Summary of the 10 Concepts
Covered in the course 60:100 “Key Concepts of Computer Science”

1. *Basics of programming*: constant programs, programs that take inputs, conditional programs, programs that repeat a process, program composition (building programs from smaller programs), e.g the following is a program which converts fahrenheit to centigrade and which consists of three parts:

```
fahrenheit_to centigrade = (*5) . (/9) . (-32)
```

   example use `fahrenheit_to_centigrade 68 => 20`

2. *Higher order programs*: e.g. the `map` program which takes an operator and a list as input and which “maps” the operator onto the list. E.g. (where `x => y` means executing the expression `x` returns the result `y`):

```
map (+2) [2,3,4] => [4,5,6]
```

   we can define new programs using `map` e.g `square_all = map (^2)`

   example use `square_all [2,3,4] => [4,9,16]`

   Another higher order program is the `foldr` program which “folds” a list up using an operator and the algebraic identity element for that operator. For example:

   example use `foldr (*) 1 [2,3,4] => 24`

   we can also define new programs using `foldr`, e.g.

```
sum_of_list = foldr (+) 1
```

   example use `sum_of_list [1,2,3] => 6`

   Now we can define `sum_of_squares` by composing `square_all` and `sum_of_list`:

```
sum_of_squares = sum_of_list  .  square_all
```

   example use `sum_of_squares  [2,3,4] => 29`

3. *Data types*. For example, where `a::t` means `a` is of type `t`

   ```
    3::num     'd':: char   (+):: num->num->num    [5,6,7]::[num]
   ```

   "*polymorphic*" types,where * means any type and ** means any type not necessarily the same as *).

   Analysing the type of programs, e.g. Given the definition of map:

   ```
                       map op inp = [op x | x <- inp]
   ```

   the above is read as " `map` takes an operator `op` and a list inp as input and returns a list of values obtained by applying `op` to each value `x` taken from the input list `inp`. Analysis of this program shows that `map` is a polymorphic program of type:

   ```
                    map:: (*->**) -> [*] -> [**]
   ```

4. *Operations on data*: e.g list comprehensions e.g `[x^2 | x <- [1,2,3]] => [1,4,9]`
   And *relational algebraic* operators that are defined using list comprehensions: `select, project, join, intersect, union and difference` which are the *basis of all database systems*.

5. *Recursion*, which repeats a process. E.g. the program which computes the sum from `0 to n,` where n is the input:

   ```
             sum_to_n 0 = 0
             sum_to_n n = n + sum_to_n (n – 1)
   ```

   example application `sum_to_n  4 => 10`

   Also recursion on lists given as input, with example programs which check a list for an element, sort a list into ascending order, merge two lists, create a list of common elements in two lists given as input, etc. Discussion of the capability of recursion to define any algorithm.

6. *Syntax and context free grammars* to define languages..e.g. a simple language which contains expressions such as "231.56" and "4523.78543" can be defined by a grammar containing the following recursive rules:

   ```
            num        ::= seq_of_dig  .   seq_of_dig

            seq_of_dig ::= dig
                             | seq_of_dig   dig
      (the rule above defines the infinite set of sequences of digits)

                   dig ::= 0 | 1 | 2 etc
   ```

7. *Semantics and attribute grammars* which contain semantic rules explaining how the meaning of a compound expression is composed from the meanings of its parts. For the language above:

```
num        ::= seq_of_dig  .   seq_of_dig’
    VAL of num = VAL of seq_of_dig / Val of seq_of_dig’

seq_of_dig ::= dig
    VAL of seq_of_dig = Val of dig

             | seq_of_dig’   dig
    VAL of seq_of_dig = 10 * VAL of seq_of_dig’ + Val of dig

        dig ::= 0
    VAL of dig = 0 etc
                       | 1 etc
```

8. Use of *mathematical induction* to prove properties of the infinite set of natural numbers. For example, to prove that for all natural numbers n, $4^n - 1$ is divisible by 3. An introduction to a generalization of mathematical induction called *structural induction* with various examples (dominoes, climbing steps, properties of geometric shapes). Use of structural induction to prove properties of programs with respect to the infinite set of program inputs. For example, given the following definition of the program `rev` which reverses its input (where `x:xs` stands for an input list whose first element is `x` and whose "tail" of remaining elements is `xs`, and where ++ appends two lists together):

```
rev    []  = []
rev (x:xs) = rev xs ++ [x]
```

example application `rev [1,2,3] => [3,2,1]`

Prove `rev (x ++ y) = rev y ++ rev x`

i.e prove that reversing a list can be achieved by cutting the list in two pieces x and y, reversing each part independently, and appending the reversed list y to the front of the reversed list x. This can be done by structural input on the length of the first input list x.

9. *Computational complexity*. Determining the relationship between the time (or space) required to execute a program and the size of the input. For example, "linear" complexity means that it takes twice as long to execute the program if the input size is doubled. The shape of the graph depicting the relationship between the time and the input size metric `n` is called the time complexity of the program with respect to n. Algorithms with exponential time complexity are useless for all but very small inputs. Some exponential algorithms can be transformed to algorithms with lower complexity. Some transformation techniques are discussed briefly.

10. *Formal logic*. With examples from propositional logic the following aspects of formal logic are defined, discussed and applied: the grammar of the language of logical expressions, the semantics of the language (usinf truth table definitions of the operators), model theory (satisfiability, universal validity, equivalence, and logical consequence), proof theory (what is a formal proof, what is a theory, what is a theorem, theorem proving in an axiom system, soundness, correctness, and an introduction to automated theorem proving.)