G52MAL Revision Guide Collection

Craig Knott

January 20, 2014

Contents

1 Languages of the World **2**

1.1 Regular languages 2

1.2 Context free languages 2

1.3 Recursively enumerable languages (Turing Recognisable) 2

1.4 Recursive languages (Turing Decidable) 2

1.5 Decidability 2

1.6 Closure 2

2 DFAs and NFAs **3**

3 Regular Expressions **4**

4 I'll pump your Lemma **6**

5 PDAs **7**

6 No, you're ambiguous! **8**

7 CFGs to PDAs **10**

8 Turing Machines **12**

1 Languages of the World

1.1 Regular languages

These are the languages that can represented by regular expressions, deterministic finite automata and non-deterministic finite automata. They are decidable and are closed under all operations (Union, Concatenation, Kleene Star, Intersection, Difference and Complement).

1.2 Context free languages

A context free language is one which can be represented by a context-free grammar, and is once again decidable. They are closed under Union, Concatenation and Kleene Star, but *not* under Difference, Intersection or Complement.

1.3 Recursively enumerable languages (Turing Recognisable)

A recursively enumerable language is one for which there is a Turing machine that will accept it, and is the only language class (that we need to know) that is not decidable. It is closed under Union, Concatenation, Kleene Star and Intersection, but not under Complement or Difference.

1.4 Recursive languages (Turing Decidable)

Finally, a recursive language is one for which there is a Turing machine that will accept it, and not accept any other word. This class of language is decidable, and closed under all operations. The difference between a recursively enumerable language and just a recursive language is that in the recursive language, the Turing machine for the language is guaranteed to reject any words not in the language, for a recursively enumerable language, the Turing machine may either reject the word or simply not terminate at all.

1.5 Decidability

A language is decidable if there is a Turing machine which accepts this language, and also halts on every input string. All regular languages, context-free languages and recursive languages are decidable.

1.6 Closure

This section will give a brief explanation of what being "closed under" an operation actually means for a given language.

Operation	Effect
Union	If L and M are languages, so is $L \cup M$
Concatenation	IF L and M are languages, so is $L \circ M$
Kleene Star	If L is a language, so is L^*
Complement	If L is a language, so is $\Sigma^* - L$
Intersection	If L and M are languages, so is $L \cap M$
Difference	If L and M are languages, so is $L - M$

2 DFAs and NFAs

Deterministic finite automata have a single initial state, and a transition for each input symbol from each state. There is no choice of direction or path, computations must follow a single strict path for any given word. Note that a DFA is a special case of an NFA where the aforementioned properties hold, that is, all DFAs are NFAs. A word is accepted by a DFA when, at the end of the input string, the state of the machine is one of the final states.

Q	$Set\ of\ states$
Σ	$Input\ symbols$
δ	$Transition\ function$
q_0	$Initial\ state$
F	$Set\ of\ final\ states$

The extended transition function for a deterministic finite automata ($\hat{\delta}$). We use this to determine if a word is in the language (if at the end of computation we are left with a final state).

$$\begin{aligned}&\hat{\delta}(q,\varepsilon)=q\\&\hat{\delta}(q,xw)=\hat{\delta}(\delta(q,x),w)\end{aligned}$$

Non-deterministic finite automata can have multiple initial states, and may have either a single transition, multiple transitions or no transitions for each input symbol from each state. There is a choice of direction and path when computing with a non-deterministic machine. This mean that, if *any* path will end the computation in a final state, the word is accepted.

Q	$Set\ of\ states$
Σ	$Input\ symbols$
δ	$Transition\ function$
S	$Set\ of\ initial\ states$
F	$Set\ of\ final\ states$

The extended transition function for a non-deterministic finite automata ($\hat{\delta}$). We use this to determine if a word is in the language (if at the end of computation we are left with a final state).

$$\begin{aligned}&\hat{\delta}(P,\varepsilon)=P\\&\hat{\delta}(q,xw)=\hat{\delta}(\bigcup\{\delta(q,x)\mid q\in P\},w)\end{aligned}$$

The Table Filling Algorithm

1. Create a table n to 1 along the top, and 0 to $n-1$ down the side.
2. Mark each pair of final and non
3. For each remaining pair, parse each letter of the alphabet. If any result in a marked pair, they are also marked.
4. Redraw the transition diagram with the non-marked states as a single state.

Subset Construction (NFA→DFA)

1. Group the initial states together
2. Model a transition table from these, adding each new state
3. Don't forget to include the $\emptyset$ state

3 Regular Expressions

A regular expression is a string made up of alphabet symbols, and special regular expression symbols. The semantics of regular expressions are as follows:

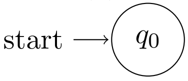
Symbol	Meaning
$\emptyset$	The empty set is a regular expression
ε	The empty word is a regular expression
$x \in \Sigma$	x is a regular expression
$E + F$	If E and F are regex's, so is $E + F$
EF	If E and F are regex's, so is EF
E^*	If E and F are regex's, so is E^*
(E)	If E and F are regex's, so is (E)

Now, what to the regular expression operators actual mean?

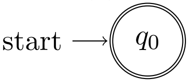
Expression	Meaning
$E + F$	Either E or F
EF	E followed by F
E^*	0 or more occurrences of E

A regular language is one that can be represented by a regular expression, and this regular expression can, in turn, be converted to a finite automata. Below is how to construct automata from a given regular expression.

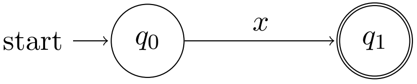
$R(\emptyset)$



$R(\varepsilon)$

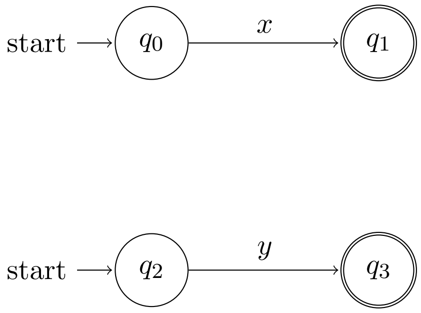


$R(x)$



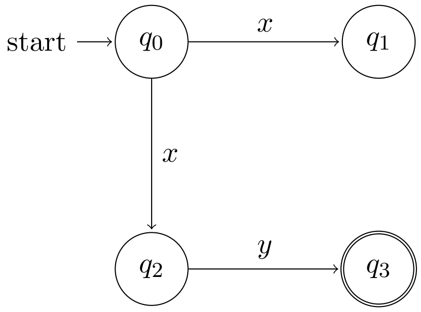
$R(x+y)$

For this diagram, we simply put the two expressions (in this case x and y) next to each other. No connections or other modifications are necessary.



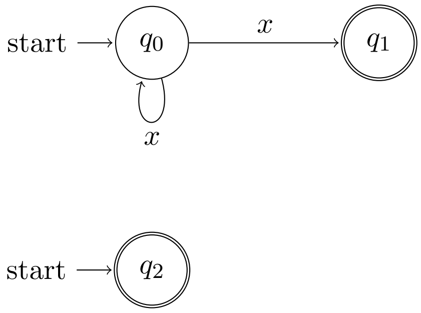
$R(xy)$

1. Put R_1 and R_2 side by side
2. Connect all accepting states in R_1 to initial states in R_2
3. Change initial states in R_2 to regular states
4. Change final states in R_1 to regular states



$R(x^*)$

1. Construct the diagram for $R(x)$
2. All states connecting to a final, must connect to *all* initials
3. An accepting initial state should also be included



4 I'll pump your Lemma

While initially confusing, the Pumping Lemma can actually be very simple, if you can remember the rules that go along with it. The general purpose of the Pumping Lemma is to take some word in some language, and prove that by "Pumping" one part of it (duplicating) you can prove that the word is no longer in the language, which further proves that the language is not regular. First, we will specify what the process of going through the Pumping Lemma is, and then we will go through some examples. For some language L:

1. $\forall n \in \mathbb{N}, \exists w \in L, |w| \geq n$
   Here we choose a word that would be in the language, that is of at least length n

2. $\forall x, y, z \ . \ w = xyz, |xy| \leq n, y \neq \varepsilon$
   Here we use our word, w, and split it into two parts, xy and z, using the length property. We can then use $y \neq \varepsilon$ to determine what the value of y must *at least* be.

3. $\exists i \in \mathbb{N}, xy^iz \notin L$
   We now "pump" y, until some property of the language no longer holds, which proves the language it not regular.

Now for a specific example, imagine we are given the example language

$$\{0^n1^m \mid n > m\}$$

We should easily be able to solve this, using our 3 step process above, let us do that now.

1. $\forall n \in \mathbb{N}, \exists w \in L, |w| \geq n$
   Here we choose some word that belongs to our language with atleast length n. In situations where there are two distinct elements in the language (for instance, our 0 and 1 elements), it is useful to split the word at the n boundary. In this case, I am going to choose the word:

   $$0^n1^{n-1}$$

2. $\forall x, y, z \ . \ w = xyz, |xy| \leq n, y \neq \varepsilon$ Now we split our word, w into two sections, xy and z. The xy section must be atleast length n (this is why splitting the word on a boundary earlier was useful), and the z section is everything that remains. In this case, we can split our word like so:

   $$xy = 0^n$$
   $$z = 1^{n-1}$$

   From this, we can decide that the value of y must be *at least* one "0" (as it can only be some number of 0's, and not ε)

3. $\exists i \in \mathbb{N}, xy^iz \notin L$ Now, we can use our value of y (at least one "0") to prove this language is not regular. We do this by specifying any single value of i that would create a word that does not belong to the language. In our case, we can simply specify i to be equal to 0. This would give us a word in which the number of 0's and 1's was equal:

   $$w = 0^{n-1}1^{n-1}$$

   But in our language definition, the number of 0's was to be strictly higher than the number of 1's. This word is not in our language, and because we have arrived at it using the pumping lemma, we can say that the language it not regular.

5 PDAs

A Push-Down Automata is a finite machine, that uses a stack for memory, accessed in a LIFO fashion. Push down automata can be used to model the context-free languages. There are defined are:

Q	*Set of states*
Σ	*Input symbols*
Γ	*Stack symbols*
δ	*The transition function*
q_0	*The initial state*
Z_0	*The initial stack*
F	*Set of final states*

The transition function is listed in the format

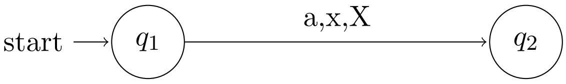
$$\delta(state, input\ symbol, stack\ symbol) \to \{(new\ state, new\ stack)\}$$

The labels of the transitions on a diagrammatic representation of a PDA would be in the format

$$(input\ symbol, stack\ symbol, new\ stack)$$

For instance, the following two representations are synonymous

$$\delta(q_1, a, x) = \{(q_2, X)\}$$

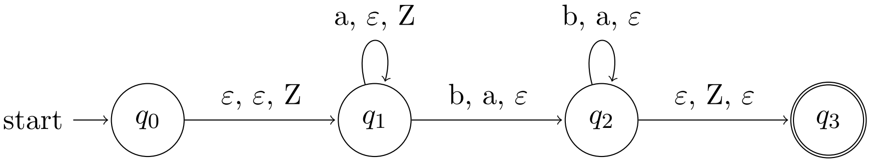


There is another notion to consider with push-down automata: ε-transitions. These transitions do not read any input (leaving the input word the same as prior to the transition), but can affect the state of the stack.

Before we move onto an example, we must cover how push-down automata accept words, as it is done in two distinct ways.

1. Acceptance by Empty Stack
   Words are accepted if there is a sequence of transitions that culminate in both the input word and stack being empty.
2. Acceptance by Final State
   Words are accepted if there is a sequence of transitions that culminate in a final state with the input word being empty.

For the following PDA



Would the word, “aabb” be accepted, by either empty stack or final state?

$$\begin{array}{lll} [q_0, aabb, \varepsilon] & \vdash & [q_1, aabb, Z] \\ & & [q_1, abb, aZ] \\ & & [q_1, bb, aaZ] \\ & & [q_2, b, aZ] \\ & & [q_2, \varepsilon, Z] \\ & & [q_3, \varepsilon, \varepsilon] \end{array}$$

Why not both?! The word is accepted through final state, because computation ends in a final state with no input word remaining *and* is accepted through empty stack, as there is a point in which the input word and stack are both empty.

6 No, you're ambiguous!

Given a context free grammar, G, with the following productions:

$$E \to E + E \mid E * E \mid (E) \mid a$$

If we were to model the following expression

$$a * a + a$$

It's clear to see that there are two distinct approaches to take. We could either try to fully expand the multiplication first, or fully expand the addition first. This means we could construct two distinct traces and, more importantly, two distinct parse trees.

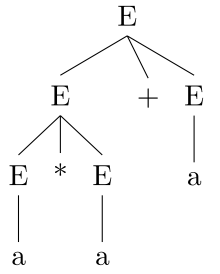


Figure 1: Addition First

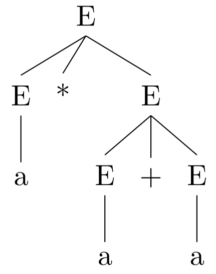


Figure 2: Multiplication First

The ability to produce two distinct parse trees for a given word from a context free grammar means that the grammar is *ambiguous*. It is a common task to convert an ambiguous language to one that is no longer ambiguous; and it is a relatively simple task.

1. Firstly, it is important to determine the order of precedence of the operations in the grammar. In our case, the precedence is as follows (this is usually given to you).

Operation	Precedence
()	Highest
*	
+	Lowest

2. Using this, we can then begin ordering our operations, which is done in reverse order of precedence. For each of the productions in the original grammar, we construct a new production that takes some new non-terminal symbol (E), and maps this to the same symbol (E), using the operation at hand (+), to a second new Non-Terminal (T). We also construct a production from the first non-terminal (E) to the second (T). This is better illustrated below.

$$E \to E + T \mid T$$

3. We continue this process with each of the different operations, using the second non-terminal from the previous production.

$$T \rightarrow T * F \mid F$$

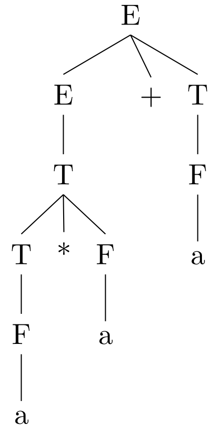
4. When we reach the final operation, we have to create a loop, which is how the ambiguity of the grammar is actually removed. This final production should take the second non-terminal from the previous production, and map this to a bracketed version of the original non-terminal, or to any of the terminal symbols of the grammar, like so.

$$F \rightarrow (E) \mid a$$

This gives us our new, unambiguous grammar

$$\begin{aligned} &E \rightarrow E + T \mid T \\ &T \rightarrow T * F \mid F \\ &F \rightarrow (E) \mid a \end{aligned}$$

Our equation from earlier now only has one possible parse tree (attempting to take any other route will cause some difference in the result, like adding brackets).



7 CFGs to PDAs

A CFG is defined as the following:

$$G = (N, \Sigma, P, S)$$

Where N is the set of Non-Terminals, Σ is the set of Terminals, P is the set of productions, and S is the initial stack symbol. So, if we were given a CFG, G

$$\begin{aligned} G &= (\{E\}, \{a\}, P, E) \\ & \quad \textbf{where} \\ & \qquad P = \{E \to Ea, E \to \varepsilon\} \end{aligned}$$

How can we construct an equivalent PDA that accepts through the "Acceptance by Empty Stack" methodology? Luckily, quite easily. As a reminder, the structure of a PDA definition is the following:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Where Q is the set of states, Σ is the set of input symbols, Γ is the set of stack symbols, δ is the transition function, q_0 is the initial state, Z_0 is the initial stack symbol, and F is the set of accepting states. For our CFG conversion, we would construct the following PDA:

$$G = (\{q_1\}, \{a\}, \{E, a\}, \delta, q_1, E)$$

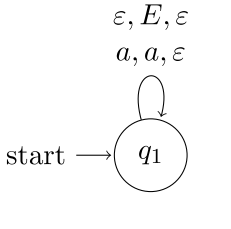
We have a single state, q_1, our input symbols are all the Terminals, and our stack symbols are all the Non-Terminals *and* Terminals. Naturally the initial state is q_1, and our initial stack symbol is the initial symbol of the CFG. Note how we have only 6 elements, as we are using acceptance by empty stack, and thus there are no final state (we could have included a 7th element, $\emptyset$ though). The transition function is the only mildly complex part of this conversion process. We must convert all the productions from P, into suitable transitions for our PDA. This is done by constructing transitions from all Terminal symbols, when reading themselves on the top of the stack, back to the single state, and removing themselves from the top of the stack, like so:

$$\delta(q_1, a, a) = (q_1, \varepsilon)$$

Now we must construct transitions for each Non-Terminal, that are ε-transitions, reading the particular symbol from the stack, and producing whatever that Non-Terminal produces, taken from P. In our case:

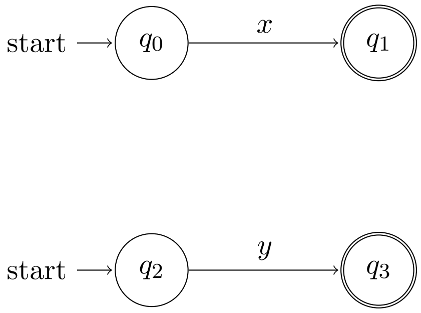
$$\begin{gathered} \delta(q_1, \varepsilon, E) = (q_1, Ea) \\ \delta(q_1, \varepsilon, E) = (q_1, \varepsilon) \end{gathered}$$

Finally, we construct a diagram of our specified PDA, and draw it (remember that for each of the transition functions $\delta(q, a, b) = (p, c)$, we construct a transition on the diagram, in the form "$a, b/c$".



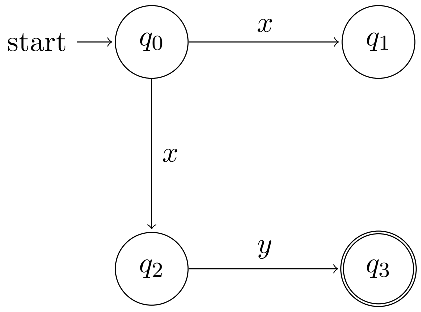
$R(x+y)$

For this diagram, we simply put the two expressions (in this case x and y) next to each other. No connections or other modifications are necessary.



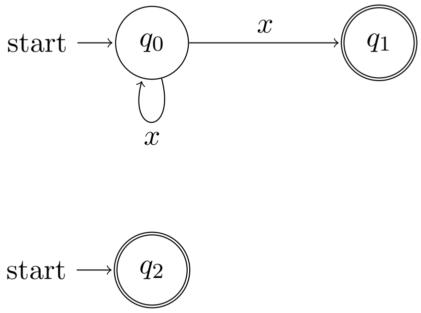
$R(xy)$

1. Put R_1 and R_2 side by side
2. Connect all accepting states in R_1 to initial states in R_2
3. Change initial states in R_2 to regular states
4. Change final states in R_1 to regular states



$R(x^*)$

1. Construct the diagram for $R(x)$
2. All states connecting to a final, must connect to *all* initials
3. An accepting initial state should also be included



8 Turing Machines

A Turing machine is a Push-Down Automata with an infinite tape, instead of a stack, and can be used to model any possible computation. If a word is accepted by a Turing machine, it will terminate on an accepting state. Words that are not accepted will either terminate on a non-accepting state, or not at all. Recursive (decidable) languages are those that can be represented as a Turing machine that will always stop (meaning words in the language will be reject, and not cause the machine to continue indefinitely). Recursively enumerable language however, do not have this guarantee.

A Turing Machine, M, can be represented as a 5-Element tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

Where

Q is the set of states
Σ is the input alphabet
Γ is the tape alphabet (encapsulates Σ)
δ is the transition function
q_0 is the initial state
B is the blank symbol ($B \in \Gamma, B \notin \Sigma$)
F is the set of final states

The transition function of a Turing machine is the following

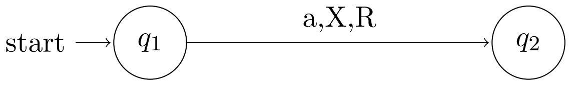
$$\delta(state, input\ symbol) \rightarrow (new\ tape\ symbol, new\ state, tape\ movement)$$

Which, when used to label a transition on a Turing machine diagram, is translated into the following format

$$(input\ symbol, new\ tape\ symbol, tape\ movement)$$

For instance, the following two representations are synonymous

$$\delta(q_1, a) = (X, q_2, R)$$



The only other thing to learn about Turing machines is how the tape is modified during computation. The "Left" and "Right" tape movement refers to how where the head of the tape will move (or how the tape moves in regard to the head) - the head of the tape being the tape symbol that is currently being observed. Initially, the tape is the input word, with the blank symbol (B from above) on either side. For instance, if we were to process the word "aa", our initial tape would be pictured as in figure 3, and our tape after the transition above would be as pictured as in figure 4.

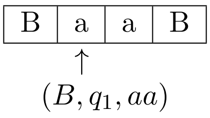


Figure 3: Tape and Instantaneous description of initial tape

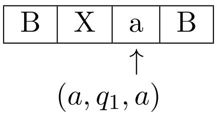


Figure 4: Tape and Instantaneous description of tape after $\delta(q_1, a) = (X, q_2, R)$ transition

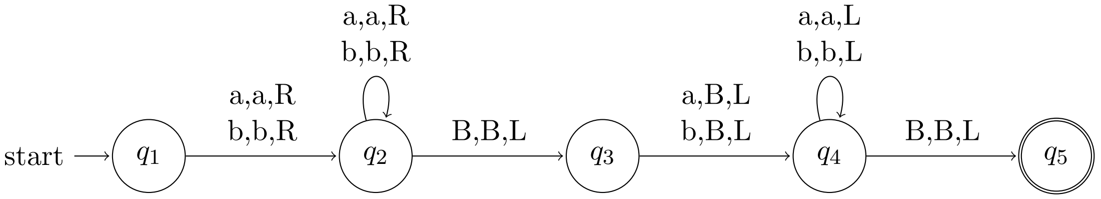


Figure 5

Below is a trace for the computation of the word “aabb” for the Turing machine in figure 5.

$$\begin{array}{lcl} [B, q_1, aabb] & \vdash & [a, q_2, abb] \\ & & [aa, q_2, bb] \\ & & [aab, q_2, b] \\ & & [aabb, q_2, B] \\ & & [aab, q_3, b] \\ & & [aa, q_4, b(B)] \\ & & [a, q_4, ab] \\ & & [B, q_4, aab] \\ & & [B, q_5, Baab] \end{array}$$

Computation of this word ends in an accepting state (q_5), meaning it is accepted. More over, it does not loop infinitely, and does not prematurely halt.

Some final terminology to be aware of: Decidability of a property, in the context of a Turing machine, means that there is a Turing machine that will produce a yes or no for the property, in any given case. If a problem is not decidable, there will be no such Turing machine.