Grammars and machines

What makes a language a language rather than an arbitrary sequence of symbols is its grammar.

A grammar specifies the order in which the symbols of a language may be combined to make up legitimate statements in the language. Human languages have rather relaxed informal grammars that we pick up as children. Computer languages are sometimes called formal languages because they obey an explicitly specified grammar.

When people came to design computer languages around the end of the 1950's they had to devise methods of formally specifying what the grammar of these new language was to be. By coincidence the linguist Chomsky had been investigating the possibility of formally specifying natural languages, and had published an influential paper in which he had classified all possible grammars into 4 classes. These classes of grammars are now refered to as Chomsky class 0, class 1, class 2 and class 3 grammars. It turns out that Chomsky class 2 and class 3 grammars are most suitable to describe programming languages. To understand what these different classes of grammars are we need to go into a little formal notation.

The syntax or grammar of a language can be thought of as being made up of a 4 tuple $(\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$ where:

$\mathcal{T}$ stands for what are called the terminal symbols of the language. In a human language these terminal symbols are the words or lexicon of the language. In a computer language they are things like identifiers, reserved words and punctuation symbols.

$\mathcal{N}$ stands for what are called the non-terminal symbols of the language. In a human language a non-terminal would be grammatical constructs like a sentence, a noun clause or a phrase. A computer language is likely to have a large number of non-terminals with names like **clause, statement, expression**.

$\mathcal{S}$ is the start symbol of the grammar. It is one of the non terminals. Its meaning will become clear shortly.

$\mathcal{P}$ is a set of productions or rewrite rules. These tell you how to expand a non-terminal in terms of other terminals and non-terminals.

This sounds a bit dry, but it will be clearer if we give an example. Suppose we wish to define a grammar that describes the ‘speech’ of a traffic light. A traffic light has a very limited vocabulary. It can say **red** or **amber** or **green** or **red-and-amber**. These are the terminal symbols of its language.

$\mathcal{T}$ = { **red, green, amber, red-and-amber** }

At any moment in time the traffic light is in a current state and after some interval it goes into a new state that becomes its current state. Each state is described by one of the colours of $\mathcal{T}$. This can be expressed as a set of non-terminal symbols which we will call:

$\mathcal{N}$ = { *going-red, going-green, going-amber, going-red-and-amber* }

We will assume that when the power is applied for the first time the light enters state going-red. Thus

$\mathcal{S}$ = *going-red*

A traffic light has to go through a fixed sequence of colours. These are the syntax of the traffic light language. Which sequence it goes through is defined by the productions of the traffic light language. If the light is in going-red then it must output a red and go into going-red-and-amber. We can write this down as:

going-red $\rightarrow$ ***red*** *going-red-and-amber*

This is an individual production in the traffic light language.

The whole set of productions is given by:

$\mathcal{P}$ = { *going-red* $\rightarrow$ ***red*** *going-red-and-amber*

going-green $\rightarrow$ ***green*** *going-amber*

going-red-and-amber $\rightarrow$ *red-and-amber going-green*

going-amber $\rightarrow$ *amber going-red*

}

This combination of $(\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$ ensures that the only sequence of colours allowed by the traffic light are thing like :

red red-and-amber green amber red *going-red-and-amber*

It turns out that traffic lights speak the simplest of the Chomsky classes of language, which perversely enough is class 3.

To distinguish between these classes of grammars the following notation will be used:

bold letters : **a b c** ... represent non-terminals

italic letters : *a b c* ... represent terminals

Class 3

Class 3 languages like that of the traffic light have all of their productions of the form:

$\mathbf{a} \rightarrow a\mathbf{b}$

or

$\mathbf{a} \rightarrow c$

The traffic light obviously only has the first type of production or it would stop at some point. These simple languages occur widely in nature. Look at the patterns of leaves round the stem of a plant. They will often alternate left or right, or form a spiral that can be described by a class 3 grammar. In the example in figure we can describe the plant shape by the grammar:

Plant generated by a regular grammar

$\mathcal{T}$ = { **flower, left, right** }

$\mathcal{N}$ = { *lstem, rstem* }

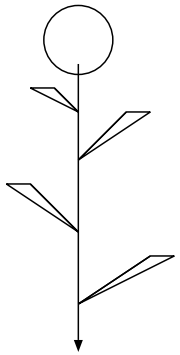
$\mathcal{S}$ = *lstem*

$\mathcal{P}$ = { *rstem*$\rightarrow$ ***flower***

lstem$\rightarrow$ ***left*** *rstem*

rstem$\rightarrow$ ***right*** *lstem*

}



Class 3 grammars are also sometimes described at regular grammars and the patterns they describe as regular expressions. It turns out that the reserved words of most computer languages can be described by class 3 grammars.

Class 2

Class 2 grammars, also called context free grammars have productions of the form

$a \rightarrow b$

where **a** is a non-terminal symbol and **b** is some combination of terminals and non terminals. We could describe the 'if' expression in an algol like language as:

if-expression$\rightarrow$ ***if*** *expression* ***then*** *expression* ***else*** *expression*

where italics are non-terminals and bold letters are terminals. Most of the syntax of algol like programming languages can be captured using class 2 grammars.

Class 1

Class 1 grammars, also called context sensitive grammars have production of the form

$abc \rightarrow axc$

where **a** and **c** are strings of terminals and non-terminals,

b

is a single non-terminal and **x** is a non-empty string of terminals and non-terminals. The string of symbols on the right hand side must be a least as long as the string on the left hand side.

The reason why these are called context sensitive is that the production of **x** from **b** can only occur in the context of **abc**. In the context free languages a non terminal can be expanded out irrespective of the context. Although the bulk of a programming language's syntax can be described in a context free fashion, some parts are context sensitive. Consider the line:

```
x:=9
```

This will only be valid if at some point previously there has been a line declaring x. The name of the variable must have been introduced earlier and it must have been specified that it was an integer or real variable. The context sensitive part of the language is dealt with by the type checking system. In untyped languages like Basic context sensitive parts are minimal. In more advanced languages they are crucial.

The class 1 grammars can be recognised by a Linear Bounded Turing machine, that is a Turing machine with a fixed length tape.

Class 0

Class 0 grammars, the most powerful class are not needed for translating programming languages. Here we have productions of the form:

$X \rightarrow Y$

where X is made up of non-terminal symbols and Y is any string of terminals or non terminals of any length.

This requires a full Turing machine to recognise it.

The Class 0 grammars also correspond to what are called the Recursively Enumerable languages.

Definition 1.

> Given string w as input, the algorithm halts and outputs YES if and only if w belongs to the language L. If w does not belong to the language L, the algorithm either runs forever, or halts and outputs NO.

Definition 2.

The algorithm takes a positive integer, say n as an argument, and produces as output a string in the language L. For every string s which is in L there must be an n so that the algorithm produces the string s.

The equivalence of these two definitions can be seen as follows:

1 -> 2 Given an algorithm A according to the first definition for language L (assumed to be non-empty), the following algorithm will enumerate L according to the second definition:

Let E be an algorithm which enumerates all strings, and so that every string appears infinitely often in the enumeration. We write E(*n*) to denote the string produced by algorithm E on input *n*. Pick a fixed string t in L (possible since L is non-empty). The following algorithm enumerates L:

> Given integer n, run algorithm A on input E(*n*) for *n* steps. If the answer is YES, output the string E(*n*). Otherwise, output string t.

Clearly this will output only strings in L, since t is in L and any string on which A halts with YES is also in L. Moreover it will output all of them, since for any string s in L we can find a *n* such that the number of steps A will take to recognise it will be less than *n* and E(*n*)=s.

Hierarchy of grammars

A programming language can be translated by using a hierarchy of grammars.

At the lowest level we use class 3 grammars to recognise the identifiers and reserved words of the language. Above that we use class 2 grammars to analyze the context free parts of the language. Finally we use type checkers to verify that the context sensitive rules of the language are being obeyed.

Program Module	Grammar
type checking	class 1
syntax analysis	class 2
lexical analysis	class 3

The hierarchy of grammars is reflected in compiler structure

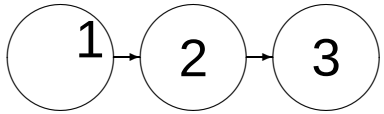
The structure of the compiler reflects this structure of the language. to each of the layers of grammar there is a module of the compiler. It also turns out that in our strategy for writing the compiler we can take advantage of a relationship which exists between classes of grammars and types of computing machines.

The idea of store and stored state will be familiar to all programmers, but a stored program computer need not in principle be anything like the Von

Neumann machines that we normally call computers. There are in principle much more general purpose designs. At the most general level digital computer capable of performing computation over time must contain a set of storage cells each capable of holding a bit. The computer is capable of existing in a number of states characterised by the values in its storage cells. If we consider these we can see that the number of states that the computer can occupy will be 2^s where s is the number of storage cells in the machine.

Computation proceeds by the computer going from one state to the next as shown below.

Computation as an evolution of numbered states



Clearly the number of state that a computer can go through in the course of a computation will be 2^s. The larger the number of storage cells in the machine the longer or more complex the sequence of state that it can go through. This relationship is familiar to us all in the way more complex programs demand more store.

To actually perform computation it is necessary to be able to modify the sequence of states that the computer goes through on the basis of input signals. To produce any useful effect the computer must generate one or

more output signals, to indicate the result of the computation. Reduced to its most simple a computer must be capable of responding to a sequence of inputs and generating appropriate outputs.

Consider a machine that has to recognise a 3 digit sequence and then respond with a yes or no according to whether or not the sequence was correct. An example might be digital door lock as shown in the diagram below. This requires the sequence 469 to be keyed in to open the lock.

This sequence of numbers can be described by a class 3 grammar: $(\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$ where

$\mathcal{T}$ = { **1, 2, 3, 4, 5, 6, 7, 8, 9, 0** }

$\mathcal{N}$ = { *s, t, u* }

$\mathcal{S}$ = *s*

$\mathcal{P}$ = {

$\boldsymbol{s} \rightarrow 4\boldsymbol{t}$

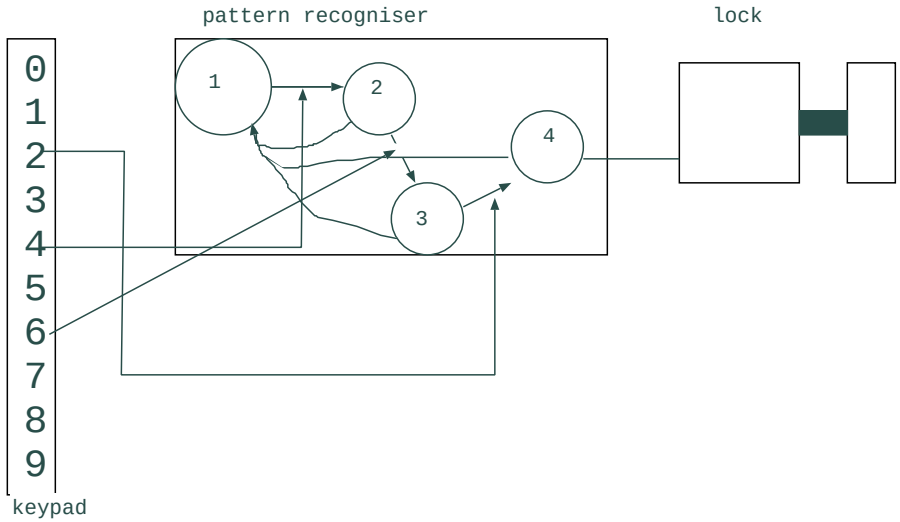
$\boldsymbol{t} \rightarrow 6\boldsymbol{u}$

$\boldsymbol{u} \rightarrow 9$

}

We define a set of numbered states corresponding to the non terminals of the grammar such that *s* = **1**, *t* = **2** etc. The machine starts in state 1 and undergoes transitions to successive states on the basis of the keys that are pressed. If any incorrect key is pressed the state reverts to the start state.

A digital door lock



The collection of numbered circles with arrows connecting them is called a state transition diagram.

How should state bits be organised?

Given that we wish to be able to represent state, it still does not follow that we will end up with a random access store. There are other possibilities, which have been tried in the past, and which still have certain limited applications. If each bit represents a state we could easily construct the door lock by stringing 4 cells together in sequence and having them activated sequentially by the and of the signal from a button and the previous state.

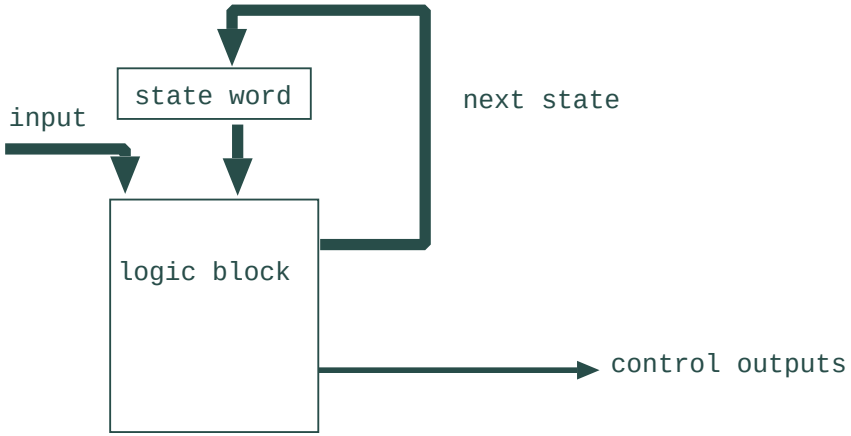
This is simple to implement and some digital logic used to be built this way, but it makes poor use of the state bits as we only get s rather than 2^s states from an s bit store.

An improved arrangement is to gather all of the bits in the computer into one word s bits long. This is then treated as a binary number and the computer program can be thought of as a mapping of the form:

program:(int x input)$\rightarrow$ (int x output)

Succesive application of the program function to the state word and the input generates a new state and an output. This is in theoretical terms the ideal way to construct a computer. For large s the number of states possible becomes astronomical. A computer with a 64 bit status word could have a state to represent every centimeter of the distance between here and the nearest star. This sort of computer is a generalised finite state automaton . If the computer is organised as shown :

A finite state machine

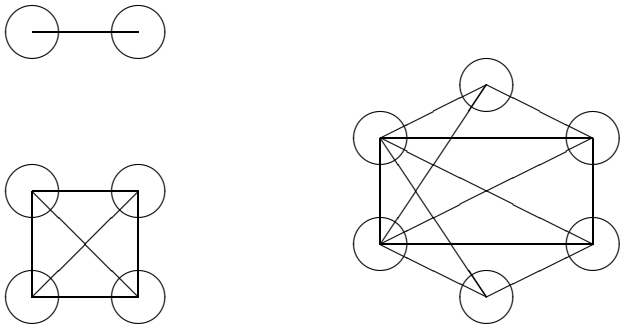


This architecture can go from any state to any other in a single step. Each bit in the current state can be taken into account in determining the next state. All values of the input bits can be taken into account likewise. A class 3 grammar can be handled by a finite state machine.

Finite state machines are widely used in computer hardware in the form of PLA's or programmable logic arrays. These are basic components of microprocessors that are used to decode machine instructions. The instruction decode unit of a microprocessor has to parse machine code. Machine code has a class 3 grammar so a finite state machine is enough.

Although this sort of machine is very fast, we have practical difficulties in scaling it up. The problem is the rate at which hardware complexity increases with the size of the computer. The number of interconnection wires required to allow each state bit to affect the next state of every other bit goes up as the square of the number of bits, and the number of logic cells (ANDs , ORs) to do this goes up quadratically in the number of state bits.

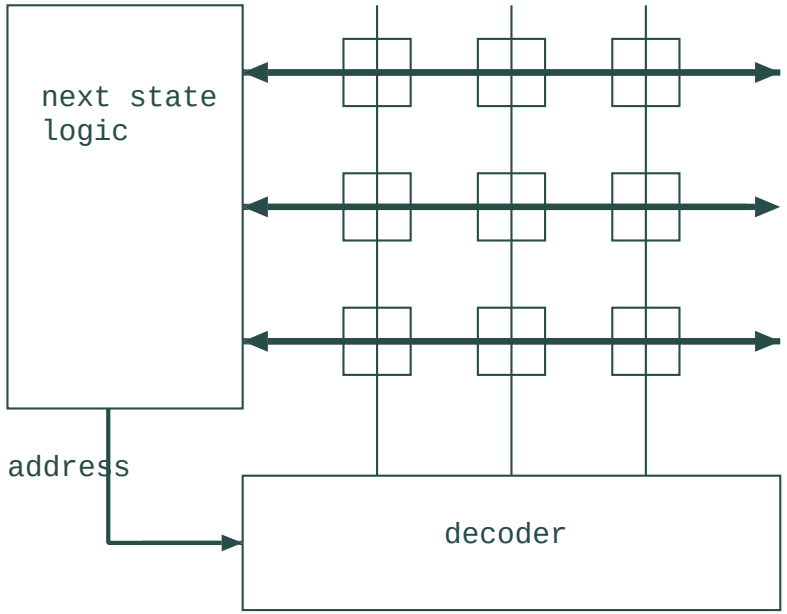
The number of wires used to connect state cells in finite automata goes up as $s(s-1)$



Random Addressed store

If instead of connecting all of the state cells to one another, we organise the state cells into subgroups of bits termed words and lead these into a common logic block then we can diminish the number of wires considerably

A random addressed store computer



arranged in a grid
9 state bits need
only 6 wires to
connect them rather
than the 72 that
a FSM would need

If we divide our *s* state bits into $w = s/b$ words each of *b* bits, and wire them up in a grid, we need only $(w + b)$ wires to join them to the common logic block. What we then have is the random access memory computer. Like a generalised finite state automaton it runs in a cycle reading the present state and modifying the state vector as a result of what it has read but it is less powerful than an FSA in that at most w bits of the state can be taken into account each cycle and at most w bits of the state altered in each cycle.

The paradox is that although the FSA is the fastest type of computing machine, used in CPU's where speed matters, it is linguistically the least competent.

Suppose that I have a class 2 grammar $(\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$

where

$\mathcal{T}$ = { **), (, 1, 2, 3** } , $\mathcal{N}$= {*s, t, u* }, $\mathcal{S}$= *s*

$\mathcal{P}$= {*s*→***(t)*** *; t*→***1u2u*→t ; u→s ; u→3*** }

This can generate sequences like

(132) or **(11322)** or **(1(132)2)**

You will find that you can not draw a state transition diagram that is capable of handling this syntax. In fact it can not be handled by a finite state machine. The machine would have to remember how many left brackets and how many 1s it had encountered and in what order they had come so that it could match them up with right brackets and 2s. Since the sequence defined by the grammar can be of arbitrary length, no finite memory could hold the information.

To handle a class 2 grammar like this you need to have an infinite stack memory. As each left bracket or one is encountered, a token representing it is pushed onto the stack. When parsing, the computer looks at the top of the stack and at the next character to decide what state to go into.

Of course in practice any stack that we build will be of finite depth. This means that looked at another way a stack machine is still a finite state automaton.

There will be sequences of symbols that are just too long to parse. For practical purposes we are willing to accept that some programs are too big to compile. But we can write our compiler as if it was going to run on a computer with an infinite stack. This technique allows us to write a program that only needs to have a small number of rules in it. The complexity of the parser is then limited by the size of the grammar itself rather than by the size of the programs it will have to compile.

When we take into account context sensitive information, we will need the full facilities of a random access memory in which we can built up information about what identifiers and types have been declared. Broadly speaking, the lexical analysis part of compiling will be handled by algorithms that mimic a finite state machine. The syntax analysis will be handled using a stack, and the type checking will use a random access heap. A machine with a finite random access store is computationally equivalent to a Linear Bounded Turing machine.