Chapter 0

Introduction

Most of your programming experience will have been acquired by writing conventional data-processing style programs that read data, process it, and generate output. Programs that simply read and write files are among the oldest uses of computers, and are known as *batch* programs — the program runs, computes and, when it is finished, the result is available.

If the program interacts with a user, and asks for data as it is needed for the computation, it is called an *interactive* program. Interactive programs were the next stage in the evolution of computers and software. You have probably written many interactive programs during your coursework so far.

However, many programs in the real world need to be able to process data as it arrives at the system. These programs are called *reactive* or *event-driven* because they need to be able to react to the data as soon as it arrives.

A familiar example of a reactive program is any program with a Graphical User Interface (GUI). Such a program does not wait for a specific button to be pressed, but is instead ready to respond to *any* button that is pressed.

Another application of reactive programs is embedded systems. These are systems that control electrical appliances such as DVD players, air conditioners, toasters, mobile phones, burglar alarms, car fuel injection systems, and many others. More than 99% of all microprocessors made in the world are hiding inside an embedded system.

Reactive programs have three main characteristics:

- *They are driven by the availability of data.* When a particular piece of data becomes available for processing, we say that an event occurs (hence event-driven programs). From a software viewpoint, the order in which events can occur is unpredictable — e.g. a DVD player cannot predict which button a user will press next. This means that the number of execution paths in a reactive program is *much* larger than in an interactive program. This fact alone makes writing correct code much more difficult (we have all experienced GUI programs that crash for no obvious reason).

- *They are concurrent.* Reactive systems are usually carrying out multiple activities at the same time. For example, a single web browser process can have multiple windows opened where a different download is in progress in each window. While doing this, the browser still responds to user mouse-clicks and key-strokes. From the viewpoint of the web browser, the arrival of data for each web-page or of a mouse-click is simply an event that can occur at any time.

- *The user is always in control.* While the program is processing data, it *cannot* choose to ignore the user, rather it *must* respond to the user's events (mouse-clicks, or key-strokes) in a timely fashion.

0.1 What are event-driven systems?

As noted above, reactive or event-driven systems can cover a broad spectrum of applications. The most common are:

Graphical User Interfaces (GUIs) It is *very* difficult to write correct code for GUIs without using event-driven techniques. In this course, we will show you how to construct *bug-free* GUIs.

Embedded Systems If you are an engineer, you will almost certainly be involved in designing or implementing an embedded control system for a product. The reason is simple: a microcontroller is the cheapest and most flexible way to implement the control section of almost every electronic product. Those of you who are studying the courses "Software Engineering and Project" or "Embedded Computer Systems" will find that knowledge of event-driven systems will significantly simplify the software for your projects.

Communication Protocols Modern computer systems are extensively networked, to allow them to exchange data. All computer communication depends on a protocol for the exchange of data. Communication protocols need to be able to respond to events, such as: arrival of a message, arrival of an acknowledgement, timeout after sending a message, and so on. All these events can occur in any order. A protocol can *only* be implemented reliably using event-driven programming techniques.

A "protocol stack" is a layered set of intercommunicating protocols that together permit computer-to-computer communication. Each layer in the stack is a reactive program. A protocol stack is thus, by its very nature, *highly* reactive, and *highly* concurrent.

Despite the variety of applications for event-driven systems, there are some well understood techniques for designing and implementing these systems. They can significantly reduce the time, effort and the likelihood of bugs when writing your code. They also will make your program *much* easier to change in the future.

By making clear design and implementation decisions at the beginning, you will retain intellectual control over the complexity inherent in all reactive systems.

0.2 A little history

The ideas behind finite state machines have been around for a long time now. In the 1930s, Alan Turing did research on computable functions, and invented the "Turing machine", a simple processor that has been proved to have the same capabilities as any current-day computer.

In the 1940s, McCulloch and Pitts modelled the behaviour of nerve networks, using ideas similar to those presented in these notes.

In the mid 1950s, papers by Mealy and Moore, from the Engineering community, had a big influence on the design of switching systems in telephone exchanges. Both described finite state machines as we now know them. We will look at the nature of their work a little later on.

At the end of the 1950s, Rabin and Scott published a paper that introduced non-deteministic finite state machines, and provided fruitful insights for further work. They received the ACM Turing award for this work.

0.3 An introductory example - the tap-light

Let us consider a simple but familiar example: an electric table lamp that is controlled by simply "tapping" it: when a user taps the lamp, it turns on; tap again, and it turns back off; tap again, and it turns back on; and so on. We can represent this behaviour in a *state-diagram*, as shown in fig. 1.

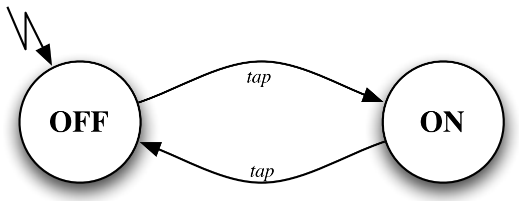

Figure 1: The tap-on/tap-off table light

In the diagram, the two circles, named off and on, represent the state of the light. The state of the light can be changed by the occurrence of an event, named *tap*. We show a transition from one state to another by a directed arc (A line with an arrow on one end) from one state to another, labelled with the name of an event. There is a special transition, shown as a “lightning-strike”, that indicates the starting state of the diagram. (In this example, the starting state is named off.) A diagram such as this is called a *finite state machine*(FSM).

This state-diagram very neatly and succinctly captures the behaviour of the light: Initially, the light is off, and the FSM is in state off. Upon receipt of a tap event, the system changes state to on, indicating that the light is now on. The FSM is now in state on. Upon receipt of a tap event, the system changes state to off, indicating that the light is now off. The FSM is now in state off.

We can show the current state by putting a *dot* inside it. For example, immediately after starting this FSM, the diagram will appear as shown in fig. 2.

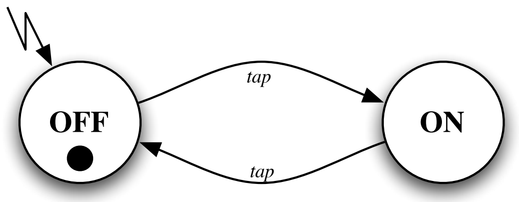

Figure 2: The tap-light in the *OFF* state

All this behaviour can be understood just by putting your finger on the current state (the one with the dot) and, upon receipt of an event, following the appropriate transition to the next state. By reading the diagram, it is easy to see what will happen in every situation.

Consider our previous diagram, with the tap-light in the *OFF* state. If we receive a tap event, the diagram changes to the *ON* state, as shown in fig. 3.

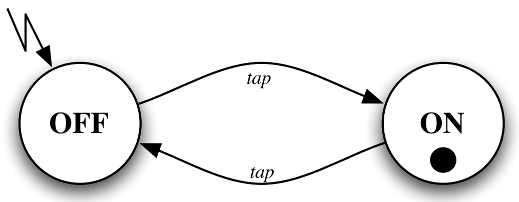

Figure 3: The tap-light in the *ON* state

0.3.1 Some nomenclature

A finite state machine has one or more *states*, represented on a state-diagram as circles. As a result of an external *event*, the FSM can make a *transition* from one state to another (possibly the same) state. A transition is represented on the diagram as a directed arc, labelled with the name of the event that causes the transition to be taken. The FSM has an *initial state*, represented on the diagram by a lightning-strike. The *current state* is represented by a large dot inside the state.

0.4 Another example - text recognition

We consider here another example, that arises in text-processing programs, such as a compiler. Suppose we wish to recognise the words "for", and "float", in a stream of text. We could construct a finite state machine, where the sequence of events is the sequence of characters in the input stream. The resulting FSM is shown in figure 4.

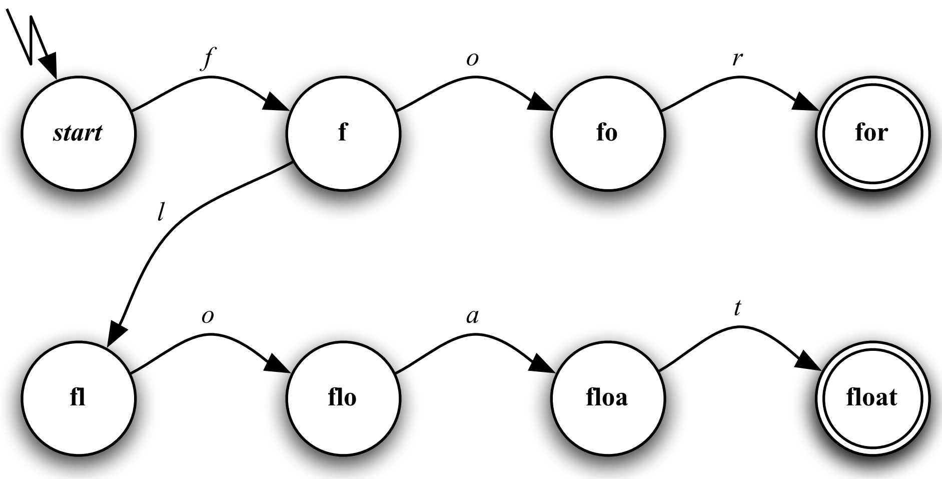

Figure 4: The text recogniser

In the FSM there are states named after the partially-recognised words: f, fo, fl, flo, and $floa$. There are also two *accepting* states named for and $float$, shown on the diagram as double-circled states. It is obvious that for each state, there are many possible input characters that will *not* be recognised. For example in state fo, the machine will not recognise the character x, because there is no transition labelled with the event x, from state fo. If an unrecognised event occurs, the machine "dies", and ceases to process symbols — it no longer has a *current state*. Effectively, the *dot* has been lost.

Diagrams of this kind often result from describing the behaviour of a *regular expression*, a topic we will deal with later.

0.5 Central concepts

There are a number of basic concepts that are central to the study of finite automata. These concepts are *alphabet* (a set of symbols), *string* (a list of symbols from an alphabet), and *language* (a set of strings from the same alphabet).

0.5.1 Alphabets

An *alphabet* is a finite, non-empty set of symbols. It is conventional to use the Greek letter Σ (sigma), to represent an alphabet. Some examples of common alphabets are:

1. $\Sigma = \{0, 1\}$, the set of binary digits.
2. $\Sigma = \{A, B, \cdots, Z\}$, the set of Roman letters.
3. $\Sigma = \{N, E, S, W\}$, the set of compass-points.

0.5.2 Strings

A *string* is a finite sequence of symbols drawn from an alphabet. A string is also sometimes called a *word*. Some examples of strings are:

1. 100101 is a string from the binary alphabet $\Sigma = \{0, 1\}$.
2. $THEORY$ is a string from the Roman alphabet $\Sigma = \{A, B, \cdots, Z\}$.
3. SE is a string from the compass-points alphabet $\Sigma = \{N, E, S, W\}$.

Empty string

The *empty string* is a string with no symbols in it, usually denoted by the Greek letter ϵ (epsilon). Clearly, the empty string is a string that can be chosen from any alphabet.

Length of a string

It is handy to classify strings by their *length*, the number of symbols in the string. The string $THEORY$, for example, has a length of 6. The usual notation for the length of a string s is $|s|$. Thus $|THEORY| = 6$, $|1001| = 4$, and $|\epsilon| = 0$.

Powers of an alphabet

We are often interested in the set of all strings of a certain length, say k, drawn from an alphabet Σ. This can be constructed by taking the *Cartesian product*, of Σ with itself k times: $\Sigma \times \Sigma \times \cdots \Sigma$. We can represent this symbolically, using exponential notation, as Σ^k.

Clearly $\Sigma^0 = \{\epsilon\}$, for any alphabet Σ, because ϵ is the only string whose length is zero.

For the alphabet $\Sigma = \{\mathrm{N, E, S, W}\}$, we find:

$$\Sigma^1 = \{\mathrm{N, E, S, W}\}$$

$$\Sigma^2 = \{\mathrm{NN, NE, NS, NW, EN, EE, ES, EW, SN, SE, SS, SW, WN, WE, WS, WW}\}$$

$$\Sigma^3 = \{\mathrm{NNN, NNE, NNS}, \cdots, \mathrm{WWS, WWW}\}$$

Σ^3, has 64 members, since it contains $4 \times 4 \times 4$ members.
The set of *all* strings that can be drawn from an alphabet is conventionally denoted, using the so-called *Kleene star*, by Σ^*, and of course has an infinite number of members. For the alphabet $\Sigma = \{0, 1\}$

$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \cdots\}$

Clearly, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$

Sometimes we do not want to include the empty string in the set. The set of *non-empty* strings is denoted by Σ^+. This is often referred to as the *Kleene plus*, by analogy with the Kleene star.

Clearly, $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$

And $\Sigma^* = \{\epsilon\} \cup \Sigma^+$

Concatenating strings

Let s be the string composed of the m symbols $s_1 s_1 s_2 \cdots s_m$, and t be the string composed of the n symbols $t_1 t_1 t_2 \cdots t_n$. The *concatenation* of the strings s and t, denoted by st, is the string of length $m + n$, composed of the symbols $s_1 s_1 s_2 \cdots s_m t_1 t_1 t_2 \cdots t_n$.

It is clear that the string ϵ can be concatenated with any other string s and that: $\epsilon s = s\epsilon = s$. ϵ thus behaves as the *identity value*, for concatenation.

0.5.3 Languages

A set of strings, all of which have been chosen from Σ^* of an alphabet Σ, is called a *language*. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is said to be a *language over* Σ.

A language over Σ does *not* need to include strings with *all* the symbols of Σ. The implication of this is that when we know that L is a language over Σ, then L is also a language over any alphabet that is a *superset* of Σ.

The use of the word "language" here is entirely consistent with everyday usage. For example the language "English" can be considered to be a set of strings drawn from the alphabet of Roman letters.

The programming language *Java*, or indeed any other programming language, is another example. The set of syntactically-correct programs is the set of strings that can be formed from the alphabet of the language (the ASCII characters).

Using the alphabets we defined earlier, we can specify some languages that might be of interest to us:

1. The language consisting of valid binary byte-values (a string of 8 0's or 1's): $\{00000000, 00000001, \cdots, 11111111\}$ This is just Σ^8.

2. The set of even-parity binary numbers (having an even number of 1's), whose first digit is a 1: $\{11, 101, 110, 1001, 1010, 1100, 1111, \cdots,\}$

3. The set of valid compass directions: $\{\mathrm{N}, \mathrm{S}, \mathrm{E}, \mathrm{W}, \mathrm{NE}, \mathrm{NW}, \mathrm{SE}, \mathrm{SW}, \mathrm{NNE}, \mathrm{ENE}, \cdots\}$

4. Σ^* is a language over an alphabet Σ.

5. $\{\epsilon\}$, the language consisting only of the empty string, is a language over any alphabet. This language has just one string: ϵ.

6. $\emptyset$, the language with *no* strings, is a language over any alphabet. Note that $\emptyset \neq \{\epsilon\}$, because $\{\epsilon\}$ contains *one* string.

Notice also that an alphabet Σ is always of a finite size, but a language over that alphabet can either be of finite or of infinite size.