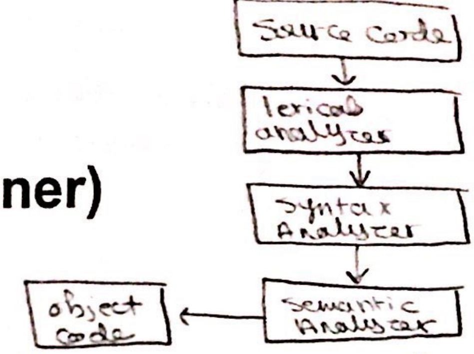
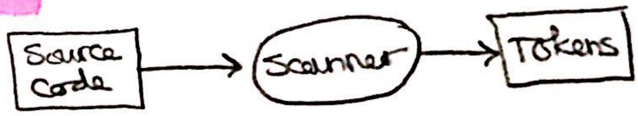


Chapter 4
Lexical Analysis(Scanner)

What is a Lexical Analyzer?

is an algorithm which recognizes the set of tokens in the source code.

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.
   a. Keywords/Reserved, which are words such as if/else/while. These names can't be used as variable names. They have a specific place and function
   b. User Defined Names, Which are the names declared by the user.
   (i-e) variable names, Constant names, function names ----
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.

⚠ Comments aren't part of the source code So they aren't considered tokens

Let us apply the scanner to this short segment of code:

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

This results in this set of tokens :

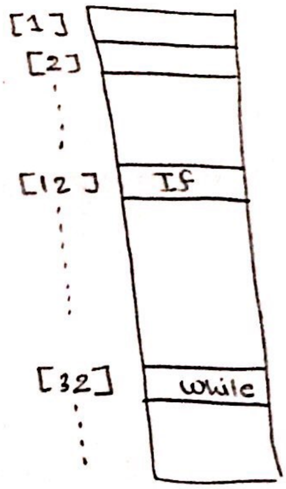
```
While , ( ,x ,>= ,100 ,) ,{ ,n ,+= ,x ,; ,x ,++ ,} .
```

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

Token-ID Token

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

* the Compiler maintains a table of the reserved words (sorted)



Leads us to these ID's :

Token-ID

Token	Internal Representation Number
While	33
(	84
x	100
>=	67
100	200
)	85
{	92
n	100
+=	77
x	100
;	81
x	100
++	75
;	81
}	93

* All the user defined names are given the same ID(index) Because the type of the name of the user defined name doesn't affect the syntax.

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

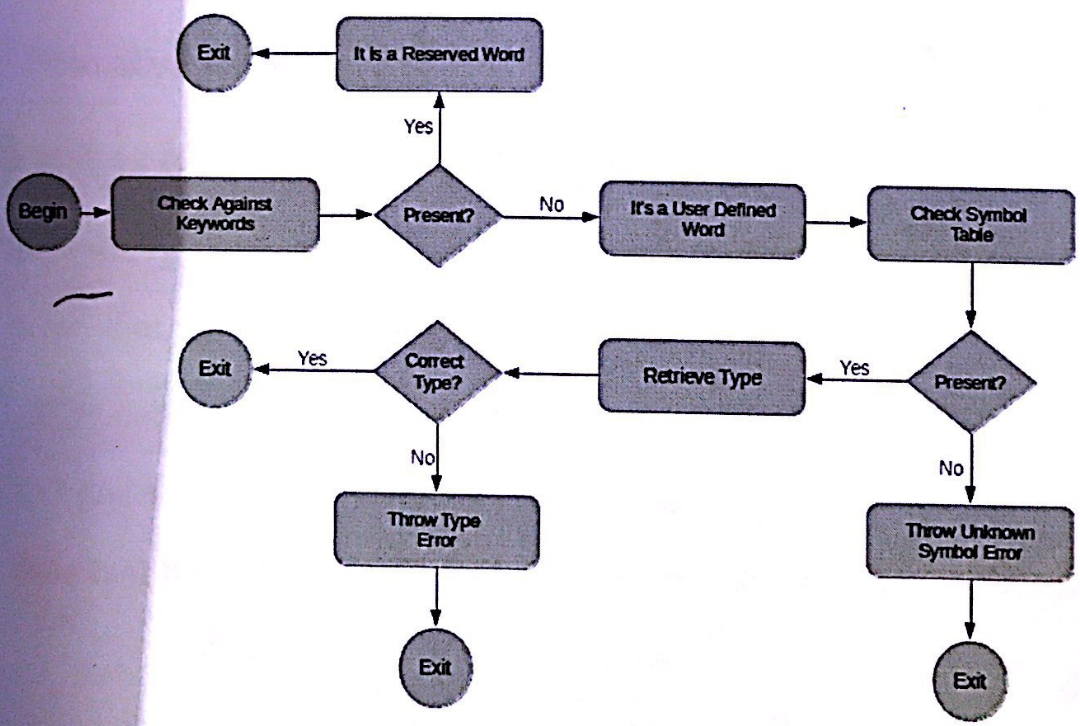
During this process of analysis, The Compiler builds what is called the **Symbol Table**. The Symbol Table is a table of the name of each user defined name(mostly variables), its type, and its values. The Symbol table for this segment of code would be:

int compute(int,int);
int n; = 6.52 <Float> → Error [Type mismatch]
float x,y;
const int m=10;
Compute (int, int) → Error [Identifier isn't declared]

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

$\Big\}$ user-defined names.

To perform type checking, the compiler takes the name, (1) and checks (2) 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 Expressions(Regular Languages)

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 :

Def: An Alphabet is a Set of Symbols.

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

V = {0 , 1} Binary alphabet

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

So if V is our alphabet, then :

abc
dsd
a
qwe
az
asa
sd

When we define * on IR then * is a binary operation defined on IR.

Are all strings defined on V.

Note: we can define an infinite number of strings on an alphabet.

Formally, we define:

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

takes 2 operations.

let V={0,1} with S defined over it, let us 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}.

غير تبديلية

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

Def: Given a set of strings S on V, then define the Concatenation Operation (∘) on S as follows:

$X \cdot Y = XY = \{XY \text{ such that } Y \text{ is following } X\}, X, Y \in S$

Let us Define a special string, called the empty string, which we denote with λ.
Notice that $\lambda X = X\lambda = X$.

Formally, we can say:

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

Where the Identity element is formally defined as :

Generally: If e is the Identity element of some operation ⊛ then: $e \circledast x = x \circledast e = x$

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

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

$V = \{a, b, c, \ldots, x, y, z\}$

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

L1={a, b, c}
L2={asdasd, qwe, asd}
L3={abb}

$L4 = \{\lambda\}$ ≡ Language which has only λ as a string.
$L5 = \{aa, abc, bc, \lambda\}$ — empty string
$L6 = \emptyset \equiv \{\}$ — a language which has no elements ≡ empty language

This definition also leads us to the conclusion :

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

$|S| = \infty$
↳ cardinality of S ≡ # of elements of S

Set of 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 "BINARY OPERATION"

Given that L, M are languages over an alphabet V, then , $L, M \in \mathcal{L}$
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 : NOTE ⚠

1.LM ≠ ML. (Concatenation on languages is not commutative).

2.L{λ} = {λ}L = L. (λ is the identity for concatenation).
{λ}

Example on the or operation:

given $L = \{ab, bc, bb\}$, $M = \{\lambda, ccc\}$

$L|M = L \cup M = \{ab, bc, bb, \lambda, ccc\}$

$M|L = M \cup L = \{\lambda, ccc, ab, bc, bb\}$

Example on the closure Operation:

given $L = \{aab, b\}$ over $V = \{a, b\}$

$L^* = L_0 \cup L_1 \cup L_2 \cdots = \bigcup_{i=0}^{\infty} L$

$L^* = \{\lambda\} \cup \{aab, b\} \cup \{aabaab, aabb, baab, bb\} \cup \ldots$

$L^* = \{\lambda, aab, b, aabaab, aabb, baab, bb, \ldots\}$

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

Example on the recursive definition of Regular language

$V = \{a, b, c\}$

1. $\emptyset = \{ \}$
2. $\lambda = \{\lambda\}$
3. $a = \{a\}$
   $b = \{b\}$
   $c = \{c\}$
4. $R = L_R$
   $S = L_S$

   a. RS a regular expression denoting the languages L_R & L_S → (i.e) $ab = \{a\} . \{b\} = \{ab\}$

   b. R|S a regular expression denoting $L_R | L_S = L_R \cup L_S$

   (i.e) $a|b = \{a\} \cup \{b\} = \{a, b\}$

   (i.e) $a|ab = \{a\} \cup \{ab\} = \{a, ab\}$

   c. $R^* = L_R^*$

   → $(a|b)^* = (\{a\} \cup \{b\})^* = (\{a, b\})^*$
   $= \{\lambda, ab, aa, ab, ba, bb, \ldots\}$

3. L{ } = { }L = { }.

The OR "|" Operation

"BINARY OPERATION"

OR is ∪ operation in set theory

Union

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

$L|M$ = "L OR M" = $\{x \mid x \in L \text{ or } x \in M\} = L \cup M$.

Note that :

NOTE

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)

$\emptyset$

The Closure "*" Operation (A Unary Operation)

we use(*) to denote this operation.

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

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

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

The Recursive Definition of Regular Languages

Given an alphabet V then :

1. $\emptyset = \{\}$ = empty language is a regular language denoting the language { }
2. $\lambda = \{\lambda\}$ is a regular language denoting the language λ
3. For every element $a \in V$, **a**= {a} is a regular language denoting the language {a}
4. Given R and S are regular languages denoting the regular languages LR and LS respectively, then :

   a) RS is a regular language denoting LRLS

   b) R|S is a regular language denoting LR|LS

   c) R* is a regular language denoting LR*

EXAMPLES:

Given R={a} and S={b}, then :

- **RS={ab},**
- **R|S={a,b},**
- **$R^*= \{a\}^0 \cup \{a\}^1 \cup ....$**

  $= \{\lambda, a, aa, aaa, ...\}$

  = A string that consists of any number of a's
- Lets say we took (RS)* then

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

  $= \{\lambda, ab, abab, ababab, abababab, ...\}$

  = A string that consists of any number of "ab"s
- Lets say we took (a|b)* , then :

  $(a|b)^* = (\{a\}|\{b\})^* = (\{a\} \cup \{b\})^*$

  $= (\{a,b\})^*$ = Any string of a's and b's

- Let us say we took (0|1)*00, then By the definitions above, this results in any binary string that ends with 00, such as {100,000,1100,0000,...} /!\ any string of 0's & 1's that ends with 00.
- Let us 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 consecutive b's such as {bbb,abbb,bbba,bbbbb,...}

Example on Exponential Notation

$[+|-]\ \underline{d}\ .\ \underline{d^+}\ E\ \underline{(+|-)}\ d^+$

must have only one digit

any digit but?

there must be + or −

⚠ All tokens in the source code are strings (members) in language of type regular expression.

Defining Tokens Using Regular Languages:

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: letter 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 :

   [+|-]digit(digit)* = [+|-]dd* = [+|-]d+ → we either take + or − or we don't.

   all digits

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

2. Floating Point Numbers :

   [+|-] d+.d+ → Fixed Point → 0.5 ✓, .5 ✗ // [+|-] d*.d+ → 0.5 ✓, .5 ✓

   [+|-] d.d+E(+|-)d+ → Exponential

   must ⚠

   these brackets mean that there must be either + or − .

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 :

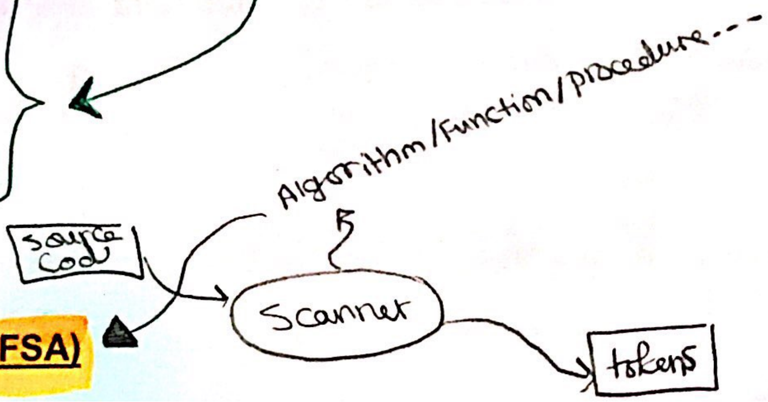
+ = {+}

* = {*}

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

or ++ , which is given by

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



Finite State Automata(FSA) ▲

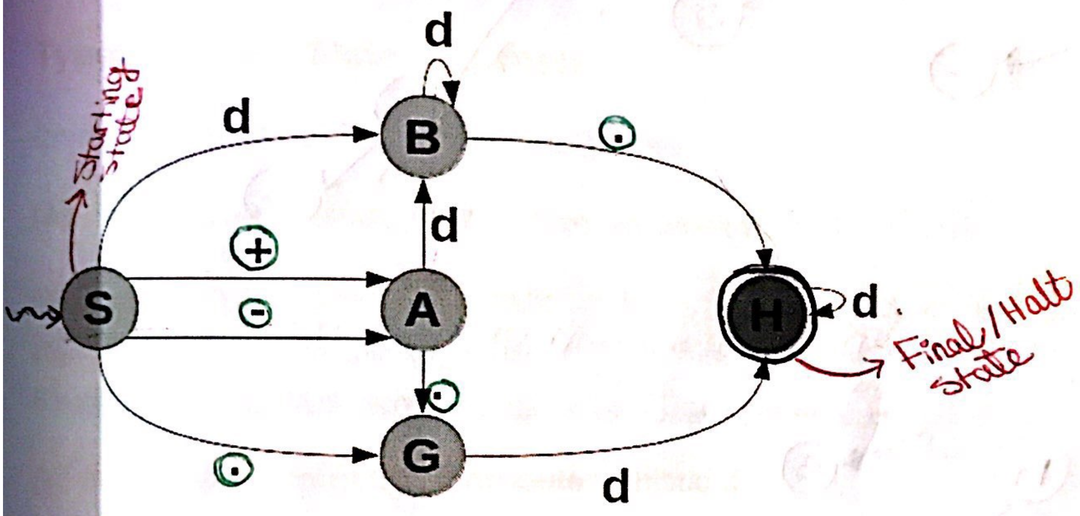
(8)

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

Tokens in source codes are strings of regular languages. The algorithm that recognizes 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. $Q = \{S, A, B, G\}$
2. Transitions between states. transitions $\to$ $S \xrightarrow{+} A$, $S \xrightarrow{-} A$, $S \xrightarrow{d} B$, $S \xrightarrow{\cdot} G$ ⋮
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(Halt) State.**

- The transitions between the states are given by{+, -, d, .}

- Given 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
```

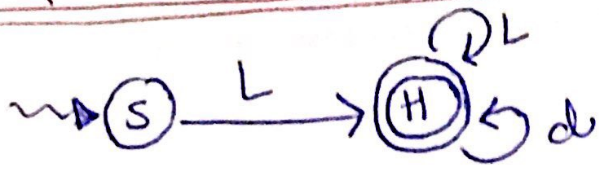
If after scanning the inputs, the machine ends up in a Final / Halt state we say

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

Def: A string is accepted or recognized if after scanning the

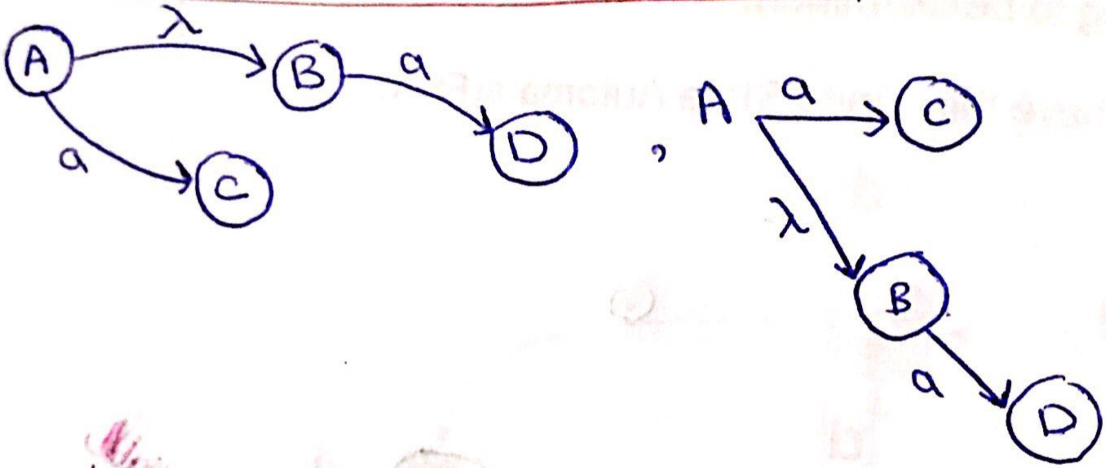
9

Example on Finite State Machine For Names

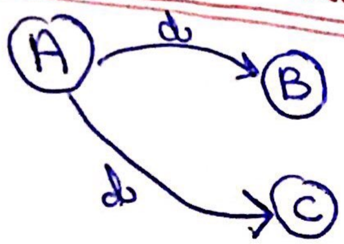


Example: {+=}

(1) If there are λ-transitions in the FSA:.



(2) Theres more than one transition on the same input from the same state.



string we end up with a final state.

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

+ / - / unsigned.

$L(M)=[+|-]\{d^+., .d^+, d^+.d^+\}$ → it accepts all floating point numbers in the Fixed point anotation.

Other examples of finite state machines are :

1. Names
2. Integers
3. <=

Types of Finite State Automata

there are 2 types of Finite State Automata

(A) Non-Deterministic Finite State Automata [NDFSA]

An algorithm is non-deterministic or fuzzy if there are options(choices) 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 choices (trial and error) to make. The only way to solve non-deterministic machines is to use backtracking. This is practical 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

(B) 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-moves(transitions).
2. AND There is **NO** more than one transition from the same state on the same input.

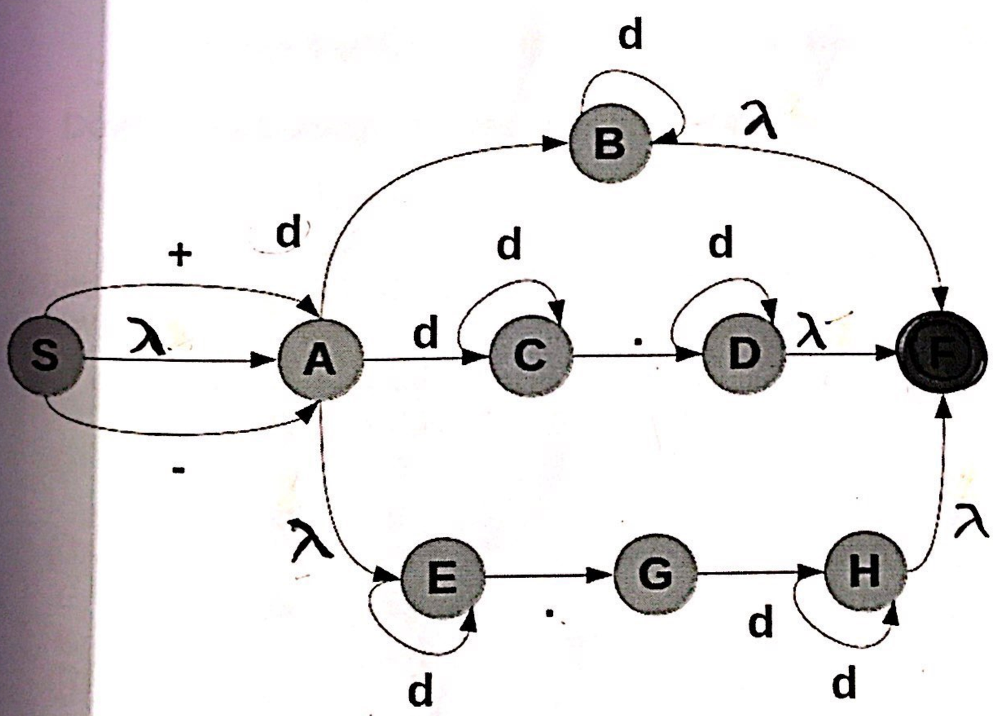
Only Deterministic Finite State Automata are used for compilers.

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.

Given the following NDFSA :



What is L(G) = ?

Integers floats

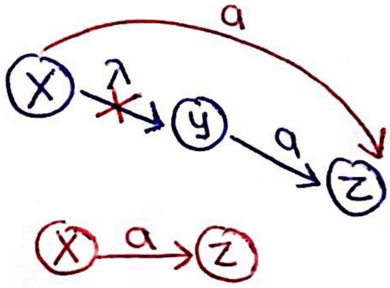
L(G)=[+|-] (d^+ | $d^+.d^+$ | $d^+.$ | $.d^+$)

Signed
Unsigned

And we want to transform it into a DFSA. Let's follow through the steps :

Let's transform the Finite state machine into a transition table: 1

State \ VT	+	-	.	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					



(1) Removal Lambda Transitions

1. Consider S-λ->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. B,D,A are marked as Final states

4. Delete the λ Column This results in this table :

State \ VT	+	-	.	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				

(2) Removal Of Non-Determinism

Which mean 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 :

O ~> Final State

State \ VT	+	-	.	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

(3) Removal of Non-Accessible States ~> the states that can't be reached.

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.

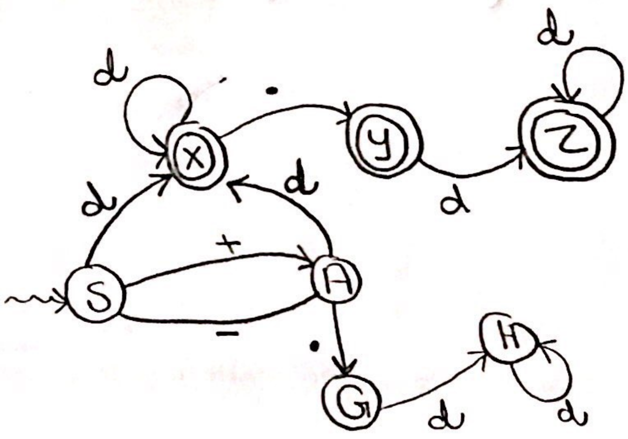
⚠ Do one more extra iteration because some times you mark states above the current state.

This results in this table :

State \ VT	+	-	.	d
✓S	A	A	G	B,C,E
✓A			G	B,C,E
X B				B
X C			D	C
X D				D
X E			G	E
✓G				H
✓H				H
X F				
✓B,C,E X			D, G	B,C,E
y ✓D,G				D,H
z ✓D,H				D,H

(4) Delete all unmarked states . This results in this simplified Table :

State \ VT	+	-	.	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



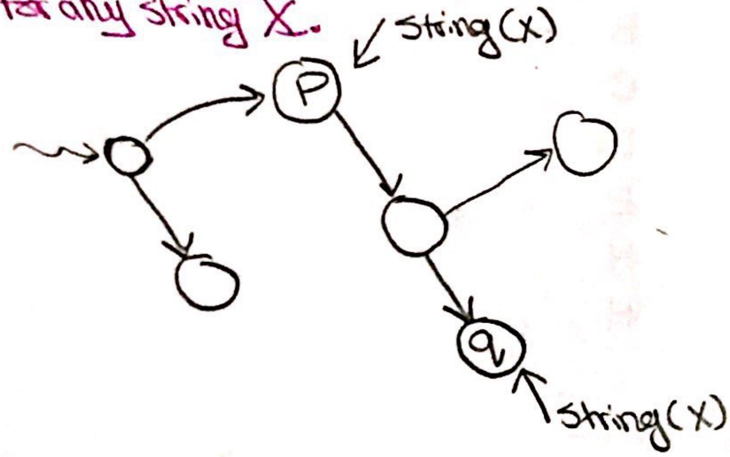
This is now a Deterministic Machine That accepts the same languages as the original NDFSA . For Clarity, Let.s rename [B,C,E] to X, [D,G] to Y, [D,H] to Z.

(4) Merging equivalent states

Definition

two states p, q in the FSA are said to be equivalent if:

p accepts the string X iff, q accepts the string X, for any string X.



p & q accept string(x)

modified

Revision

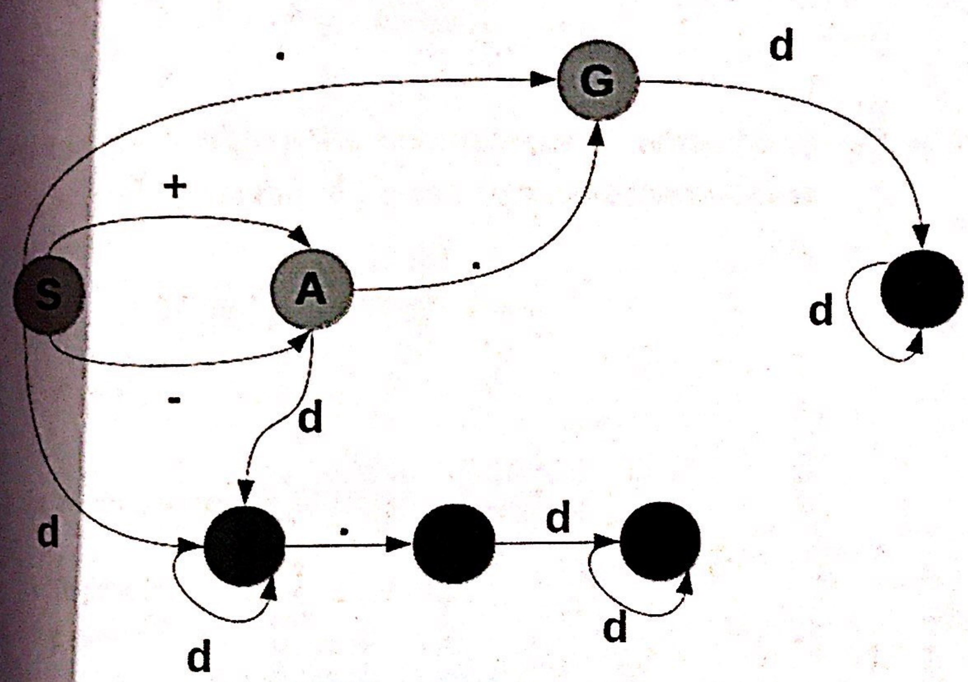
A state q in machine M & state p in machine M' are said to be Equivalent if given a string X, the:

machine M in state q accepts X iff machine M' in state P accepts X for every string X.

$$\{P \equiv q\}$$

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

And the graph now looks like this :



(4) Merging Equivalent States

We will use the **feasible-pairs table** method in merging equivalent states. (we look for)

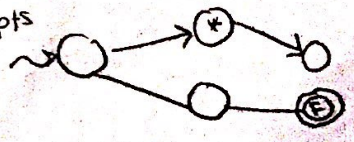
A state pair (p,q) is a feasible pair if: (to be equivalent.)

1. $\{p,q\} \subset F$ OR $\{p,q\} \subset Q-F$ ie, either both {p,q} are final states or both {p,q} are not final states.
2. for every token(symbol) $a \in VT$, either both {p,q} have transitions on "a", or both {p,q} don't have transitions on "a".
3. $p \neq q$;

Note that $(p,q) \equiv (q,p)$.

State * doesn't accept λ, but Final state accepts λ so they're not Feasible

(i.e)



Definition

A Feasible state pair (p,q) is marked if there's a transition from (p,q) to a pair (r,s) such that (r,s) is either marked or not among the feasible pairs. & $r \neq s$.

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

State \ VT	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

Table #1

final

(1,2) can't be feasible since there's transition to C(2), but there's no transition to C(1)

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} "Non-Final"

F = {4,6} "Final"

this results in this feasible-pairs table :

feasible pairs\VT	a (r,s)	b (r,s)	c (r,s)
(1,3)	2,5x	5,2x	
(2,5)	3,1x	4,4x	1,1x
✓ (2,7)	3,3x	4,5✓	1,3x
✓ (5,7)	1,3x	4,5✓	1,3x
(4,6)	6,4x		1,1x

(1-3) → (2,5) ↳ not marked ✓ ↳ among feasible pairs X } X

→ (5,2) ≡ (2,5) X

Those pairs are feasible to be EQUIVALENT

⚠ repeat the iteration one more time.

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

1. $r \neq s$.

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

≡ machine in Table #1.

This is the machine with the minimum number of states.
Let's go back to our example 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
Z				Z

Lets quickly apply what we learned on this table.

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\VT	+	-	.	d
(H,Y)				H,Z
(H,Z)				H,Z
(Y,Z)				Z,Z

1. Marking feasible pairs

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

2. Merge

and Replace

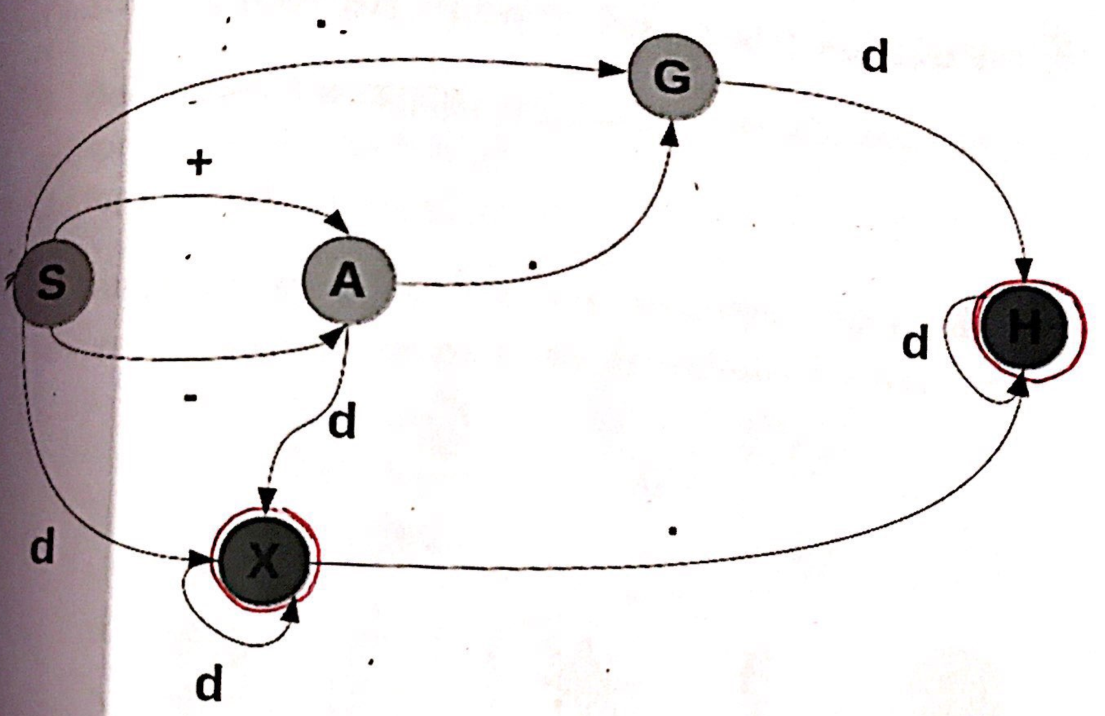
H ≡ Y ≡ Z --> H

Resulting in this state table :

State \ VT	+	-	.	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 machine accepts the language.

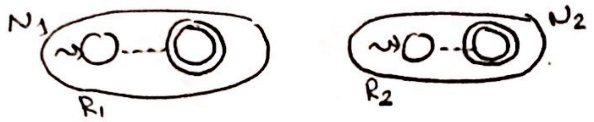
L(G) = [+|-] { dddd, ddd.dd, dddd., .dddd }

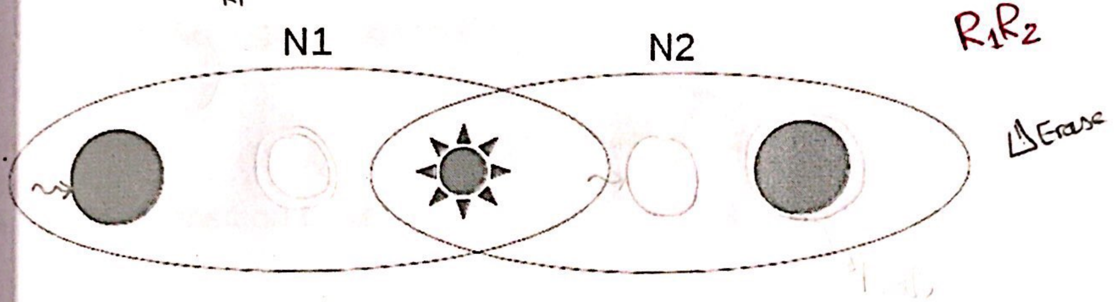
Which is the same language of our original machine.

Creating a NDFSA From a Regular Expression

1. Decompose the regular expression to its primitive components :
   - a. for λ, X-λ->Y.
   - b. for a, X-a->Y.

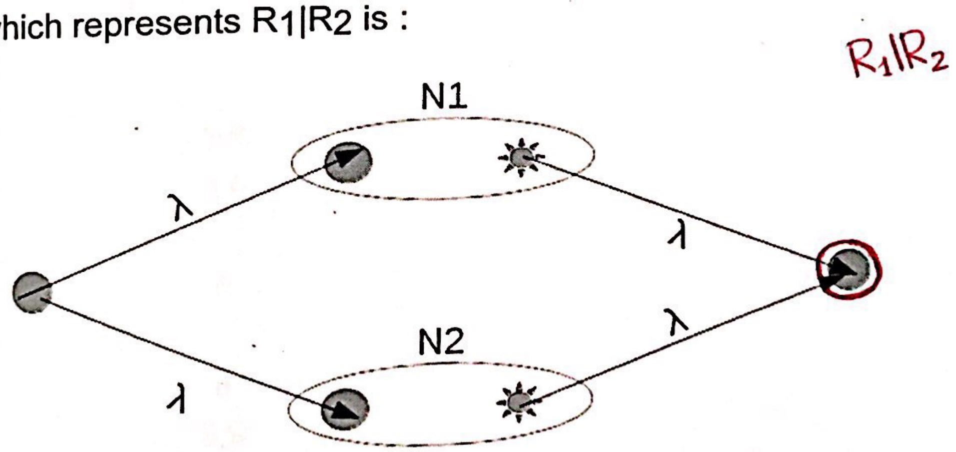
2. Supposed that N_1, N_2 are transition diagrams for the regular expressions R_1, R_2 respectively, N_1 accepts R_1 & N_2 accepts R_2, then :



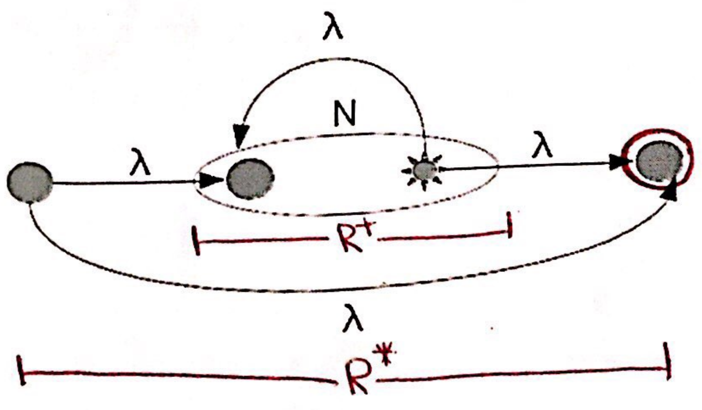


a. which represents R_1R_2 is :

b. which represents $R_1|R_2$ is :

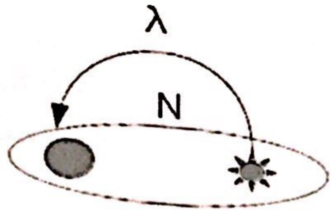


c. which represents R_x* is :



d. x N
+

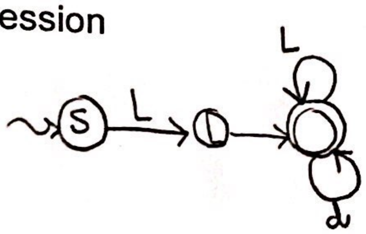
which represents R $^+$ is :



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

<u>Example</u>: Given the regular expression

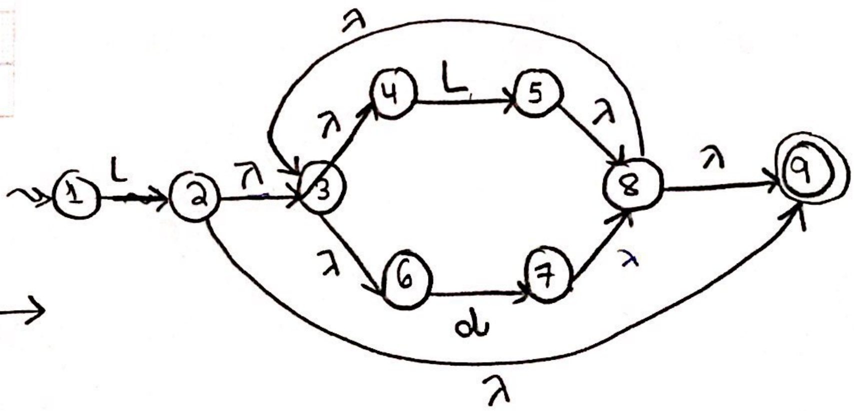
L(L|d)* "Names"



Which 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			

NDFSA →
L(L|D)*



State\ VT	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			

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

State \ VT	L	d
✓1	2	
✓2	5	7
✓5	5	7
✓7	7	7

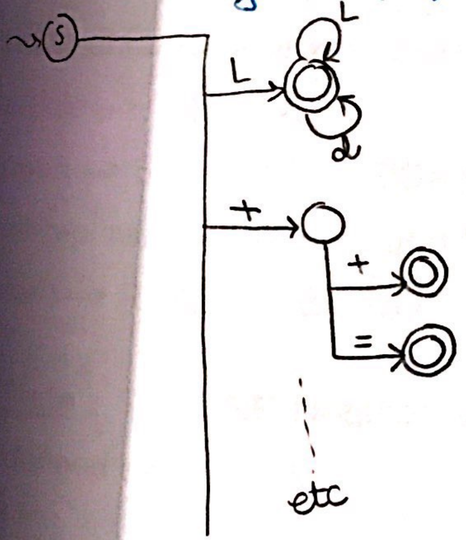
feasible pairs \ VT	L	d
(2,5)	(5,5)	(7,7)
(7,7)	(5,5)	(7,7)
(5,7)	(5,5)	(7,7)

2 ≡ 5 ≡ 7

State\ VT	L	d
1	2	-
2	2	2

* Our Scanner [lexical Analyzer] Algorithm, Contains a set of DFSA.

- getchar(ch)



Token

Name

+	x = n+5;
++	X++;
+=	X+ =5;

Chapter 5

Syntax Analysis(Parser)

A Syntax analyzer is formally defined as :

An Algorithm that Groups the Set of Tokens Sent by the Scanner to Form **Syntax Structures** Such As Expressions, Statements, Blocks, etc.

Simply, the parser examines if the source code written follows the grammar(production rules) of the language.

The Syntax structure of programming languages and even spoken languages can be expressed in what is called **BNF** notation, which stands for **Bakus Naur Form.**

For example, in spoken English, we can say the following:

sentence --> noun-phrase verb-phrase First Notation

noun-phrase --> article noun

article --> THE | A | ...

noun --> STUDENT | BOOK | ...

verb-phrase --> verb noun-phrase

verb --> READS | BUYS | ....

Note : The BNF Notation uses different symbols, for example, a sentence is defined as :

< sentence > ::= < noun-phrase > < verb-phrase > Second notation.

مزعج

But this is very cumbersome, so we use the first notation, since it is easier to use. Now, let us derive a sentence :

sentence --> **noun-phrase** verb-phrase

--> **article** noun verb-phrase

--> THE **noun** verb-phrase

--> THE STUDENT **verb-phrase**

--> THE STUDENT **verb** noun-phrase

--> THE STUDENT READS **noun-phrase**

--> THE STUDENT READS **article** noun

--> THE STUDENT READS A **noun**

--> THE STUDENT READS A BOOK

⚠ STEP BY STEP

In the same way, the parser tries to **derive** your source program from the starting symbol of the grammar. Let us say we have these sentences :

dream

THE BOOK BUYS A STUDENT
THE BOOK WRITES A DISH

Syntax-wise, all of these sentences are correct. However, their meaning is not correct, and they are not useful. What differentiates 2 sentences that are grammatically correct is their meaning or their **semantics**. You and I can agree that the meaning of a grammatically correct sentence is not correct, but how does the computer do it?

Grammar of a programming language:-

A grammar $G=(V_N, V_T, S, P)$ where:

1. V_N : A finite set of nonterminals(nonterminals set).
2. V_T : A finite set of terminals(terminals set).
3. $S \in V_N$: The Starting symbol of the grammar. "non-terminal"
4. P = A set of **production rules**(productions).<-- Pending <==> Basically the whole grammar.

Note :

1. $V_N \cap V_T = \varnothing$.
2. $V_N \cup V_T = V$ (the vocabulary of the grammar).

Note : We will use:

1. Uppercase Letters A,B,...,Z for non-terminals. V_N
2. Lowercase Letters a,b,...,z for terminals. V_T
3. Greek letters α,β,γ,... for strings V

formed from V_N OR $V_T = V$.

$V_N \cup V_T$

eg, if $V_N = \{S,A,B\}$, $V_T = \{0,1\}$

then

α = A11B
β = S110B
γ = 0010

Productions

Contains at least one non-terminal

1. A Production $\boxed{\alpha \text{-->} \beta}$ (alpha derives beta) is a rewriting rule such that the occurrence of **α** can be substituted by **β** in any string.

Note that α must contain at least one nonterminal from V_N, i.e. $\in V_N$. For example, Assume we have the string γασ,

γασ --> γβσ

2. A Derivation is a sequence of strings $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_n$, then :

$\alpha_0 \to \alpha_1 \to \alpha_2 \to \ldots \to \alpha_n$

Given a grammar G, then :

L(G) = Language Generated By the Grammar.

for example, Given the Grammar, $G = (\underbrace{\{S,B,C\}}_{V_N},\underbrace{\{a,b,c\}}_{V_T},S,P)$,

P :
S--> aSBC
S--> abC
CB --> BC
bB--> bb
bC--> bc
cC--> cc

∴ contains at least one non-terminal.
∴ Non-terminal represented by capital letter.

What is L(G)=? ← mark it !

Let us do some derivations: "Derive sentences"

S --> ab**C** --> $\boxed{\text{abc}}$(all terminals) ∈ L(G) <--- A sentence

S --> aSBC --> aa**bC**BC --> aabbcBC --> blocked, so we try another path

S --> aSBC --> aab**CB**C --> aab**B**CC --> aab**bC**C --> aabb**cC**

--> $\boxed{\text{aabbcc}}$ ∈ L(G) <--- A sentence

S --> aSBC -->........--> $\boxed{\text{aaabbbccc}}$ ∈ L(G) <--- A sentence

Therefore, $L(G)=\{a^n,b^n,c^n \mid n \geq 1\}$

As another Example, we have these productions

E--> E+T <-- we can write the productions 1 and 2 as a single production $\boxed{\text{E --> E+T | T}}$
E--> T
T--> T*F ←-- $\boxed{\text{T --> T * F | F}}$
T--> F
F--> (E) <-- we can write the productions 5 and 6 as a single production $\boxed{\text{F --> (E) | n}}$
F--> n

$V_N = \{E, T, F\}$
$V_T = \{+, *, (,), n\}$
$S = E$

Let us follow through some derivations

E --> **T** --> **F** --> n ∈ L(E)

E --> **E**+T --> T+**T** --> T+**F** --> **T**+n --> **F**+n --> n+n ∈ L(E)

E --> **E**+T --> **T**+T --> **F**+T --> n+**T** --> n+**F** --> n + (**E**) --> n+(**T**) --> n+(**T***F) -->
n+(**F***F) --> n+(n***F**) --> n+(n*n) ∈ L(E)

L(G) :- The set of all arithmetic operations ~~derived~~ with + & * operations.

4

⚠ The Algorithm we used in the Derivation steps in Backtracking "Try & Error" and the Language is Non-Deterministic Because it has options!

Therefore, L(G) = {Any arithmetic expression with * and + operations}, n is an operand here.

Note that, if we add the productions

E --> E+T | E-T | T

T --> T*F | T/F | T%F

We would have a language to express all arithmetic expressions with (* , \ , + , -) operations.

Let us Take another <u>Example</u> (Tokens between double quotes are terminals)

Program --> block "#"

block --> "{" stmt-List "}"

stmt-List --> statement ";" stmt-List | λ

statement --> if-stmt | while-stmt | read-stmt | write-stmt | assignment-stmt | block if-

if-stmt-> "if" condition....

while-stmt --> "while" condition.....

....

....

read-stmt --> "read"

write-stmt --> "write"

THOSE TOKENS DON'T APPEAR IN THE SOURCE CODE.

VN = {Program, block, stmt-List, statement, if-stmt, while-stmt,

read-stmt, write-stmt, assignment-stmt}

VT = { "{", "}", "#", ";", "if", "while", "read", "write" }

THOSE TOKENS APPEAR IN THE SOURCE CODE.

Let us Follow through some derivations :

Program --> **block** # --> { **stmt-list** } # --> { λ } #

Program --> **block** # --> {**stmt-list**} # --> {statement ; **stmt-list**} # -->
{statement ; statement ; **stmt-list**} # -->
{statement; statement ; λ} # --> {**statement** ; statement ;} # --> {**READ-statement** ;
statement ;} # -->{READ ; **statement** ;} # -->
{READ ; write-statement ;} # -->
{READ ; WRITE ;} #

The language of this language is defined as

L(G) = {Set of all programs that can be written in this language}.
Set of all programs derived from this grammer.

This is only a simple example, of a simple language. For something more complex such as C or Pascal, there are hundreds of productions.

Algorithms for Derivation

A Leftmost derivation is a derivation in which we replace the **leftmost** nonterminal in each derivation step.

A Rightmost derivation is a derivation in which we replace the **rightmost** nonterminal in each derivation step.

DEFINITIONS

For example, given

the grammar

V --> S R$ Stop Symbol.

S --> + | - | λ

R --> .dN | dN.N

N --> dN | λ

V_N = {V,R,S,N}
V_T = {+, - , ., d, $}

Let us follow through on the leftmost derivation

V --> **S**R$ --> -**R**$ --> -d**N**.N$ --> -dd**N**.N$ --> -dddN.N$ --> -ddd.**N**$ --> -ddd.d**N**$ --> -ddd.d$ <-- A sentence.

Let us follow through on the rightmost derivation

V --> S**R**$ --> SdN.**N**$--> SdN.d**N**$ --> Sd**N**.d$ --> sdd**N**.d$ --> sddd**N**.d$ --> **S**ddd.d$ --> -ddd.d$ <-- A sentence.

Derivation Trees

A Derivation Tree is a Tree that displays the derivation of some sentence in the language. For example, let us look at the tree for the previous example

Examples on Leftmost & Rightmost Derivations

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid n$

(*1) $E \to E+T \xrightarrow{Lm} T+T \xrightarrow{Lm} F+T \xrightarrow{Lm} n+T \xrightarrow{Lm} n+F \xrightarrow{Lm} n+n \in L(G)$

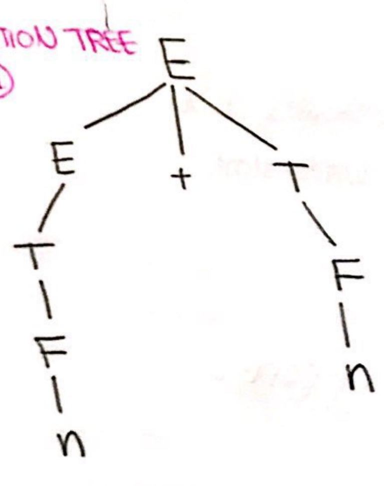
↳ LEFTMOST

(*2) $E \xrightarrow{RM} E+T \xrightarrow{RM} E+F \xrightarrow{RM} E+n \xrightarrow{RM} T+n \xrightarrow{RM} F+n \xrightarrow{RM} n+n \in L(G)$

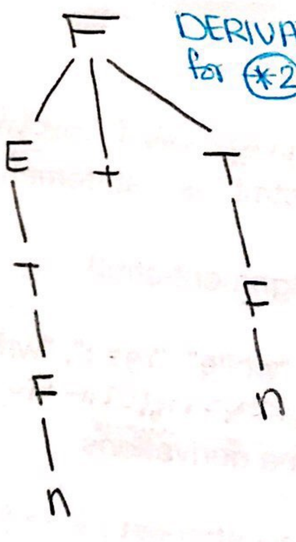
↳ RIGHTMOST

⚠ Note:- If we traverse the derivation tree INORDER considering only the leaves, we obtain the Sentence.

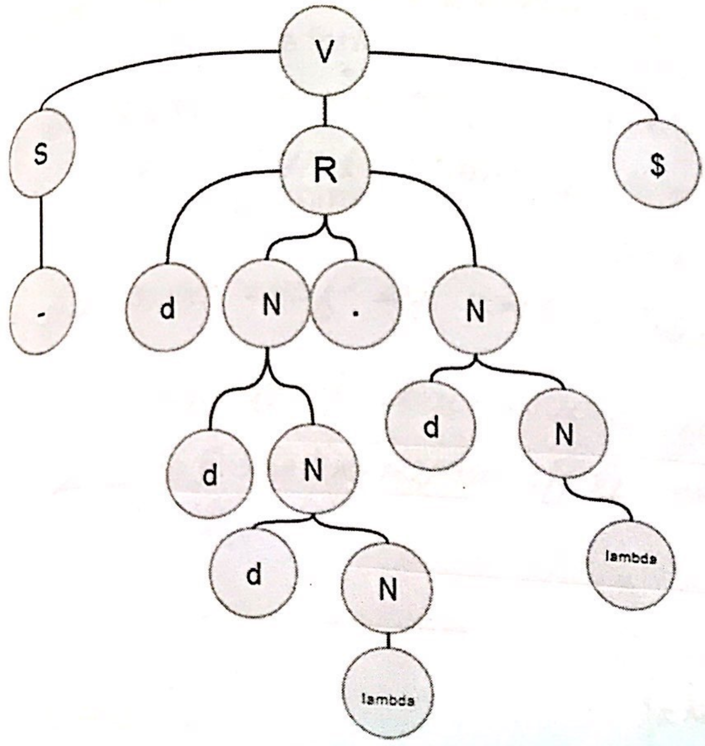
DERIVATION TREE for (*1)



DERIVATION TREE for (*2)



⚠ Same Derivation TREE for the Sentence n+n



Note that if we traverse the tree in order, recording **only** the leaves, we obtain the sentence.

Classes of Grammars

According to Chomsky, There are 4 classes of grammars :

1. Unrestricted Grammars : No restrictions whatsoever except the restriction by definition that the left side of the production contains at least one nonterminal from V_N. This grammar is not practical and we cannot work with it.

   its very hard to work with

2. Context-Sensitive Grammars : For each production α --> β, |α| ≤ |β|, i.e. , the **length of alpha(α)** is less than or equal to the **length of Beta(β)**. This means that in this class of grammar, there are no λ productions in the form A --> &lambda, since |λ| = 0 and A ≥ 1.

   $(A \to \lambda)$ X

3. Context-Free Grammar(CFG) : Each production in this grammar class is of the form A --> α , where A ∈ V_N and α ∈ V^*

   that is to say, the left hand side is **only** one nonterminal.

   This is the most important class of grammar. Most programming languages structures are context-free. We will mostly be working with this class of grammar. Most of the examples we have taken are CFG.

4. Regular Grammar (Regular Expressions) : Each production in this grammar class is of the form A -- > aB or A --> a, where A,B ∈ VN and a ∈ VT, **with the exception** of S --> λ

↳ Starting Symbol.

5.
For example, given the grammar:

A --> aA
A --> a

Therefore, we get L(G)=a^+

However, adding the production A --> λ

Results in the grammar

G(L)=a^*

⚠ Regular Grammar is a subset of Context Free Grammar

Parsing Techniques

There are 2 main parsing techniques used by a compiler.

Top-Down Parsing

① leftmost derivation until it derives the sentence or deadlock.
③ Major question: which production the parser must select for the root derivation step.

In Top-Down Parsing, the parser builds the derivation tree from the root(S : the starting symbol) down to the leaves(sentence).

In Simple words, the parser tries to derive the sentence using leftmost derivation. For example, say we have this grammar : V --> SR$

S --> + | - | λ

R --> .dN | dN.N

N --> dN | λ

$L(G) = [+|-]\left\{\begin{matrix} d^+ . \\ d^+ . d^+ \\ . d^+ \end{matrix}\right\}$

Let us examine if the sentence dd.d$ if it is derived from this grammar.

V --> SR$ --> λR$ --> dN.N$ --> ddN.N$ -->

dd.N$ --> dd.dN$ --> dd.d$

Problem. The Parser does not know which production it should select in each derivation statement. We will learn how to solve these issues later in the course.

DEFENETION

PARSER [Syntax Analyzer]:

is an algorithm which takes your source Code and tries to:-

(1) Build the derivation tree for your source Code

OR

(2) Derive your source Code from the Production rules of the grammar. (Using left-most Derivation)

TOP-DOWN PARSING PROBLEM

$E \to T$

$T \to F$

$F \to a$ the only Sentence.

⚠ Not having multiple productions means that we only have one sentence.

Bottom-Up Parsing

In Bottom-Up Parsing, the parser builds the derivation tree from the leaves(sentence) up to the root(S : Starting Symbol). This type of tree, built from the leaves to the root, is called a B-Tree.

In Simple words, the parser starts with the given sentence, does **reduction**(opposite of derivation) steps, until the starting symbol is reached.

Note that the string λ is present everywhere in the string,

and we can use it wherever we like. Let us follow the

reduction of the example given above.

+dd.d$ --> +ddλ.d$ --> +ddN.d$ --> +dN.d$ --> +dN.dλ$ -->

+dN.dN$ --> +dN.N$ --> +R$ --> SR$ --> V Which means that the

sentence is in the grammar.

Note that we can run into deadlocks here. say we took this path instead :

+dd.d$ --> +ddλ.d$ --> +ddN.d$ --> +dN.d$ --> +dN.dλ$ --> +dN.dN$ -->

+dNR$ **--> +NR$ --> SNR$** --> Deadlock This technique also has a major

problem : Which substring should we select to reduce in each reduction

step?

how do we solve this?

Ambiguity

Ambigious ≡ Not Deterministic ≡ If there are choices.

non-deterministic Algorithm.

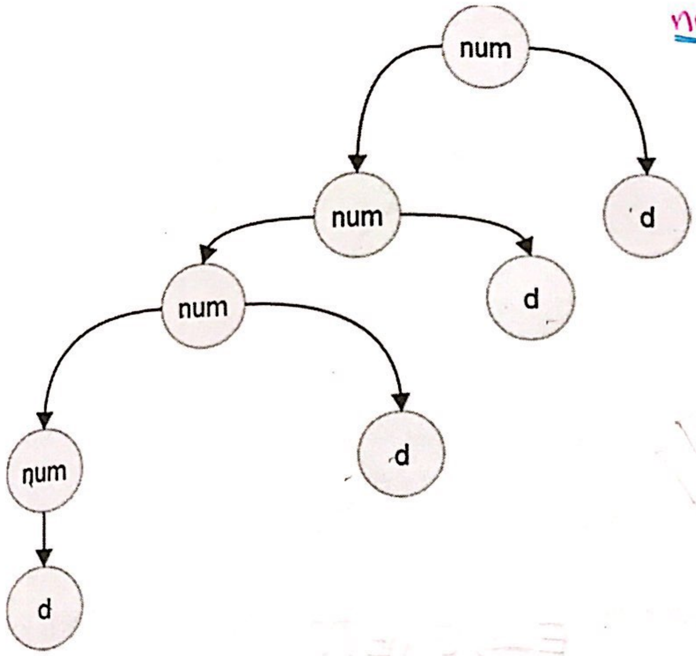
Given the following grammar :

num --> num d
num --> d

Similar to regular grammar.

Let us draw the derivation tree for the sentence dddd

9



num → num d → num d d → num d d d → d d d d ∈ L(G)

⚠ One sentence → one derivation tree

↳ Unambigious.

Question : is there another derivation tree that represents the sentence? The answer is **no**.

If there is only one derivation tree representing the sentence, this means there is only one way to derive the sentence. Based on this, we can say that

↳ Unambigious.

:

Def: A Grammar G is said to be ambiguous if there is at-least one sentence with more than one derivation tree. That is, there is more than one way to derive the sentence.

This means that our algorithm is **non-deterministic.**

EX: given the grammar:

E --> E + E

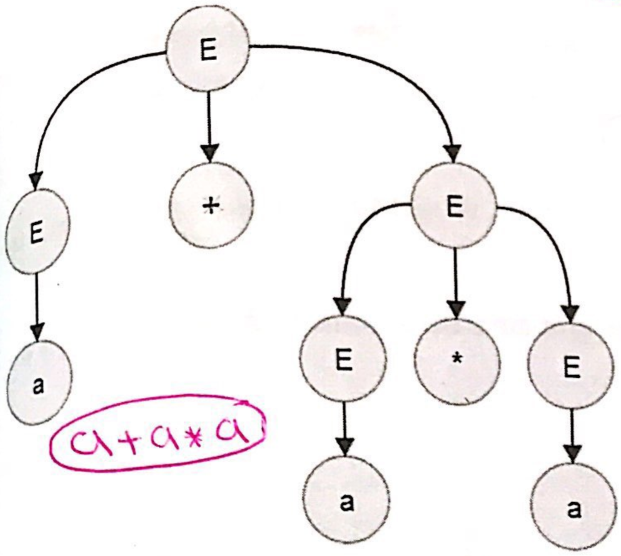
E --> E * E

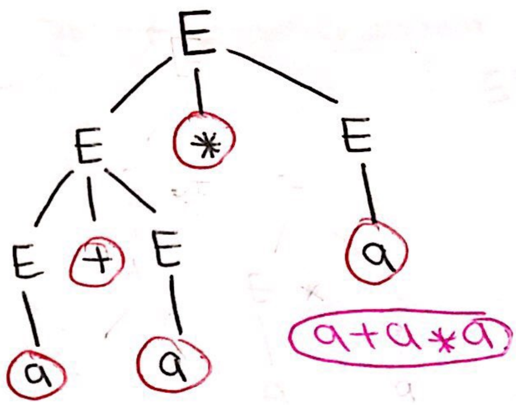
E --> (E) | a

Take the sentence : a + a * a

Let us draw the derivation tree

$E \to \underline{E}+E \to a+\underline{E} \to a+\underline{E}*E \to a+a*E \to \boxed{a+a*a} \in L(G)$





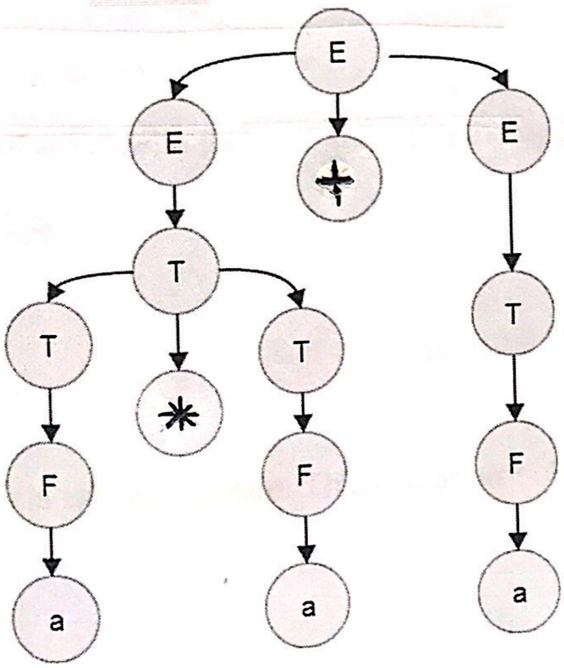
⚠ Same sentence, different derivation trees. "ambigious"

Due to the fact that we have 2 trees that give the same result, we can say that this grammar is ambiguous. In this case, to enforce the associativity rule, this grammar can be re-written as :

E--> E + E | T
T--> T*T | F
F--> (E) | a

_ambiguity.
-Precedency.
-associativity.

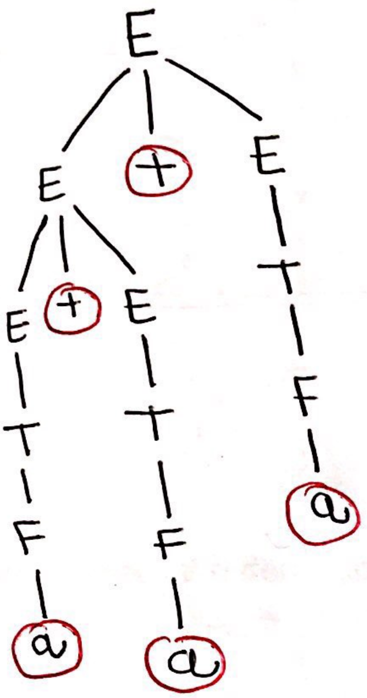
Now, Take the sentence a + a * a and find the derivation tree now.



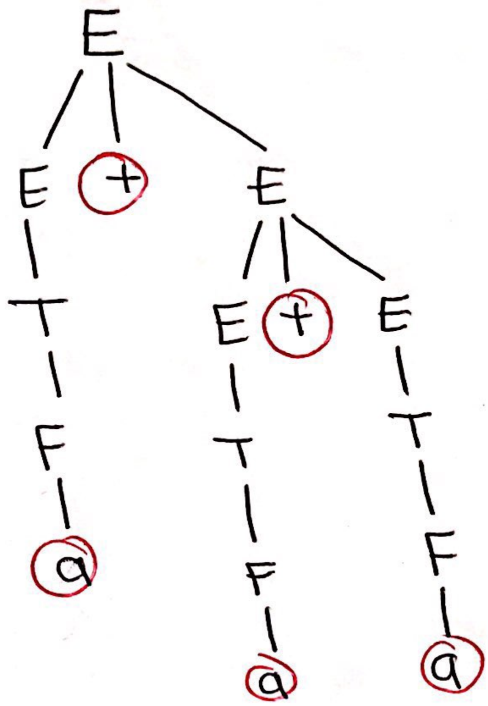
There is only possible derivation trees now. This solves the associativity problem with + and * of the grammar before with the operations.

But Let us say we have the sentence : a + a + a

Let us try to find the derivation tree and any alternative trees.



a+a+a



a+a+a

We can see here that there is more than 1 derivation tree, and the language is still ambiguous.

We can solve this if we rewrite the grammar with the **left-associative rule**

E --> E + T | T
T --> T * F | F
F --> (E) | a

The grammar now is left-associative. This grammar solves the problems of :

- ambiguity.
- precedence.
- associativity.

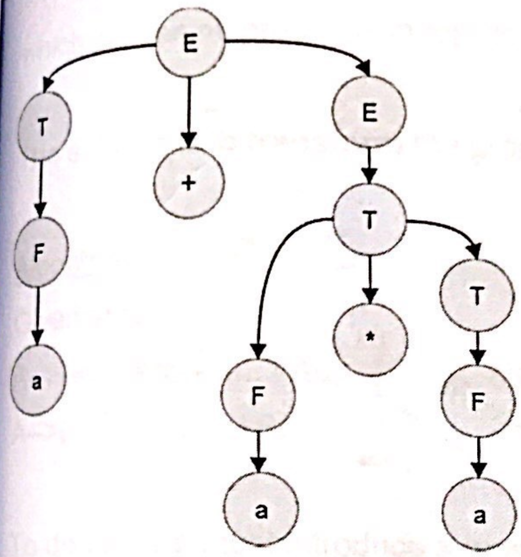
Let us try rewriting it with the **right-associative rule**

E --> T + E | T

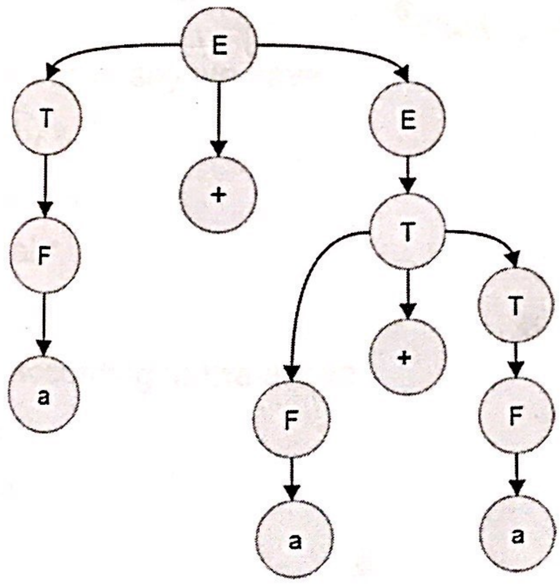
T --> F * T | F

F --> (E) | a

Let us try creating the derivation tree of a + a * a



Now let us check the derivation tree of a + a + a



This new grammar is not ambiguous, , however it does not solve the fact
That associativity issue according to our standard.

left-recursive Grammar

This causes problems when it comes to Top-down parsing techniques(see why later).

A grammar is said to be **left recursive** if there is a production of the form:

A-->Aα

Conversely, a grammar is right-recursive if there is a production of the form:

A-->αA

which causes no problems in top-down parsing.

The solution is to **transform** the grammar to a grammar which is not left-recursive.

Algorithm.

Given that:

$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \mid A\alpha_n$

$A \to \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_m$

Transform

To do this, we must introduce a new non-terminal, say A\`.

The grammar now becomes :

To

$A \to \beta_1 A\` \mid \beta_2 A\` \mid \ldots\ldots \mid \beta_m A\`$

$A\` \to \alpha_1 A\` \mid \alpha_2 A\` \mid \ldots\ldots \mid \alpha_n A\` \mid \lambda$

IMPORTANT!

- INPUT: Left-Recursive grammer.
- OUTPUT: An equivalent non Left-Recursive grammer

For example, say we have

A-->Ab → α_1

A-->a → β_1

L(G)=ab*

Then according to the above

A-->aA\`

A\`-->bA\` | λ

which results in the same grammar.

Let us apply this to the grammar :

E --> E + T | T   (α_1, β_1)   • Left_Recursion.

T --> T * F | F

F --> (E) | a

E → E+T|T → E → E (+T) → α_1
E → T → β_1

Then the new grammar :

E --> T E`

E` --> + T E` | λ

T --> F T`

T` --> * F T` | λ

F --> (E) | a

ألف ألف

This grammar is now **perfect**. It solves all our ambiguity issues, and this is a grammar we can use to construct the production rules for our programming language.

Another ambiguity in programming languages is the **if...else** statement.

stmt --> if-stmt | while-stmt | ....
if-stmt --> IF condition stmt
if-stmt --> IF condition stmt ELSE stmt
condition --> C
stmt --> S

RESERVED WORDS

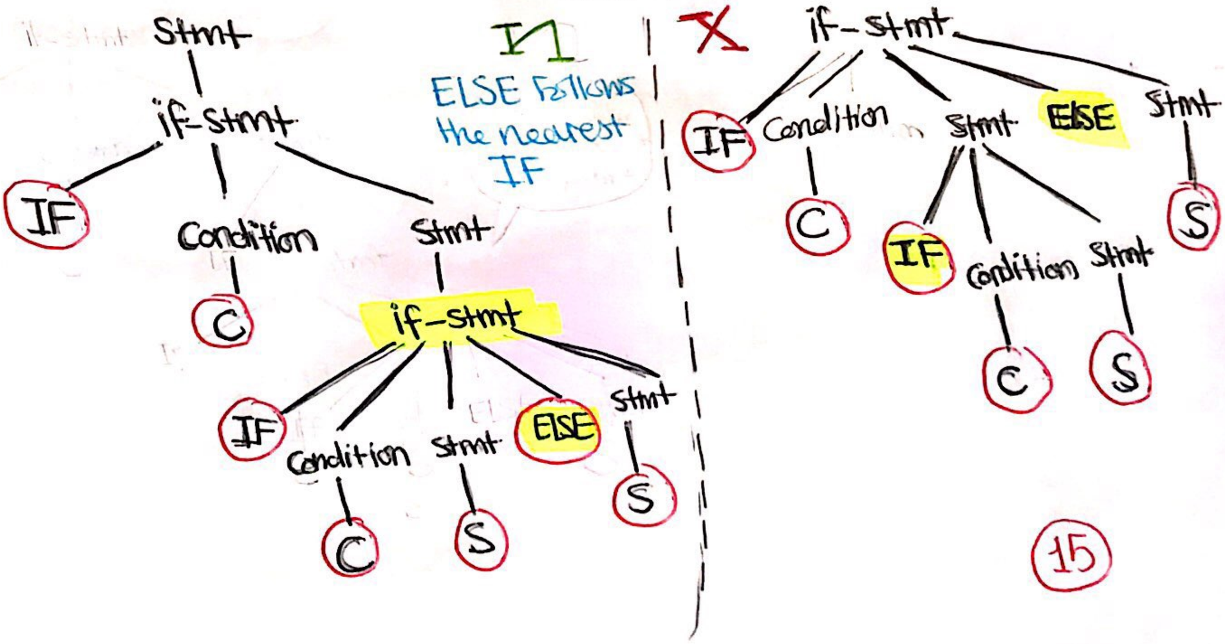
This grammar is **ambiguous**.

Let us take the following nested i**f...else** statement :

IF C
IF C
S
ELSE
S

This statement results in 2 derivations trees.

Draw the two derivation Trees



The first results in the ELSE belonging to the first IF , while the second results in the ELSE belonging to the second IF .

The second tree is the correct one since we know that the ELSE statement follows the nearest IF .

But how can the compiler behaves in this case?

There are a bunch of solutions to this problem:

1. Add a delimiter to the IF statement, such as ENDIF or END or FI to the end of the statement, resulting in these productions :

if-stmt --> IF condition stmt **ENDIF**

if-stmt --> IF condition stmt ELSE stmt **ENDIF**

Resulting in this statement :

```
IF C
   IF C
      S
   ELSE
      S
   ENDIF
ENDIF
```

The grammar is now unambiguous, since we have to clearly state whenan ```IF``` statement ends.

However, this is not a pretty solution, and is extra work for both the programmer and compiler, and result in less readable code.

2. Make the compiler **always** prefers to shift the ELSE when it sees the ELSE in the source code.

Left Factoring

Consider the productions :

A --> αβ

A --> αγ

Note how the **first part** of the productions is the same. This grammar can be transformed by introducing a new non-terminal B,

So what happens now is:

A --> αB

B --> β|γ

For our grammar, this results in

if-stmt --> IF condition stmt

if-stmt --> IF condition stmt ELSE stmt

becomes:

if-stmt --> IF conditon stmt else-part

else-part --> ELSE stmt | λ

Does this solve the ambiguity? No, but it helps in removing choices, since the if-stmt is now one production. If we look at the

statement :

IF C
IF C
S
ELSE
S

It still has 2 derivation trees

Extended BNF Notation

So far, we have been using **BNF Notation**(Production rules) to express languages. However, there is another form to

Express a language, which is **Extended BNF Notation**

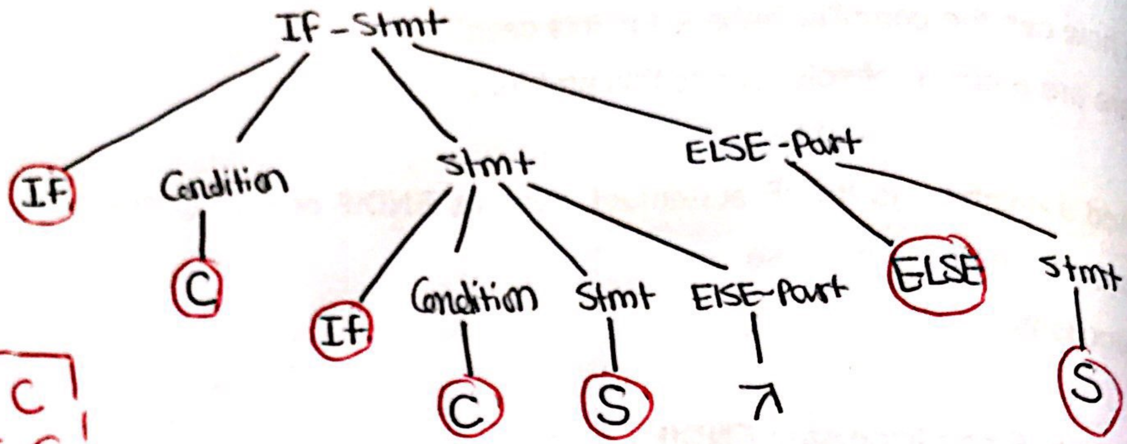
if there is repetition in the grammar, say in the example of the grammar

17

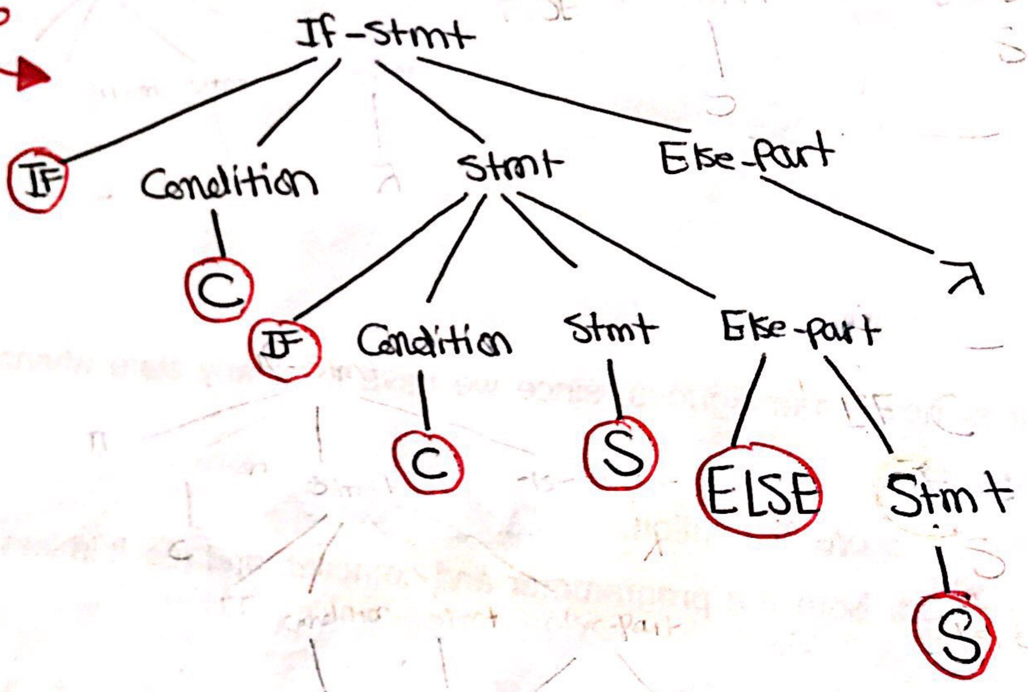
E --> E + T | T

T --> T * F | F

F --> (E) | a



IF C
IF C
S
ELSE
S



which can give us a derivation in the form of

E --> E + T --> E + T + T --> E + T + T + T --> T + T + T + T....+T

or in the same line,

T --> T* F --> T* F* F --> T* F*F* F --> T* F*F*F.... F

We can express this grammar as :

E --> T { + T }

T --> F { * F }

F --> (E) | a

We know that [x] means that we take x 0 or 1 time only.

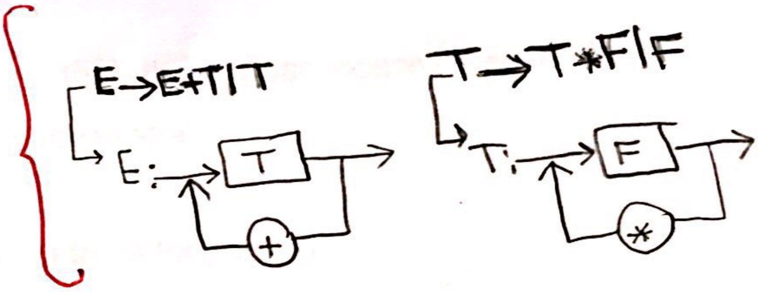
However, { x } means we take x zero or any number of times. This is equivalent to (x)*

We can also express this grammar as:

E --> T (+ T)*

T --> F (* F)*

F --> (E) | a



Syntax Diagrams

Another way to express languages are **Syntax Diagrams**. These are used only with Extended-BNF notation.

A square shape represents a nonterminal and an oval shape represents a terminal.

DRAW

18

Parsing Techniques

Recall : The parser is an algorithm which accepts or rejects a sentence in the programming language.

Recall : There are 2 kinds of parsers :

1. **Top-Down Parsers** : In This parsing technique, The parser starts with S using leftmost derivation to derive the sentence.

The **Major problem** with this parsing technique is that the parser doesn't know which production it should select in each derivation step.

2. **Bottom-Up Parsers** : The parser in this parsing technique starts from the sentence, doing reduction steps, until it reaches the starting symbol S of the grammar.

The **Major problem** with this technique is that the parser doesn't know which substring the parser should select in each reduction step.

In Top-Down parsing, we have 2 available algorithms for parsing :

1. Recursive Descent Parsing.
2. LL(1) Predictive Parsing.

* We can Build deterministic Parser for the grammar.
* Those subsets are powerful enough to define any Programming Language.

In Bottom-Up parsing, we have 2 available algorithms for parsing :

1. LR Parsers.
2. Operator Precedence Parsers --> Uses matrix manipulation.

* We can Build deterministic parser for the grammar.

Before we continue, we need to define a few functions

The FIRST() Function

Given a string $\alpha \in V^*$, then

zero or more derivations.

FIRST(α) = { a | α --*--> aw, a$\in V_T$, w$\in V^*$}

$V_T \cup V_N$ including λ.

in addition, if α --> λ, then we add λ to FIRST(α), that is $\lambda \in$ FIRST(α).

That is to say, FIRST(α) = Set of all terminals that may begin strings derived from α.

any String

For example

α --*--> cBx

α --*--> ayD

α --*--> ab

α -----> ddd

Then

FIRST(α) = {c,a,d}

Assume as well that

α --*--> λ

then

FIRST(α) = {c,a,d,λ}

That is to say, **λ appears in the FIRST() function.**

The FOLLOW() Function

We define the FOLLOW() function for **only** non-terminals. That is to say FOLLOW(A), A∈VN, then

Terminal. Terminals

FOLLOW(A) = { a | S --*-->(LM) uAβ, where a ∈FIRST(β)}

That is, S --*--> $\boxed{u A\beta}$, u∈VT*, A∈VN, β∈V* and FOLLOW(A)=FIRST(β)

Sentential form

That is to say, FOLLOW(A) = The set of all terminals that may appear after A in the any derivation.

any non-terminal.

⚠ λ : doesn't appear in the follow Function.

لازم يكون في إشي في ال follow

S --*--> aaXdd

S --*--> Xa

S --*--> BXc

Then

FOLLOW(X) = {d,a,c}

* TGP-DOWN
* Left-most derivation

S --*--> uAB

left most non-terminal

- every thing on the left is terminals

20

X)

Rules To Compute FIRST() and FOLLOW() Sets

1. FIRST(λ) = {λ}.
2. FIRST(a) = { a }. $a \in V_T$
3. FIRST(aα)= {a}. $a \in V_T$, $\alpha \in V^*$
4. FIRST(XY) = FIRST(FIRST(X).FIRST(Y)) **OR** preferred
   FIRST(X.FIRST(Y)) **OR**
   FIRST(FIRST(X).Y).
5. Given the production A --> αXβ, Then :
   a. FIRST(β) ⊂ FOLLOW(X) if β ≠ λ.
   b. FOLLOW(A) ⊂ FOLLOW(X) if β = λ.

Note that the FIRST() and FOLLOW() sets are made of **terminals only**

Notes :

1. λ **may** appear in FIRST() but it doesn't appear in FOLLOW(). We will see this when we define augmented grammars.
2. Generally, we start computing the FIRST() from bottom to top, But FOLLOW() from top to bottom.
3. When we compute FOLLOW(X), we search for X in the right side of any production.

Augmented Grammars

DEFENETION

Given the grammar G=(VN,VT,S,P), then the augmented grammar G`=(VN`,VT`,S`,P`) can be obtained from G as follows:

1. V_N` = $V_N$ U {S`}.
2. V_T` = V_T U { $}. ex Stoping Symbol
3. S` = new starting point.
4. P` = P U {S` -->S$}

For example :

E --> E + T | T

T --> T * F | F

F --> (E) | a

Becomes :

G --> E$

E --> E + T | T

T --> T * F | F

F --> (E) | a

This is because we want to create a FOLLOW() set for S.

Example 1 :

S` --> S$

S --> AB

A --> a | λ

B --> b | λ

Let us compute the FIRST() sets for this grammar :

FIRST(A) = {a,λ}

FIRST(B) = {b,λ}

FIRST(S) = FIRST(AB) = FIRST(FIRST(A).FIRST(B))

= FIRST({a,λ},{b,λ})

= FIRST({ab, a, b, λ})

= {a,b,λ}

FIRST(S`) = FIRST(S$) = FIRST(FIRST(S).FIRST($))

= FIRST({a,b,λ}.$)= FIRST(a$,b$,$)

= {a,b,$}

⚠ we don't Calculate Follow(S') ?!

Now Let us compute the FOLLOW() sets for this grammar :

FOLLOW(S) = {$}

FOLLOW(A) = {b,$}

FOLLOW(B) = {$}

Follow(A) = First(B) "B ≠ λ" → Rules Page #21

First(b) = {b, λ}

→ λ doesn't appear in Follow()

Follow(A) = First(B) U Follow(S) " First(B) Contains λ" → Rules Page #21

= {b} U {$}

= {b, $}

Example 2 :

S` --> S$

S --> aAcb

S --> Abc

A --> b | c | λ

Let us take the FIRST() for this grammar :

FIRST(A) = {b,c,λ}

FIRST(S) = FIRST(aAcb)∪FIRST(Abc) = {a,} ∪{b,c}

= {a,b,c}

FIRST(S`) = FIRST(S$) = FIRST(FIRST(S).FIRST($))

= FIRST({a,b,c}.{$})

= {a,b,c}

Now let us take the FOLLOW() :

FOLLOW(S) = {$}

FOLLOW(A)= {c,b}

Example 3:

G --> E$

E --> E + T | T

T --> T * F | F

F --> (E) | a

Let us calculate FIRST() :

FIRST(F) = {(,a}

FIRST(T) = FIRST(T*F)∪FIRST(F) = FIRST(T*F)∪{(,a}

= {(,a} (Because every T will eventually become an F)

FIRST(E) = FIRST(E + T) ∪ FIRST(T) = {(,a} ∪ {(,a}

= {(,a}

FIRST(G) = FIRST(E$) = {(,a}

Now let us Calculate FOLLOW() :

FOLLOW(E) = {$,+,)}

FOLLOW(T) = FOLLOW(E) ∪ {*} = {$,+,*,)}

FOLLOW(F) = FOLLOW(T) = {$,+,*,)}

But what makes all this so important?

Well, All of the parsing techniques we are going to depend will heavily on FIRST() and FOLLOW().

FIRST({a, b, c} ∪ {\$}) = {a, b, c}

↳ since this doesn't contain λ
we ignore \$

FIRST({a, b, c, λ} ∪ {\$}) = {a, b, c, \$}

↳ λ exists here so we
take \$ in concederation

Recursive Descent Parsing

This parsing technique is better with Extended BNF

Recursive Descent Parsing is very simple. It works like this :

Divide the grammar into primitive/simple components

1- For the token "a" :

```
If(token == "a")
        get-next()
else
        report-error()
```

⚠ Disadvantage: Slow Because it depends on function calling.

2- For $X = \alpha_1\ \alpha_2 \ldots \alpha_n$:

$X \rightarrow \alpha_1\ \alpha_2 - \alpha_n$

Code(X):

```
{       Code(α1);
        Code(α2);
        .
        .
        Code(αn);
}
```

3- For $X = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, If none of the α_i's = λ

$X \rightarrow \alpha_1 | \alpha_2 | - | \alpha_n$

Code(X):

```
{       If (token ϵ FIRST(α1))
            Code(α1);
        Else
                If (token ϵ FIRST(α2))
                    Code(α2);
                Else
        .
        .
                Else
                        If (token ϵ FIRST(αn))
                            Code(αn);
                        Else
                            Report-error();
}
```

TOKEN DOESN'T EXIST

SYNTAX ERROR!

4- For $X = \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n = \lambda$, If one of the α_i's = λ, say $\alpha_n = \lambda$

$X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n = \lambda$

Code(X):

```
{       If (token ϵ FIRST(α1))
            Code(α1);
        Else
                If (token ϵ FIRST(α2))
                    Code(α2);
                Else
        .
        .
                Else
                        If (token ϵ FIRST(αn-1))
                                Code(αn-1);
                        Else
                                If (token  is not ϵ FOLLOW(X))

                                        Report-error();
}
```

5- For $X = \alpha^*$

Code(X):

```
While (token ϵ FIRST(α))
        Code(α);
```

Notes :

1. Every nonterminal has a code(a function).
2. S` in augmented grammar is represented by the function "main".
3. We only start with calling "get-token" in function "main".

Example:

G --> E$

E --> T(+ T)*

T --> F(* F)*

F --> (E) | a

```
main(){        //represents G
get-token;  only in the main.
call E();
if(token!="$")
      Error;
else

}

function E(){                    // E → T (+ T)*
call T();
while(token == "+"){
get-token();
call T()
}
}
```

⚠ Here we don't need get token Because we already have a token.

While(token ∈ First(α))
α = +T   // First(α) = +

```
function T(){        //T--> F (* F)*
call F();
while(token == "*"){
get-token();
call F();
}
}

function F(){        //F--> ( E ) | a
if(token == "(")
{
      get-token();
      call E();
      if(token == ")")
            get-token();
      else
            ERROR;
}
Else
      if(token=="a")
            get-token();
      else
            ERROR;
}
```

Note that ERROR is a function we should write.

Example:

Given the grammar :
Program --> body .
body --> Begin stmt (; stmt)* End
stmt --> Read | Write | body | λ

and
VN = { Program, body, stmt, block}
VT = { ., Begin, ;, End, Read, Write}

__examples__ of programs of this language would be:

```
Begin
    Read;
    Write;
    Read;
    Write;
End.
```

Or

```
Begin
    Read;
End.
```

Or

```
Begin
    Read;
    Begin
        Read;
        Write;
    End.   $/Stop Symbol.
    Write;
End.
```

Or

```
Begin;
;
;
;
;
End.
```

Let us write the <u>recursive descent code</u> for this programming language.

```
main(){
get-token();
call body();
if(token != "."){
        ERROR;

}
Else  {
        SUCCESS;
        }

}

function body()
{ if(token == "Begin")
        {get-token();
        call stmt();
        while(token == ";")
                { get-token();
                  call stmt();
                }
        if(token == "End")
                get-token();
        else
                ERROR;
        }
else
        ERROR;
}

function stmt()
{
if(token == "Read")
        get-token();
else  if (token == "Write")
                get-token();
        else if(token == "Begin")
                        call body();
                else
                        if(token != ";" && token != "End" )
                                ERROR();

}
```

LL(1) Parsing

This Parsing method is a **table-driven** parsing method. The LL(1) parsing table selects which production to choose for the next derivation step.

Formal Definition of LL(1)

The Formal definition of LL(1) grammars is given by :

Given the Productions :

A --> α_1

A --> α_2

A --> α_3

.

.

A --> α_n

then the grammar is LL(1) if :

1. FIRST(α_i)∩FIRST(α_J) = Ø for all i,j
2. if one of α_i is λ,α_n = λ, in addition to 1

   FIRST(α_i)∩follow(A) = Ø , for , ∀ i < n

For example, Given the grammar :

S` --> S$

S --> aABC

A --> a | bbD

B --> a | λ

C --> b | λ

D --> ~~C~~ c | λ

let us see if it is LL(1)

FIRST(a)∩FIRST(bbD) = Ø

FIRST(a)∩FOLLOW(B) = Ø

FIRST(b)∩FOLLOW(C) = Ø

FIRST(c)∩FOLLOW(D) = Ø

Then this grammar is LL(1).

en another Grammar :

-> S$

-> aAa | λ

-> abS | λ

ST(aAa)∩FOLLOW(S) = {a}∩{$,a} = {a} ≠ ∅

s grammar is not LL(1).

L(1) Parsing Table Building Algorithm

t us assume that we have a grammar that is LL(1). How do we build the LL(1) parsing ble?

1. For each production A --> α in the grammar G,
   Add to the table entry T[A,a] the production A --> α, where a ∈ FIRST(α)
   If λ ∈ FIRST(α), Add to the table entry T[A,b] the production A --> α ,
   ∀ b ∈ FOLLOW(A).
2. All Remaining Entries are Error Entries.

r **example**, given the grammar :

-CGF-

-> SR $$_1$

-> +$_2$ | -$_3$ | λ4

-> dN.N$_5$ | .dN$_6$

-> dN$_7$| λ$_8$

L(G) = [+|-] { ddd.ddd / ddd. / .ddd }

ote that the superscript denotes the production number.

FIRST(SR $) = {+,-,d,.}

FIRST(+) = {+}

FOLLOW(S) = {d,.}

FIRST(R) = {d, .}

FIRST(d) = { d }

FOLLOW(N) = { ., $}

First(SR$) = First(First(S) . First(R) . {$})
= First({+, -, λ} . {d, .} . {$})
= ({+d, -d, d, +., -., .} . {$})
= {+, -, d, .}

VN \ VT	+	-	d	.	$
V	1	−1	1	1	
S	2	3	4	4	
R			5	6	
N			7	8	8

There should be no conflict(multiple entries) in the LL(1) table.
L(G) of this grammar = all floating point numbers.

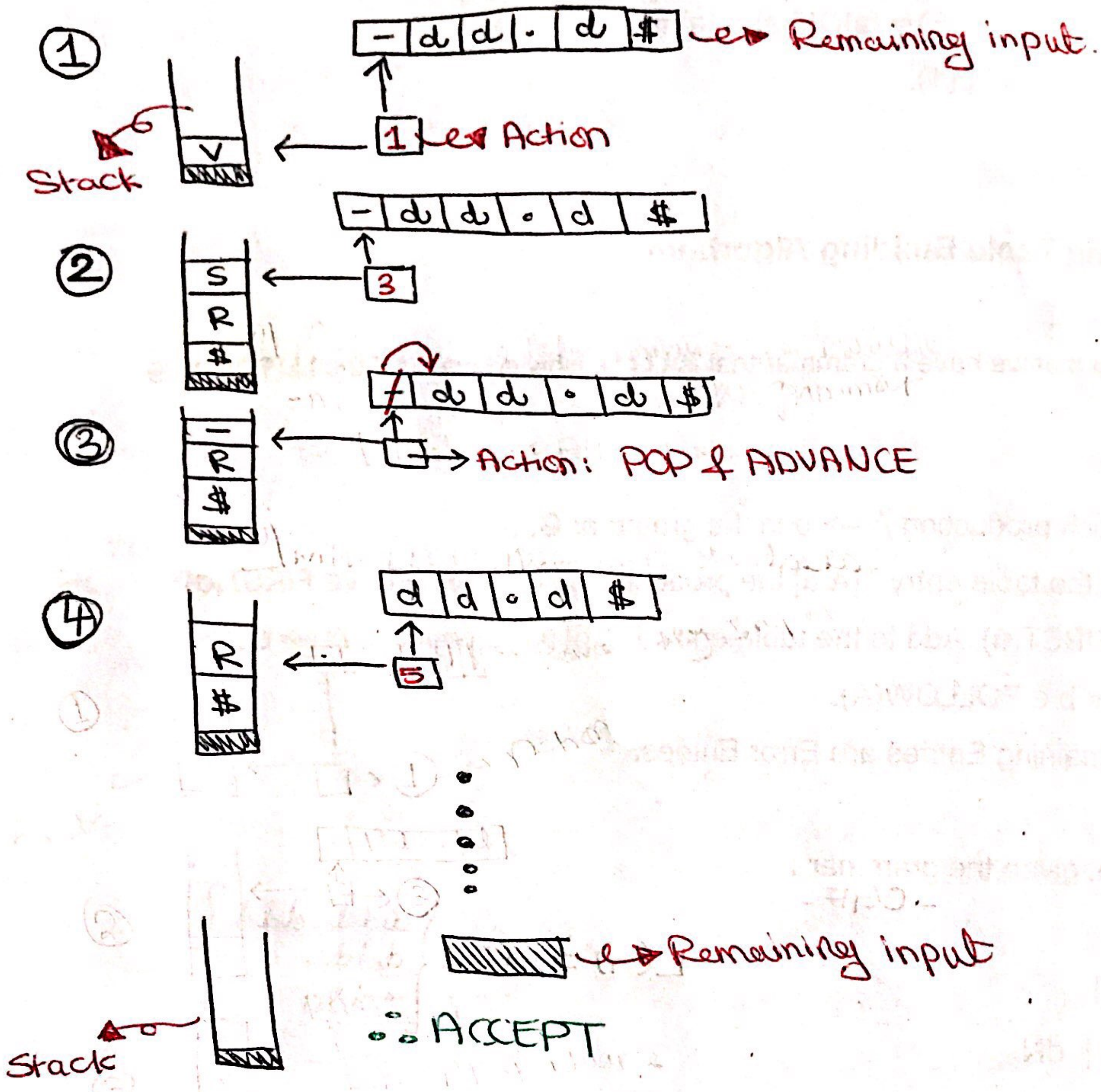
The parser works like this

Stack	Remaining Input	Action
V	-dd.d$	Production 1
SR$	-dd.d$	Production 3
-R$	-dd.d$	Pop & advance input
R$	dd.d$	Production 5
dN.N$	dd.d$	Pop & advance input
N.N$	d.d$	Production 7
dN.N$	d.d$	Pop & advance input
N.N$	.d$	Production 8
.N$	.d$	Pop & advance input
N$	d$	Production 7
dN$	d$	Pop & advance
N$	$	Production 8
$	$	Pop and Advance
λ	λ	Accept

If at any point the parser reaches a place where the input and the stack have 2 different terminal symbols, it throws a syntax error.

⚠ If the parsing table contains no multiple enteries (Conflicts) then the grammar is **LL(1) Grammar.**

⚠ With LL(1) there are **NO** deadlocks.



et us Take another example. Let the Grammar be :

rogram --> block $ $_1$

lock --> { decls ; stmnts } $_2$

ecls --> D ; decls $_3$ | λ $_4$

tmnts --> statement ; stmts $_5$ | λ $_6$

tatement --> if $_7$ | while $_8$ | ass $_9$ | scan $_{10}$ | print$_{11}$ | block $_{12}$ | λ$_{13}$

VT = {$,{,},D,;,if,while,ass,scan,print}

First(block) = { '{' }
First(program) = { '{' }

V N \ V T	if	while	ass	scan	print	{	}	D	;	$
Program						1				
block						2				
decls	4	4	4	4	4	4	4	3	4	
stmts	5	5	5	5	5	5	6		5	
statement	7	8	9	10	11	2			13	

Another example is the If..else statement with a delimiter. the grammar looks like this :

S' --> S$

S --> iCSE

E --> eS | λ

S --> a ELSE

C --> c

V N \ V T	i	a	e	c	$
S `	1	1			
S	2	5			
E			3,4 !		4
C				6	

⚠ OUR IF/ELSE GRAMMAR ISN'T LL(1)

There is a conflict. To solve this, we can add a delimiter.

S` --> S$

S --> iCSEd

E --> eS | λ

S --> a

C --> c

VN \ VT	i	a	e	c	d	$
S `	1	1				
S	2	5				
E			3		4	
C				6		

The grammar is now unambiguous.

Alternatively, we can just remove out the production 4 in the conflict entry from the LL(1) table.

The New table is:

VN \ VT	i	a	e	c	d	$
S `	1	1				
S	2	5				
E			3		4	
C				6		

Note:

- if a grammar is LL(1), then it is unambiguous. However, the opposite is not necessarily true.
- Another thing to note is that in Top-Down parsing, we should avoid a grammar that is not LL(1).

Bottom-Up Parsing

Recall that in Bottom-Up parsing, the parser starts from the given sentence, applying reductions until it reaches the starting symbol of the grammar or a deadlock.

The **major problem** with Bottom-Up parsing is which substring we should select in each reduction step.

The **answer** to the above question is :
In each reduction step, we select what is called **the handle.** Solution for the Problem of bottom-UP Parsing.

The Handle is obtained by a **rightmost** derivation **in reverse.**

For **example,** Given the grammar :

V --> S R $
S --> +|-|λ
R --> .dN | dN.N
N --> dN | λ

and the sentence
-dd.d$

First, we derive the sentence rightmost.
V --rm--> SR$ --rm--> SdN.N$ --rm--> SdN.dN$ --rm--> SdN.dd$ --rm--> SddN.d$ --rm--> Sdd.d$ --rm--> -dd.d$

So our handles would be :

V <-- SR$ <-- SdN.N$<-- SdN.dN$ <-- SdN.dλ$ <-- SddN.d$ <-- Sddλ.d$ <-- -dd.d$

But Compilers do not work like this. We already derived the sentence, why would we go back and do it again?

We could not build a Bottom-Up parser for every Context-Free Grammar. However, we are fortunate enough that there exist subsets of the Context-Free Grammar for which we can build a deterministic Bottom-Up parser i.e. the parser can determine/decide precisely where the handle is in each reduction step.
Some of these subsets are:

LR Parsers :

- SLR(Simple-LR).
- LALR(Look-Ahead LR).
- LR.

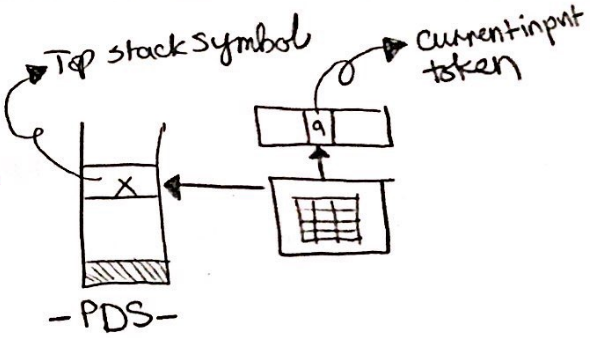
Operator Precedence. "depends on matrix manipulation"

We will only be talking about the LR parsers, just to get an idea of how Bottom-Up parsing works.

SLR Parsing

SLR parsing, and LR parsing in general, is a table driven parsing method.
All LL(1) grammars are a subset of SLR grammars.

All LR parsers contains :



1. A parsing table.
2. A stack. → Push Down Stack.
3. The input string.

As a reminder, the LL(1) parser contains :

1. A parsing table.
2. A stack.
3. The input string.

⚠ Question! How we build the LR parsing table?!

However, the way we build it is different.

Building the SLR Parsing Table

Def: An LR(0) item of a grammar G is a production in G with a dot(.) at some position in the right side.

For example, the production

A --> aBY

This production generates the following LR(0) items :

A --> .aBY

A --> a.BY

already had seen a string derived from aB

$A \rightarrow aB.Y$

A --> aB.Y

expect to see on input a string derived from Y.

A --> aBY. --> complete item

Note that for A --> λ, this generates only A --> λ. .

Generally speaking, if the right side of the production is of length n, then there are n+1 LR(0) items.

For $A \rightarrow \alpha$, $\alpha \neq \lambda$

$|\alpha| = n$,

LR(0) generates n+1 items.

The LR(0) item

A --> aB.Y

Means that the parser has scanned on the input a string derived from aB and expects to see a string derived from Y .

We need to define the following 2 functions.

The CLOSURE function

function CLOSURE(I) **//I is a set of LR(0)items**

Non-Terminal

{ Repeat $A \rightarrow \alpha.B\beta$

For (every LR(0) item **in I, and for every production B--> δ in G,**

Add the LR(0) item B-->.δ to I)

Until no more items to be added;

}

Let us apply this to our grammar :

(0) E` --> E$

(1) E --> E+T

(2) E --> T

(3) T --> T*F

(4) T --> F

(5) F --> (E)

(6) F --> a

⚠ This grammar isn't LL(1)! because there will be conflict.

This grammar is not LL(1) because

FIRST(T*F)∩FIRST(F) = {(,a} ≠ ∅

We will need to build the **LR(0) sets of items.**

we start with : I_0: E` --> .E

CLOSURE(I_0)

I_0: E`-->.E

E-->.E+T

E-->.T

T--> .T*F

T-->.F

F-->.(E)

F-->.a

The **GOTO function**

function GOTO(I,X) =

CLOSURE(all items A-->αX.β Where A-->α.Xβ in I)

$\bar{\bar{E}}$ Σ V$_N$

Let us apply this to the grammar above. groups:

I1 : E-->.E, E-->.E+T

I2 : E-->.T, T--> .T*F

I3 : T-->.F

I4 : F-->.(E)

	a	(	+	x	)	$
E`	0	0				
E	1,2	1,2				
T	3,4	3,4				
F	6	5				

⚠ It's Not LL(1) Because there are confilicts

I5 :F-->.a

and take the CLOSURE for all these sets. The resultant is :

I0:
E`-->E
(1) E-->.E+T
(2) E-->.T
(3) T--> .T*F
(4) T-->.F
(5) F-->.(E)
(6) F-->.a

since they are the same they go to a new state.

I1:
E`-->E.
E-->E.+T
Complete

I2:
E-->T.
T--> T.*F
Complete

I3:
T-->F.
Complete

I4:
F-->(.E)
E -->.E+T
E-->.T
E-->.T*F
T-->.F
F-->.(E)
F-->.a

I8

I2

I5:
F-->.a

I6:
E-->E+.T
E-->.T*F
T-->.F
F-->.(E)
F-->.a

I9

I7:
E-->T*.F
F-->.(E)
F-->.a

I8:
F-->(E.)
E-->E.+T

I9:
E --> E+T.
T --> T.*F

I10:
T --> T*F

I11:
F --> (E)

LR(0) Sets of items
{I0, I1, I2, ..., I11}

I_0: $E' \to \boxed{\cdot E}$ I_1

① $E \to \boxed{\cdot E}+T$ I_1

② $E \to \boxed{\cdot T}$ I_2

③ $T \to \boxed{\cdot T}*F$ I_2

④ $T \to \cdot F$ I_3

⑤ $F \to \cdot (E)$ I_4

⑥ $F \to \cdot a$ I_5

I_1: $E' \to E \cdot$ Complete
$E \to E \cdot +T$ I_6

I_2: $E \to T \cdot$ Complete
$T \to T \cdot *F$ I_7

I_3: $T \to F \cdot$ Complete

I_4: $F \to (\cdot E)$ I_8
$E \to \cdot E+T$ I_8
$E \to \cdot T$ I_2
$T \to \cdot T*F$ I_2
$T \to \cdot F$ I_3
$F \to \cdot (E)$ I_4
$F \to \cdot (a)$ I_5

I_5: $F \to a \cdot$ Complete

I_6: $E \to E+\cdot T$ I_9
$T \to \cdot T*F$ I_9
$T \to \cdot F$ I_3
$F \to \cdot (E)$ I_4
$F \to \cdot a$ I_5

I_7: $T \to T*\cdot F$ I_{10}
$F \to \cdot (E)$ I_4
$F \to \cdot a$ I_5

I_8: $F \to (E \cdot)$ I_{11}
$F \to E \cdot +T$ I_6

I_9: $E \to E+T \cdot$ Complete
$T \to T \cdot *F$ I_7

I_{10}: $T \to T*F \cdot$ Complete

I_{11}: $F \to (E) \cdot$ Complete

LR(0) Sets of items

$\{I_0, I_1, I_2, \ldots, I_{11}\}$

Constructing the SLR table

Input : LR(0) sets of items

Output : SLR(1) parsing table

1. For every item A-->α.aB in I_i, a ∈ V_T , and GOTO(I_i,a)=I_j, then set: ACTION[i,a]=Sj(shift and push j on top stack).
2. For item A-->α.(complete item) in I_i, ACTION[i,b]=Reduce by A-->α FOR ALL b ∈ FOLLOW(A).
   Terminals.
3. For S` --> S. in Ii, ACTION[i,$] = Accept.
4. If GOTO(Ii,A) = Ij then set, GOTO(i,A) = j.
5. All remaining entries are error entries.

Let us apply this to the example above and generate the table

Follow (E)={+,), $}
Follow (T)={+,*,), $}
Follow(F)={+,*,), $}

ACTION [V_T] GOTO [V_N]

State / token	a	+	*	(	)	$	E	T	F
0	S5			S4 shift & goto 4			1	2	3
1		S6				A accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1		S7	R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Production number.

action section
→ . a → shift & go to

go-to section
→ . A → go to

No conflict --> SLR(1) grammar

Parsing The SLR Table

E --> E+T
E --> T
T --> T*F
T --> F
F --> (E)
F --> a

Let us examine the sentence a + a $

Stack	Remaining	Action	
0	a + a $	S5	
0 a 5 H	+ a $	R6	R: F→a
0 F 3 بشيل 0 بحط F H	+ a $	R4	R: T→F
0 T 2 H	+ a $	R2	
0 E 1	+ a $	S6	
0 E 1 + 6	a $	S5	
0 E 1 + 6 a 5 H	$	R6	
0 E 1 + 6 F 3 H	$	R4	
0 E 1 + 6 T 9 H	$	R1	
0 E 1	$	Accept	

H : Handles

LR Parsing Techniques

→ Definition

The main difference between LR and SLR is the CLOSURE function,

function CLOSURE(I) //I is a set of LR(1)items
{ Repeat
for(every LR(1) item [A-->α.Bβ, a] in I, and for every production B--> δ in G,
Add the LR(1)item [B-->. δ, b] where b belongs to FIRST(βa)to I)
Until no more items to be added;

}

Where An LR(1) item is an LR(0) item with a **Look-ahead Symbol.**

For example [A --> α.**β** , a] where a is the look-ahead. The look-ahead symbol "a" has no effect whatsoever on an item **[A --> α. β,a] β ≠ λ.**

(not complete item) However, if the item is a complete [A --> α.,a], this means we reduce by the production A --> α on token "a".

For example, Give the grammar :

S` --> S
(1) S --> CC
(2) C --> cC
(3) C --> d

I_0:

S` --> .S,	$	I_1
S --> .CC	$	I_2
C --> .cC	c,d	I_3
C --> .d	c,d	I_4

First (C$)

A → α.Bβ a → S` → .S $
B → .δ S → .CC

β = λ

First (β$) = First(λ$) = {$}

I1:

S` --> S.	$	Accept

I2:

→ λ

S --> C.Cβ	$	I_5
C --> .cC	$	I_6
C --> .d	$	I_7

I3:

C --> c.C	c,d	I_8
C --> .cC	c,d	I_3
C --> .d	c,d	I_4

Definition:

An LR(1) item is an LR(0) item with look Ahead symbol (token). that's, $A \rightarrow \alpha . \beta$ is an LR(0) item, then LR(1) item is : $[A \rightarrow \alpha . \beta \, , a]$, $a \in V_T$

* We start building the LR(1) sets of items,
We start with LR(1) item $I_0 : [S' \rightarrow . S \, , \$]$

look-ahead

$[A \rightarrow \alpha . B \beta \, , a]$

* Add

$[B \rightarrow . \gamma \, , b]$ $b \in First(\beta a)$

* look-Ahead has no effect if the item isn't complete.
* look-Ahead affects the parser if the item is complete.

I4: $\boxed{C \to d.}$ c,d Complete

I5: S --> CC. $ Complete

I6:

C --> c.C	$	I_9
C --> .cC	$	I_6
C --> .d	$	I_7

I7: $\boxed{C \to d.}$ $ Complete

I8: $\boxed{C \to cC.}$ c,d Complete

I9: $\boxed{C \to cC.}$ $ Complete

ACTION V_T GOTO V_N

V	c	d	$	S	C
0	S3	S4		1	2
1			A		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

No conflict, the grammar is an LR grammar. (1)

multiple enteries

If we look at the above example, we can see that some sets of items have the same core items(LR(0) items), but the look-ahead is different.

For example (I7 , I4), (I3 , I6), (I8 , I9). Let us say we merge the states.

	ACTION			GOTO	
V	c	d	$	S	C
0	S3	S4		1	2
1			A		
2	S3	S4			5
3	S3	S4			8
4	R3	R3	R3		
5			R1		
8	R2	R2	R2		

⚠ Some of the LR(1) sets of items, the core items are identical.

- I_3, I_6
- I_4, I_7
- I_8, I_9

We merge these sets of Items.

This is now a simplified table. if the parsing table after merging has no conflicts(like in the above example), then the grammar is an LALR(1) Grammar.

Conflict Types:

given S(shift) - R(Reduce)

→ Possibilities of Conflicts :-

(1) SS (shift-shift)

In one of the LR(0) sets of items.

I_1: ⋮

$A \to \alpha \boxed{.a}\beta$ I_6

$B \to \gamma \boxed{.a}\delta$ I_5

⚠ Impossible to happen

since .a both should go to the same state.

∴ S.S conflict Impossible.

(2) SR (shift - Reduce)

I_i: ⋮

$A \to \alpha . a \beta$ I_j

$B \to \gamma .$ a

	a
⋮	
i	s_j $R_{B\to\gamma}$

(3) RR (Reduce - Reduce)

I_i: ⋮

$A \to \alpha .$ a   c

$B \to \gamma .$ a   c

	a
⋮	
i	$R_{A\to\alpha}$ $R_{B\to\gamma}$

⚠ In LALR parssing table there's only Reduce - Reduce Conflict.

- Assume the grammar is LR(1) & there's a Shift - Reduce Conflict in the LALR parssing table.

Since there's a S-R conflict in the LALR(1), this means, in one of the LALR(1) sets of items there are:

I_i : ⋮

$A \rightarrow \alpha . a \beta \quad c \quad I_j$

$B \rightarrow \gamma . \quad a \quad c$

	a
⋮	
i	S_j $R_{B \rightarrow \gamma}$

But the items $A \rightarrow \alpha . a \beta \quad c$

$B \rightarrow \gamma . \quad a$

Came from the LR(1) parssing table so it's not LR(1)

∴ this contradicts our assumption.