Lexical Analysis(Scanner)

What is a Lexical Analyzer?

A Lexical Analyzer or Scanner is an algorithm that groups the characters of the source code to form the **Tokens**. Afterwards, it returns the **Internal Representation Number** of these tokens, which is an ID that matches the reserved word fetched from a **keywords table** which the implementer of the compiler predefines.

These tokens are divided into 3 kinds:

1. Names : Which is any name we have in a program. These in turn are divided into 2 types: a. Keywords/Reserved, which are words such as if/else/while. These names cant be used as variable names. They have a specific place and function b. User Defined Names, Which are the names declared by the user.

2. Values : such as integers(1, 2, 3, 4) or floating point(1.1, 2.34, 5234.123) etc

3. Special Symbols/Tokens : And these are the logical(==, &, ||) and arithmetic operations(+, -, *, /), parenthesis([], {}, ()), or any other tokens that are not from the first or second kind.

Lets apply the scanner to this short segment of code:

```
while(x>=100)
{
    n +=x;
    x++;
}
```

This results in this set of tokens :

`While` , `(` , `x` , `>=` , `100` , `)` , `{` , `n` , `+=` , `x` , `;` , `x` , `++` , `}` .

Referencing these tokens against a certain keywords table like this one :

index	Symbol
..	..
33	`While`
..	..
67	`>=`
..	..
..	..

Leads us to these ID's :

Token	Internal Representation Number
`While`	33
`(`	84
`x`	100
`>=`	67
`100`	200
`)`	85
`{`	92
`n`	100
`+=`	77

Token	Internal Representation Number
`x`	100
`;`	81
`x`	100
`++`	75
`;`	81
`}`	93

note that all user defined names have the same number. This is because to the syntax analyzer, it doesn't matter what the variable is, it just matters that there is a variable there.

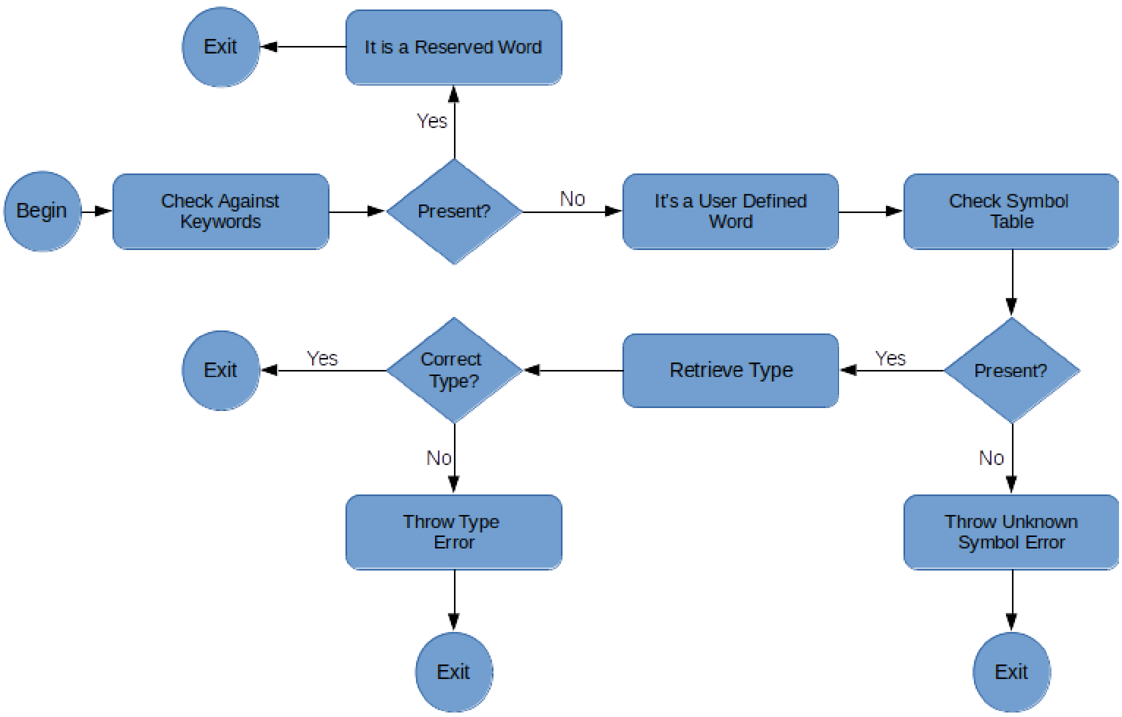
Type Checking implementation

During this process of analysis, The Compiler builds what is called the **Symbol Table**. The Symbol Table is a table of the name of each user defined name(mostly variables), its type, and its values. The Symbol table for this segment of code would be:

```
.
.
.
int compute(int,int);
int n;
float x,y;
const int m=10;
.
.
.
```

Name	Type	Value
`compute`	function-name	0
`n`	Integer	0
`x`	float	0
`y`	float	0
`m`	const-int	10

To perform type checking, the compiler takes the name, and checks the keywords table. If it is not in the keywords table, it is a user defined variable. if it is a user defined variable, it then goes to check the symbol table. If it is not defined in the symbol table, it returns that the variable is not defined (unknown symbol/variable deceleration error), if it is in the symbol table, it retrieves its type. If the operation being performed on the variable is not compatible with the type of the variable, it returns that the operation is not compatible(type error).



Regular Languages(Regular Expressions) in Lexical Analysis

Regular Languages are a class of language that are important for lexical analysis, since we use them to define and generate tokens. This Class of languages is defined recursively.

Defining a Language as a Set

We Say that :

> An Alphabet is a Set of Symbols.

For example, our alphabet is the set V = {a,b,...,y,z}.

Expanding on the definition above, we say that :

> A String is a Sequence of Symbols Taken From the Alphabet.

So if V is our alphabet, then :

abcd

dsda

qweaz

asasd

Are all strings defined on V. we can define **an infinite number of strings on an alphabet**.

Formally,

> S is the set of all strings over some alphabet V.

let V={0,1} with S defined over it, lets define a binary operation on S

given that x,y ∈ S, then we can define a concatenation operation on x and y as follows:

> XY={the set of strings formed by following x with y}.

Note that XY ≠ YX . We say that the concatenation operation is **not** commutative.

Lets Define a special string, called the empty string, which we denote with λ.

Notice that λX is the same as X. Formally, we can say:

> λ is the **identity** element of the concatenation operation.

Where the Identity element is formally defined as :

> an element of a set that, if combined with another element by a specified binary operation, leaves that element unchanged.

Putting all of this together, We can define a language as :

> Given an alphabet V, a language L over V is a set of strings formed from V.

By this definition, these sets Are *all* languages:

L_1={a, b, c}

L_2={asdasd, qwe, asd}

L_3={abb}

This definition also leads us to the conclusion :

> There are an ∞ number of languages defined on an alphabet.

Set Operations On Languages

Given an Alphabet V, assume that :

1. L = {set of all languages defined on V}
2. L = {L1, L2, L3,....Ln}

We will define a 3 operations on L , ie, the operands are languages belonging to L.

Concatenation Operation

Given that L, M are languages over an alphabet V, then

> LM = "L concatenated with M" = {xy | x ∈ l, y ∈ m}.

For Example let L={a,b,c} and M={aa,bb}, then :

LM={aaa,abb,baa,bbb,caa,cbb}

ML={aaa,aab,aac,bba,bbb,bbc}

Note that :

1. LM ≠ ML. (Concatenation on languages is not commutative).
2. L{λ} = {λ}L = L. (λ is the identity for concatenation).
3. L{ } = { }L = { }.

The OR "|" Operation

OR is ∪ operation in set theory

Given that L, Mare languages over an alphabet V, then

> L|M = "L OR M" = {x | x ∈ l ∪ x ∈ m} = L ∪ M.

Note that :

1. L|M = M|L. (OR on language is commutative)

2. L|{ } = { }|L = L. (The empty set is the identity element for the OR operation)

The Closure "*" Operation (A Unary Operation)

The Closure Operation is the "NOT" Operation in Logic.

Given that L is a language over an alphabet V then L* is:

> $L^* = L$^0 $\cup L^1 \cup L^2 \cup \ldots.. \cup L^\infty$

L^+ is given by

> $L^+ = L^* - \{\lambda\}$

The Recursive Definition of Regular Languages

Given an alphabet V then :

> 1. ∅ = { } = empty language is a regular language denoting the language { }
> 2. **λ** = {λ} is a regular language denoting the language λ
> 3. For every element a ∈ V , **a**= {a} is a regular language denoting the language {a}

lets say that V = {a, b, c}, then according to 1, 2, 3, **∅**={ },**λ**={λ},**a**={a},**b**={b},**c**={c}, are all regular languages

Given R and S are regular languages denoting the regular languages L_R and L_S respectively, then :

> a. RS is a regular language denoting $L_R L_S$
>
> b. R|S is a regular language denoting $L_R|L_S$
>
> c. R* is a regular language denoting L_R^*

say that R={a} and S={b}, then :

RS={ab},

R|S={a,b},

R*= $\{a\}^0 \cup \{a\}^1 \cup$....

= {λ,a,aa,aaa,....}

= A string that consists of any number of a's

Lets say we took (RS)* then

(RS)* = $\{ab\}^0 \cup \{ab\}^1 \cup$.....

= {λ,ab,abab,ababab,abababab,....}

= A string that consists of any number of "ab"s

Lets say we took (a|b)* , then :

(a|b)* = ({a}|{b})* = ({a}∪{b})*

= ({a,b})* = A string of a's and b's

Lets say we took (0|1)*00, then By the definitions above, this results in any binary string followed by 00, such as {100,000,1100,0000,...}

Lets say we took (a|b)*bbb(a|b)*, then By the definitions above, this results in any string of a's and b's that contains at least 3 b's such as {bbb,abbb,bbba,bbbbb,...}

Defining Tokens Using Regular Languages

Remember that we have 3 types of tokens:

1. Names
2. Values
3. Special Symbols

The scanner must recognize these and be able to distinguish them.

Names

In programming languages, names are letters followed by letters or digits. The regular language for names is :

> Letter(Letter|digit)* = L(L|d)*

Values

In programming languages, there are multiple types of values, and they can all be defined using a regular language

1. Integers : Integers consist of a + or - followed by a digit followed by a set of digits. the regular language for them is

   > [+|-]digit(digit)* = [+|-]dd* = [+|-]d^+

   Note : [x] means we take x zero or one time **only**.

2. Floating Point Numbers :

   a. Fixed Floating Point : Fixed Floating Point Numbers consist of a+ or more followed by 1 or more digits followed by a . followed by 1 or more digits. They are given by the regular language : > [+|-]d^+.d^+ b. Exponential Notation : Exponential Floating Point Numbers consist of a + or - followed by 1 or more digits followed by a . followed by one or more digits followed by an E followed by a + or - followed by 1 or more digits. They are given by the regular language :

   > [+|-]d^+.d^+E[+|-]d^+

Special Symbols

The Set of special symbols {+,-,<=,....} are each given by its own regular languages. For example, the symbol + has its own regular languages given by :

> + = {+}

or the * symbol is given by

> * = {*}

or the <= symbol is given by

> <= = {<}{=} = {<=}

or ++ , which is given by

> ++ = {+}{+} = {++}

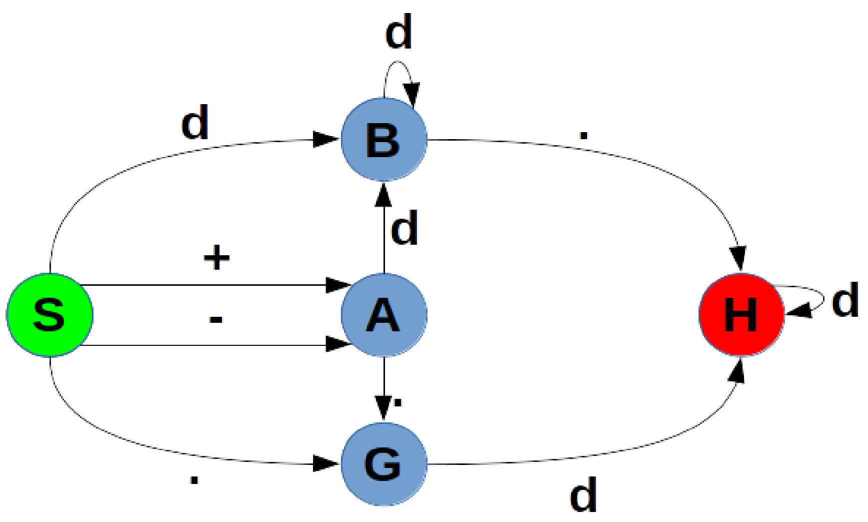
The Question remains: How do we build an algorithm to recognize(accept) strings whose languages are regular languages?

Finite State Automata(FSA)

Tokens in source codes are strings of regular languages. The algorithm that recognises these strings is called the **Finite State Automata** or the **Finite State Machine** or the **Finite State System**. The Finite State Automata contains :

1. A Set of states.
2. Transitions between states.
3. Input string to be examined.

Given that we have this Finite State Automata(FSA) :



- The set of states Q={S,A,B,G,H}
  - S is called the **Starting State**.
  - H is called the **Final State**.
- The transitions between the states are given by{+, -, d, .}
- Lets Say that the input string is "-ddd.dd" eg "-511.32".

Tracking through the states on this string, we start at state S :

```
  -    d    d    d    .    d    d
S--->A--->B--->B--->B--->H--->H--->H
```

Since H is the final state, we say that the **string is accepted**, or more formally :

> A string is accepted or recognized if after scanning the string we end up with a final state.

The Regular Language(expression) generated(accepted) by this machine L(M) is given by :

> $L(M)=[+|-]\{d^+., .d^+, d^+.d^+\}$

Other examples of finite state machines are :

1. Names : $L(N) = \{l^+\}$.
2. Integers : $L(I) = \{d^+\}$.
3. Greater Or Equal : $L(G_e) = \{>=\}$.

Types of Finite State Automata

there are 2 types of Finite State Automata

Non-Deterministic Finite State Automata

An algorithm is non-deterministic or fuzzy if there are options in the algorithm. An example of a non-deterministic algorithm is the solution of the Knight Tour Problem, which is based on Backtracking Techniques.

A Finite State Automata is non-deterministic if :

1. There are λ-transitions(moves) in the FSA :
2. **Or** There is more than one transition from the same state on the same input :

In Both cases, There is a choice(trial and error) to make. The only way to solve non-deterministic machines is to use backtracking. This wont do in a compiler, because backtracking is a very compute-heavy method and is extremely slow.

Fortunately, There are algorithms to transform any NDFSA to a DFSA. Therefore, we can assume always in the assumption that our machine is deterministic

Deterministic Finite State Automata

If A Machine is **not** non-deterministic, we call it a **Deterministic Finite State Automata**, ie :

1. There are **NO** λ-states.
2. **AND** There is **NO** more than one transition from the same state on the same input.

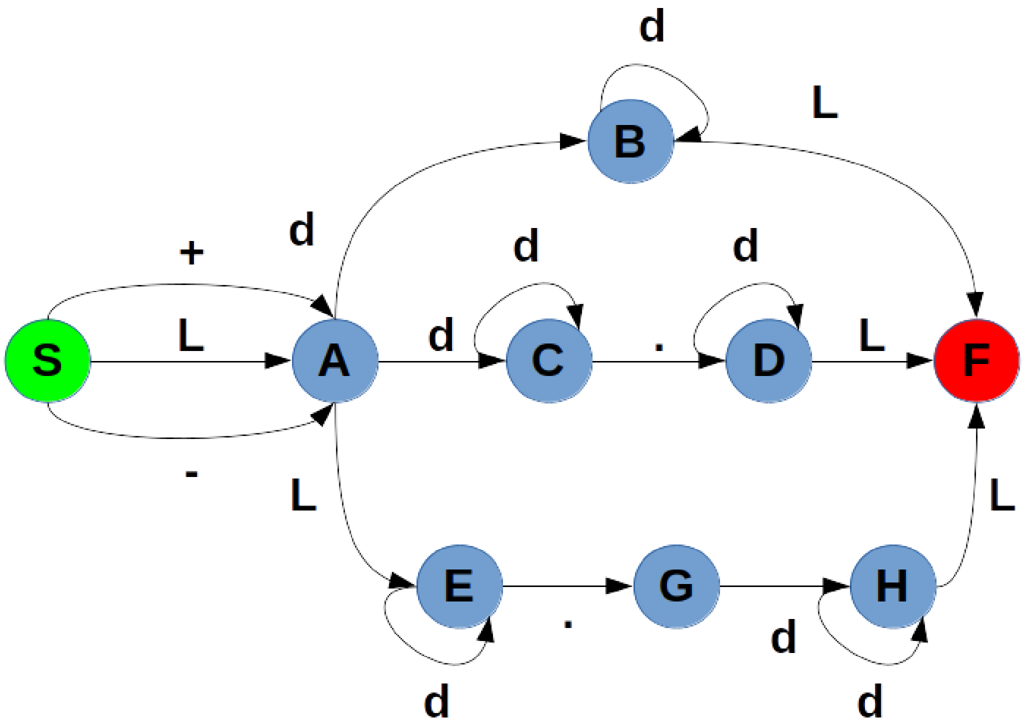
Only Deterministic Finite State Automata are used for compilers.

Transformation of NDFSA to DFSA

The Algorithm that transforms an NDFSA to a DFSA consists of the following steps :

1. Removal of λ transitions.
2. Removal of non-determinism.
3. Removal of inaccessible states.
4. Merging equivalent states.

Lets Say we have this NDFSA :



Which has this language:

> L(G)=[+|-]{ ddddddd,
>
> ```
> dddd.ddd,
>
> dddddd.,
>
> .dddd
> ```
>
> }

Or in short

> L(G)=[+|-](d^+|$d^+.d^+$|d^+.|.d^+)

And we want to transform it into a DFSA. Lets follow through the steps :

Lets Break down the Finite state machine into a transition diagram :

State \ V_T	**+**	**-**	**.**	**d**	**λ**
S	A	A			A
A				B,C	E
B				B	F
C			D	C	
D				D	F
E			G	E	
G				H	
H				H	F
F					

Remove Lambda Transitions

1. Consider S-$^{\lambda}$->A

   Add all transition in Row A to S.

2. Repeat Step(1) for all States with λ Transitions
3. Mark all states from which there is a λ Transition to a final State. Mark it as the final State
4. Delete the λ Column

This results in this table :

State \ V_T	**+**	**-**	**.**	**d**
S	A	A	G	B,C,E
A			G	B,C,E
B				B
C			D	C
D				D
E			G	E
G				H
H				H
F				

Removal Of Non-Determinism

Which means not having more than 1 transition on 1 input.

1. Consider [B,C,E]. Lets add this and treat it as a new state in the table.
2. If at least one of the states [B,C,E] is a final state, then we make it a final state.
3. Repeat steps (1) and (2) for all non-deterministic states
4. The Machine is now deterministic

This results in this table :

State \ V_T	**+**	**-**	**.**	**d**
S	A	A	G	B,C,E
A			G	B,C,E
B				B
C			D	C
D				D

State \ V_T	+	-	.	d
E			G	E
G				H
H				H
F				
B,C,E			D,G	B,C,E
D,G				D,H
D,H				D,H

Removal of Non-Accessible States

1. Mark the Initial State
2. Mark all states for which there is a transition from S
3. Repeat step (2) for all marked states.

   This results in this table :

State \ V_T	+	-	.	d
✓ S	A	A	G	B,C,E
✓ A			G	B,C,E
B				B
C			D	C
D				D
E			G	E
✓ G				H
✓ **H**				H
F				
✓ **B,C,E**			D,G	B,C,E
✓ **D,G**				D,H
✓ **D,H**				D,H

4. Delete all non-marked states . This results in this simplified Table :

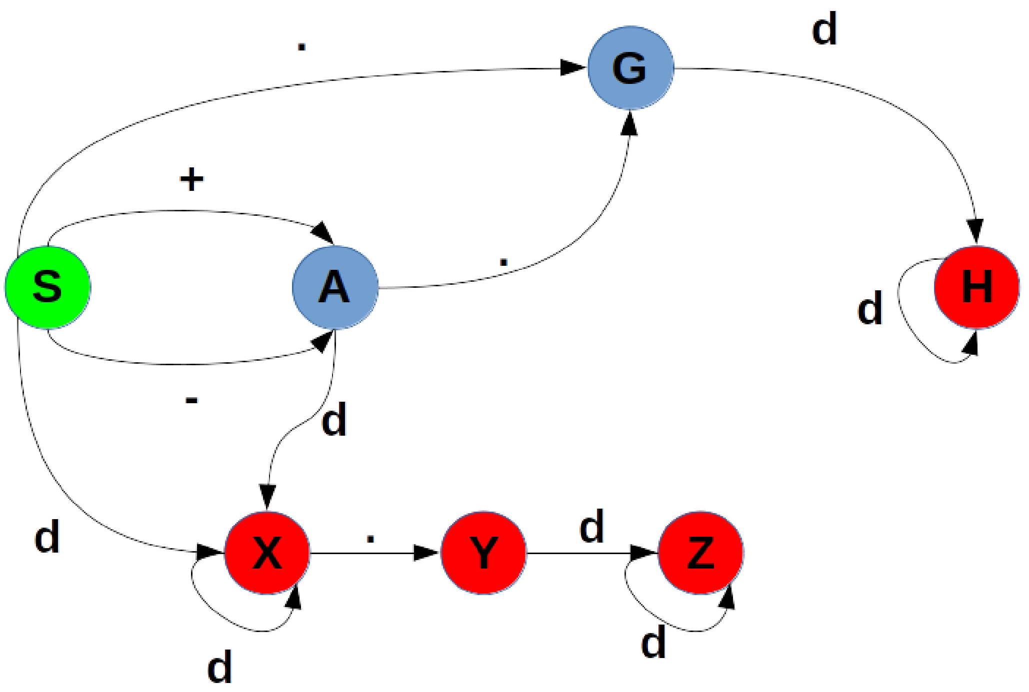
State \ V_T	+	-	.	d
✓ S	A	A	G	B,C,E
✓ A			G	B,C,E
✓ G				H
✓ **H**				H
✓ **B,C,E**			D,G	B,C,E
✓ **D,G**				D,H
✓ **D,H**				D,H

This is now a Deterministic Machine That accepts the same languages as the original NDFSA . For Clarity, Lets rename [B,C,E] to X, [D,G] to Y, [D,H] to Z.

State \ V_T	+	-	.	d
✓ S	A	A	G	X
✓ A			G	X
✓ G				H

State \ V_T	+	-	.	d
✓ **H**				H
✓ **X**			Y	X
✓ **Y**				Z
✓ **Z**				Z

And the graph now looks like this :



But this machine is not in its simplest state.

Merging Equivalent States

We will use the **feasible-pairs table** method in merging equivalent states.

A state pair (p,q) is a feasible pair if:

1. {p,q}⊂F OR {p,q}⊂Q-F ie, either both {p,q} are final states or both {p,q} are **not** final states.
2. for every token(symbol) a∈V_T, either both {p,q} have transitions on "a", or both {p,q} don't have transitions on "a".
3. p ≠ q ;

Note that (p,q) ≡ (q,p).

for example, given the following NDFSA, represented by the following transition table :

State \ V_T	a	b	c
1	2	5	
2	3	4	1
3	5	2	
4	6		1
5	1	4	1
6	4		1
7	3	5	3

To find the feasible pairs, first we must separate the set of final states from the set of non-final states. therefore, we have these 2 sets :

> Q-F = {1,2,3,5,7}
>
> F = {4,6}

this results in this feasible-pairs table :

feasible pairs \ V_T	**a**	**b**	**c**
(1,3)	2,5	5,2	
(2,5)	3,1	4,4	1,1
(2,7)	3,3	4,5	1,3
(5,7)	1,3	4,5	1,3
(4,6)	6,4		1,1

We then mark all feasible pairs (p,q) where There is a transition to a pair (r,s) such that :

1. r ≠ s.
2. (r,s) is either marked OR not among the feasible pairs.

This results in this feasible-pairs table :

feasible pairs \ V_T	**a**	**b**	**c**
(1,3)	2,5	5,2	
(2,5)	3,1	4,4	1,1
✓ (2,7)	3,3	4,5	1,3
✓ (5,7)	1,3	4,5	1,3
(4,6)	6,4		1,1

We go through the table once more, in case we marked something later on in the table that would effect the pairs in the top of the table.

if a pair (p,q) remains unmarked, that means that p is equivalent to q. therefore, we merge p and q, choosing one of them :

1. 1 ≡ 3 ---> 1
2. 2 ≡ 5 ---> 2
3. 4 ≡ 6 ---> 4

We then merge, replacing every 3 with a 1, every 5 with a 2, and every 6 with a 4, resulting in this state table :

State \ V_T	**a**	**b**	**c**
1	2	2	
2	1	4	1
4	4		1
7	1	2	1

This is the machine with the minimum number of states.

Lets go back to our example(the FNDSA we were already working on). Last time, we reached this state table :

State \ V_T	**+**	**-**	**.**	**d**
✓ S	A	A	G	X
✓ A			G	X
✓ G				H
✓ **H**				H
✓ **X**			Y	X
✓ **Y**				Z

State \ V_T	**+**	**-**	**.**	**d**
✓ **Z**				Z

Lets quickly apply what we learned on this table.

1. Separate the final from the non-final states :

   > Q-F ={S,A,G}
   >
   > F = {H,X,Y,Z}

   Constructing the feasible pairs table :

feasible pairs \ V_T	**+**	**-**	**.**	**d**
(H,Y)				H,Z
(H,Z)				H,Z
(Y,Z)				Z,Z

2. Marking feasible pairs

feasible pairs \ V_T	**+**	**-**	**.**	**d**
(H,Y)				H,Z
✓ (H,Z)				H,Z
✓ (Y,Z)				Z,Z

3. Merge and Replace

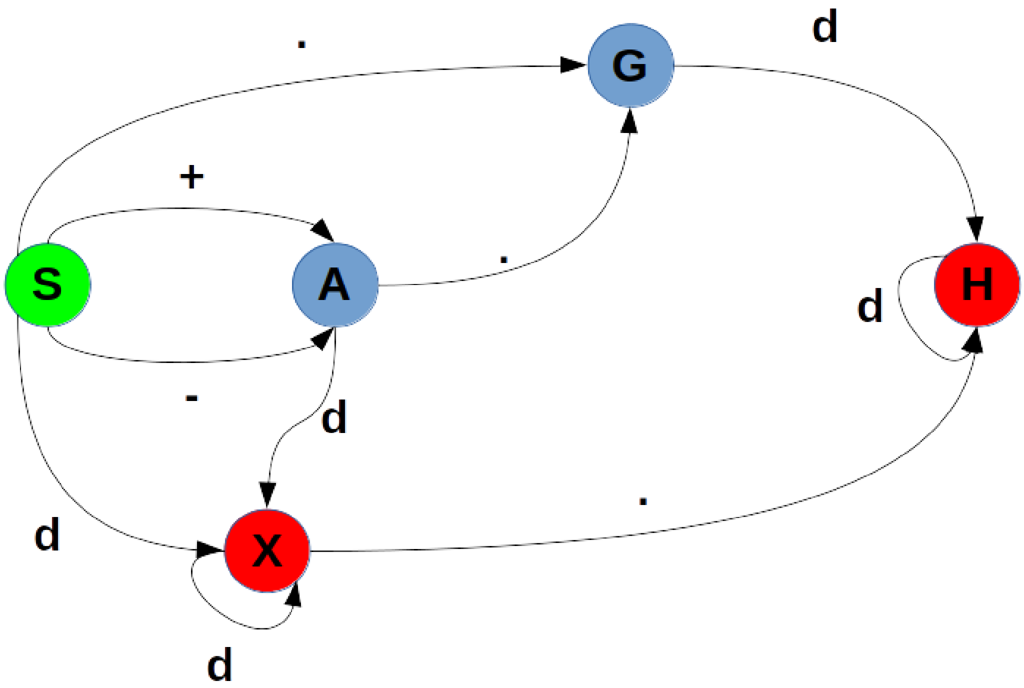
   H ≡ Y ≡ Z---> H

   Resulting in this state table :

State \ V_T	**+**	**-**	**.**	**d**
✓ S	A	A	G	X
✓ A			G	X
✓ G				H
✓ **H**				H
✓ **X**			H	X

   This is the simplest form of the machine.

Now we must check if the machine accepts the same language as our original machine.



This macchine accepts the language L(G) where :

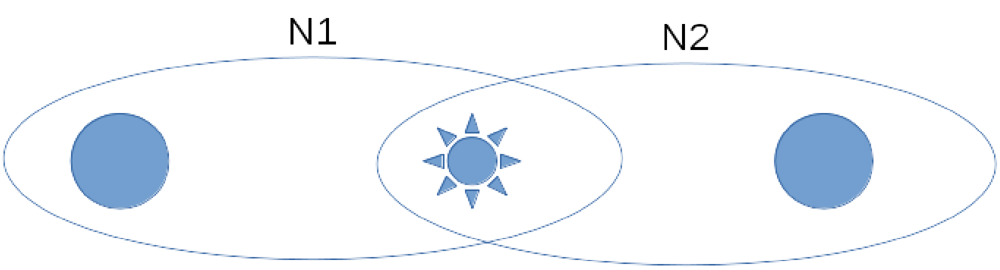
> L(G)=[+|-]{ ddddddd,
>
> ```
> dddd.ddd,
>
> dddddd.,
>
> .dddd
> ```
>
> }

Which is the same language of our original machine.

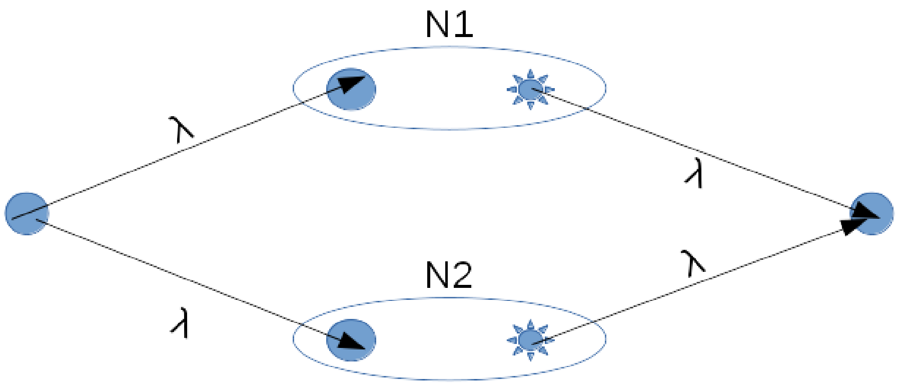
Creating a NDFSA From a Regular Expression

1. Decompose the regular expression to its primitive components : a. for λ, X-$^{\lambda}$->Y. b. for a, X-^{a}->Y.
2. Supposed that N_1, N_2 are transition diagrams for the regular expressions R_1, R_2 respectively, N_1 accepts R_1 & N_2 accepts R_2, then :

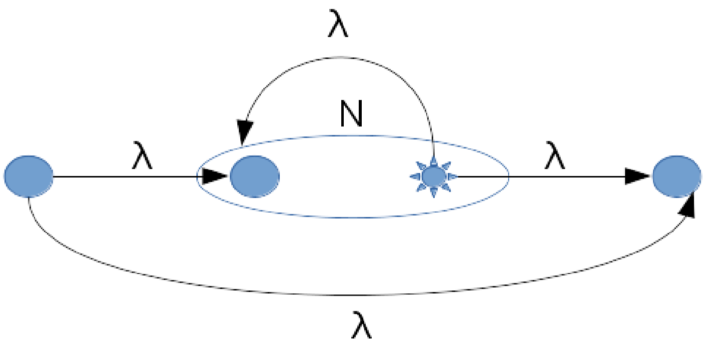
   a. N_{12} which represents R_1R_2 is :



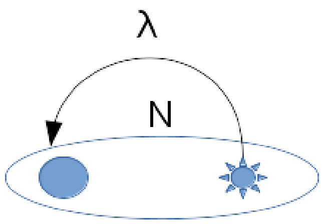
   b. $N_{1|2}$ which represents $R_1|R_2$ is :



c. N_x* which represents R_x* is :



d. N_x^+ which represents R_x^+ is :



Note that this is the same as R_x* except we removed all the states that result in a λ

for example, lets say we have the regular expression

> L(L|d)*

Whch has this transition table

State \ V_T	L	d	λ
1	2		
2			3,9
3			4,6
4	5		
5			8
6		7	
7			8
8			3,9
9			

Turning this into an DFSA :

1.

State \ V_T	**L**	**d**	**λ**
1	2		
2	5	7	3,9,8,6
3	5	7	4,6,8,3,9
4	5		
5	5	7	8,3,9,4,6
6		7	
7	5	7	8,3,9,4,6
8	5	7	3,9,8,4,6
9			

2.

State \ V_T	**L**	**d**
✓ 1	2	
✓ **2**	5	7
3	5	7
4	5	
✓ **5**	5	7
6		7
✓ **7**	5	7
8	5	7
9		

3.

State \ V_T	**L**	**d**
✓ 1	2	
✓ **2**	5	7
✓ **5**	5	7
✓ **7**	7	7

feasible pairs \ V_T	**L**	**d**
(2,5)	(5,5)	(7,7)
(7,7)	(5,5)	(7,7)
(5,7)	(5,5)	(7,7)

4.

State \ V_T	**L**	**d**
✓ 1	2	-
✓ **2**	5	7
✓ **5**	5	7
✓ **7**	5	7

programmatically, this results in :

```
getchar(ch);
case(ch){
    letter : get-name;
    digit : get-number
```

```
    < : getchar(ch)
    ...
    ...
}
```