lists can have any length, it will be list.length^n, where n = depth of combinations.
- So here, an if-statement checks if 0 is lower than the limit and if so, continues. This will be carried out recursively until the limit is reached, guaranteeing that all possible combinations are included. At each possible combination, a call to the function *add* is made with the parameters *depth* and *limit*. Each call of this function represents on sub-list of a combination in the final output. All of these sub-lists are concatenated by the function *makeOutput*, concluding in the output of a List[List[Any]].
- The function *add* also has two parameters, depth and limit, just like the previous function. This function makes yet another call to the function *mapToInput* n number of times. The actual number of calls is decided by the depth of each combinations, so for instance if depth = 2, then two calls to *mapToInput* are made recursively, and both of these calls are concatenated into a list (i.e. the sub-list which will contain one possible combination).
- The function *mapToInput* has two input parameters, n and inputList which corresponds to the inputList parameter from the *combinationList* function. When it is called from

within *add*, its parameter n consists of the remainder of limit % inputList.length. By using this number, the map function can map the combination to the actual numbers given in the input list.

2.2.2 Sample Computation

Let's look at this example, where only the left recursive side is computed:

combinationList(2, List("x", 7, 10)

- ⇨ makeOutput(2, 0)
    - if (limit < math.pow(inputList.length, depth))
    - (0 < 9) = true
    - List(add(depth, limit)) ::: makeOutput(depth, limit+1)
    - List(add(2, 0))
    - …increase the limit until 3^2 is reached and add all results to right side of the list
- ⇨ add(2, 0)
    - if (depth > 1)
    - (2 > 0) = true
    - add(depth-1, limit/inputList.length) ::: List(mapToOutput(limit%inputList.length, inputList)
        - recursive part: if (depth > 1)
        - (1 > 1) = false
    - List(mapToOutput(limit%inputList.length, inputList)) ::: List(mapToOutput(limit%inputList.length, inputList))
    - List(mapToOutput(0, inputList)) ::: List(mapToOutput(0, inputList))
- ⇨ mapToOutput(0, inputList)
    - if (n == 0)
    - (0 == 0) = true
    - inputList.head
    - "x"
- ⇨ List(x, x)
- ⇨ List(List(x, x), …)

When increasing *limit+1*, the part where *mapToOutput* is called, will receive different input parameters *n*.

How combinations are computed for depth = 2 and a list with three elements can be seen in below table.

Limit < list.length^{depth}	**limit / inputList.length**	**limit % inputList.length**	**Combination**	**Combination of list elements**
Limit = 0	0 / 3 = 0	0 % 3 = 0	0, 0	hd, hd
Limit = 1	1 / 3 = 0	1 % 3 = 1	0, 1	hd, tl.hd
Limit = 2	2 / 3 = 0	2 % 3 = 2	0, 2	hd, tl.tl.hd
Limit = 3	3 / 3 = 1	3 % 3 = 0	1, 0	tl.hd, hd
Limit = 4	4 / 3 = 1	4 % 3 = 1	1, 1	tl.hd, tl.hd
Limit = 5	5 / 3 = 1	5 % 3 = 2	1, 2	tl.hd, tl.tl.hd
Limit = 6	6 / 3 = 2	6 % 3 = 0	2, 0	tl.tl.hd, hd
Limit = 7	7 / 3 = 2	7 % 3 = 1	2, 1	tl.tl.hd, tl.hd
Limit = 8	8 / 3 = 2	8 % 3 = 2	2, 2	tl.tl.hd, tl.tl.hd

The “binary” representation is essentially indicating which element of a list to take for each combination. The elements in the list are indexed from 0 to list.length – 1.

3. Exercise 2

3.1. Function "baseChange"

Consider a function that given a number and a base makes a base conversion and writes the number into the new base. Given the number n and the base b, the function should return a list of numbers between 0 and b-1 corresponding to the representation in the new basis.

For example, consider the following calls to the function and the corresponding results (related to change 4532 and 45 to base 10 and base 2, respectively).

```
scala> baseChange (4532,10)

res34: List[Int] = List(4, 5, 3, 2)

scala> baseChange (45,2)

res35: List[Int] = List(1, 0, 1, 1, 0, 1)
```

As a summary, implement the function baseChange recursively.

3.1.1 Mathematical Background[3]

The most commonly used numeral system is known as decimal system, which is the way that we write numbers in everyday life. However, there are some other well-known numeral systems, such as

- Binary, known from binary code (Computer Science), e.g. $2_{10} = 10_2$
- Octal, e.g. $25_{10} = 31_8$
- Hexadecimal, e.g. $23.288_{10} = 5AF8_{16}$

Generally, the "base b" number

$$d_n\ d_{n-1}\ d_{n-2}\ \dots\ d_{0\ (b)}$$

equals

$$d_n * b^n + d_{n-1} * b^{n-1} + d_{n-2} * b^{n-2} + \dots + d_0 * b^0$$

or more specifically in the case of $2_{10} = \mathbf{10}_2$

$$\mathbf{1} * 2^1 + \mathbf{0} * 2^0$$

[3] Oxford Math Center (w.y.)

From this, we can derive a general algorithms of how to carry out base changes from base$_{10}$ to any other base.

The original decimal number will be divided by the base which the number should be converted to. While the remainder will be added to the head of a list, the rest of the original number is divided again and again until the number cannot be divided by the base anymore.

In practice:

Converting 45_{10} to Base$_{2:}$

45/2 = 22 | 1
22/2 = 11 | 0
11/2 = 5 | 1
5/2 = 2 | 1
2/2 = 1 | 0
*1/2 = **0** | 1* <= division stops once the result is less than 1 (here 0.5)

Now we can add the remainders to get 45_2 = 101101

Note: The first remainder will be last in the list. In this case, you cannot see this, but it will be important for other cases.[4]

3.1.2 Implementation

The Scala function *baseChange*

- Has two input parameters, both of type Integer, where the first one is the original value to be converted and the second on is the base to which the original value should be converted to.
- The output is given in form of a list of Integers.
- Before any conversion takes place, if-statements check if the original number is less than 0 and also if the new base is less or equal to 0. This mirrors some of the restrictions described in the following chapter.
- The logic is based on pattern matching where n, the original number, can either be 0 or anything else (but negative, of course). If the number 0 needs to be converted, this is fairly straight forwards as 0_{10} equals 0_{any} in any other numeral system. If n however has a different value, a function which actually does the conversion calculation is called. This function implements the generic conversion (repeated divide) algorithm that was

[4] Chalk Street (2016)

described earlier. It is recursively adding the remainder of the division to a list until it has reached its base case of 0.

When testing the function baseChange, it proves to be working correctly as it gives out correct lists of numbers for any given numbers conversion to any base. It also handles exceptions, e.g. negative original number or non-accepted bases.

3.1.3 Restrictions of the Implementation

Converting to the $Base_0$ and $Base_1$ does not provide much value in most use cases, because in case of $Base_0$, this numeral system could only display zeroes and in the case of $Base_1$, the system would only provide us with a corresponding number of symbols.

For example, by our previous definition

27_{10} => $Base_0$ cannot be represented in

$$d_n * 0^n + d_{n-1} * 0^{n-1} + d_{n-2} * 0^{n-2} + \ldots + d_0 * 0^0$$

because no matter which numbers will be inserted for d and n, 0^n will always be 0

6_{10} => $Base_1$ would be represented as

$$1 * 1^5 + 1 * 1^4 + 1 * 1^3 + 1 * 1^2 + 1 * 1^1 + 1 * 1^0$$

and thus, $6_1 = 111111$ [5]

Hence, the solution in Scala disregards these two cases.

Additionally, once converting from decimal system to any numeral system larger than $Base_{10}$, these numeral systems not only include numbers, but also letters for encoding. The Scala implementation will provide the correct sequence of numbers, but lacks to convert the corresponding numbers into letter.

For example:

$23.288_{10} = 5AF8_{16}$

and the Scala function will return List(5, 10, 15, 8)

where the elements 10 and 15 should be converted to A and F respectively

[5] McGill School of Computer Science (2008/2009)

But since the exercise specified that the return value will be a list of numbers with numbers from 0 to b-1, the Scala function does fulfill this requirement even without the final conversion to corresponding letters.

List of References

Chalk Street (2016): Base System, https://www.chalkstreet.com/aptipedia/knowledgebase/base-system/, Date of Retrieval: October 11, 2016

Math is Fun (2015: Binary Digits, https://www.mathsisfun.com/binary-digits.html, Date of Retrieval: October 12, 2016

McGill School of Computer Science: Numeral System, http://www.cs.mcgill.ca/~rwest/link-suggestion/wpcd_2008-09_augmented/wp/n/Numeral_system.htm, Date of Retrieval: October 11, 2016

Oxford Math Center (w.y.): Numbers in Different Bases, http://www.oxfordmathcenter.com/drupal7/node/18, Date of Retrieval: October 11, 2016

Scala (w.y.): Scala – API Docs, http://www.scala-lang.org/api/current/#package, Date of Retrieval: October 11, 2016