1 Imperative programming

1.1 Introduction

Imperativ programming focusses on how a problem is to be solved as a list of *statements* and and a set of *states*, where states may change over time. An example is a baking recipe, e.g.,

· imperative programming
· statement
· states

1. Mix yeast with water
2. Stir in salt, oil, and flour
3. Knead until the dough has a smooth surface
4. Let the dough rise until it has double size
5. Shape dough into a loaf
6. Let the loaf rise until double size
7. Bake in oven until the bread is golden brown

Each line in this example consists of one or more statements that are to be executed, and while executing them states such as size of the dough, color of the bread changes, and some execution will halt execution until certain conditions of these states are fulfilled, e.g., the bread will not be put into the oven for baking before it has risen sufficiently.

Statements may be grouped into procedures, and structuring imperative programs heavily into procedures is called *Procedural programming*, which is sometimes considered as a separate paradigm from imperative programming. *Object-oriented programming* is an extension of imperative programming, where statements and states are grouped into classes and will be treated elsewhere.

· Procedural programming
· object-oriented programming

Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming language, and almost all machine langues follow the imperative programming paradigm.

· machine code

Functional programming may be considered a subset of imperative programming, in the sense that functional programming does not include the concept of a state, or one may think of functional programming as only have one unchanging state. Functional programming has also a bigger focus on what should be solved, by declaring rules but not explicitly listing statements describing how these rules should be combined and executed in order to solve a given problem. Functional programming will be treated elsewhere.

· Functional programming

1.2 Generating random texts

1.2.1 0'th order statistics

Listing 1.1 randomTextOrder0.fsx:

```
let histToCumulativeProbability hist =
  let appendSum (acc : float array) (elem : int) =
    let sum =
      if acc.Length = 0 then
        float elem
      else
        acc.[acc.Length-1] + (float elem)
    Array.append acc [| sum |]

  let normalizeProbability k v = v/k

  let cumSum = Array.fold appendSum Array.empty<float> hist
  if cumSum.[cumSum.Length - 1] > 0.0 then
    Array.map (normalizeProbability cumSum.[cumSum.Length - 1]) cumSum
  else
    Array.create cumSum.Length (1.0 / (float cumSum.Length))

let lookup (hist : float array) (v : float) =
  Array.findIndex (fun h -> h > v) hist

let countEqual A v =
  Array.fold (fun acc elem -> if elem = v then acc+1 else acc) 0 A

let intToIdx i = i - (int ' ')

let idxToInt i = i + (int ' ')

let legalIndex size idx =
  (0 <= idx) && (idx <= size - 1)

let analyzeFile (reader : System.IO.StreamReader) size pushForward =
  let hist = Array.create size 0
  let mutable c = Unchecked.defaultof<int>
  while not(reader.EndOfStream) do
    c <- pushForward (reader.Read ())
    if legalIndex size c then
      hist.[c] <- hist.[c] + 1
  hist

let sampleFromCumulativeProbability cumulative noSamples =
  let rnd = System.Random ()
  let rndArray = Array.init noSamples (fun _ -> rnd.NextDouble ())
  Array.map (lookup cumulative) rndArray

let filename = "randomTextOrder0.fsx"
let noSamples = 200
let histSize = 126 - 32 + 1

let reader = System.IO.File.OpenText filename
let hist = analyzeFile reader histSize intToIdx
reader.Close ()
let idxValue = Array.mapi (fun i v -> (idxToInt i, v)) hist
```

1.2.2 1'th order statistics

Listing 1.2 randomTextOrder1.fsx:

```
let histToCumulativeProbability hist =
  let appendSum (acc : float array) (elem : int) =
    let sum =
      if acc.Length = 0 then
        float elem
      else
        acc.[acc.Length-1] + (float elem)
    Array.append acc [| sum |]

  let normalizeProbability k v = v/k

  let cumSum = Array.fold appendSum Array.empty<float> hist
  if cumSum.[cumSum.Length - 1] > 0.0 then
    Array.map (normalizeProbability cumSum.[cumSum.Length - 1]) cumSum
  else
    Array.create cumSum.Length (1.0 / (float cumSum.Length))

let lookup (hist : float array) (v : float) =
  Array.findIndex (fun h -> h > v) hist

let countEqual A v =
  Array.fold (fun acc elem -> if elem = v then acc+1 else acc) 0 A

let intToIdx i = i - (int ' ')

let idxToInt i = i + (int ' ')

let legalIndex size idx =
  (0 <= idx) && (idx <= size - 1)

let analyzeFile (reader : System.IO.StreamReader) size pushForward =
  let hist = Array2D.create size size 0
  let mutable c1 = Unchecked.defaultof<int>
  let mutable c2 = Unchecked.defaultof<int>
  let mutable nRead = 0
  while not(reader.EndOfStream) do
    c2 <- pushForward (reader.Read ())
    if legalIndex size c2 then
      nRead <- nRead + 1
      if nRead >= 2 then
        hist.[c1,c2] <- hist.[c1,c2] + 1
      c1 <- c2;
  hist

let Array2DToArray (arr : 'T [,]) = arr |> Seq.cast<'T> |> Seq.toArray

let Array2DOfArray (a : 'T []) = Array2D.init 1 a.Length (fun i j -> a.[j])

let hist2DToCumulativeProbability hist =
  let rows = Array2D.length1 hist
  let cols = Array2D.length2 hist
  let cumulative = Array2D.zeroCreate<float> rows cols
  for i = 0 to rows - 1 do
    let histi = Array2DOfArray (histToCumulativeProbability
```