# Regular Languages

From the general properties of formal languages presented in the last section we already know a lot about the type 3 or *regular languages*. We know that they are recursive and in fact can be easily recognized by linear bounded automata (since they are a subclass of the type 1 languages). We know also that they are closed under the operations of union, concatenation, and Kleene star.

Now recall the definition of regular sets and some of the characterization results for this class. In particular, we know that the class of regular sets is the smallest class of sets containing the finite sets which is closed under union, concatenation, and Kleene closure. Now note that any finite set can be very easily generated by a regular grammar. For example to generate the set {aa, ab} we use the grammar:

$$S \to aA,\ \ S \to aB,\ \ A \to a,\ \ B \to b$$

This allows us to immediately state a theorem and corollary.

> **Theorem** 1. *Every regular set is a regular (type 3) language.*
>
> **Corollary.** *Sets accepted by finite automata can be generated by regular grammars.*

This is a very good result considering that we do not know exactly how to generate any regular sets, just that they can be generated by some grammar. We do know quite a few things about type 3 languages since they've now been placed in our automata hierarchy as a superclass of the regular sets and a subclass of the context sensitive languages or the class of sets accepted by linear bounded automata. But, we have not even seen one yet and must remedy this immediately. Let's examine one which is very familiar to all computer scientists. Namely numerical constants which are used in programs. These are strings which look like this:

$$-1752.40300E+25$$

There seem to be three parts, an integer, a fraction (or decimal), and an exponent. Let's label these I, F, and X respectively and construct a list of the possible combinations. Some are:

I.F, I, I., and .F

for positive constants with no exponents. Imagine how many combinations there are with signs and exponents!

Now we need a grammar which will generate them. We'll use C as a starting symbol. Next, we note that they all begin with a sign, an integer, or a dot. Here are our beginning productions.

$$C \rightarrow -I \qquad C \rightarrow .F$$
$$C \rightarrow dI \qquad C \rightarrow d$$

(We shall use the terminal symbol d instead of actual digits since we do not want to have to do *all* of the productions involving 0 through 9.) Note that all of these productions begin with a terminal. Also we should note that we could not just use $C \rightarrow I$ since it is not of the proper form.

Next comes the integer part of the constant. This is not hard to do.

$$I \rightarrow dI$$
$$I \rightarrow .F$$
$$I \rightarrow d$$

Note that we allowed the choice of ending the constant at this integer stage or allowing it to progress into the fractional portion of the constant.

Fractional parts of the constant are easily done at this point. The decimal point has already been generated and if desired, the exponential part may come next. These productions are:

$$F \rightarrow dF$$
$$F \rightarrow EA$$
$$F \rightarrow d$$

There might seem to be a production of the wrong kind above in the middle. But this is not so since E is actually a terminal. Also, the nonterminal A has appeared. Its job will be to lead in to exponents. Here is how it does this.

$$A \rightarrow +X \qquad X \rightarrow dX$$
$$A \rightarrow -X \qquad X \rightarrow d$$

Here is the entire grammar as figure 1. It has {d,.,+,-,E} as its alphabet of terminals and {C,I,F,A,X} as the nonterminal set.

| | | | | |
|---|---|---|---|---|
| C → –I | I → dI | F → dF | A → +X | X → dX |
| C → dI | I → .F | F → EA | A → –X | X → d |
| C → .F | I → d | F → d | | |
| C → d | | | | |

**Figure 1 - Grammar for Constants**

It is interesting to see what derivations look like for type 3 grammars. A derivation tree for the string dd.dE-dd appears as figure 2.

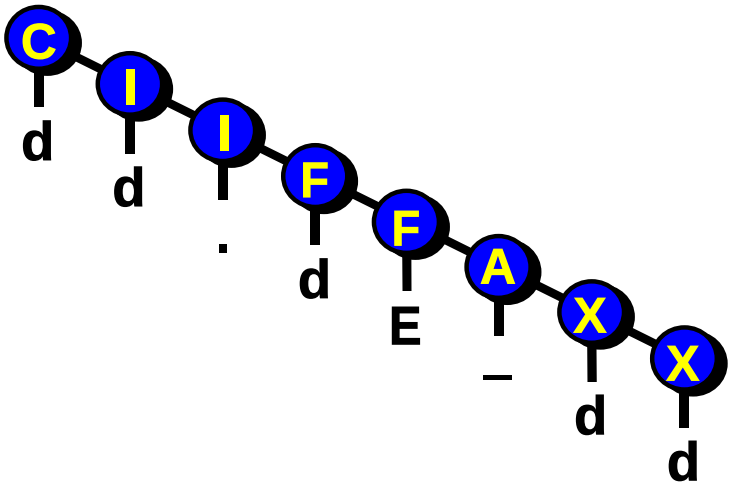


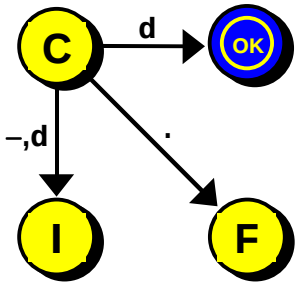

**Figure 2 - Regular Grammar Derivation Tree**

There is not much in the way of complicated structure here. It is exactly the sort of thing that a finite automaton could handle. In fact, let us now turn the grammar into a finite automaton.

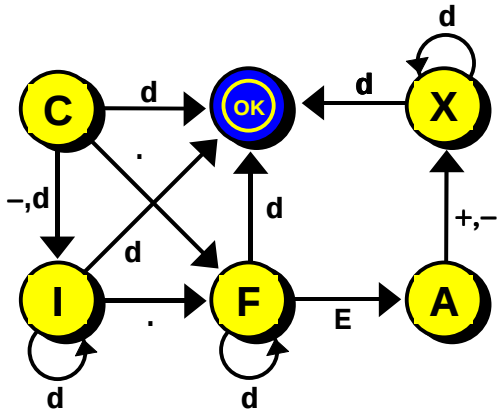
In order to do this conversion we must analyze how a constant is assembled. We know that the derivation begins with a C so we shall use that as a starting state and examine what comes next. This is fairly clear. It must be either a minus sign, a digit, or a dot. Here is a chart.

| Production | Action |
|---|---|
| C → –I | look for a minus and an integer |
| C → dI | look for a digit and an integer |
| C → .F | look for a dot and a fraction |
| C → d | look for a digit and quit |

Our strategy will be to go from the state which refers to the part of the constant we're building (i.e. the nonterminal) to the next part of the constant. This is a machine fragment which covers the beginning productions of our grammar for constants.




We should note that we are building a *nondeterministic* finite automaton. The above fragment had a choice when it received a digit (d) as input. To continue, we keep on connecting the states (nonterminals) of the machine according to productions in the grammar just as we did before. When everything is connected according to the productions, we end up with the state diagram of a machine that looks like this:




At this point we claim that a derivation from our grammar corresponds to a computation of the machine. Also we note that there are some missing transitions in the state graph. These were the transitions for symbols which were out of place or errors. For example: strings such as --112, 145.-67, 0.153E-7+42 and the like. Since this *is* a nondeterministic machine, these are not necessary but we shall invent a new state R which rejects all incorrect inputs and put it in the state table below. The states H and R are the halting (we used OK above) and rejecting states.

| State | Inputs | | | | | Accept? |
|---|---|---|---|---|---|---|
| | d | . | – | + | E | |
| C | H, I | F | I | R | R | no |
| I | H, I | F | R | R | R | no |
| F | H, F | R | R | R | A | no |
| A | R | R | X | X | R | no |
| X | H, X | R | R | R | R | no |
| H | R | R | R | R | R | yes |
| R | R | R | R | R | R | no |

From the above example it seems that we can transform type 3 or regular grammars into finite automata in a rather sensible manner. (In fact, a far easier way than changing regular expressions into finite automata.) It is now time to formalize our method and prove a theorem.

**Theorem 2.** *Every regular (type 3) language is a regular set.*

**Proof Sketch.** Let G = (N, T, P, S) be a type 3 grammar. We must construct a nondeterministic finite automaton which will accept the set generated by the language of G. As in our example, let's use the nonterminals as states. Also, we shall add special halting and rejecting states (H and R). Thus our machine is:

$$M = (N \cup \{H, R\}, T, \delta, S, \{H\})$$

which accepts the strings in L(G).

The transition relation $\delta$ of M is defined as follows for all states (or nonterminals) A and C as well as all input symbols (or terminals) b.

a) $C \in \delta(A, b)$ iff $A \rightarrow bC$
b) $H \in \delta(A, b)$ iff $A \rightarrow b$
c) $R \in \delta(A, b)$ iff it is not the case that $A \rightarrow bC$ or $A \rightarrow b$
d) $\{R\} = \delta(R, b) = \delta(H, b)$ for all $b \in T$

We need one more note. If there is a production of the form $S \rightarrow \varepsilon$ then S must also be a final state.

This is merely a formalization of our previous example. The machine begins with the starting symbol, and examines the symbols which appear in the input. If they could be generated, then the machine moves on to the next nonterminal (or state).

The remainder of the proof is the demonstration that derivations of the grammar are equivalent to computations of the machine. This is merely claimed here and the result follows.

Another example occurs in figure 3. It contains a grammar which generates $0^*10^*$ and the finite automaton which can be constructed from it using the methods outlined in the theorem. Note that since there are choices in both states A and S, this is a nondeterministic finite automaton.

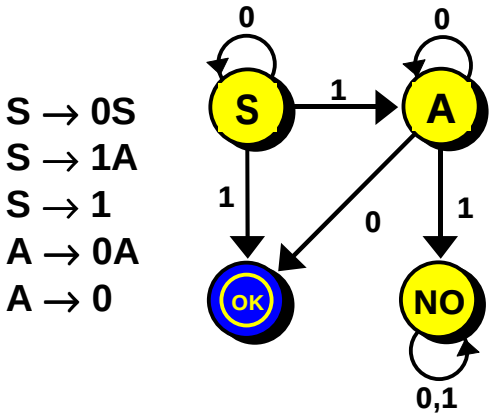



**Figure 3 - Automaton and Grammar for $0^*10^*$**

With this equivalence between the regular sets and the type 3 languages we now know many more things about this class of languages. In particular we know about some sets they cannot generate.

> **Theorem 3.** *The set of strings of the form $a^n b^n$ cannot be generated by a regular (type 3) language.*

> **Corollary.** *The regular (type 3) languages are a proper subclass of the context free (type 2) languages.*