Regular Expressions and Deterministic Finite Automata

Given an alphabet Σ, a finite set of symbols, a **language** over the alphabet Σ is any set of strings made up of the symbols from Σ. For example, if Σ = $\{a, b\}$, then the following are some examples of languages over Σ:

L1 = { aab, aba, bab, aa }

L2 = { w | w has equal number of a's and b's } = { abab, aaabbb, abba, λ, … }, here λ is the empty string.

L3 = { w | w is made up only a's and has a length which is a prime number } = { aa, aaa, aaaaa, … }

We define 3 operations on Languages. Let L, L1, and L2 be languages. Then,

1. L1.L2 = { w1.w2 | w1 ∈ L1 and w2 ∈ L2 }, where w1.w2 is the string concatenation of w1 and w2.
2. L1 ∪ L2 = { w | w ∈ L1 or w ∈ L2 }, called the union
3. L* = { λ } ∪ L ∪ L.L ∪ L.L.L ∪ …

I. Regular Expressions

Regular expressions are a mathematical mechanism to define a class of languages called regular languages. Given an alphabet of symbols, Σ, a **regular expression** is defined as follows:

1. Every symbol in Σ is a regular expression.
2. ϵ is a regular expression
3. if r and s are regular expressions, then so are the following
   - (rs)
   - $(r+s)$
   - $(r)*$

(rs) is called the concatenation of r and s, $(r+s)$ is called the union of r and s, and $(r)*$ is called the Kleene closure of r. The parentheses may be left out with the understanding that the $*$ operator has highest precedence, the concatenation operator has the next level of precedence, and the $+$ operator the lowest precedence. Some examples of regular expressions over the alphabet $\{a, b\}$ are:

- r1 = ab(a+b)*ab
- r2 = (a+b)*
- r3 = aa+bb

Each regular expression r represents a language $L(r)$ which is defined as follows:

1. L(a) = { a }, for any a in Σ
2. L(ϵ) = { λ }
3. L(rs) = L(r).L(s)
4. L($r + s$) = L(r) + L(s)
5. L($r*$) = L(r)*

Apply this definition to the earlier 3 examples of regular expressions, we get the following:

L(ab(a+b)*ab) = { w I w starts with ab and ends with ab }

L((a+b)*) = set of all strings made up of any number of a's and b's in any order including the empty string.

L(aa+bb) = { aa, bb }

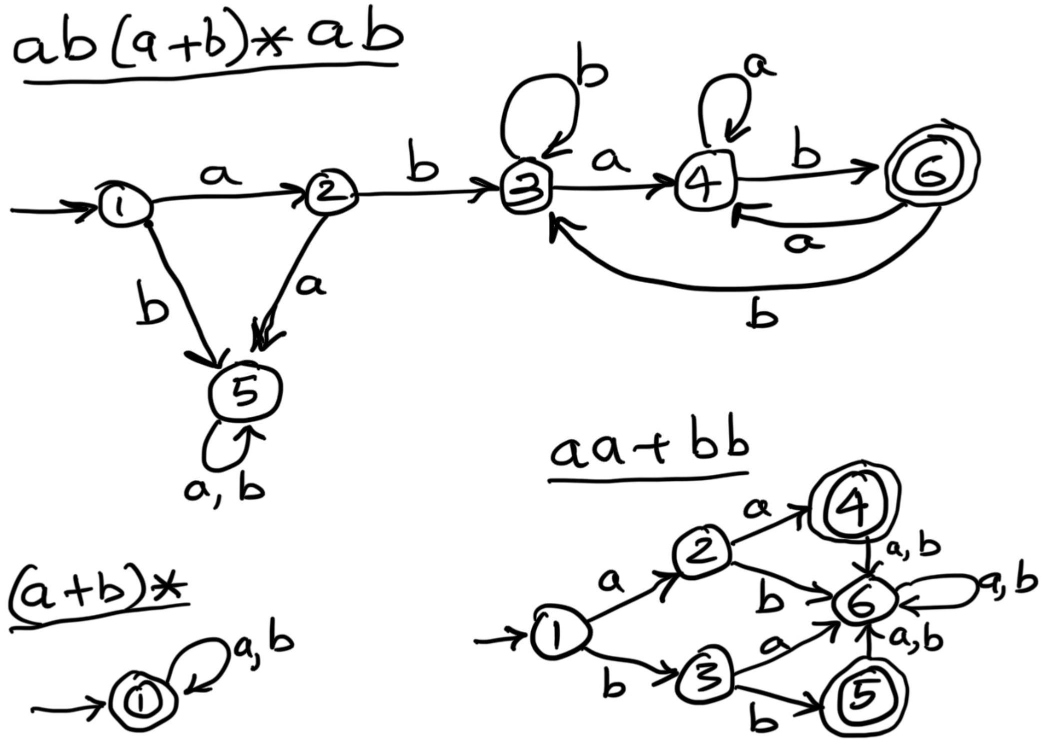
II. Deterministic Finite Automata

A deterministic Finite Automata (DFA) is a mathematical model of a simple computational device that reads a string of symbols over the input alphabet Σ, and either accepts or reject the input. The set of strings accepted by the DFA is referred to as the language of the DFA.

A deterministic finite automata (DFA) is defined as a 4-tuple (Q,T,S,F), where

- Q is a finite set of states
- S $\in$ Q is designated as a start state
- F $\subseteq$ Q is a designated set of final states
- T is a transition function from Q x $\Sigma \rightarrow$ Q

A DFA can be pictured as a **graph** with states as the nodes and the transitions as directed edges from one node to another. The transitions/edges will be labeled by the alphabet symbol. Start state will be designated by an arrow mark and final states will be designated by double circles. Here are DFAs for the three regular expressions discussed before:



The DFA transition functions can also be represented in tabular form as follows:

ab(a+b)*ab

Start State = 1

Final States = 6

FROM	SYMBOL	TO
1	a	2
1	b	5
2	a	5
2	b	3
3	a	4
3	b	3
4	a	4
4	b	6
5	a	5
5	b	5
6	a	4
6	b	3

(a+b)*

Start State = 1

Final States = 1

FROM	SYMBOL	TO
1	a	1
1	b	1

aa+bb

Start State = 1

Final States = 4, 5

FROM	SYMBOL	TO
1	a	2
1	b	3
2	a	4
2	b	6
3	a	6
3	b	5
4	a	6
4	b	6
5	a	6
5	b	6
6	a	6
6	b	6

How does a DFA work?

A DFA can be used to verify if a string belongs to a language or not. All strings that are "accepted" by a DFA belong to it’s language and those that are "rejected" do not belong to it’s language. How do we determine "acceptance" and "rejection"?

A configuration for a DFA is a pair, (q, s), where q is a state and s is a string made up of symbols from the alphabet. Given an input string, w, and a DFA with start state q0, the initial configuration is (q0,w). DFA moves from one configuration to the next as follows:

(q, ax) => (T(q,a), x)

until it reaches the following configuration

(p,λ)

We say that a string w is accepted by a DFA if (q0,w) =>* (f,λ) and f is a final state; otherwise it is rejected.

Let us see if the input string abaaab is accepted or rejected by the DFA for ab(a+b)*ab shown earlier.

(1,abaaab) => (2,baaab) => (3,aaab) => (4,aab) => (4,ab) => (4,b) => (5,λ)

Since 5 is a final state, the DFA accepts the string abaaab.

The input string abaaba is rejected because (1,abaaba) => (2,baaba) => (3,aaba) => (4,aba) => (4,ba) => (6,a) => (4,λ) and 4 is not a final state.

Language of DFA, D, L(D) = set of all strings accepted by D

III. Regular Expression to DFA (Direct Algorithm)

It turns out that for every regular expression there is an equivalent DFA (i.e. the language defined by the regular expression equals the language accepted by the equivalent DFA).

This equivalent DFA is what the PLY and similar compiler-compiler systems use to extract the tokens from the input string!

ALGORITHM: Convert Regular Expression to DFA

INPUT: regular expression, r

OUTPUT: DFA, D, such that Language(D) = L(r)

METHOD: (To illustrate each step of the algorithm, we will use the regular expression (a+b)*abb as an example, however the method is general that it will work for any regular expression)

Step 1: Expression Tree

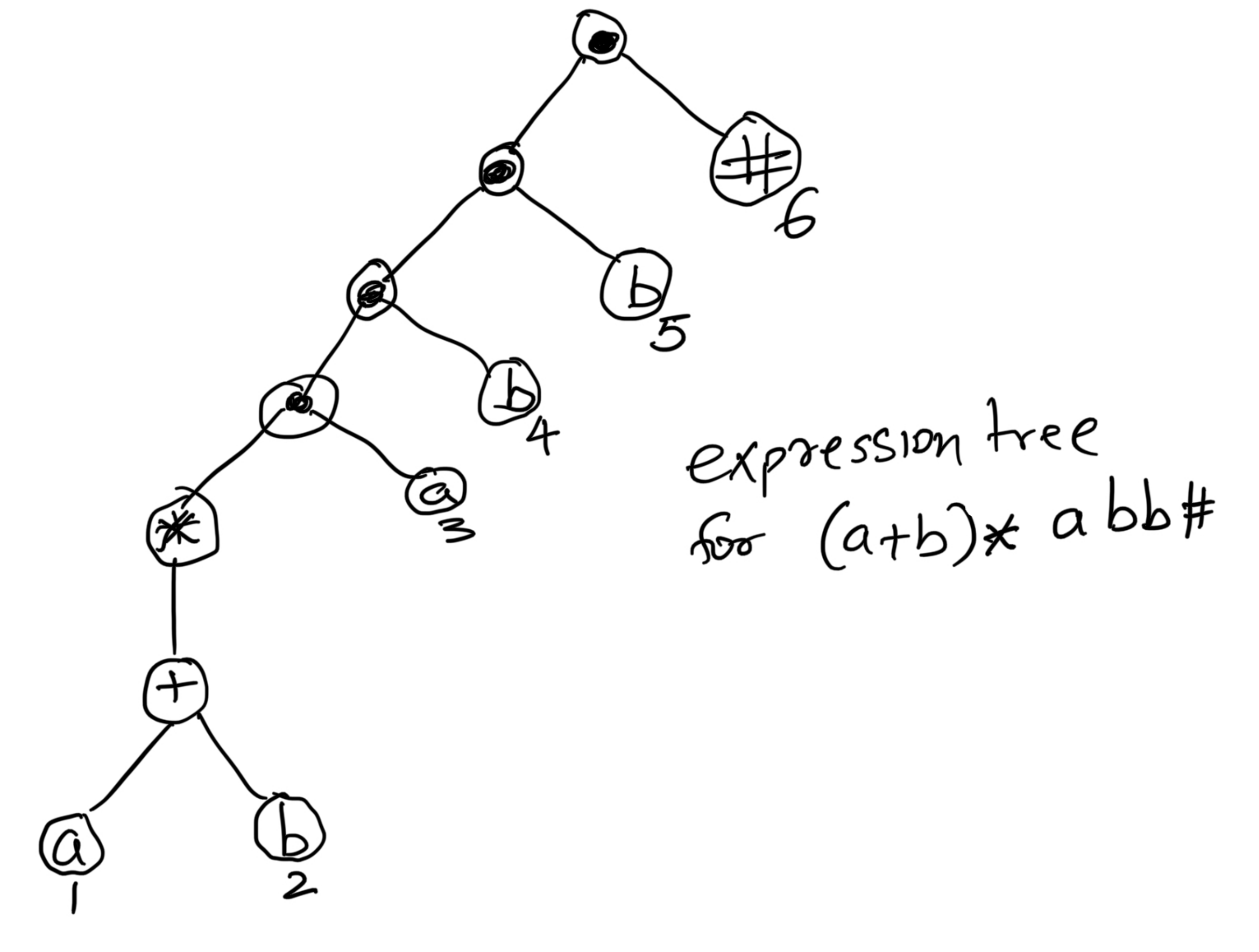
Augment r with a special end symbol # to get r#, e.g. (a+b)*abb#

Using the following grammar, construct an expression tree for r#

```
re : term | re PLUS term
term : factor | term factor
factor : niggle | factor STAR
niggle : LETTER | EPSILON | LPAREN re RPAREN
```

Step 2: Unique Number for Leaf Nodes

Assign a unique integer to each leaf node (except for the ϵ leaf) of the expression tree.



Step 3: nullable(n), firstpos(n), lastpos(n)

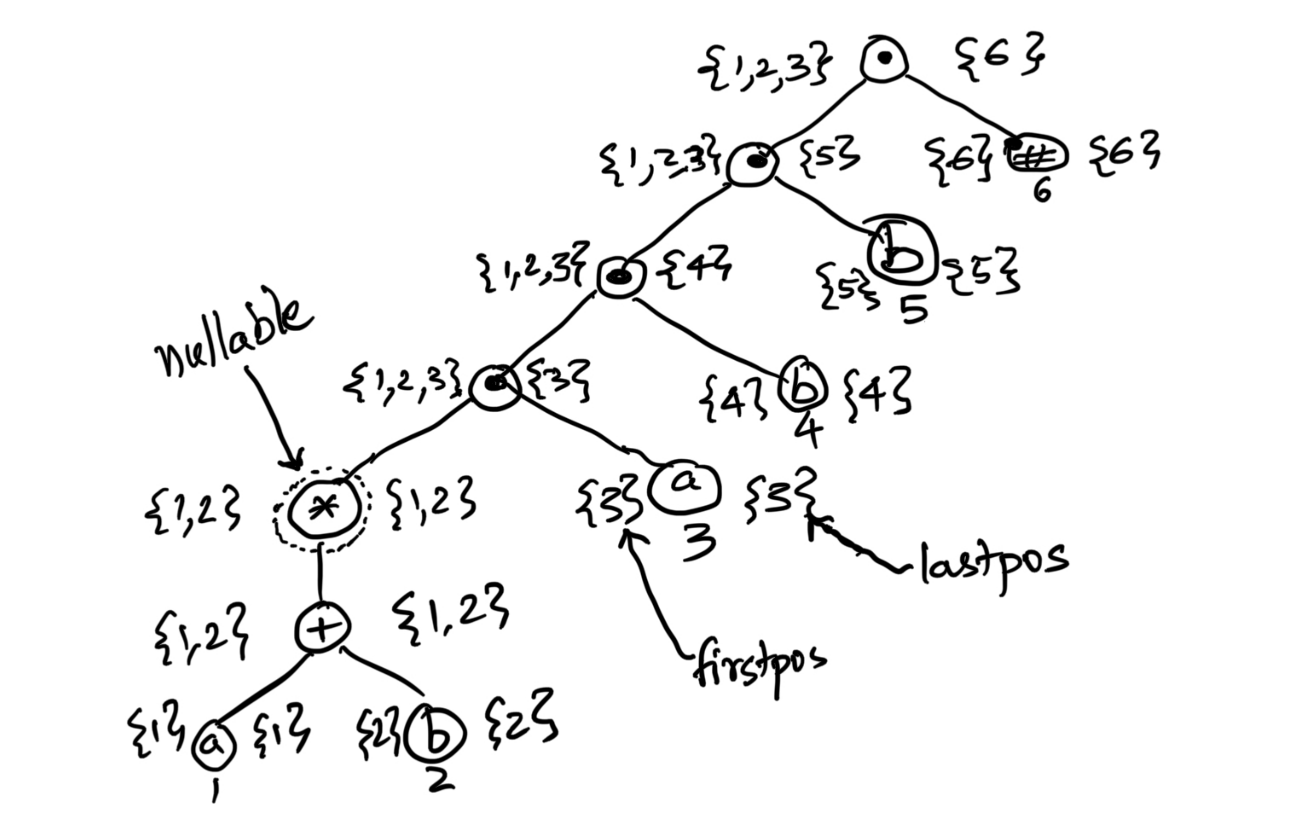
Traverse the tree to compute **nullable(n)**, **firstpos(n)**, and **lastpos(n)** for each node, **n** in the tree using the following definitions:

Node n	nullable(n)	firstpos(n)	lastpos(n)
Leaf ϵ	true	{ }	{ }
Leaf i	false	{ i }	{ i }
(c1 + c2)	nullable(c1) or nullable(c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
(c1 . c2)	nullable(c1) and nullable(c2)	if nullable(c1) then firstpos(c1) ∪ firstpos(c2) else firstpos(c1)	if nullable(c2) then lastpos(c1) ∪ lastpos(c2) else lastpos(c2)
(c1)*	true	firstpos(c1)	lastpos(c1)

The intuition behind these functions are as follows. Let L(n) be the language generated by the subtree rooted at node n.

- nullable(n) = L(n) contains the empty string λ
- firstpos(n) = set of positions under n than can match the first symbol of a string in L(n)
- lastpos(n) = set of positions under n than can match the last symbol of a string in L(n)

For the example regular expression, the following shows the values of these functions:



Step 4: followpos(n)

Compute **followpos(n)** for leaf nodes/positions.

followpos(i) = set of positions that can follow position i in any generated string.

followpos(n) can be computed using the following algorithm:

```
for each node n in the tree do
       if n is a concat node with left child c1 and right child c2 then
           for each i in lastpos(c1) do
               followpos(i) = followpos(i) U firstpos(c2)
       else if n is a Kleene star node
           for each i in lastpos(n) do
               followpos(i) = followpos(i) U firstpos(n)
       else
           pass
```

Applying the algorithm to our example, we get the following values of **followpos(n):**

Node n	followpos(n)
1	{ 1, 2, 3 }
2	{ 1, 2, 3 }
3	{ 4 }
4	{ 5 }
5	{ 6 }
6	{ }

Step 5: Generate DFA

```
s0 = firstpos(root-node); designate it the start state
states = { s0 } and is unmarked
while (there is an unmarked state T in states) do
      mark T
      for each input symbol 'a' in the alphabet do
          let U be the union of followpos(p) for all positions p in T such that
                     the symbol at position p is 'a'
          if U is not empty and not in states then
              add U as an unmarked state in states
          trans[T,a] = U
 Designate any state containing the #-position as a final state
```

Applying this algorithm to our example, we get:

Initially

s0 = {1,2,3}

states = { {1,2,3} }

Iteration 1 or while loop

T = {1,2,3}

Of the elements of T, 1,3 correspond to a and 2 corresponds to b

{1,2,3} on a transitions to followpos(1) U followpos(3) = {1,2,3,4}

{1,2,3} on b transitions to followpos(2) = {1,2,3}

i.e.

trans[{1,2,3},a] = {1,2,3,4}

trans[{1,2,3},b] = {1,2,3}

Iteration 2 or while loop

T = {1,2,3,4}

Of the elements of T, 1,3 correspond to a and 2,4 corresponds to b

{1,2,3,4} on a transitions to followpos(1) U followpos(3) = {1,2,3,4}

{1,2,3,4} on b transitions to followpos(2) U followpos(3) = {1,2,3,5}

i.e.

trans[{1,2,3,4},a] = {1,2,3,4}

trans[{1,2,3,4},b] = {1,2,3,5}

Iteration 3 or while loop

T = {1,2,3,5}

Of the elements of T, 1,3 correspond to a and 2,5 corresponds to b

{1,2,3,5} on a transitions to followpos(1) U followpos(3) = {1,2,3,4}

{1,2,3,5} on b transitions to followpos(2) U followpos(5) = {1,2,3,6}

i.e.

trans[{1,2,3,5},a] = {1,2,3,4}

trans[{1,2,3,5},b] = {1,2,3,6}

Iteration 4 or while loop

T = {1,2,3,6}

Of the elements of T, 1,3 correspond to a, 2 corresponds to b, and 6 corresponds to #

{1,2,3,6} on a transitions to followpos(1) U followpos(3) = {1,2,3,4}

{1,2,3,6} on b transitions to followpos(2) = {1,2,3}

i.e.

trans[{1,2,3,6},a] = {1,2,3,4}

trans[{1,2,3,6},b] = {1,2,3}

We designate {1,2,3,6} as a final state since it contains the position of #

Note: The "marking" of states is not shown above, but we can worry about this in the implementation!

Taking all the values of T and the values of trans, we obtain the following DFA