SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN::BHIMAVARAM

DEPARTMENT OF INFORMATION TECHNOLOGY

Automata and Compiler Design Lecture Notes

UNIT-1
FORMAL LANGUAGE AND REGULAR EXPRESSIONS

Alphabets

An Alphabet is a finite, non empty set of symbols. It is denoted by S. The symbols are called the letters of the alphabet.

Examples

1)S={0,1}, the binary alphabet

1 The ASCII alphabet, the set of ASCII characters

2 S={a,b,c,…z}

3 S={0,1,a,b,c}

Strings

A string over an alphabet S is a finite sequence of symbols from alphabet S

Examples

1) Binary alphabet S={0,1}

Strings are 0111100,111,000,…. Etc

2) S= {a,b,c,…..z}

aa,bb,afasdsasd, are the strings of the alphabet

Languages and Operations on Languages

A language L over the alphabet ∈ is a subset ∈* . That is, a language is a set of strings over the given alphabet.

Note1: A language can be empty L =ϕ

Note2. Empty language is not equal to L={€}

Operations:

Union: Union is the simplest operation on two languages. If L1 and L2 are two languages, then union, denoted by L1 U L2 is a language containing all strings(w) from both the languages .

Concatenation: The concatenation AB of languages A and B is defined by,

$AB=\{uv| u\in A, v\in B)$

Kleene/Star closure: This operation defines on a set S a derived set S* , having as member, the empty string and all strings formed by concatenating a finite number of strings in S or alternatively.

$S^*=S^0 \cup S^1 \cup S^2$

Where $S^0=\epsilon$

Positive closure: The positive closure of a language is defined as

$S^+ = S^0 \cup S^1 \cup S^2$

Formal Definition of Computation

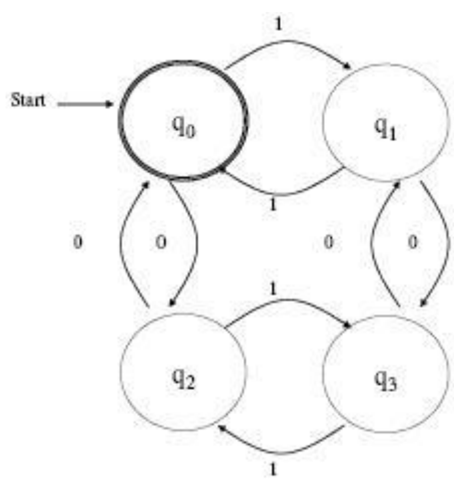
• Let M = (Q, ∑,∂, q0, F) be a finite automaton and let w = w1w2…wn be a string where each wi is a member of alphabet ∑

• M accepts w if a sequence of states r0r1…rn in Q exists with three conditions:

1. r0 = q0
2. ∂ (ri, wi+1) = ri+1 for i=0, … , n-1
3. r ϵ F

We say that M recognizes language A if A = {w | M accepts w }

In other words, the language is all of those strings that are accepted by the finite automata.



The above figure gives the examples of the DFA

Here is a DFA for the language that is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even:

Acceptance of Strings and languages by Finite Automata

A string X is accepted by a finite automata M=(Q,∑,∂,qo,F) if ∂(qo,x) = q for some q ∈ F. This is basically acceptability of a string by the final state.

DFA starts with initial state qo, Let a1,a2,……an is a sequence of input symbols and ∂ is transition function.

Step1: ∂(qo,a) = q1, DFA in Qo on input a , enters state q1

Step2; ∂(q1,a2)= q2, DFA in state q1, on input a2, enters state q2

Similarly ∂(qi-1,ai) = qi for each i.

Here Q={q0,q1,q2,q3}

∑={0,1}

F={q0}

Input string 110101

∂(q0,110101) -> ∂(q1,10101)

∂(q0,0101)

∂(q2,101)

∂(q3,01)

∂(q1,1)

∂(q0,ε)

Qo is a accepting state

1 1 0 1 0 1

Hence q0------->q1----------q0 --------->q2 ----------> q3 ------------>q1 ------------q0

A *deterministic finite automaton* (or DFA) is a deterministic automaton with a finite input alphabet and a finite number of states. It can be formally defined as a 5-tuple (Q, ∑,∂, q0,

4. , where

Q is a non-empty finite set of *states*,

∑ is the alphabet (defining what set of input strings the automaton operates on),

∂ is the *transition function*,

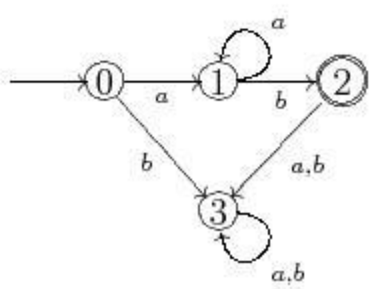
*q*0 is the starting state, and

F is a set of final (or accepting states).

A DFA works exactly like a general automaton: operation begins at $q0$, and movement from state to state is governed by the transition function . A word is accepted exactly when a final state is reached upon reading the last (rightmost) symbol of the word.

DFAs represent regular languages, and can be used to test whether any string in is in the language it represents. Consider the following regular language over the alphabet

This language can be represented by the DFA with the following state diagram:

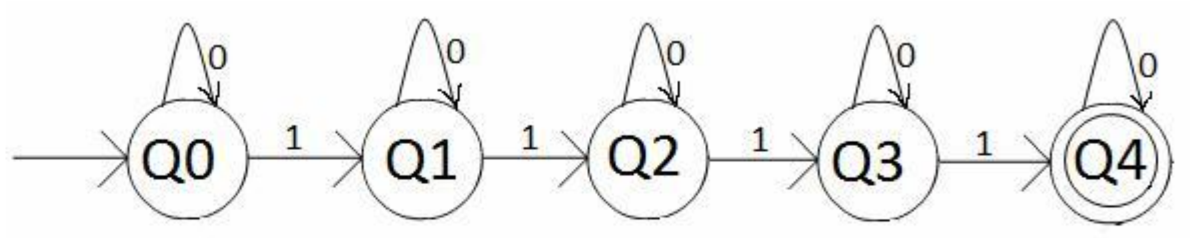


The vertex 0 is the initial state $q0$, and the vertex 2 is the only state in F . Note that for every vertex there is an edge leading away from it with a label for each symbol in . This is a requirement of DFAs, which guarantees that operation is well-defined for any finite string.

If given the string aaab as input, operation of the DFA above is as follows. The first a is removed from the input string, so the edge from 0 to 1 is followed. The resulting input string is aab. For each of the next two as, the edge is followed from 1 to itself. Finally, b is read from the input string and the edge from 1 to 2 is followed. Since the input string is now , the operation of the DFA halts. Since it has halted in the accepting state 2, the string aaab is accepted as a sentence in the regular language implemented by this DFA.

Now let us trace operation on the string aaaba. Execution is as above, until state 2 is reached with a remaining in the input string. The edge from 2 to 3 is then followed and the operation of the DFA halts. Since 3 is not an accepting state for this DFA, aaaba is *not* accepted.

Design finite automata which accepts set of strings containing four 1's in every string over alphabet {0,1}



Q={Q0,Q1,Q2,Q3,Q4}

E={0,1}

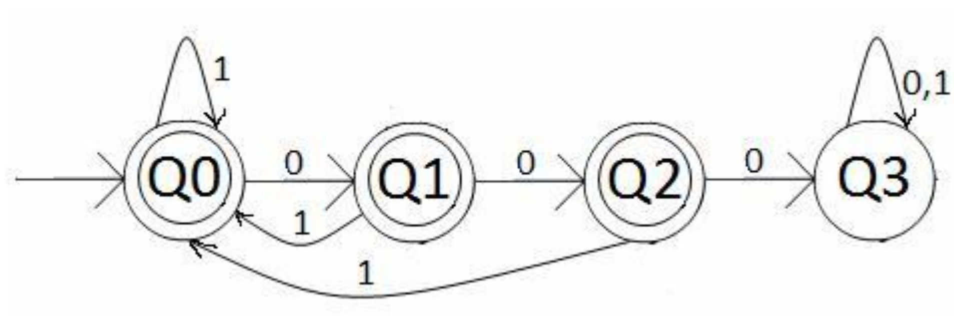
S=Qx ε ->Q

Initial state=Q0

Final state,f={Q4}

	0	1
Q0	Q0	Q1
Q1	Q1	Q2
Q2	Q2	Q3
Q3	Q3	Q4
Q4	Q4	-

Design DFA that accepts all strings without three consecutive zeroes



Q={Q0,Q1,Q2,Q3}

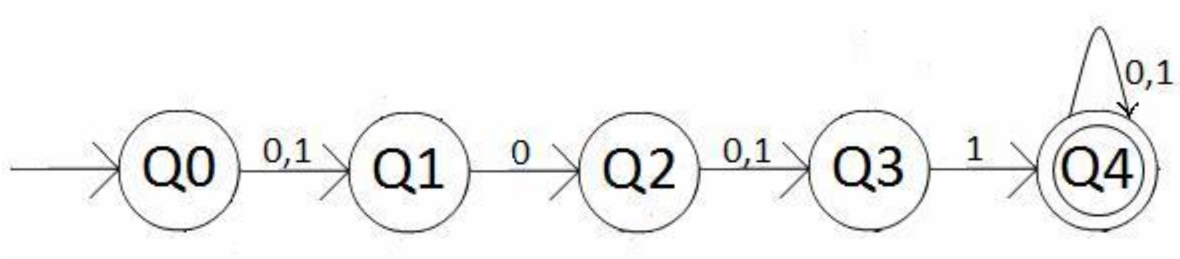
ε={0,1}

Initial state=Q0

Final states,f={Q0,Q1,Q2}

	0	1
Q0	Q1	Q0
Q1	Q2	Q0
Q2	-	Q0

Design finite automata that accepts the language L={W belongs to (0,1)*/second symbol of W is 0 and the fourth symbol is 1}



Q={Q0,Q1,Q2,Q3,Q4}

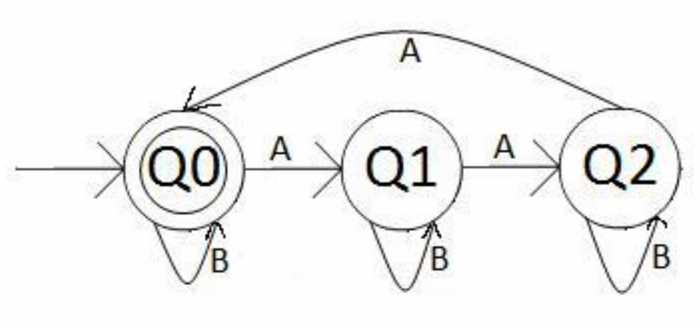
ε ={0,1}

Initial state=Q0

Final state=Q4

	0	1
Q0	Q1	Q1
Q1	Q2	--
Q2	Q3	Q3
Q3	--	Q4
Q4	Q4	Q4

Design DFA to accept strings with no of A's and B's such that numbers of A divisible by 3.



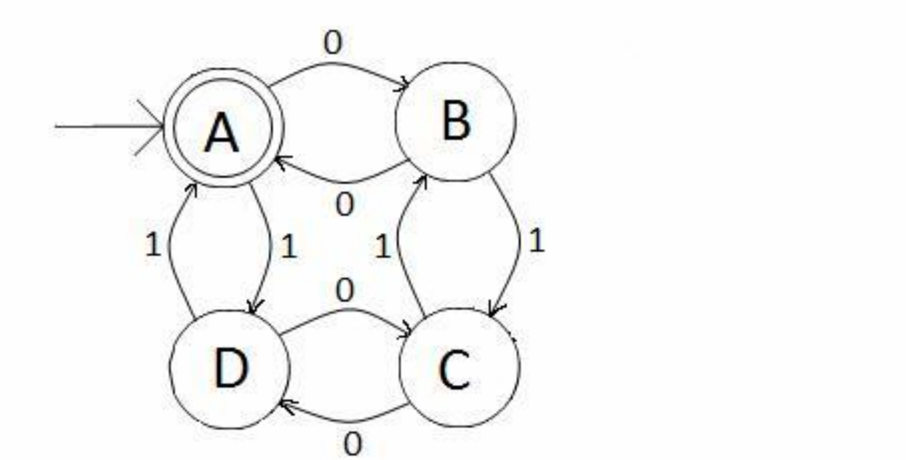
Q={Q0,Q1,Q2}

ε={A,B}

Initial state=Q0

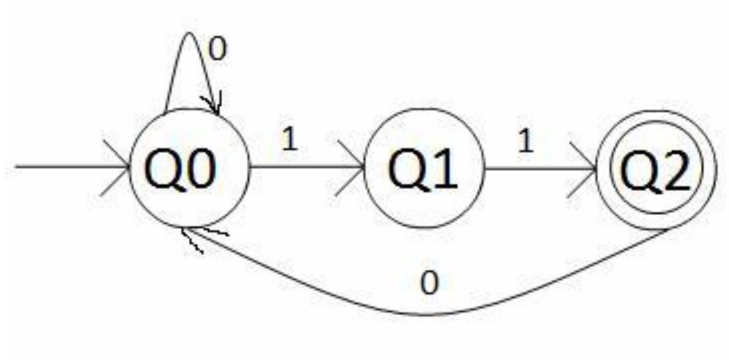
Final state={Q0}

Consider the below transition diagram and verify whether the following strings will be accepted or not



(1) 0011—accepted

Write a DFA that accepts set of all strings over 0,1 that ends with 11.



Q={Q0,Q1,Q2}

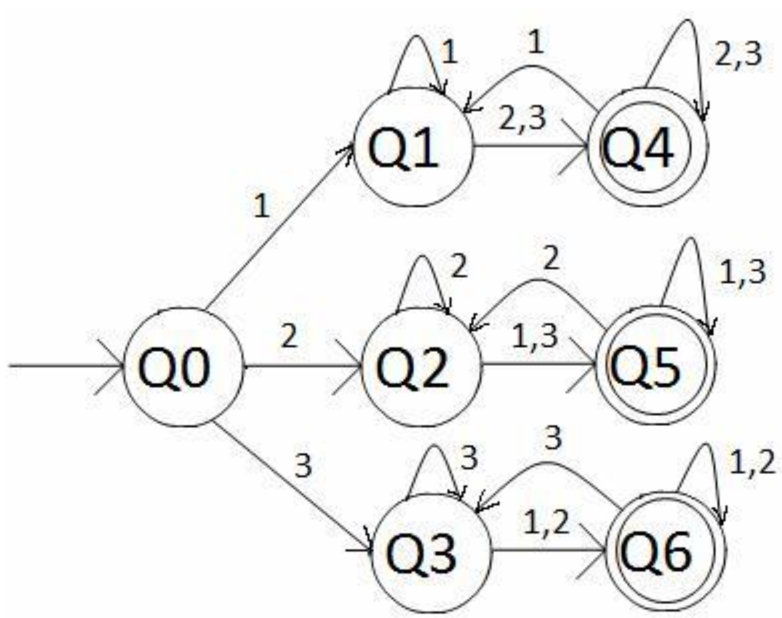
E={0,1}

Initial state=Q0

Final state, F={Q2}

	0	1
Q0	Q0	Q1
Q1	--	Q2
Q2	Q0	--

Construct a DFA that accepts set all of strings that start and end with different symbol. (over 1,2,3)



Q={Q0,Q1,Q2,Q3,Q4,Q5,Q6}

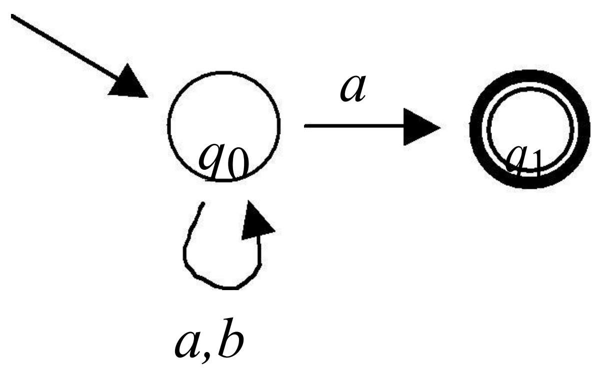
E={1,2,3}

Initial state=Q0

Final states, F={Q4,Q5,Q6}

	1	2	3
Q0	Q1	Q2	Q3
Q1	Q1	Q4	Q4
Q2	Q5	Q2	Q5
Q3	Q6	Q6	Q3
Q4	Q1	Q4	Q4
Q5	Q2	Q2	Q2
Q6	Q3	Q3	Q3

Nondeterministic Finite Automaton (NFA)



4 Does not have exactly one transition from every state on every symbol:
- Two transitions from *q*0 on *a*
- No transition from *q*0 (on either *a* or *b*)

5 Though not a DFA, this can be taken as defining a language, in a slightly different way

6 We'll consider all possible sequences of moves the machine might make for a given string

7 For example, on the string *aa* there are three:
- From *q*0 to *q*0 to *q*0, rejecting

– From $q0$ to $q0$ to $q1$, accepting

– From $q0$ to $q1$, getting stuck on the last *a*

8 Our convention for this new kind of machine: a string is in $L(M)$ if there is *at least one* accepting sequence

9 An NFA for a language can be smaller and easier to construct than a DFA

10 Strings whose next-to-last symbol is 1:

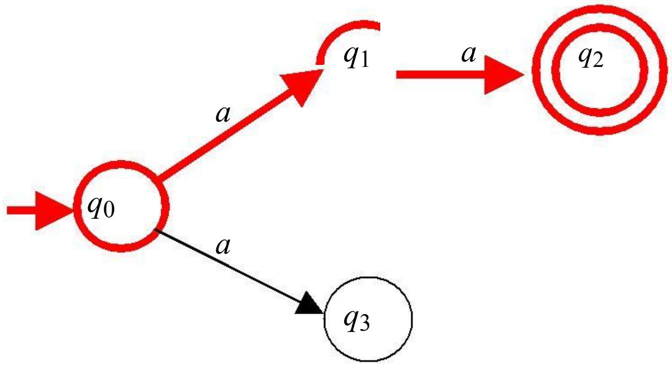
An NFA accepts a string:

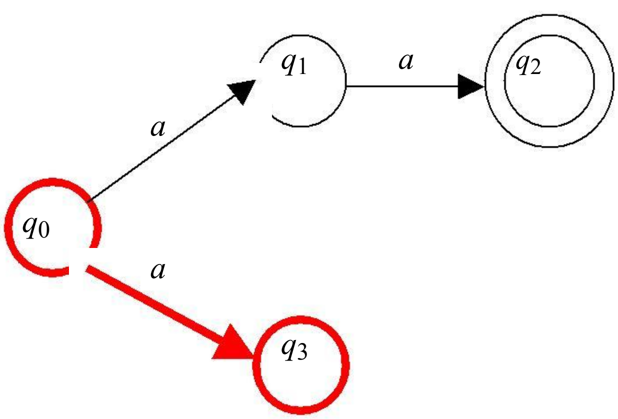
when there is a computation of the NFA that accepts the string

There is a computation: all the input is consumed and the automaton is in an accepting state

Example : aa is accepted by the NFA

“accept”





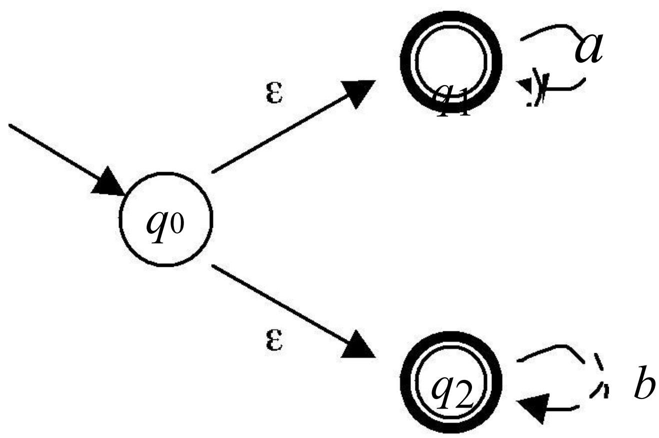
”reject”

NFA with ε-Transactions

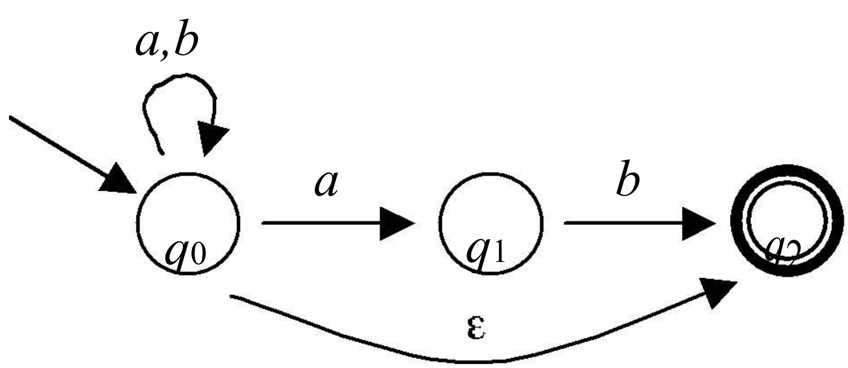
11 An NFA can make a state transition spontaneously, without consuming an input symbol

12 Shown as an arrow labeled with ε

13 For example, $\{a\}^* \cup \{b\}^*$:



5. An ε-transition can be made at any time
6. For example, there are three sequences on the empty string
   - No moves, ending in $q0$, rejecting
   - From $q0$ to $q1$, accepting
   - From $q0$ to $q2$, accepting
7. Any state with an ε-transition to an accepting state ends up working like an accepting state too
8. ε-transitions are useful for combining smaller automata into larger ones
9. This machine is combines a machine for $\{a\}^*$ and a machine for $\{b\}^*$
10. It uses an ε-transition at the start to achieve the union of the two languages



- Formally, $M = (Q, \Sigma, \delta, q0, F)$, where
  - $Q = \{q0,q1,q2\}$
  - $\Sigma = \{a,b\}$ (we assume: it must contain at least a and b)
  - $F = \{q2\}$
  - $\delta(q0,a) = \{q0,q1\}$, $\delta(q0,b) = \{q0\}$, $\delta(q0,\varepsilon) = \{q2\}$,
    $\delta(q1,a) = \{\}$, $\delta(q1,b) = \{q2\}$, $\delta(q1,\varepsilon) = \{\}$

    $\delta(q2,a) = \{\}$, $\delta(q2,b) = \{\}$, $\delta(q2,\varepsilon) = \{\}$
- The language defined is $\{a,b\}^*$
- An *instantaneous description* (ID) is a description of a point in an NFA's execution
- It is a pair (q,x) where
  - $q \in Q$ is the current state
  - $x \in \Sigma^*$ is the unread part of the input
- Initially, an NFA processing a string x has the ID $(q0,x)$
- An accepting sequence of moves ends in an ID (f,ε) for some accepting state $f \in F$

Conversion of NFA to DFA

14 For any string x, there may exist none or more than one path from initial state and associated with x.

15 Consider an NFA M=(Q, Σ, δ, s, F).

16 For x in Σ*, define

[x] = {q in Q | there exists a path s   q}

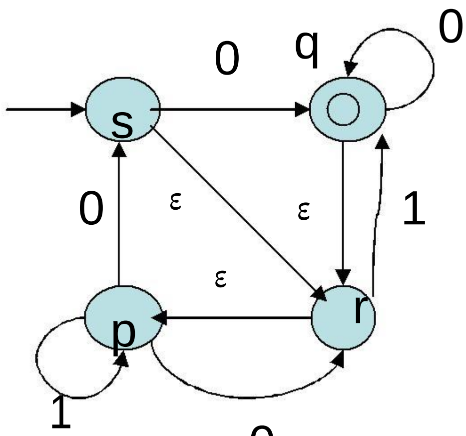
• Define DFA M'=(Q', Σ, δ', s', F'}: Q' = {
[x] | x in Σ* },

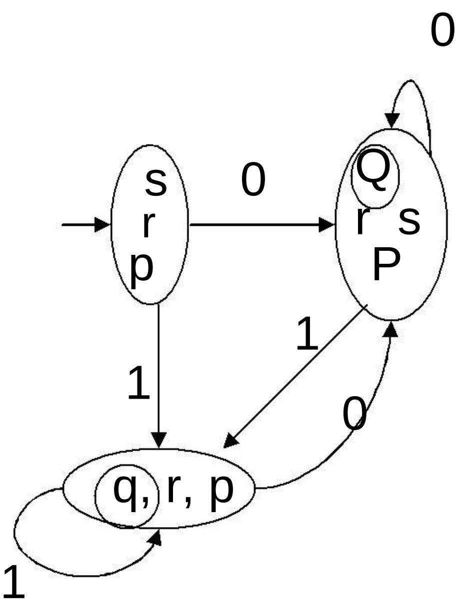
δ ([x], a) = [xa] for x in Σ* and a in Σ, s' =
[ε],

F' = { [x] | x in L(M) }

Convert the following NFA to DFA.

Ex1:





Find a DFA equivalent to NFA M= ({q0,q1,q2},{a,b,}, ∂,q0,{q2}) ∂ is given by

	a	B
qo	{q0,q1}	{q2}
q1	{qo}	{q1}
q2	-	{q0,q1}

qo is the initial state q2

is the final state

construction of DFA

M'=(Q',∑,∂',qo',F')

11. $Q' = 2^q$ ={ǿ,[q0],[q1],[q2],[q0,q1],[q0,q2],[q1,q2],[q0,q1,q2]}

12. [q0] is initial state

13. F'={[q2],[q0,q2],[q1,q2],[q0,q1,q2]}

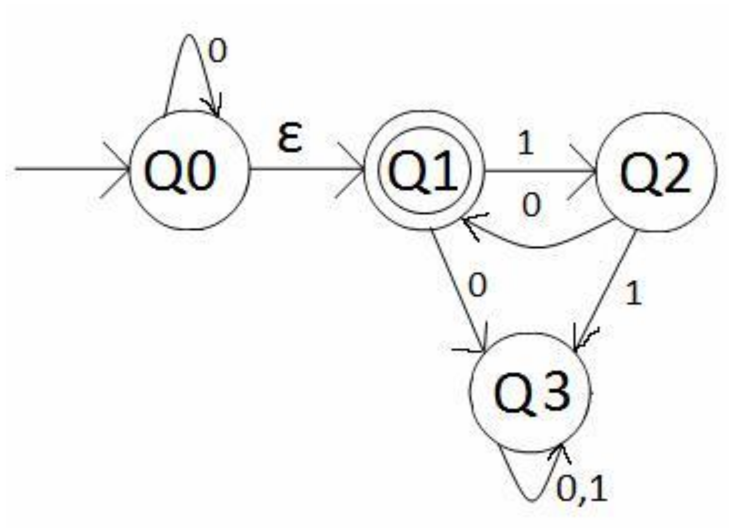
14. ∂' as follows

	A	b
{q0}	{q0,q1}	{q2}
{q1}	{q0}	{q1}
{q2}	ǿ	{q0,q1}
{q0,q2}	{q0,q1}	{q1}
{q1,q2}	{q0}	{q0,q1}
{q0,q1,q2}	{q0,q1}	{q0,q1,q2}

qo is the initial state

{q0,q1,q2} is the final state

Convert the following NFA with ϵ moves into equivalent NFA ϵ without moves.



The above is NFA with ϵ moves

M={Q, ϵ,^,q0,F}

={{q0,q1,q2,q3},{0,1},^,q0,q1}

M*={Q, ϵ,^ ‘,q0,F’}

17 -closure(q0)={q0,q1}
“ (q1)={q1}

“ (q2)={q2} “

(q3)={q3}

.

	0	1	є
q0	q0	---	1
q1	q3	q2	---
q2	q1	q3	---
q3	q3	q3	---

^’(q0,0)= Є-closure(^(^(q0,Є),0))

= “ (^(Є-closure(q0),0))

= “ (^(q0,q1,0)

= “ (^(q0,0)u^(q1,0))

= “ (q0uq3)

= {q0,q1,q3}

^’(q0,1)=Є-closure(^(^’(q0,Є),1))

= “ (^(Є-closure(q0),1))

= “ (^(q0,q1,1))

= “ (^(q0,1)u^(q1,1))

= “ (q2)

= {q2}

^’(q1,0)= Є-closure(^(^’(q1,Є),0))

= “ (^(Є-closure(q1),0))

= “ (^(q1,0))

= “ (q3)

= {q3}

^’(q1,1)= Є-closure(^(^’(q,Є),1))

= “ (^(Є-closure(q1),1))

= “ (^(q1,1))

= “ (q2)

1. {q2}

^’(q2,0)= ϵ-closure(^(^’(q2,ϵ),0))

= “ (^(ϵ-closure(q2),0))

= “ (^(q2,0))

= “ (q1)

= {q1}

^’(q2,1)= ϵ-closure(^(^’(q2,ϵ),1))

= “ (^(ϵ- closure(q2),1))

= “ (^(q2,1)

= (q3)

= {q3}

^’(q3,0)= ϵ-closure(^(^’(q3,ϵ),0))

2 “ (^(ϵ closure(q3),0))

=“ (^(q3,0))

= “ (q3)

= {q3}

^’(q3,1)= ϵ closurϵ(^(^’(q3,ϵ),1))

= “ (^(ϵ closurϵ(q3),1))

= “ (^(q3,1))

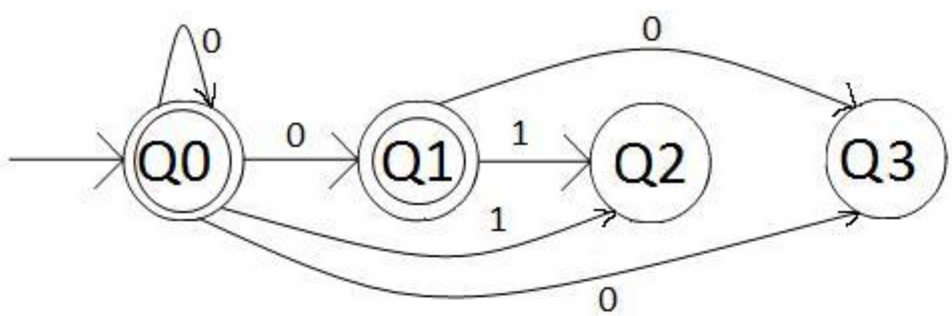
= “ (q3)

= {q3}

here final states are q0 and q1

Since

F’={Fu{q1} if ϵ closure(q1) contain a state F}



Regular sets

A special class of sets of words over S, called regular sets, is defined recursively as follows.

18Every finite set of words over S is a regular set.

19If U and V are regular sets over S, then U u V and UV are also regular.

20If S is a regular set over S, then so is its closure S*.

21No set is regular unless it is obtained by a finite number of applications of Definitions (1) to (3).

i.e., the class of regular sets over S is the smallest class containing all finite sets of words over S and closed under union, concatenation and star operation.

Examples

1. Let [redacted] = {1} then the set of strings {1, 11, 111 …} is a regular set.
2. Let [redacted] = {0,1} then the set of strings {10,01} is a regular set.

Regular Expression

Let be an alphabet. The regular expressions over and the sets that they denote are defined recursively as follows.

1. ϕ is a regular expression and denotes empty set.

. λ is a regular expression and denotes the set {λ}

For each a in Σ,a is a regular expression and denotes the set {a}.

These are called primitive regular expressions.

15. If r and s are regular expressions denoting the languages R and S, respectively, then (r+s), (rs) and (r*) are regular expressions that denote the sets R U S, RS and R* respectively.

3.A string is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules in (2).

Examples

i) For Σ = {a,b}, the expression r = (a+b)* (a + bb) is regular. It denotes the language,

L(r) ={a,bb,aa,ab,ba,bbb,…………}

The first part (a+b)* stands for any string of a's and b's. The second part (a+bb) represents either an 'a' or a double 'b'. Consequently L(r) is the set of all strings on {a,b} terminated by either an 'a' or 'bb'

Languages Associated with Regular Expressions

Regular expressions can be used to describe some simple languages. If r is a regular expression, we will let L(r) denote the language associated with 'r'.

The languageL(r) denoted by any regular expression r is defined by the following rules

1. ϕ is a regular expression denoting the empty set.

2. λ is a regular expression denoting {λ}.

3. for every a $\in \Sigma$, a is a regular expression denoting {a}. If r and s are regular expressions, then

4. L(r+s)=L(r) U L(s),

5. L(rs)=L(r) L(s)

6. L(r)=L(r)

7. L(r*)=(L(r))*

Identity rules

I1 [redacted] + R = R

I2 [redacted].R = R.[redacted] = [redacted]

I3 ^R = R^ = R

I4 ^* =^ and [redacted]* =^

I5 R + R = R

I6 R*R* = R*

I7 R.R* = R*.R

I8 (R*)* = R

I9 ^ +RR* = R* = ^ + R*.R

I10 (PQ)*P = p.(QP)*

I11 (P + Q)* = (P*Q*)* = (P* + Q*)*

I12 (P+Q)R = PR +QR and

R(P + Q)= RP + RQ

Manipulation of Regular Expression(Arden's Theorem)

Let p and q be two regular expressions over [illegible]. If p does not contain [illegible], then the following eqution in r, r=q+p has a unique solution given by r= qp*.

Example

.Prove that the regular expression r = [illegible] + 1*(011)*(1*(011)*)* describes the same set of strings as that of (1 +011)*

Sol:

r= [illegible]+ P1P1*

where p1= 1*(011)*

1 p1* using I9

2 (1*(011)*)*

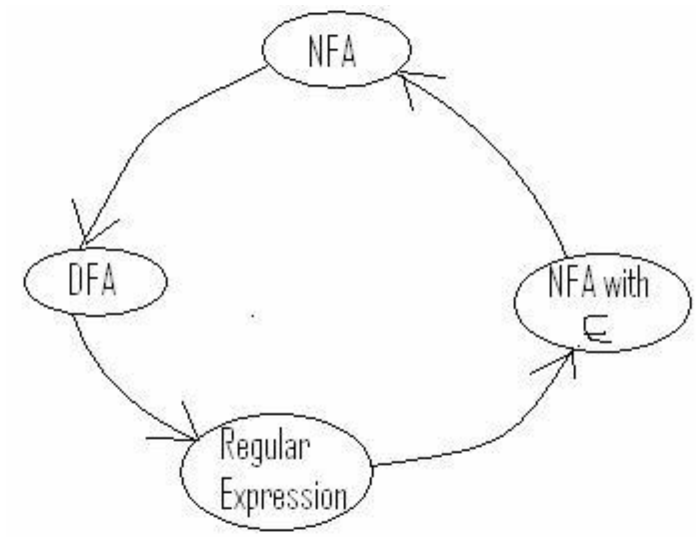
3 (p2*p3*)* letting p2=1, p3=011

4 (p2 + p3)* using I11

=(1+011)*

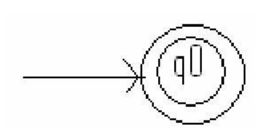
Hence proved

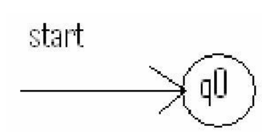
Equivalence and conversion between FA and R.E



Constructing Finite Automata for a given Regular Expressions

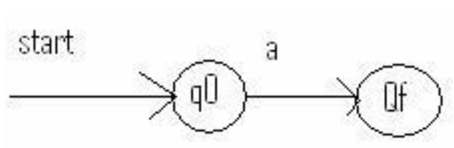
1. r=[redacted] start



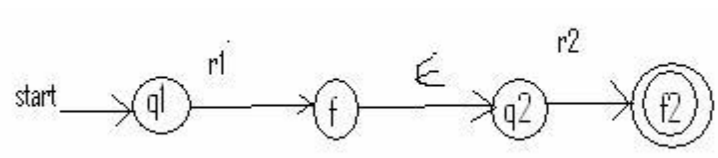


2. r = ϕ

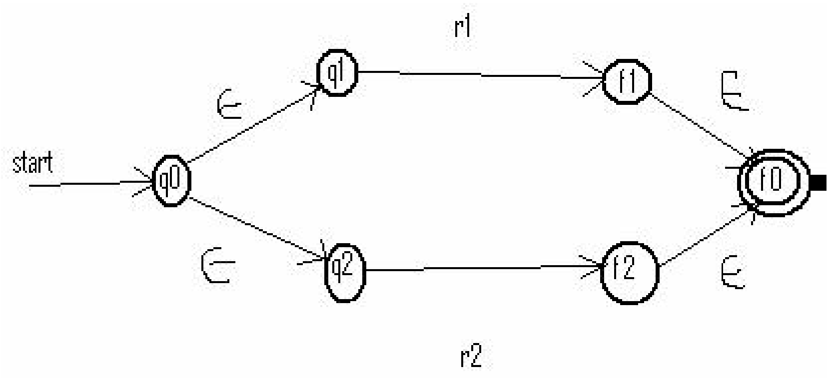
3. r = a



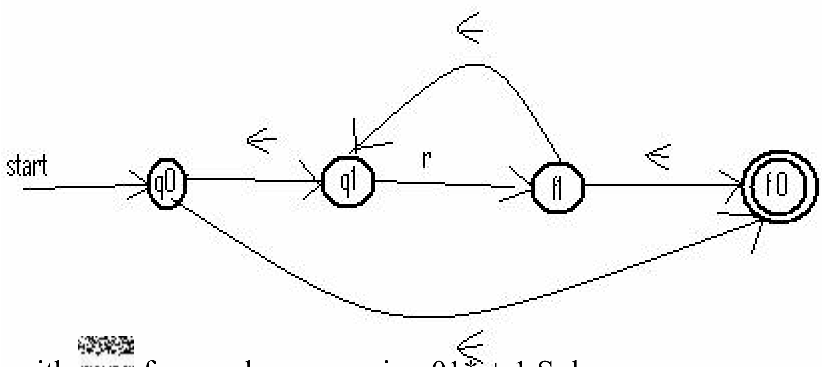
4.r = r1.r2 (concatenation of regular expression)



5. r = r1 + r2



6. Kleen closure (r*)

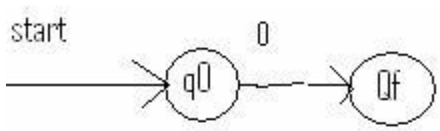


Example

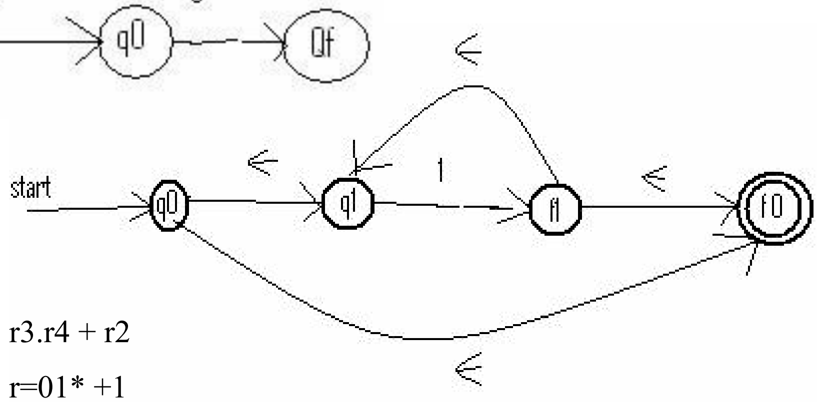
Construct NFA with for regular expression 01* + 1 Sol:

r=01* + 1 r=r1 + r2 r1 = r3.r4 r3=0 r4=1*

r3

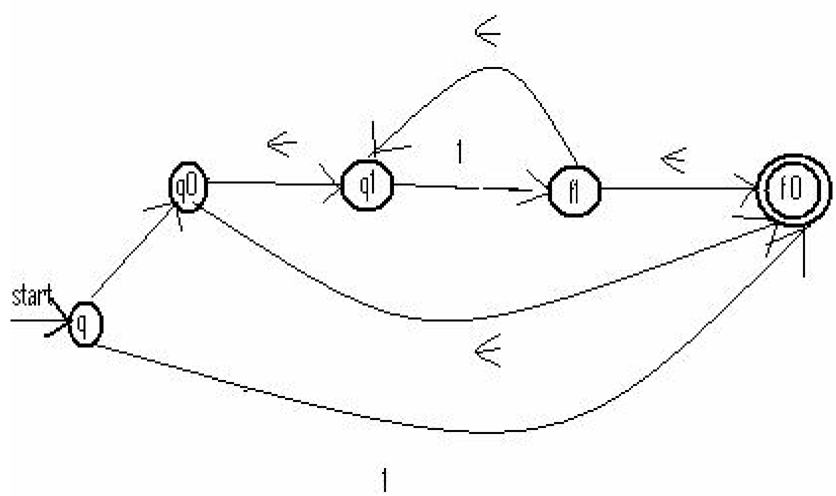


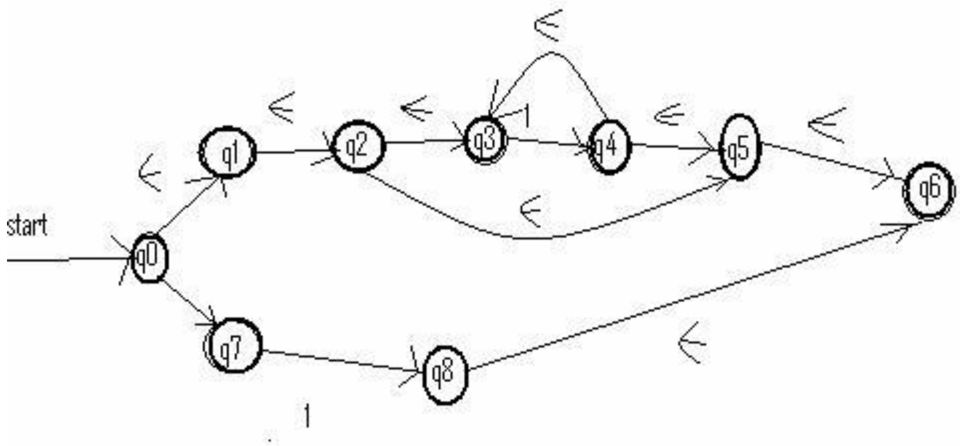
r4



r3.r4 + r2

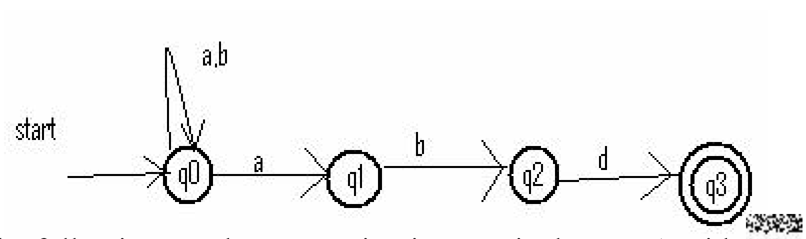
r=01* +1





Convert the regular expression to NFA

1.(a+b)* abd Sol:



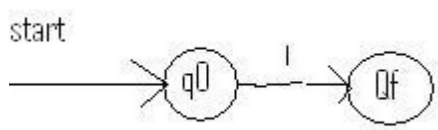
2.Convert the following regular expression into equivalent NFA with ∈-transaction r= (10*)*

Sol:

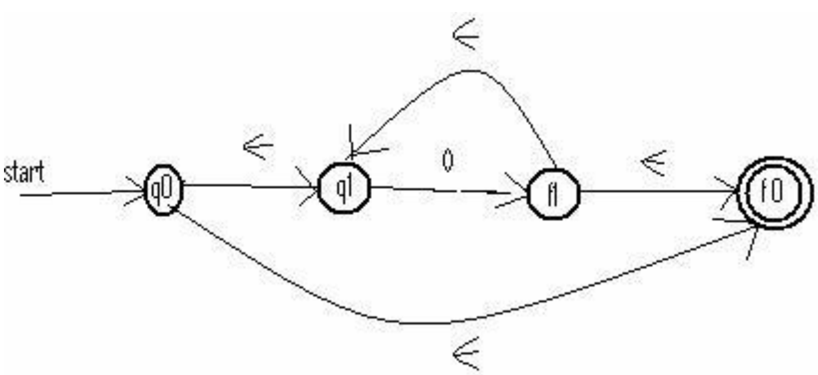
Given r = (10*)* r1 = 1

r2=0* r3= r1.r2, r =
(r3)*

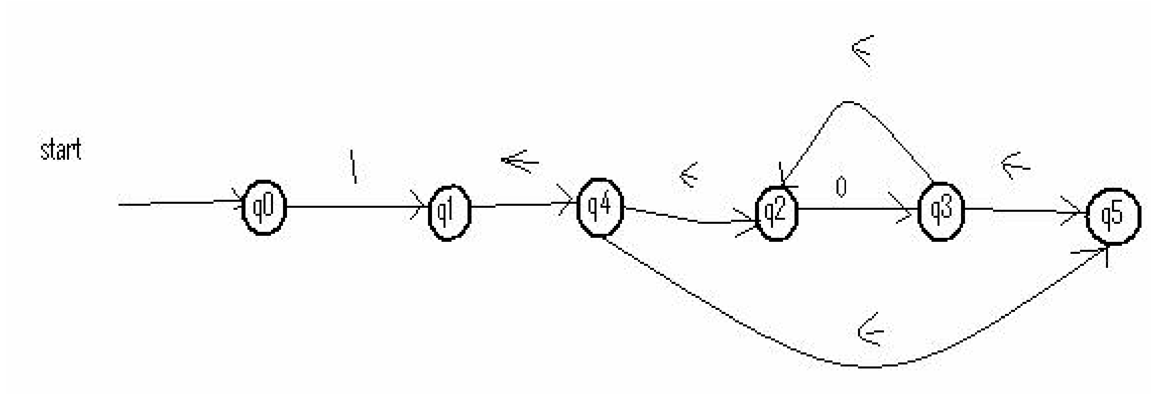
r1 is



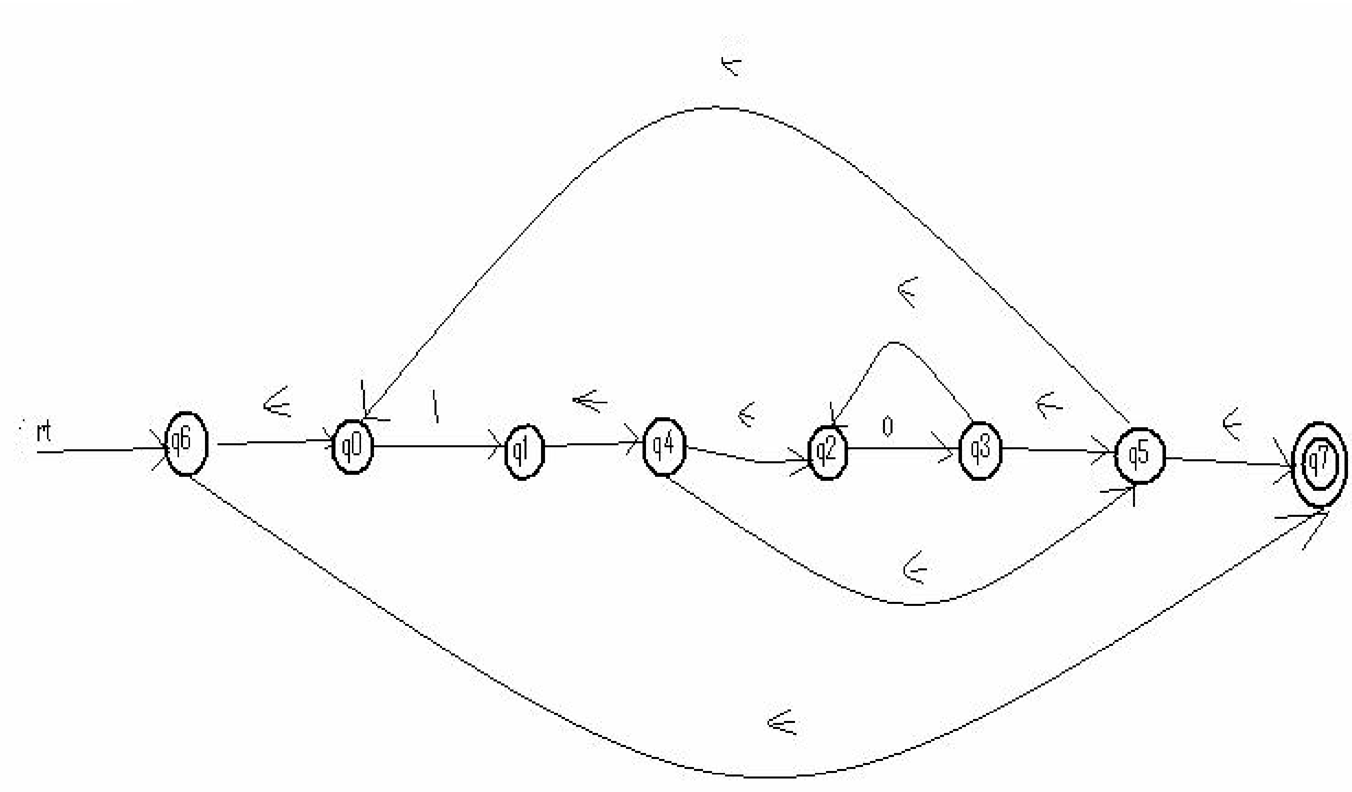
r2 is



r3 is



r = (r3)*



Therefore r = (10*)*

Conversion of Finite Automata to Regular Expressions

<u>Arden's Threorm:</u>

Let P and Q be 2 regular expressions over . If P does not contain ^(epsilon) then following relation R namely

R = Q + RP has a unique solution given by R = QP*

This method is used to find regular expression recognized by a transitions are made regarding transition system

1.The transition graph does not have ^ moves 2. Its vertices are V1 to Vn

3.Vi the regular expression represents set of strings accepted by system even though Vi is final state.

4. [redacted]i,jdenotes regular expression representing set of labels of edges from Vi to Vj when there is no edge [redacted]ij = [redacted] consequently we get following set of equations in V1 to Vn

V1 = V1 [redacted]11 + V2 [redacted]21 +…………+ Vn [redacted]n1+ ^

V2 = V1 [redacted]12 + V2 [redacted]22 + …………+ Vn [redacted]n2

:

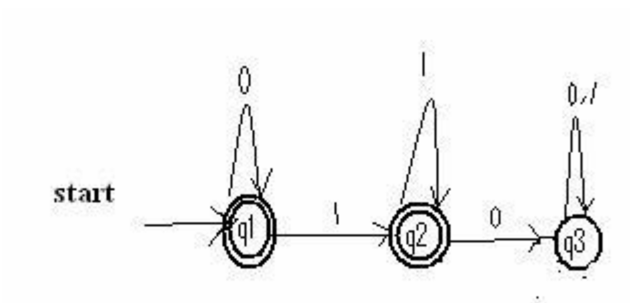
:

:

Vn = V1 [redacted]1n + V2 [redacted]2n +…………+Vn [redacted]nn

Example

1.Find the Regular Expression for a given transition diagram



Sol q1 = q1.0 + ^ equ1

q2 = q1.1 + q2 equ2

q3 = q2.0 + q3.0 + q3.1equ3

q3= q2.0 + q3(0+1)

q1=^ + q1.0

q1 =^.0* =0*

put in equ2

q2 = 0*.1 + q2.1

q2 =0*.1(1)*

q1+q2 = 0* + 0*1(1)*

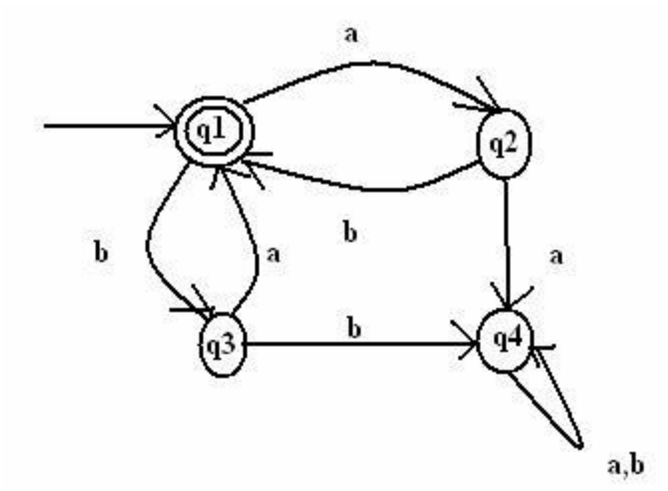
= 0*(^ + 11*)

=0*1*   (from I9)

$\boxed{\text{q1 + q2 = 0*1*}}$

is the required solution.

2.Construct regular expression for a given Finite Automata



Sol:

q1 = q2.b + q3.a + ^   equ1

q2 = q1.a   equ2

q3 = q1.b   equ3

q4 = q2.a + q3.b + q4(a+b) equ4

substitute equ2 and equ3 in equ1

q1 = (q1.a))b + (q1.b)a + ^

q1 = ^ + q1(ab +ba)

The above is in R = Q +R.P form

Where R = q1

Q = ^

P = (ab+ba)

Therefore the solution is R = QP*

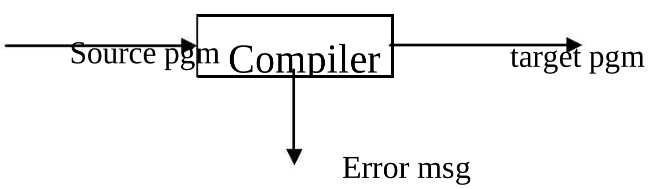
Which is nothing but q1= ^(ab +ba)*

$\boxed{q1 = (ab + ba)^*}$

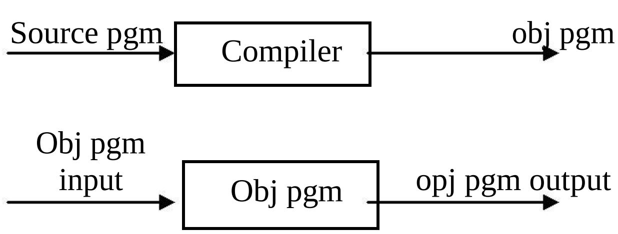
[Since ^(ab +ba)* = (ab+ba)*]

COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

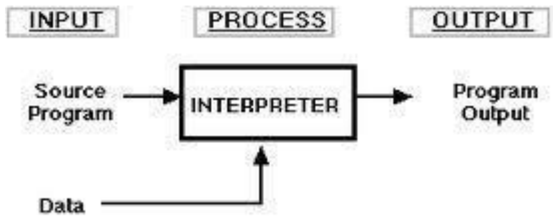


Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.



ASSEMBLER*:* programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER*:* An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

2. Lexical analysis
3. Synatx analysis
4. Semantic analysis
5. Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.

Type of object that denotes a various may change dynamically.

Debugging a program and finding errors is simplified task for a program used for interpretation.

The interpreter for the language makes it machine independent.

Disadvantages:

The execution of the program is *slower*.

Memory consumption is more.

3 ***Loader and Link-editor:***

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To over come this problems of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could"relocate" directly behind the user's program. The task of adjusting programs o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

STRUCTURE OF THE COMPILER DESIGN

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

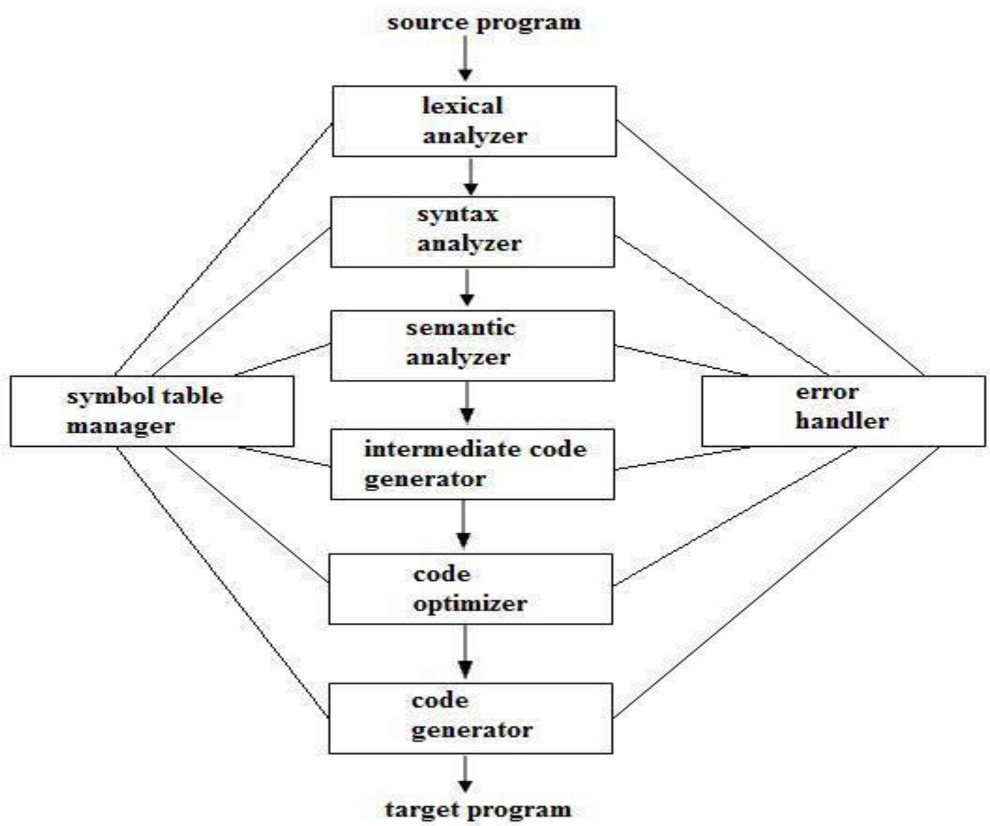


Fig 1.5 Phases of a compiler

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called **tokens.**

Syntax Analysis:-

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced.

This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:-

This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a
'Symbol Table'.

Error Handlers:-

It is invoked when a flaw error in the source program is detected.

The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression.

Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Example, (A/B*C has two possible interpretations.)

1, divide A by B and then multiply by C or

2, multiply B by C and then use the result to divide A.

each of these two interpretations can be represented in terms of a parse tree.

Intermediate Code Generation:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands.

The output of the syntax analyzer is some representation of a parse tree. the intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the some job as the original, but in a way that saves time and / or spaces.

1, Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If **A** > **B** goto **L2**

Goto **L3**

L2 :

This can be replaced by a single statement If

A < B goto **L3**

Another important local optimization is the elimination of common sub-expressions

A := B + C + D
E := B + C + F

Might be evaluated as

T1 := B + C

A := T1 + D
E := T1 + F

Take this advantage of the common sub-expressions **B + C.**

2, Loop Optimization:-

Another important source of optimization concerns about **increasing the speed of loops**. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.

Code generator :-

Cg produces the object code by deciding on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Table Management OR Book-keeping :-

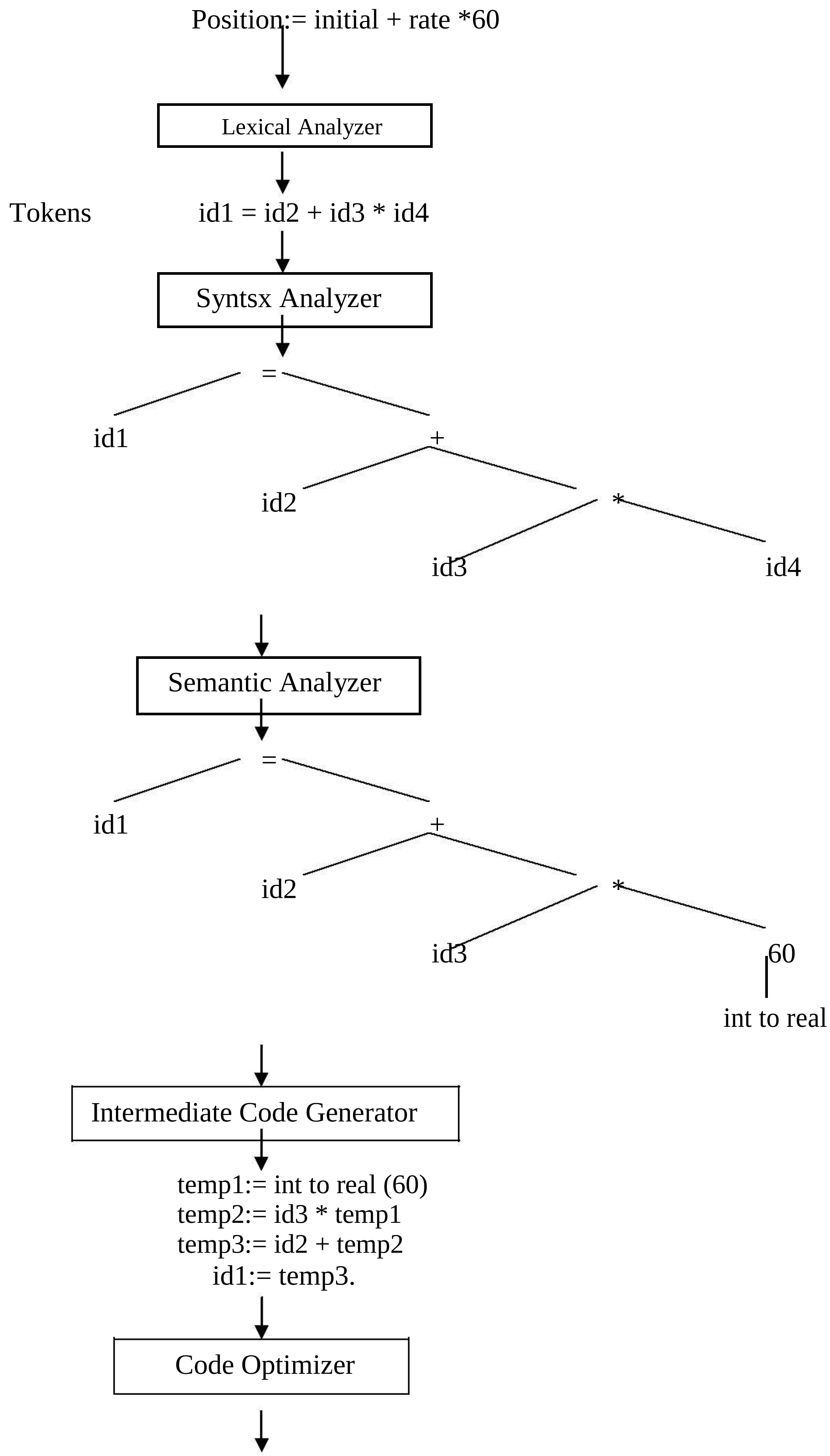
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

Error Handing :-

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:


Position:= initial + rate *60
Lexical Analyzer
Tokens
id1 = id2 + id3 * id4
Syntsx Analyzer
=
id1
+
id2
*
id3
id4
Semantic Analyzer
=
id1
+
id2
*
id3
60
int to real
Intermediate Code Generator
temp1:= int to real (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.
Code Optimizer

Temp1:= id3 * 60.0
Id1:= id2 +temp1

↓

Code Generator

↓

MOVF id3, r2
MULF *60.0, r2
MOVF id2, r2
ADDF r2, r1
MOVF r1, id1

TOKEN

LA reads the source program one character at a time, carving the source program into a sequence of automatic units called ‘Tokens’.

1, Type of the token.
2, Value of the token.

Type : variable, operator, keyword, constant
Value : N1ame of variable, current variable (or) pointer to symbol table.

If the symbols given in the standard format the LA accepts and produces token as output. Each token is a sub-string of the program that is to be treated as a single unit. Token are two types.

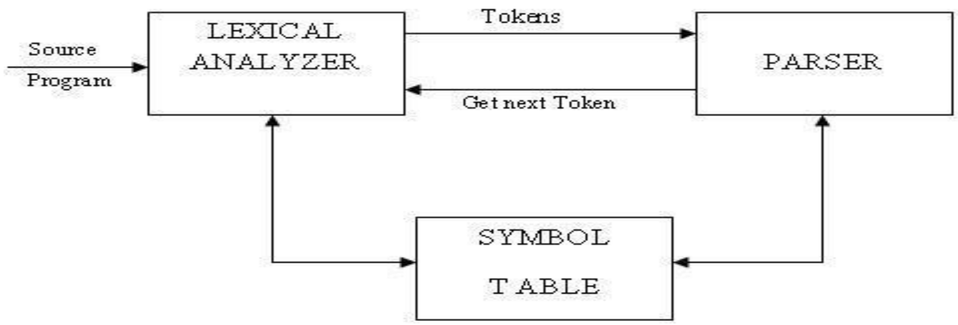
1, Specific strings such as IF (or) semicolon.
2, Classes of string such as identifiers, label, constants.

OVER VIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

ROLE OF LEXICAL ANALYZER

the LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

LEXICAL ANALYSIS VS PARSING:

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence). A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).

TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

Token	**lexeme**	**pattern**
Const	const	const
If	if	If
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
I	pi	any numeric constant
Nun	3.14	any character b/w “and “except"
Literal	"core"	pattern

A patter is a rule describing the set of lexemes that can represent a particular token in source program.

LEXICAL ERRORS:

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a *lexeme* as a valid *token* for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognised valid tokens don't match any of the right sides of your grammar rules. simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

i. Delete one character from the remaining input.
ii. Insert a missing character in to the remaining input.
iii. Replace a character by another character.
iv. Transpose two adjacent characters.

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.

Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.

List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.

An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.

The compiler produce object code whereas interpreter does not produce object code. In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. hence interpreter is less efficient than compiler.

Examples of interpreter: A *UPS Debugger* is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files. example of compiler: *Borland c compiler* or Turbo C compiler compiles the programs written in C or C++.

REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string.

Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R2R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each 'a' in ∑, is a regular expression denoting { a }, the language with only one string consisting of the single symbol 'a' .
- If R and S are regular expressions, then

(R) | (S) means LrULs
R.S means Lr.Ls
R* denotes Lr*

REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter -
Digits - Id -

A | B | ……| Z | a | b |……| z|
0 | 1 | 2 | …. | 9
letter (letter / digit)*

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

Stmt → if expr then stmt
 | If expr then else stmt
 | є

Expr → term relop term | term

Term → id |number

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```
digit     -->[0,9]
digits    -->digit+
number  -->digit(.digit)?(e.[+-]?digits)?
letter    -->[A-Z,a-z]
 id       -->letter(letter/digit)*
 if       --> if
 then      -->then
 else     -->else
relop     --></>/<=/>=/==/< >
```

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws → $(blank/tab/newline)^+$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	Token Name	Attribute Value
Any ws	_	_
If	if	_
Then	then	_
Else	else	_
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE

TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .

Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.

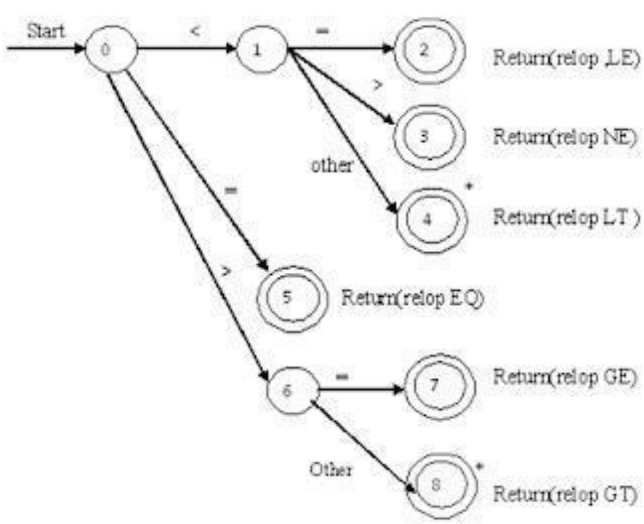
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

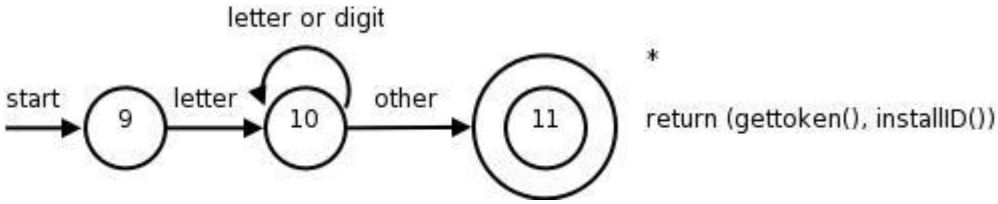
1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.

2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.

3. One state is designed the state ,or initial state ., it is indicated by an edge labeled “start” entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

If	=	if
Then	=	then
Else	=	else
Relop	=	< | <= | = | > | >=
Id	=	letter (letter | digit) *|
Num	=	digit |

AUTOMATA

An automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1, an automation in which the output depends only on the input is **called an automation without memory.**

2, an automation in which the output depends on the input and state also is **called as automation with memory**.

3, an automation in which the output depends only on the state of the machine is **called a Moore machine**.

3, an automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

DESCRIPTION OF AUTOMATA

1, an automata has a mechanism to read input from input tape,

2, any language is recognized by some automation, Hence these automation are basically language ‘acceptors’ or ‘language recognizers’.

Types of Finite Automata

- Deterministic Automata
- Non-Deterministic Automata.

DETERMINISTIC AUTOMATA

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

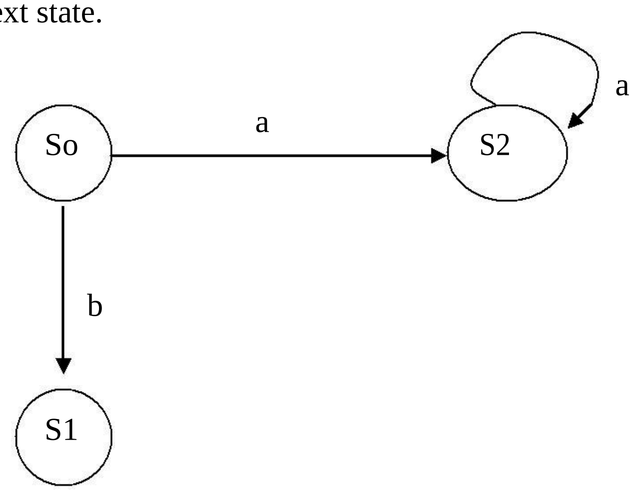
1, it has no transitions on input € ,
2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where
Q is a finite 'set of states', which is non empty.
∑ is 'input alphabets', indicates input set.
qo is an 'initial state' and qo is in Q ie, qo, ∑, Q
F is a set of 'Final states',
δ is a 'transmission function' or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

Regular expression → NFA → DFA → Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



From state S0 for input 'a' there is only one path going to S2. similarly from S0 there is only one path for input going to S1.

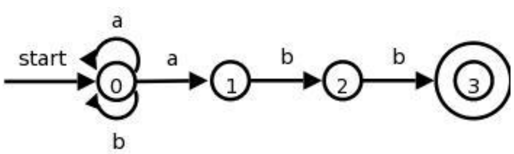
NONDETERMINISTIC AUTOMATA

- A NFA is a mathematical model that consists of
  - A set of states S.
  - A set of input symbols ∑.
  - A transition for move from one state to an other.
  - A state so that is distinguished as the start (or initial) state.
  - A set of states F distinguished as accepting (or final) state.
  - A number of transition to a single symbol.

- A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, In which the nodes are the states and the labeled edges represent the transition function.

- This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol € as well as by input symbols.

- The transition graph for an NFA that recognizes the language (a | b) * abb is shown



DEFINITION OF CFG

It involves four quantities.

CFG contain terminals, N-T, start symbol and production.

- Terminal are basic symbols form which string are formed.
- N-terminals are synthetic variables that denote sets of strings
- In a Grammar, one N-T are distinguished as the start symbol, and the set of string it denotes is the language defined by the grammar.
- The production of the grammar specify the manor in which the terminal and N-T can be combined to form strings.
- Each production consists of a N-T, followed by an arrow, followed by a string of one terminal and terminals.

DEFINITION OF SYMBOL TABLE

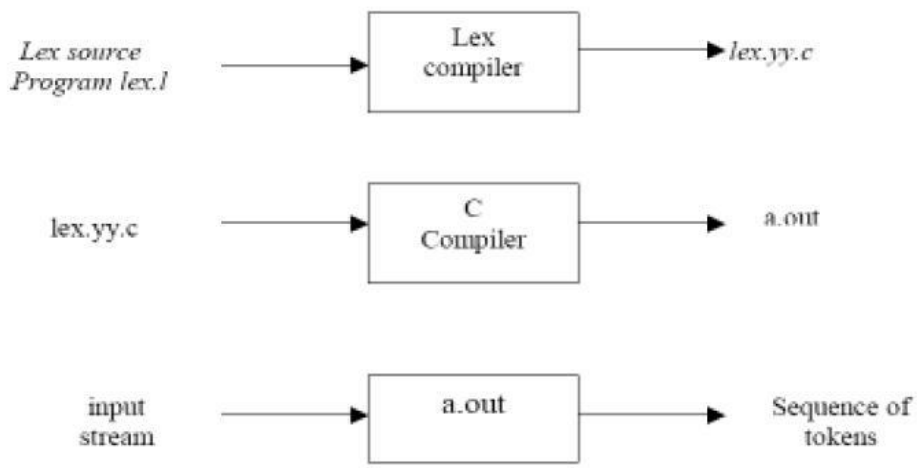
- An extensible array of records.
- The identifier and the associated records contains collected information about the identifier.

FUNCTION identify (Identifier name)

RETURNING a pointer to identifier information contains

- The actual string
- A macro definition A
- keyword definition
- A list of type, variable & function definition
- A list of structure and union name definition
- A list of structure and union field selected definitions.

Creating a lexical analyzer with Lex



Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

   p1 *{action 1}*
   p2 *{action 2}*
   p3 *{action 3}*
   ... ...
   ... ...

   where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions.*Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.

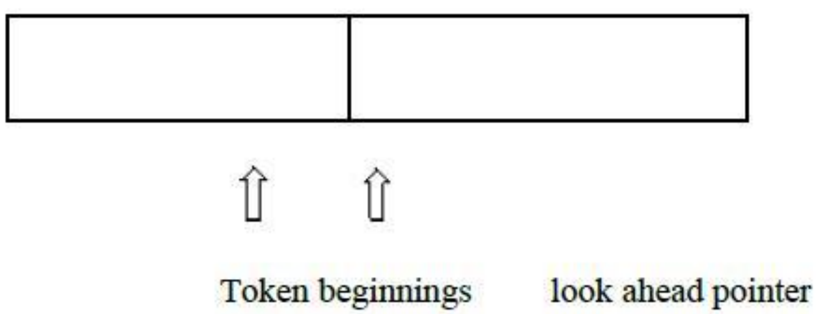
INPUT BUFFERING

The LA scans the characters of the source pgm one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for thelexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two haves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and thecharacter next to be read. In practice each buffering scheme adopts one convention either apointer is at the symbol last read or the symbol it is ready to read.



Token beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. For example, in a PL/I program we may see: DECALRE (ARG1, ARG2… ARG *n*) Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, ifthe look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been groupedinto tokens. While we can make the buffer larger if we chose or use another buffering scheme,we cannot ignore the fact that overhead is limited.

UNIT-2
CONTEXT FREE GRAMMARS AND PARSING

ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)
2.Bottom up parser: which build parse trees from leaves and work up the root.
Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing

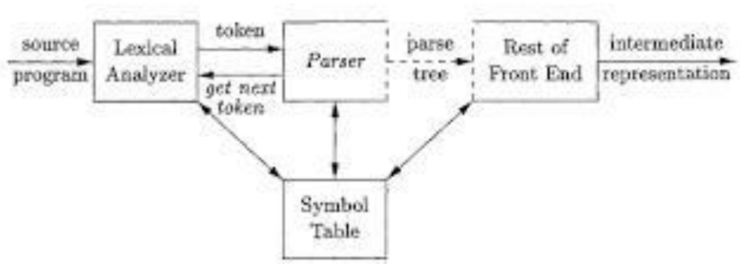


Figure 4.1: Position of parser in compiler model

TOP-DOWN PARSING

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-downparsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form . A syntax of the form defines zero or one occurrence of the form . A syntax of the form defines zero or more occurrences of the form .

A usual implementation of an LL(1) parser is:

- initialize its data structures,
- get the lookahead token by calling scanner routines, and
- call the routine that implements the start symbol.

Here is an example.

proc syntaxAnalysis()
begin

initialize(); // initialize global data and structures
nextToken(); // get the lookahead token

program(); // parser routine that implements the start
symbol end;

FIRST AND FOLLOW

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is {X}.

2. If X->e is a production, then add e to FIRST(X).

3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for

some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1) that is, Y1.......Yi-$_1$=*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=*>e, then we add FIRST(Y2) and so on.

To compute the FIRST(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol and $ in the input right endmarker.

2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).

3. If there is aproduction A->aB or a production A->aBs where FIRST(s) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Follow.Find the first and follow of all nonterminals in the Grammar-

E -> TE'

E'-> +TE'|e

T -> FT'

T'-> *FT'|e

F -> (E)|id

Then:

FIRST(E)=FIRST(T)=FIRST(F)={(,id}

FIRST(E')={+,e}

FIRST(T')={*,e}

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

FOLLOW(T)=FOLLOW(T')={+,),$}

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

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with i=1 in each case, since FIRST(id)=(id) and FIRST('(')= {(} by rule 1. Then by rule 3 with i=1, the production T -> FT' implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW,we put $ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied toproduction F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E-> TE', $ and right parenthesis are in FOLLOW(E').

3.5 CONSTRUCTION OF PREDICTIVE PARSING TABLES

For any grammar G, the following algorithm can be used to construct the predictive parsing table. The algorithm is

Input : Grammar G

Output : Parsing table M

Method

1. 1.For each production A-> a of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(a), add A->a, to M[A,a].
3. If e is in First(a), add A->a to M[A,b] for each terminal b in FOLLOW(A). If e is in FIRST(a) and $ is in FOLLOW(A), add A->a to M[A,$].
4. Make each undefined entry of M be error.

.LL(1) GRAMMAR

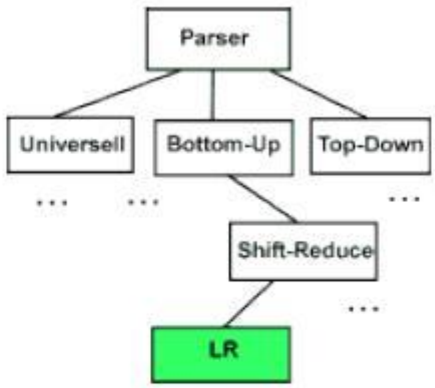
The above algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control constructs and to use operator precedence for expressions.however, if an lr parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

UNIT-3
BOTTOM UP PARSERS

LR PARSING INTRODUCTION

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.



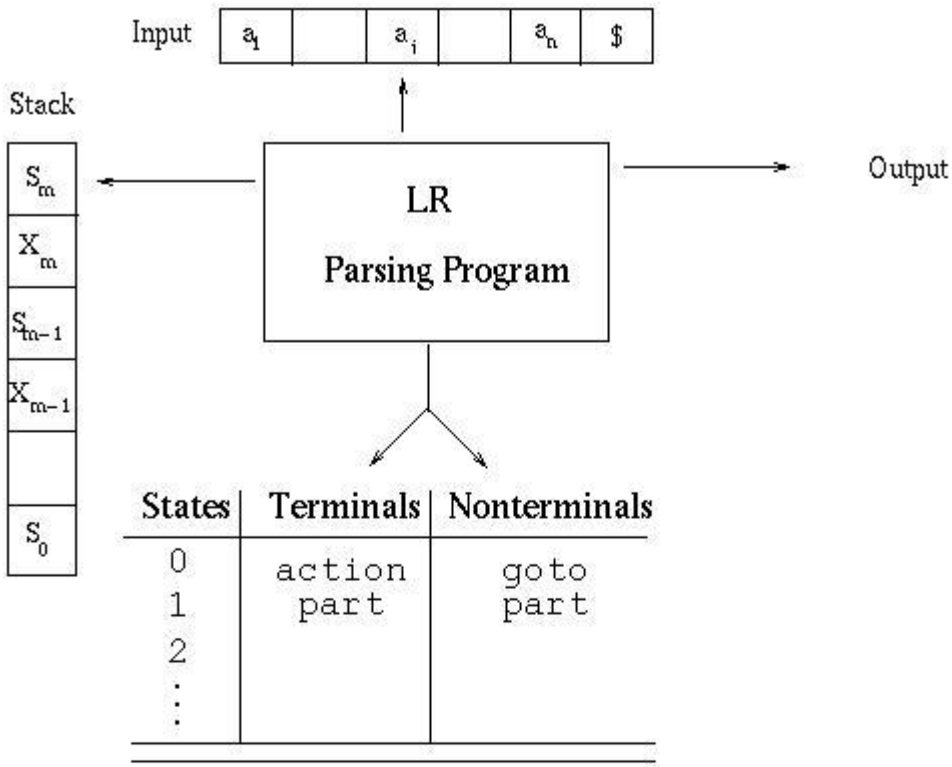
WHY LR PARSING:

- ✓ LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- ✓ The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- ✓ The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- ✓ An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

.MODELS OF LR PARSERS

The schematic form of an LR parser is shown below.



LR Parser Table

The program uses a stack to store a string of the form s0X1s1X2...Xmsm where sm is on top. Each Xi is a grammar symbol and each si is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shiftreduce parsing decision. The parsing table consists of two parts: a parsing action function action and a goto function goto. The program driving the LR parser behaves as follows: It determines sm the state currently on top of the stack and ai the current input symbol. It then consults action[sm,ai], which can have one of four values:

- shift s, where s is a state
- reduce by a grammar production A -> b
- accept
- error

The function goto takes a state and grammar symbol as arguments and produces a state.

For a parsing table constructed for a grammar G, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shiftreduce parser because they do not extend past the rightmost handle.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

(s0 X1 s1 X2 s2... Xm sm, ai ai+1... an$)

This configuration represents the right-sentential
form X1 X1 ... Xm ai ai+1 ...an

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading ai and sm, and consulting the parsing action table entry action[sm, ai]. Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later.

The configurations resulting after each of the four types of move are as follows:

If action[sm, ai] = shift s, the parser executes a shift move entering the configuration
(s0 X1 s1 X2 s2... Xm sm ai s, ai+1... an$)

Here the parser has shifted both the current input symbol ai and the next symbol.

If action[sm, ai] = reduce A -> b, then the parser executes a reduce move, entering
the configuration,

(s0 X1 s1 X2 s2... Xm-r sm-r A s, ai ai+1... an$)

where s = goto[sm-r, A] and r is the length of b, the right side of the production. The parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state sm-r. The parser then pushed both A, the left side of the production, and s, the entry for goto[sm-r, A], onto the stack. The current input symbol is not changed in a reduce move.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production reduced.

If action[sm, ai] = accept, parsing is completed.

SHIFT REDUCE PARSING

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS,replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input. The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

The operation of the parser is controlled by a couple of tables:

ACTION TABLE

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state s and the current lookahead terminal is t, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

Shift s'

Shift state s' onto the parse stack. Reduce r

Reduce by rule r. This is explained in more detail below. Accept

Terminate the parse with success, accepting the input. Error

Signal a parse error

GOTO TABLE

The goto table is a table with rows indexed by states and columns indexed by nonterminal symbols. When the parser is in state s immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1. Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

2. Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

· If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

· If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r. Then consult the goto table and push the state given by goto[s'][N] onto the stack. The lookahead token is not changed by this step.

- If the action table entry is accept, then terminate the parse with success.
- If the action table entry is error, then signal an error.

3. Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

0) $S: stmt <EOF>

1) stmt: ID ':=' expr

2) expr: expr '+' ID

3) expr: expr '-' ID

4) expr: ID

which describes assignment statements like a:= b + c - d. (Rule 0 is a special augmenting production added to the grammar).

One possible set of shift-reduce parsing tables is shown below (sn denotes shift n, rn denotes reduce n, acc denotes accept and blank entries denote error entries):

Parser Tables

Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		

A trace of the parser on the input a:= b + c - d is shown below:

Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3
0/$S 1/a 3/:=	<EOF>	g6 on expr
0/$S 1/a 3/:= 6/expr	<EOF>	r1
0/$S	<EOF>	g2 on stmt
0/$S 2/stmt	<EOF>	s4
0/$S 2/stmt 4/<EOF>		*accept*

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

SLR PARSER

An *LR(0) item* (or just *item*) of a grammar *G* is a production of *G* with a dot at some position of the right side indicating how much of a production we have seen up to a given point.

For example, for the production E -> E + T we would have the following items:

[E -> .E + T]

[E -> E. + T]

[E -> E +. T]

[E -> E + T.]

Stack	State	Comments
Empty	[E'-> .E]	can't go anywhere from here
	e-transition	so we follow an e-transition
Empty	[F -> .(E)]	now we can shift the (
(	[F -> (.E)]	building the handle (E); This state says: "I have (on the stack and expect the input to give me tokens that can eventually be reduced to give me the rest of the handle, E)."

CONSTRUCTING THE SLR PARSING TABLE

To construct the parser table we must convert our NFA into a DFA. The states in the LR table will be the e-closures of the states corresponding to the items SO...the process of creating the LR state table parallels the process of constructing an equivalent DFA from a machine with e-transitions. Been there, done that - this is essentially the subset construction algorithm so we are in familiar territory here.

We need two operations:
closure() and goto().

closure()

If I is a set of items for a grammar *G*, then closure(I) is the set of items constructed from I by the two rules: Initially every item in I is added to closure(I)

If *A* -> a.*B*b is in closure(I), and *B* -> g is a production, then add the initial item [*B* -> .g] to I, if it is not already there. Apply this rule until no more new items can be added to closure(I).
From our grammar above, if I is the set of one item {[E'-> .E]}, then closure(I) contains:

I0: E' -> .E

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

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

F -> .(E)

F -> .id
goto()

goto(I, *X*), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items [*A* -> a*X*.b] such that [*A* -> a.*X*b] is in I. The idea here is fairly intuitive: if I is the set of items that are valid for some viable prefix g, then goto(I, *X*) is the set of items that are valid for the viable prefix g*X*.

SETS-OF-ITEMS-CONSTRUCTION

To construct the canonical collection of sets of LR(0) items for
augmented grammar G'.

procedure items(G')
begin
C := {closure({[S' -> .S]})};
repeat

for each set of items in C and each grammar symbol X
such that goto(I, X) is not empty and not in C do

add goto(I, X) to C;

until no more sets of items can be added to C
end;

ALGORITHM FOR CONSTRUCTING AN SLR PARSING TABLE

Input: augmented grammar G'

Output: SLR parsing table functions action and goto for G'

Method:

Construct C = {I0, I1 , ..., In} the collection of sets of LR(0) items for
G'. State i is constructed from Ii:

if [A -> a.ab] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.
if [A -> a.] is in Ii, then set action[i, a] to "reduce A -> a" for all a in FOLLOW(A). Here A may not be S'.

if [S' -> S.] is in Ii, then set action[i, $] to "accept"

If any conflicting actions are generated by these rules, the grammar is not SLR(1) and the algorithm fails to produce a parser. The goto transitions for state i are constructed for all *nonterminals A* using the rule: If goto(Ii, A)= Ij, then goto[i, *A*] = j.

All entries not defined by rules 2 and 3 are made "error".

The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S]. Let's work an example to get a feel for what is going on,

An Example

(1) E -> E * B

(2) E -> E + B

(3) E -> B

(4) B -> 0

(5) B -> 1

The Action and Goto Table The two LR(0) parsing tables for this grammar look as follows:

	action					*goto*	
state	*****	**+**	**0**	**1**	**$**	**E**	**B**
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

CANONICAL LR PARSING

By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle a for which there is a possible reduction to *A*. As the text points out, sometimes the FOLLOW sets give too much informationand doesn't (can't) discriminate between different reductions.

The general form of an LR(k) item becomes [A -> a.b, s] where A -> ab is a production and s is a string of terminals. The first part (A -> a.b) is called the core and the second part is the lookahead. In LR(1) |s| is 1, so s is a single terminal.

A -> ab is the usual righthand side with a marker; any a in s is an incoming token in which we are interested. Completed items used to be reduced for every incoming token in FOLLOW(A), but now we will reduce only if the next input token is in the lookahead set s..if we get two productions A -> a and B -> a, we can tell them apart when a is a handle on the stack if the corresponding completed items have different lookahead parts. Furthermore, note that the lookahead has no effect for an item of the form [A -> a.b, a] if b is not e. Recall that our problem occurs for completed items, so what we have done now is to say that an item of the form [A -> a., a] calls for a reduction by A -> a only if the next input symbol is a. More formally, an LR(1) item [A -> a.b, a] is valid for a viable prefix g if there is a derivation

S =>* s abw, where g = sa, and either a is the first symbol of w, or w is e and a is $.

ALGORITHM FOR CONSTRUCTION OF THE SETS OF LR(1) ITEMS

Input: grammar G'

Output: sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'

Method:

closure(I)

begin

repeat

for each item [A -> a.Bb, a] in I,

each production B -> g in G',

and each terminal b in FIRST(ba)

such that [B -> .g, b] is not in I do

add [B -> .g, b] to I;

until no more items can be added to I;

end;

goto(I, X)

begin

let J be the set of items [A -> aX.b, a] such that

[A -> a.Xb, a] is in I

return closure(J);

end;

procedure items(G')

begin

C := {closure({S' -> .S, $})};

repeat

for each set of items I in C and each grammar symbol X such

that goto(I, X) is not empty and not in C do

add goto(I, X) to C

until no more sets of items can be added to C;

end;

An example,

Consider the following grammer,
S’->S
S->CC
C->cC
C->d
Sets of LR(1) items
I0: S’->.S,$
S->.CC,$
C->.Cc,c/d
C->.d,c/d

I1:S’->S.,$
I2:S->C.C,$
C->.Cc,$
C->.d,$

I3:C->c.C,c/d
C->.Cc,c/d
C->.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

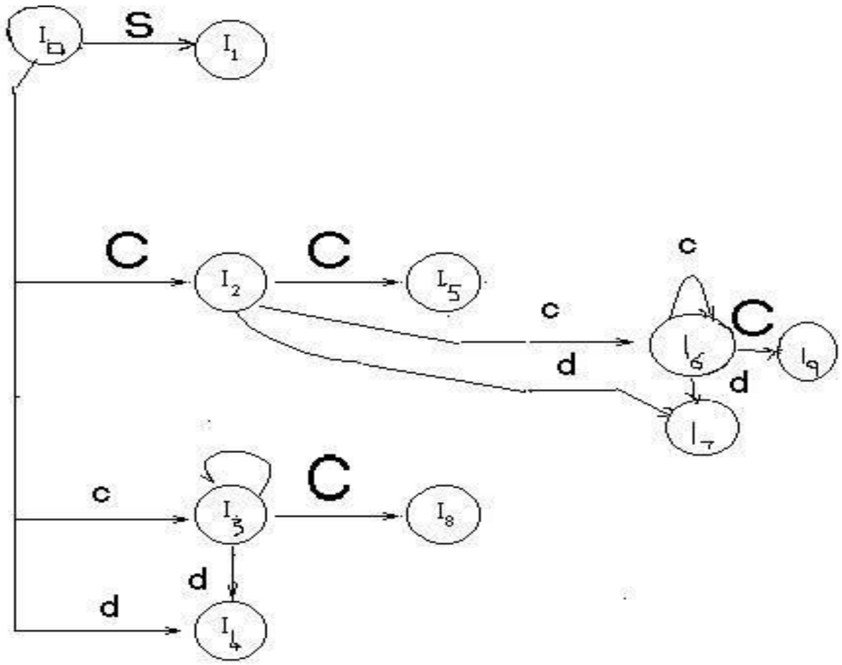
I6: C->c.C,$
C->.cC,$
C->.d,$

I7:C->d.,$

I8:C->cC.,c/d

I9:C->cC.,$

Here is what the corresponding DFA looks like



Parsing Table:state	c	d	$	S	C
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

ALGORITHM FOR CONSTRUCTION OF THE CANONICAL LR PARSING TABLE

Input: grammar G'
Output: canonical LR parsing table functions action and goto

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.State i is constructed from Ii.
2. if [*A* -> a.ab, b>] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.
3. if [*A* -> a., a] is in Ii, then set action[i, a] to "reduce *A* -> a" for all a in FOLLOW(*A*). Here *A* may *not* be *S*'.
4. if [*S*' -> *S*.] is in Ii, then set action[i, $] to "accept"
5. If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.
6. The goto transitions for state i are constructed for all *nonterminal*s *A* using the rule: If goto(Ii, A)= Ij, then goto[i, *A*] = j.
7. All entries not defined by rules 2 and 3 are made "error".
8. The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S, $].

LALR PARSER:

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of core (or first) components and differ only in their second component, the lookahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing. This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same core. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the lookahead. But it is possible for the merger to produce a reduce/reduce conflict.

Our second observation is that we are really only interested in the lookahead symbol in places where there is a problem. So our next thought is to take the LR(0) set of items and add lookaheads only where they are needed. This leads to a more efficient, but much more complicated method.

ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE

Input: G'

Output: LALR parsing table functions with action and goto for G'.

Method:

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.
2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.
3. Let C' = {J0, J1 , ..., Jm} be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as in the construction of the canonical LR parsing table.
4. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.
5. The goto table is constructed as follows: If J is the union of one or more sets of LR(1) items, that is, J = I0U I1 U ... U Ik, then the cores of goto(I0, X), goto(I1, X), ..., goto(Ik, X) are the same, since I0, I1 , ..., Ik all have the same core. Let K be the union of all sets of items having the same core asgoto(I1, X).
6. 6. Then goto(J, X) = K. Consider the above example,

I3 & I6 can be replaced by their union

I36:C->c.C,c/d/$

C->.Cc,C/D/$

C->.d,c/d/$

I47:C->d.,c/d/$

I89:C->Cc.,c/d/$

Parsing Table

state	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

UNIT 4

SYNTAX DIRECTED TRANSLATION

SYNTAX DIRECTED TRANSLATION

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - We associate Attributes to the grammar symbols representing the language constructs.
  - Values for attributes are computed by Semantic Rules associated with grammar productions.
- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.

There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).

2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

• **Syntax Directed Definitions** are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of **Attributes**;

2. Productions are associated with **Semantic Rules** for computing the values of attributes.

- Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,X.a indicates the attribute a of the grammar symbol X).

- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes

Syntax Directed Definitions: An Example

• **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

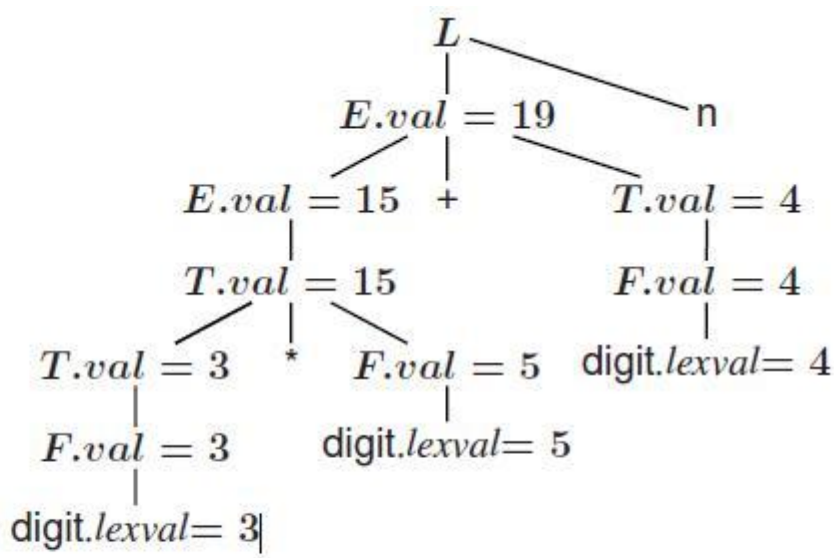
S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

• **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

• **Example.** The above arithmetic grammar is an example of an S-

Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



L-attributed definition

Definition: A SDD its *L-attributed* if each inherited attribute of Xi in the RHS of A ! X1 : :Xn depends only on

1. attributes of X1;X2; : : : ;Xi 1 (symbols to the left of Xi in the RHS)

2. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.

2. An action must not refer to synthesized attribute of any symbol to the right of that action.

3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

SYMBOL TABLES

A symbol table is a major data structure used in a compiler. Associates attributes with identifiers used in a program. For instance, a type attribute is usually associated with each identifier. A symbol table is a necessary component Definition (declaration) of identifiers appears once in a program .Use of identifiers may appear in many places of the program text Identifiers and attributes are entered by the analysis phases. When processing a definition (declaration) of an identifier. In simple languages with only global variables and implicit declarations. The scanner can enter an identifier into a symbol table if it is not already there In block-structured languages with scopes and explicit declarations:

The parser and/or semantic analyzer enter identifiers and corresponding attributes

Symbol table information is used by the analysis and synthesis phases

To verify that used identifiers have been defined (declared)

To verify that expressions and assignments are semantically correct – type checking

To generate intermediate or target code

✓

Symbol Table Interface

The basic operations defined on a symbol table include:

- allocate – to allocate a new empty symbol table
- free – to remove all entries and free the storage of a symbol table
- insert – to insert a name in a symbol table and return a pointer to its entry
- lookup – to search for a name and return a pointer to its entry
- set_attribute – to associate an attribute with a given entry
- get_attribute – to get an attribute associated with a given entry

Other operations can be added depending on requirement For example, a delete operation removes a name previously inserted Some identifiers become invisible (out of scope) after exiting a block

This interface provides an abstract view of a symbol table

Supports the simultaneous existence of multiple tables

Implementation can vary without modifying the interface

Basic Implementation Techniques

First consideration is how to insert and lookup names

Variety of implementation techniques

Unordered List

Simplest to implement

Implemented as an array or a linked list

Linked list can grow dynamically – alleviates problem of a fixed size array

Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average

Ordered List

If an array is sorted, it can be searched using binary search – $O(\log 2\ n)$

Insertion into a sorted array is expensive – $O(n)$ on average

Useful when set of names is known in advance – table of reserved words

Binary Search Tree

Can grow dynamically

Insertion and lookup are $O(\log 2\ n)$ on average

HASH TABLES AND HASH FUNCTIONS

- ✓ A hash table is an array with index range: 0 to *TableSize* – 1
- ✓ Most commonly used data structure to implement symbol tables
- ✓ Insertion and lookup can be made very fast – $O(1)$
- ✓ A hash function maps an identifier name into a table index
- ✓ A hash function, *h*(*name*), should depend solely on *name*
- ✓ *h*(*name*) should be computed quickly
- ✓ *h* should be uniform and randomizing in distributing names
- ✓ All table indices should be mapped with equal probability.
- ✓ Similar names should not cluster to the same table index

HASH FUNCTIONS

_ Hash functions can be defined in many ways . . .

_ A string can be treated as a sequence of integer words _ Several characters are fit into an integer word

_ Strings longer than one word are folded using exclusive-or or addition _ Hash value is obtained by taking integer word modulo *TableSize*

_ We can also compute a hash value character by character:

_ $h(name) = (c0 + c1 + \ldots + cn{-}1) \bmod TableSize$, where *n* is *name* length _ $h(name) = (c0 * c1 * \ldots * cn{-}1) \bmod TableSize$

_ $h(name) = (cn{-}1 + \_\_\_\ cn{-}2 + \ldots + \_\_\_\ c1 + \_\_c0))) \bmod TableSize$

_ $h(name) = (c0 * cn{-}1 * n) \bmod TableSize$

RUNTIME ENVIRONMENT

- ➢ Runtime organization of different storage locations
- ➢ Representation of scopes and extents during program execution.

- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime:
  - Generated code for various procedures and programs.

forms text or code segment of your program: size known at compile time.
  - Data objects:

Global variables/constants: size known at compile time

Variables declared within procedures/blocks: size known

Variables created dynamically: size unknown.

  - Stack to keep track of procedure activations.

Subdivide memory conceptually into code and data areas:

- Code: Program instructions
- Stack: Manage activation of procedures at runtime.
- Heap: holds variables created dynamically

SYNTAX TREES

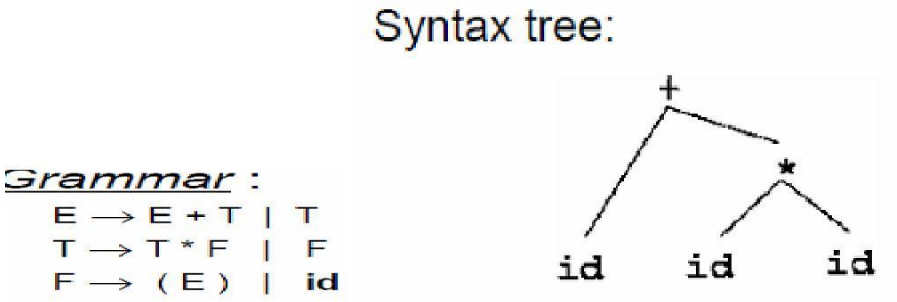
Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. Nodes represent constructs in source program and the children of a node represent meaningful components of the construct. Syntax trees are suited for static type checking.

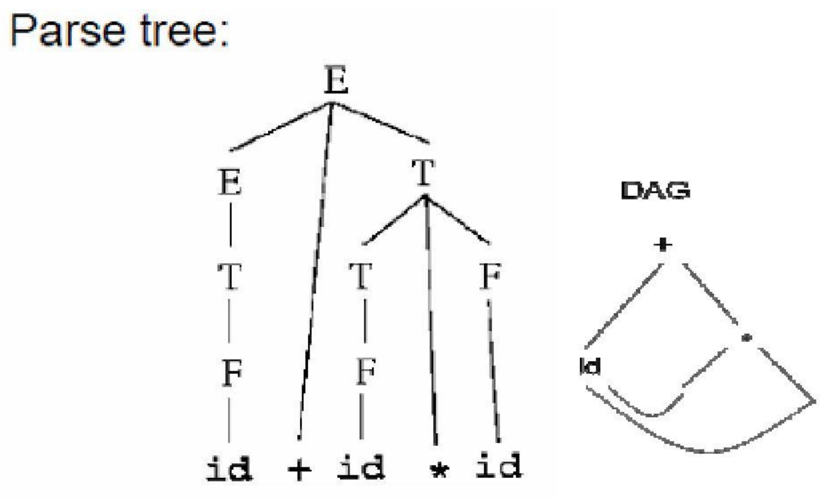
Variants of Syntax Trees: DAG

A directed acyclic graph (DAG) for an expression identifies the common sub expressions (sub expressions that occur more than once) of the expression. DAG's can be constructed by using the same techniques that construct syntax trees.

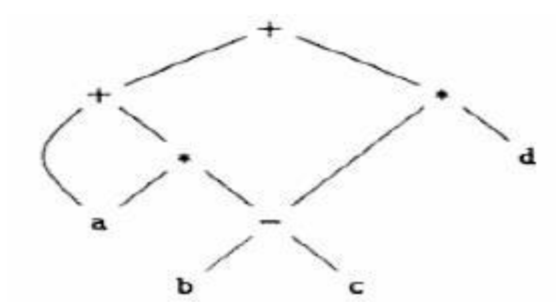
A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely. It gives clues to compiler about the generating efficient code to evaluate expressions.

Example 1: Given the grammar below, for the input string id + id * id , the parse tree, syntax tree and the DAG are as shown.





Example : DAG for the expression a + a * (b - c) + (b - c) * d is shown below.



Using the SDD to draw syntax tree or DAG for a given expression:-

• Draw the parse tree

• Perform a post order traversal of the parse tree

• Perform the semantic actions at every node during the traversal

– Constructs a DAG if before creating a new node, these functions check whether an identical node already exists. If yes, the existing node is returned.

SDD to produce Syntax trees or DAG is shown below.

	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

For the expression a + a * (b – c) + (b - c) * d, steps for constructing the DAG is as below.

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

BASIC BLOCKS

A basic block is a sequence of consecutive statements in which flow of control

enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

t1 := a*a

t2 := a*b

t3 := 2*t2

t4 := t1+t3

t5 := b*b

t6 := t4+t5

A three-address statement x := y+z is said to define x and to use y or z. A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 1: Partition into basic blocks.
Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block. Method:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are the following:

I) The first statement is a leader.

II) Any statement that is the target of a conditional or unconditional goto is a leader.

III) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example 3: Consider the fragment of source code shown in fig. 7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

begin

prod :=

0; i := 1;

do begin

prod := prod + a[i] *
b[i]; i := i+1;

end

while i<=
20 end

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic

blocks. statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block.

(1) prod := 0

(2) i := 1

(3) t1 := 4*i

(4) t2 := a [t1]

(5) t3 := 4*i

(6) t4 :=b [t3]

(7) t5 := t2*t4

(8) t6 := prod +t5

(9) prod := t6

(10) t7 := i+1
(11) i := t7

(12) if i<=20 goto (3)