ANNAMACHARYA
INSTITUTE OF TECHNOLOGY AND SCIENCES
(AUTONOMOUS)

Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapur.

Three B. Tech Programmes (CSE , ECE & CE) are accredited by NBA, New Delhi,Accredited by NAAC with 'A' Grade , Bangalore.

A-grade awarded by AP Knowledge Mission. Recognized under sections 2(f) & 12(B) of UGC Act 1956.

Venkatapuram Village, Renigunta Mandal, Tirupati, Andhra Pradesh-517520.

(COMPUTER SCIENCE AND ENGINEERING - INTERNET OF THINGS AND CYBER SECURITY INCLUDING BLOCKCHAIN TECHNOLOGY)

Academic Year 2023-24

III. B.Tech I Semster

Automata Theory and Compiler Design (20APE3603)

Prepared By

Mr. J Chandra Babu
Assistant Professor
Department of CSE, AITS

AUTOMATA & COMPILER DESIGN

UNIT - I:

Formal Language and Regular Expressions : Languages, Definition Languages regular expressions, Finite Automata – DFA, NFA. Conversion of regular expression to NFA, NFA to DFA.Applications of Finite Automata to lexical analysis, lex tools.

Context Free grammars and parsing : Context free grammars, derivation, parse trees, ambiguity LL(K) grammars and LL(1) parsing

UNIT - II:

Bottom up parsing handle pruning LR Grammar Parsing, LALR parsing, parsing ambiguous grammars, YACC programming specification.

Semantics : Syntax directed translation, S-attributed and L-attributed grammars, Intermediate code – abstract syntax tree, translation of simple statements and control flow statements.

UNIT - III:

Context Sensitive features – Chomsky hierarchy of languages and recognizers. Type checking, type conversions, equivalence of type expressions, overloading of functions and operations.

UNIT - IV:

Run time storage : Storage organization, storage allocation strategies scope access to now local names, parameters, language facilities for dynamics storage allocation.

Code optimization : Principal sources of optimization, optimization of basic blocks, peephole optimization, flow graphs, Data flow analysis of flow graphs.

UNIT - V:

Code generation : Machine dependent code generation, object code forms, generic code generation algorithm, Register allocation and assignment. Using DAG representation of Blocks

TEXT BOOKS:

1. Introduction to Theory of computation.Sipser, 2nd Edition, Thomson.
2. Compilers Principles, Techniques and Tools Aho, Ullman, Ravisethi, Pearson Education.

REFERENCES:

1. Modern Compiler Construction in C , Andrew W.Appel Cambridge University Press.
2. Compiler Construction, LOUDEN, Thomson.
3. Elements of Compiler Design, A. Meduna, Auerbach Publications, Taylor and Francis Group.
4. Principles of Compiler Design, V. Raghavan, TMH.
5. Engineering a Compiler, K. D. Cooper, L. Torczon, ELSEVIER.
6. Introduction to Formal Languages and Automata Theory and Computation - Kamala Krithivasan and Rama R, Pearson.
7. Modern Compiler Design, D. Grune and others, Wiley-India.
8. A Text book on Automata Theory, S. F. B. Nasir, P. K. Srimani, Cambridge Univ. Press.
9. Automata and Language, A. Meduna, Springer.

INDEX

UNIT -1

Fundamentals

Symbol – An atomic unit, such as a digit, character, lower-case letter, etc. Sometimesa word.*[Formal language does not deal with the "meaning" of thesymbols.]*

Alphabet – A finite set of symbols, usually denoted byΣ.
Σ ={0, 1}
Σ = {0, a,9, 4}
Σ = {a, b, c,d}

String – A finite length sequence of symbols, presumably from some alphabet. w=0110
y=0aa
x=aabcaa
z = 111

Special string: ε (also denoted by λ)
Concatenation: wz = 0110111
Length: |w| = 4 |ε| = 0 |x| = 6
Reversal: y^R = aa0

Some special sets ofstrings:
Σ^* All strings of symbols fromΣ
Σ^+ Σ^* -{ε}

Example: Σ = {0,1}
Σ^* = {ε, 0, 1, 00, 01, 10, 11, 000, 001,…}
Σ^+ = {0, 1, 00, 01, 10, 11, 000, 001,…}

A **language**is:
A set of strings from some alphabet (finite or infinite). In otherwords,
Any subset L ofΣ^*

Some speciallanguages:
{}The empty set/language, containing nostring.
{ε}A language containing one string, the emptystring.

Examples:
Σ = {0,1}
L = {x | x is in Σ^* and x contains an even number of 0"s}

Σ = {0, 1, 2,…, 9, .}
L = {x | x is in Σ^* and x forms a finite length real number}

= {0, 1.5, 9.326,…}

Σ = {a, b, c,…, z, A, B,…, Z}
L = {x | x is in Σ^* and x is a Pascal reserved word}

= {BEGIN, END, IF,…}

Σ = {Pascal reserved words} U { (,), ., :, ;,…} U {Legal Pascal identifiers} L = {x | x is in Σ^* and x is a syntactically correct Pascal program}

Σ = {English words}
L = {x | x is in Σ^* and x is a syntactically correct English sentence}

Regular Expression

- A regular expression is used to specify a language, and it does soprecisely.
- Regular expressions are veryintuitive.
- Regular expressions are very useful in a variety ofcontexts.
- Given a regular expression, an NFA-ε can be constructed from itautomatically.
- Thus, so can an NFA, a DFA, and a corresponding program, allautomatically!

Definition:

Let Σ be an alphabet. The regular expressions over Σare:
Ø Represents the empty set {}
E Represents the set{ε}
Represents the set {a}, for any symbol a inΣ
Let r and s be regular expressions that represent the sets R and S, respectively.
r+sRepresents the set RUS (precedence3)
rsRepresents thesetRS (precedence2)
r^* Represents thesetR* (highest precedence)
(r) Represents thesetR (not an op, providesprecedence)
If r is a regular expression, then L(r) is used to denote the correspondinglanguage.

Examples:

Let Σ = {0,1}
(0 +1)* All strings of 0‟s and1‟s0(0 +1)* All strings of 0‟s and 1‟s, beginning with a0
(0 +1)*1 All strings of 0‟s and 1‟s, ending with a1
(0 + 1)*0(0+1)* All strings of 0‟s and 1‟s containing at least one 0 (0 + 1)*0(0 + 1)*0(0+1)* All strings of 0‟s and 1‟s containing at least two
0‟s (0+1)*01*01* All strings of 0‟s and 1‟s containing at least two
0‟s (101*0)* All strings of 0‟s and 1‟s containing an even number of 0‟s
1*(01*01*)* All strings of 0‟s and 1‟s containing an even number of 0‟s
(1*01*0)*1* All strings of 0‟s and 1‟s containing an even number of0‟s

Identities:

1. Øu = uØ=Ø Multiply by0
2. εu = uε=u Multiply by1
3. Ø* =ε
4. ε* =ε
5. u+v =v+u
6. u + Ø =u
7. u + u = u
8. u* =(u*)*
9. u(v+w) =uv+uw

10. (u+v)w =uw+vw

11. (uv)*u = u(vu)*

12. (u+v)* = (u*+v)*

=u*(u+v)*

=(u+vu*)*
= (u*v*)*

=u*(vu*)*

=(u*v)*u*

Finite State Machines

A finite state machine has a set of states and two functions called the next-state function and the outputfunction
The set of states correspond to all the possible combinations of the internal storage
If there are n bits of storage, there are 2^n possiblestates
The next state function is a combinational logic function that given the inputs and the current state, determines the next state of thesystem
The output function produces a set of outputs from the current state and theinputs

- There are two types of finite statemachines
- In a Moore machine, the output only depends on the currentstate
- While in a Mealy machine, the output depends both the current state and the currentinput
- We are only going to deal with the Mooremachine.
- These two types are equivalent incapabilities

A Finite State Machine consistsof:
Kstates:S = {s_1, s_2, … ,s_k}, s_1 is initial state Ninputs:I = {i_1, i_2, …,i_n}
Moutputs:O = {o_1, o_2, …,o_m}
Next-state function T(S, I) mapping each current state and input to next state Output Function P(S) specifies output

Finite Automata

- Two types – both describe what are called regularlanguages
    - Deterministic (DFA) – There is a fixed number of states and we can only bein one state at atime

- Nondeterministic (NFA) –There is a fixed number of states but wcan bein multiple states at onetime

- While NFA‟s are more expressive than DFA‟s, we will see that addingnondeterminism does not let us define any language that cannot be defined by aDFA.

- One way to think of this is we might write a program using a NFA, but then when it is “compiled” we turn the NFA into an equivalentDFA.

Formal Definition of a Finite Automaton

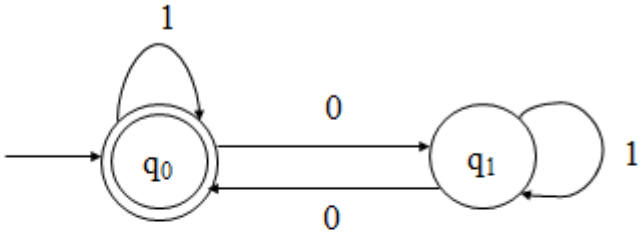
- Finite set of states, typicallyQ.
- Alphabet of input symbols, typically∑
- One state is the start/initial state, typically q0 // q0 ∈Q
- Zero or more final/accepting states; the set is typically F. // F⊆Q
- A transition function, typicallyδ. Thisfunction
- Takes a state and input symbol asarguments.

Deterministic Finite Automata (DFA)

- A DFA is a five-tuple: M = (Q, Σ, δ, q0, F)
  Q=A finite set ofstates
  Σ=A finite inputalphabet
  q0=The initial/starting state, q0 is inQ
  F=A set of final/accepting states, which is a subset ofQ
  Δ=A transition function, which is a total function from Q x Σ toQ
  δ: (Q x Σ)–>Q δ is defined for any q in Q and s in Σ, and δ(q,s)=q‟‟is equal to another state q‟‟ inQ.
  Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q.

- For Example #1:

$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

- Let M=(Q,Σ,δ,q,F)beaDFAandletwbeinΣ*.Thenwis*accepted*byMiff
  0

  $\delta(q_0, w) = p$ for some state p in F.

- Let M = (Q, Σ, δ, q , F) be a DFA. Then the *language accepted* by M is theset:
  0

  $L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } \delta(q_0, w) \text{ is in } F\}$

- Another equivalentdefinition:

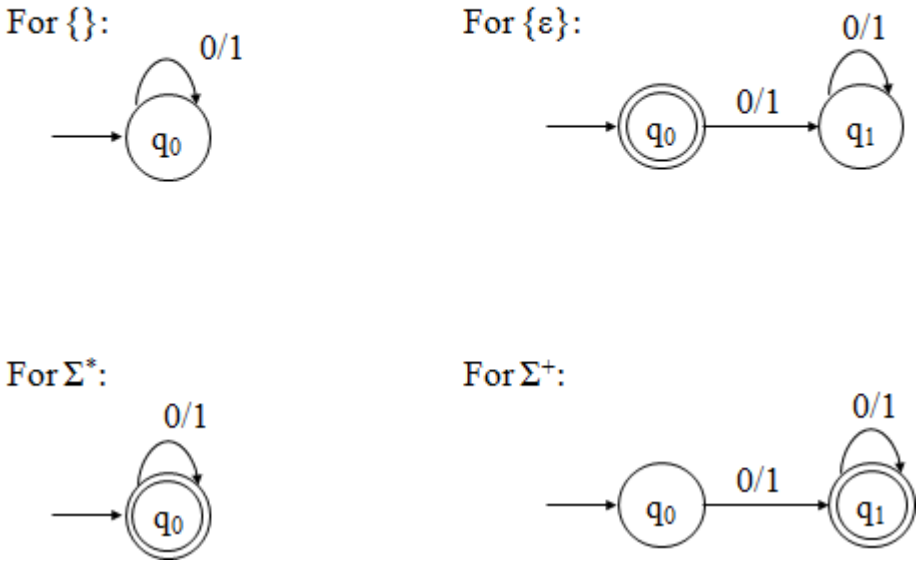
L(M) = {w | w is in Σ* and w is accepted by M}

- Let L be a language. Then L is a *regular language* iff there exists a DFA M such that L =L(M).

Notes:

- A DFA M = (Q, Σ, δ,q0,F) partitions the set Σ* into two sets: L(M)and Σ* -L(M).
- If L = L(M) then L is a subset of L(M) and L(M) is a subset ofL.
- Similarly, if L(M1) = L(M2) then L(M1) is a subset of L(M2) and L(M2) is a subset of L(M1).
- Some languages are regular, others are not. For example,if L1 = {x | x is a string of 0's and 1's containing an even number of 1's} and L2 = {x | x = 0^n1^n for some n >= 0}then L1 is regular but L2 is not.

- Let Σ = {0, 1}. Give DFAs for {}, {ε}, Σ*, and Σ+.



Nondeterministic Finite Automata (NFA)

An NFA is a five-tuple: M = (Q, Σ, δ, q_0,F)

Q	A finite set ofstates
Σ	A finite inputalphabet
q0	The initial/starting state, q0 is inQ
F	A set of final/accepting states, which is a subset ofQ
δ	A transition function, which is a total function from Q x Σ to2^Q

δ: (Q x Σ)->**2^Q** -2^Q is the power set of Q, the set of all subsets of Q δ(q,s) -The set of all states p such that there is atransition

labeled s from q to p δ(q,s) is a function from Q x S to 2^Q (but not to Q)
Let M = (Q, Σ, δ,q_0,F) be an NFA and let w be in Σ*. Then w is *accepted* by M iffδ({q_0}, w) contains at least one state inF.

Let M = (Q, Σ, δ,q_0,F) be an NFA. Then the *language accepted* by M is theset: L(M) = {w | w is in Σ* and δ({q_0},w) contains at least one state inF}
Another equivalentdefinition:
L(M) = {w | w is in Σ* and w is accepted by M}

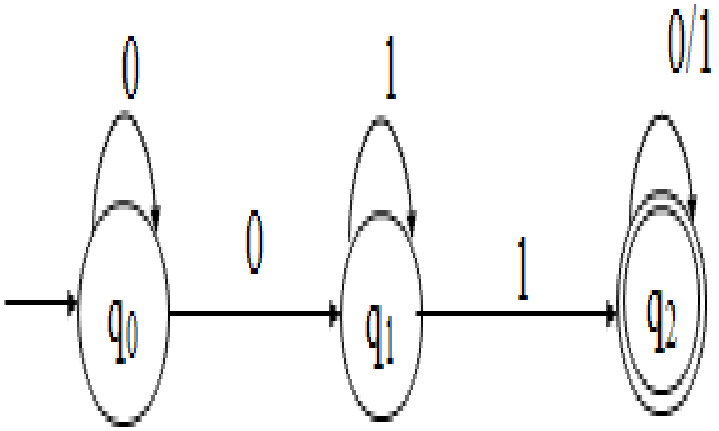
- Example: some 0's followed by some 1's

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_2\}$



δ:

	0	1
q_0	$\{q_0, q_1\}$	$\{\}$
q_1	$\{\}$	$\{q_1, q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

$= \{\epsilon, 0, 00, 1, 11, 111, 01, 10, \ldots\ldots\}$

$= \{ \epsilon$, any combination of 0's, any combination of 1's, any combination of 0 and 1 $\}$

Hence, L. H. S. = R. H. S. is proved.

3.4 RELATIONSHIP BETWEEN FA AND RE

There is a close relationship between a finite automata and the regular expression we can show this relation in below figure.

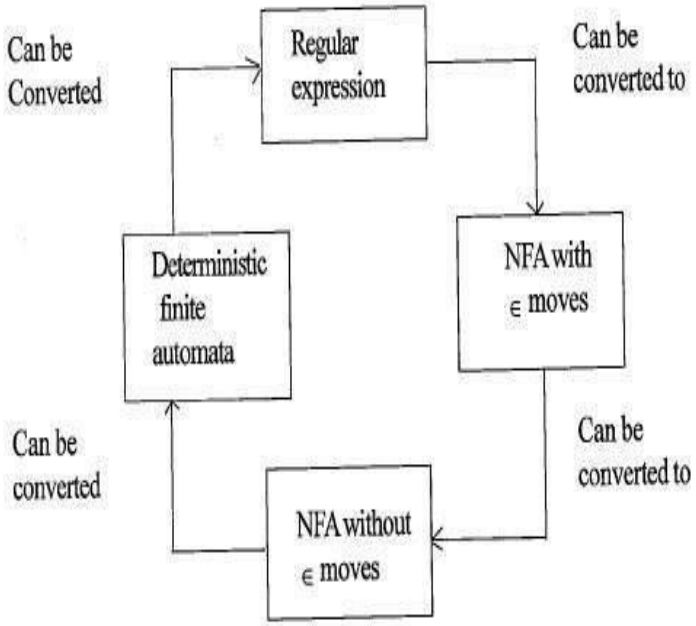


FIGURE : Relationship between FA and regular expression

The above figure shows that it is convenient to convert the regular expression to NFA with ϵ moves. Let us see the theorem based on this conversion.

3.5 CONSTRUCTING FA FOR A GIVEN REs

Theorem : If r be a regular expression then there exists a NFA with ϵ - moves, which accepts $L(r)$.

Proof : First we will discuss the construction of NFA M with ϵ - moves for regular expression r and then we prove that $L(M) = L(r)$.

Let r be the regular expression over the alphabet Σ.

Construction of NFA with ϵ - moves

Case 1 :

(i) $r = \phi$

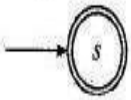
NFA $M = (\{s, f\}, \{\}, \delta, s, \{f\})$ as shown in Figure1 (a)

(No path from initial state s to reach the final state f.)

Figure 1 (a)

(ii) $r = \in$

NFA $M = (\{s\}, \{\}, \delta, s, \{s\})$ as shown in Figure 1 (b)

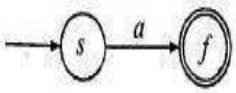


(The initial state s is the final state)

Figure 1 (b)

(iii) $r = a$, for all $a \in \Sigma$,

NFA $M = (\{s, f\}, \Sigma, \delta, s, \{f\})$



(One path is there from initial state s to reach the final state f with label a.)

Figure 1 (c)

Case 2 : $|r| \geq 1$

Let r_1 and r_2 be the two regular expressions over Σ_1, Σ_2 and N_1 and N_2 are two NFA for r_1 and r_2 respectively as shown in Figure 2 (a).

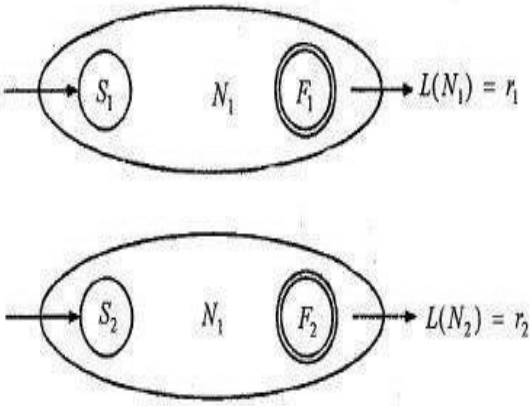
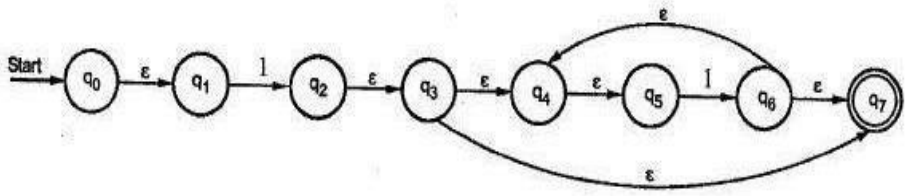


Figure 2 (a) NFA for regular expression r_1 and r_2

The final NFA is



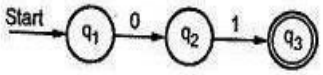
Example 4 : Construct NFA for the r. e. $(01+2^*)0$.

Solution : Let us design NFA for the regular expression by dividing the expression into smaller units

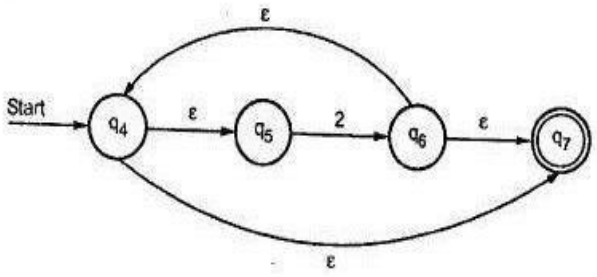
$$r = (r_1 + r_2)r_3$$

where $r_1 = 01$, $r_2 = 2^*$ and $r_3 = 0$

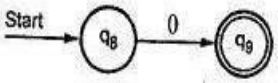
The NFA for r_1 will be



The NFA for r_2 will be



The NFA for r_3 will be



Conversion from NFA to DFA

Suppose there is an NFA N < Q, ∑, q0, δ, F > which recognizes a language L. Then the DFA D < Q‟, ∑, q0, δ‟, F‟ > can be constructed for language L as:

Step 1: Initially Q‟ = ϕ.

Step 2: Add q0 to Q‟.

Step 3: For each state in Q‟, find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q‟, add it to Q‟.

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

Example
Consider the following NFA shown in Figure 1.

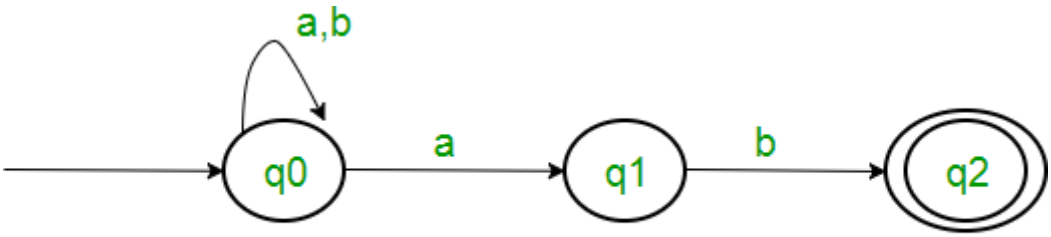


Figure 1

Following are the various parameters for NFA.
Q = { q0, q1, q2 }
∑ = (a, b)
F = { q2 }
δ (Transition Function of NFA)

State	a	b
q0	q0,q1	q0
q1		q2
q2		

Step 1: Q‟ = ϕ
Step 2: Q‟ = {q0}
Step 3: For each state in Q‟, find the states for each input symbol.
Currently, state in Q‟ is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA

δ‟ (Transition Function of DFA)

State	a	b
q0	{q0,q1}	q0

Now { q0, q1 } will be considered as a single state. As its entry is not in Q‟, add it to Q‟.
So Q‟ = { q0, { q0, q1 } }

Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like:
δ‟ ({ q0, q1 }, a) = δ (q0, a) U δ (q1, a) = { q0, q1 }
δ‟ ({ q0, q1 }, b) = δ (q0, b) U δ (q1, b) = { q0, q2 }
Now we will update the transition table of DFA.

δ‟ (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

Now { q0, q2 } will be considered as a single state. As its entry is not in Q‟, add it to Q‟.
So Q‟ = { q0, { q0, q1 }, { q0, q2 } }

Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like:
δ‟ ({ q0, q2 }, a) = δ (q0, a) U δ (q2, a) = { q0, q1 }
δ‟ ({ q0, q2 }, b) = δ (q0, b) U δ (q2, b) = { q0 }
Now we will update the transition table of DFA.

δ‟ (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., { q0, q2 }

Following are the various parameters for DFA.
Q‟ = { q0, { q0, q1 }, { q0, q2 } }
∑ = (a, b)
F = { { q0, q2 } } and transition function δ‟ as shown above. The final DFA for above NFA has been shown in Figure 2.

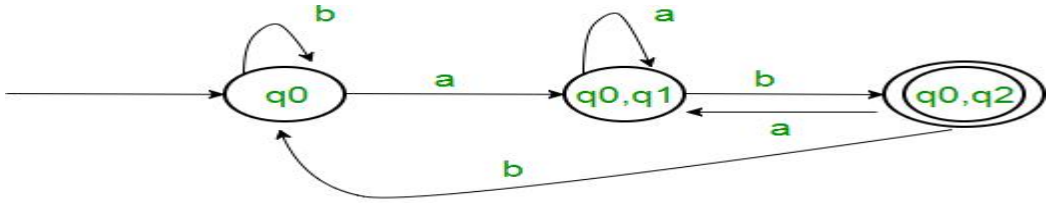


Figure 2

Note : Sometimes, it is not easy to convert regular expression to DFA. First you can convert regular expression to NFA and then NFA to DFA

Application of Finite state machine and regular expression in Lexical analysis: Lexical analysis is the process of reading the source text of a program and converting that source code into a sequence of tokens. The approach of design a finite state machine by using regular expression is so useful to generates token form a given source text program. Since the lexical structure of more or less every programming language can be specified by a regular language, a common way to implement a lexical analysis is to; 1. Specify regular expressions for all of the kinds of tokens in the language. The disjunction of all of the regular expressions thus describes any possible token in the language. 2. Convert the overall regular expression specifying all possible tokens into a deterministic finite automaton (DFA). 3. Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer. To recognize identifiers, numerals, operators, etc., implement a DFA in code. State is an integer variable, δ is a switch statement Upon recognizing a lexeme returns its lexeme, lexical class and restart DFA with next character in source code.

CONTEXT FREE-GRAMMAR

Definition: Context-Free Grammar (CFG) has 4-tuple: G = (V, T, P,S)

Where,

V -A finite set of variables or*non-terminals*
T -A finite set of *terminals* (V and T do not intersect)
P -A finite set of *productions,* each of the form A –>α,
Where A is in V and α is in (V U T)*
Note: that α may be ε.
S -A starting non-terminal (S is inV)

Example :CFG:

G = ({S}, {0, 1}, P, S) P:
S–>0S1 or just simply S –> 0S1 |ε
S –>ε
ExampleDerivations:

S	=> 0S1	(1)
S	=> ε	(2)
	=> 01	(2)
S	=> 0S1	(1)
	=> 00S11	(1)
	=> 000S111	(1)
	=> 000111	(2)

- Note that G "generates" the language $\{0^k1^k \mid k>=0\}$

Derivation (or Parse) Tree

- **Definition:** Let G = (V, T, P, S) be a CFG. A tree is a derivation (or parse) treeif:
  - Every vertex has a label from V U T U{ε}
  - The label of the root isS
  - If a vertex with label A has children with labels X_1, X_2,…, X_n, from left to right, then

synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

Terminals

A –> $X_1, X_2, \ldots, X_n$ must be a production in P

The first L stands for "Left-to-right scan of input". The second L stands for "Left-most derivation". The „1"

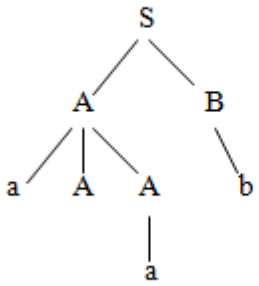
stands for "1 token of look ahead".

No LL (1) grammar can be ambiguous or left recursive.

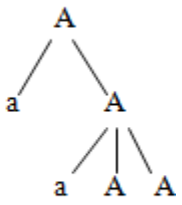
LL (1) Grammar:

- If a vertex has label ε, then that vertex is a leaf and the only child of its"parent
- More Generally, a derivation tree can be defined with any non-terminal as theroot.
- **Example:**

S –> AB
A –> aAA
A –> aA
A –> a
B –> bB
B –> b



yield = aAab



yield = aaAA

Notes:

- Root can be anynon-terminal
- Leaf nodes can be terminals ornon-terminals

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of

- A derivation tree with root S shows the productions used to obtain a sentential form.

LL(k)

LL(k) grammar performs a top-down, leftmost parse after reading the string from left-to-right Here, kk is the number of look-aheads allowed.
With the knowledge of kk look-aheads, we
calculate FIRSTkFIRSTk and FOLLOWkFOLLOWk where:

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

- FIRSTkFIRSTk: kk terminals that can be at the beginning of a derived non-terminal
- FOLLOWkFOLLOWk: kk terminals that can come *after* a derived non-terminal

The basic idea is to create a lookup table using this information from which the parser can then simply go and check what derivation is to be made given a certain input token.

Now, the following text from here explains strong LL(k)LL(k):
In the general case, the LL(k)LL(k) grammars are quite difficult to parse directly. This is due to the fact that the left context of the parse must be remembered somehow.
Each parsing decision is based both on what is to come as well as on what has

already been seen of the input.

The class of LL(1)LL(1) grammars are so easily parsed because it is strong. The strong LL(kLL(k) grammars are a subset of the LL(k)LL(k) grammars that can be parsed *without* knowledge of the left-context of the parse. That is, each parsing decision is based only on the next k tokens of the input for the current nonterminal that is being expanded.
Formally,

A grammar (G=N,T,P,S)(G=N,T,P,S) is strong if for any two distinct A-productions in the grammar:
A→αA→α
A→βA→β
FIRSTk(αFOLLOWk(A))∩FIRSTk(βFOLLOWk(A))=∅FIRSTk(αFOLLOWk(A))∩FIRSTk(βFOLLOWk(A))=∅
That looks complicated so we‟ll see it another way. Let‟s take a textbook example to understand, instead, when is some grammar “weak” or when exactly would we need to know the left-context of the parse.

S→aAaS→aAa
S→bAbaS→bAba
A→bA→b
A→ϵA→ϵ
Here, you‟ll notice that for an LL(2)LL(2) instance, baba could result from either of the SSproductions. So the parser needs some left-context to decide whether baba is produced by S→aAaS→aAa or S→bAbaS→bAba.
Such a grammar is therefore “weak” as opposed to being a strong LL(k)LL(k) grammar.

Unit-II

BOTTOM UPPARSING:

Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

(The point of parsing is to construct a derivation. A derivation consists of a series of rewrite steps)

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow - - - \Rightarrow r_{n-}$

$\longleftarrow$

$_1 \Rightarrow r_n \Rightarrow$ sentence Bottom-up

Assuming the production $A \rightarrow \beta$, to reduce r_i r_{i-1} match some RHS β against r_i then replace β with its corresponding LHS, A.In terms of the parse tree, this is working from leaves to root.

Example – 1:

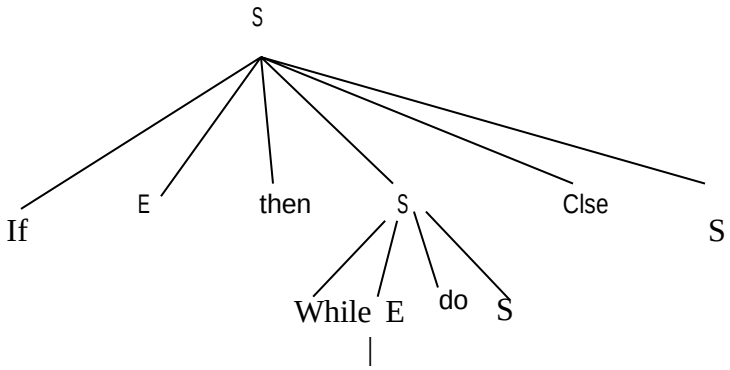
S→if E then S else S/while E do S/

print E→ true/ False/id

Input: if id then while true do print else print.

Parse tree:

Basicidea: Given input string a, "reduce" it to the goal (start) symbol, by looking for substring that match productionRHS.



⇒ **if E then S elseS**
lm

⇒ **if id then S elseS**
lm

⇒ **if id then while E do S elseS**
lm

⇒ **if id then while true do S elseS**
lm

⇒ **if id then while true do print elseS**
lm

$\Rightarrow_{lm}$ **if id then while true do print elseprint**

$\Leftarrow_{rm}$ **if E then while true do print elseprint**

$\Leftarrow_{rm}$ **if E then while E do print elseprint**

$\Leftarrow_{rm}$ **if E then while E do S elseprint**

$\Leftarrow_{rm}$ **if E then S elseprint**

$\Leftarrow_{rm}$ **if E then S elseS**

$\Leftarrow_{rm}$ **S**

HANDLE PRUNING:

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example:

E→E+E/E*E/(E)/id

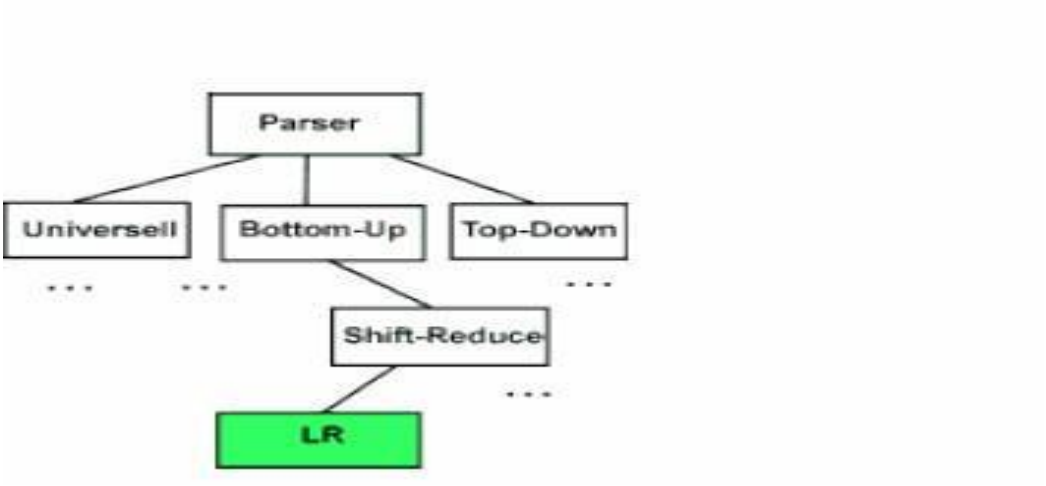
Right-sentential form	Handle	Reducing production
a+b*c	A	E→id
E+b*c	B	E→id

E+E*C	C	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step. We can use parser-generators that compute the handles for us

LR PARSINGINTRODUCTION:

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:

1. LRparsers can be constructed to recognize virtually all programming-language constructs for which context-free grammarscan be written.

2. TheLRparsing method is the most general non-backtracking shift- reduce parsing method known, yetitcanbeimplementedas efficiently as other shift-reducemethods.

3. 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 predictiveparsers.

4,AnLR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of theinput.

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 taskeasy.

LR-PARSERS:

LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are k=0 and k=1. LR(1) is of practical relevance.

„L"stands for "Left-to-right" scan of input.

„R" stands for "Rightmost derivation (in reverse)".

K"standsfornumber ofinput symbolsoflook-a-head thatareusedin makingparsingdecisions.When (K) is omitted, „K"is assumed to be 1.

LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar.

A grammar is LR(1) if, given a right-most derivation

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 - - - r_{n-1} \Rightarrow r_n \Rightarrow$ sentence.

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i.

Parser accepts input when stack contains only the start symbol and no remaining input symbol areleft.

LR(0)item: (no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.

Ex– 1: $S^{l} \rightarrow .S\$$

$S \rightarrow .$

x $S \rightarrow .(L)$

Ex-2: $A \rightarrow XYZ$ generates 4LR(0) items

$A \rightarrow .XYZ$

$A \rightarrow X.$

YZ $A \rightarrow XY.$

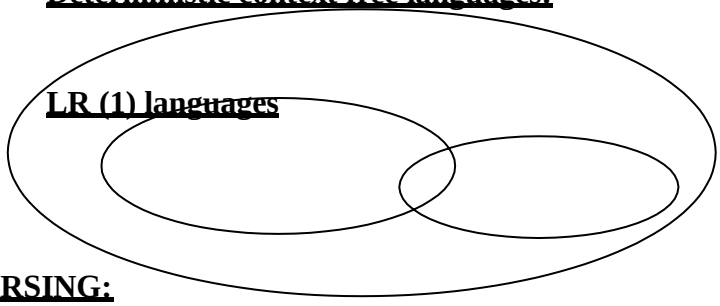
Z $A \rightarrow XYZ.$

$A \rightarrow XY.Z$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z.

→ **LR(0) items play a key role in the SLR(1) table constructionalgorithm.**

→ **LR(1) items play a key role in the LR(1) and LALR(1) table constructionalgorithms. LR parsers have more information available than LL parsers when choosing aproduction:**

- LR knowseverything derived fromRHS plus„K"lookaheadsymbols.
- **LL just knows„K"lookaheadsymbols into what"sderived fromRHS.**
- **Deterministic context free languages:**
- **LR (1) languages**



LALR PARSING:

Example:

Construct C={I0,I1,…….....,In} The collection of sets of LR(1)items

For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (clus them into a singleterm)

I_0 →same asprevious
I_1 → “
I_2 → “
I_{36} – Clubbing item I3 and I6 into one I36 item.
C →cC,c/d/$
C→cC,c/d/$
C→d,c/d/$
I_5→some as previous
I_{47}→C→d,c/d/$
I_{89}→C→cC, c/d/$

LALR Parsing table construction:

State	Action			Goto	
	c	d	$	S	C
I_o	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
89	r_2	r_2	r_2		

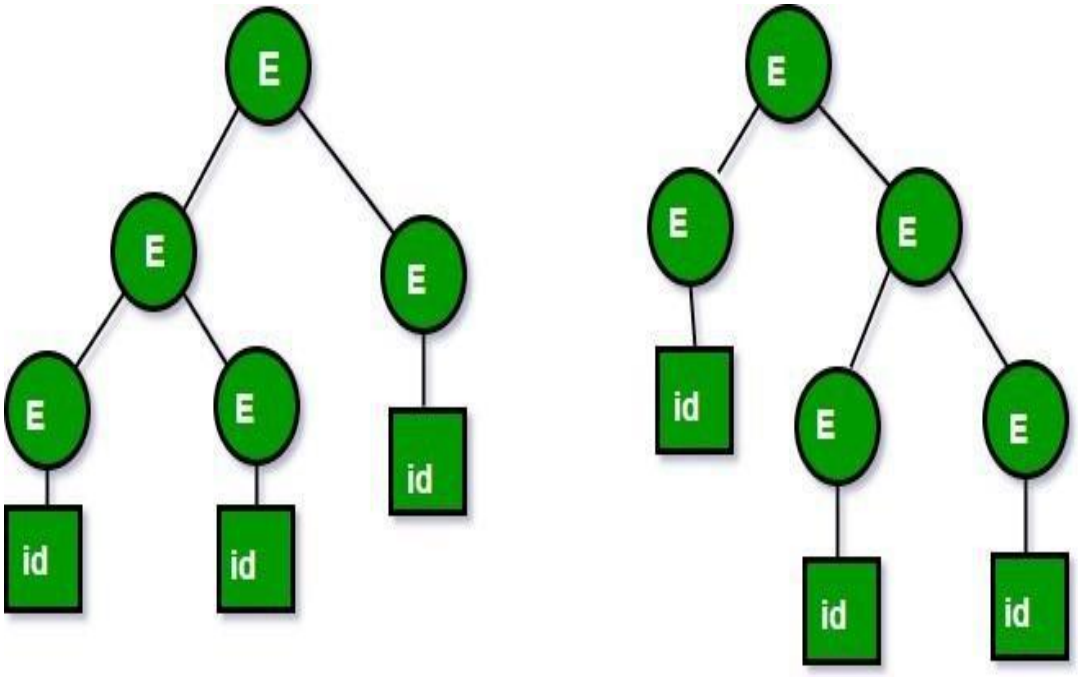
Ambiguous grammar:

A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft**M**ost **D**erivation **T**ree (LMDT) or **R**ight**M**ost **D**erivation **T**ree (RMDT).

Definition: G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than on parse tree.
where V is a finite set of variables.
T is a finite set of terminals.
P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a designated variable called the start symbol.

For Example:

1. Let us consider this grammar : **E ->E+E|id**
We can create 2 parse tree from this grammar to obtain a string **id+id+id** :
The following are the 2 parse trees generated by left most derivation:



Both the above parse trees are derived from same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

YACC PROGRAMMING

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:
YACC input file is divided in three parts.

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER

```
%token ID
```

- Yacc automatically assigns numbers for tokens, but it can be overridden by
```
%token NUMBER 621
```

- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{**and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:
```
%start nonterminal
```

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:
```
#include "lex.yy.c"
```

- YACC input file generally finishes with:
```
.y
```

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Semantics

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming languageconstructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semanticrules

Definition of (syntax Directed definition) SDD :

- SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)

Where a is an attributes obtained from the function f.

A syntax-directed definition is a generalization of a context-free grammar inwhich:

- Each grammar symbol is associated with a set ofattributes.
- Thissetofattributesforagrammarsymbolispartitionedintotwosubsetscalledsynthesized and inherited attributes of that grammar symbol.
- Each production rule is associated with a set of semanticrules.
- Semantic rules set up dependencies between attributes which can be represented by a dependencygraph.
- This dependency graph determines the evaluation order of these semanticrules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing avalue.

The two attributes for non terminalare :

The two attributes for non terminalare :
Synthesized attribute (S-attribute) :(↑)

An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node

Inherited attribute:(↑,→)

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of thatnode.

- The attribute can be string, a number, a type, a, memory location or anythingelse.
- The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.Terminals can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an Annotated parsetree.
- The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semanticrules.

Ex1:1) Synthesized Attributes : Ex: Consider the CFG :
S→ EN
E→E+T
E→E-T

E→ T
T→ T*F
T→T/F
T→F
F→(E)
F→digit N→;

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production

Productionrule	**Semanticactions**
S→EN	S.val=E.val
E→E1+T	E.val =E1.val +T.val
E→E1-T	E.val = E1.val –T.val
E→T	E.val=T.val
T→T*F	T.val = T.val *F.val
T→T|F	T.val =T.val | F.val
F→ (E)	F.val=E.val
T→F	T.val=F.val
F→digit	F.val =digit.lexval
N→;	can be ignored by lexical Analyzer as;I is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute "Val".

The taken digit has synthesized attribute "lexval".

In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.

Following steps are followed to Compute S attributed definition

Write the SDD using the appropriate semantic actions for corresponding production rule of the givenGrammar.

The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom upmanner.

The value obtained at the node is supposed to be final output.

L-attributed SDT

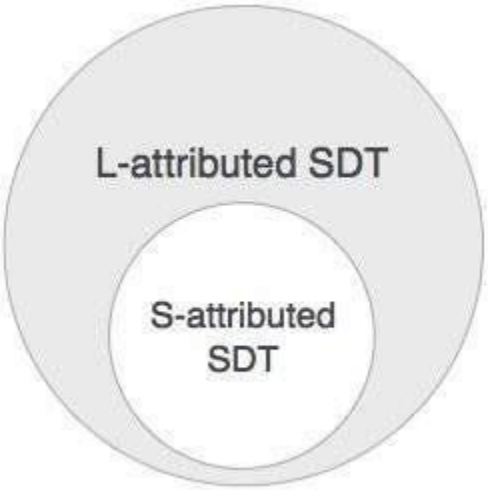
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

```
S → ABC
```

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions

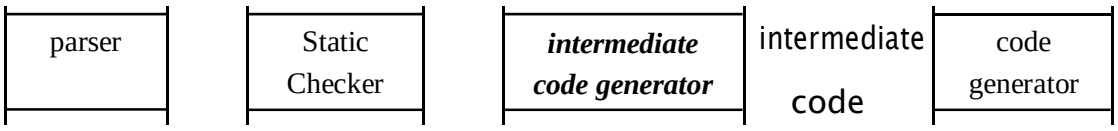
Intermediate Code

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an objectmodule.

Analysis + syntheses=translation

Createsan generate targe

code Intermediatecode



In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.

We assume that the source program has already been parsed and statically checked..the various intermediate code forms are:

a) Polishnotation
b) Abstract syntax trees(or)syntaxtrees
c) Quadruples
d) Triples three address code
e) Indirecttriples
f) Abstract machinecode(or)pseudocopde

postfix notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. the postfix (or postfix polish)notation for the same expression places the operator at the right end, asab+.

In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 is indicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfixexpression.

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming languageconstructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semanticrules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)
Where a is an attributes obtained from the function f.

- A syntax-directed definition is a generalization of a context-free grammar inwhich:
- Each grammar symbol is associated with a set ofattributes.

Thissetofattributesforagrammarsymbolispartitionedintotwosubsetscalledsynthesized and inherited attributes of that grammar symbol.

- Each production rule is associated with a set of semanticrules.

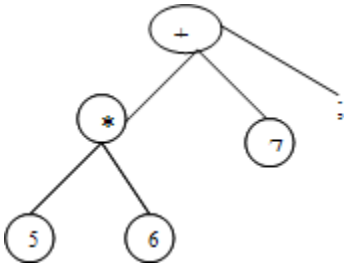
- Semantic rules set up dependencies between attributes which can be represented by a dependencygraph.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an Annotated parsetree.

- The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.Of course, the order of these computations depends on the dependency graph induced by the

Syntax tree:



Annotated parse tree :

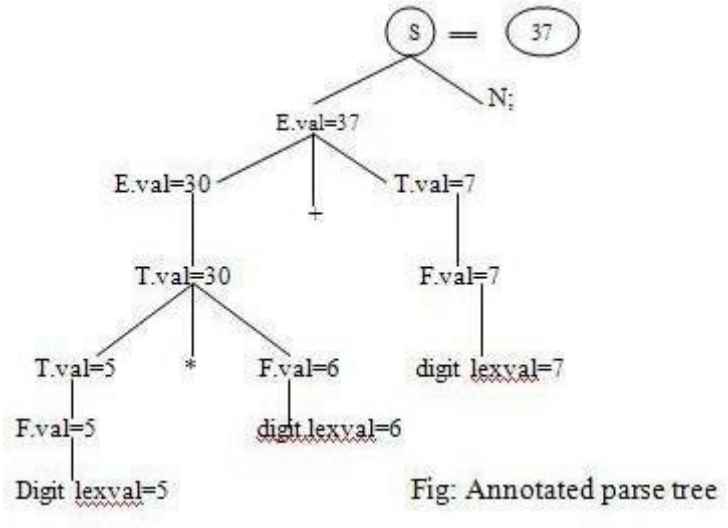


Fig: Annotated parse tree

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar. P M D

M ε
D D ; D | **id** : T | **proc id** ; N D ; S
N ε

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

S →id := E { p : = lookup (**id**.name);
ifp≠nil **then**
emit(p ' : =' E.place)
elseerror }

E → E1 + E2 { E.place : = newtemp;

emit(E.place ‘: =’ E1.place ‘ + ‘ E2.place) }

E → E1 * E2 { E.place : = newtemp;
emit(E.place ‘: =’ E1.place ‘ * ‘ E2.place) }

E →- E1 { E.place : = newtemp;
emit (E.place ‘: =’ ‘uminus’ E1.place) }

E →(E1) { E.place : = E1.place }

E → id { p : = lookup (**id**.name);
if p ≠ nil **then**
E.place : = p
else error }

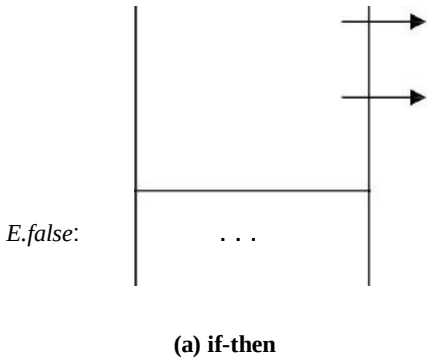
Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

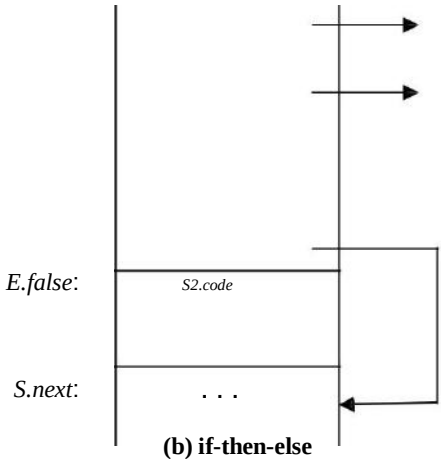
S → **if** E **then** S1
| **if** E **then** S1 **else** S2
| **while** E **do** S1

In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

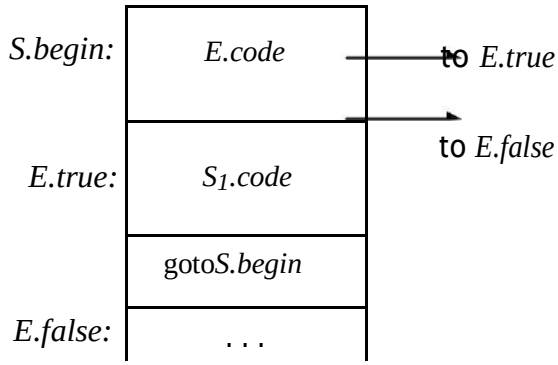
- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for **Code for if-then , if-then-else, and while-do statements**



(a) if-then



(b) if-then-else



(c) while-do

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	*E.true : = newlabel;* *E.false : = S.next;* S_1*.next : = S.next;* *S.code : = E.code || gen(E.true ,, : '') ||* S_1*.code*
$S \rightarrow$ **if** E **then** S_1 **else** S_2	*E.true : = newlabel;* *E.false : = newlabel;* S_1*.next : = S.next;* S_2*.next : = S.next;* *S.code : = E.code || gen(E.true ,, : '') ||* S_1*.code ||* *gen(,,***goto***'' S.next) ||* *gen(E.false ,, : '') ||* S_2*.code*
$S \rightarrow$ **while** E **do** S_1	*S.begin : = newlabel;* *E.true : = newlabel;* *E.false : = S.next;* S_1*.next : = S.begin;* *S.code : = gen(S.begin ,, : '')|| E.code ||* *gen(E.true ,, : '') ||* S_1*.code ||* *gen(,,***goto***'' S.begin)*

UNIT-III

Context Sensitive features – Chomsky hierarchy of languages and recognizers. Type checking, Type conversions, Equivalence of type expressions, overloading of functions and operators.

---

1. Chomsky hierarchy of languages and recognizers:-

Chomsky Hierarchy represents the class of languages that are accepted by the different machine. The category of language in Chomsky's Hierarchy is as given below:

1. Type 0 known as Unrestricted Grammar.
2. Type 1 known as Context Sensitive Grammar.
3. Type 2 known as Context Free Grammar.
4. Type 3 Regular Grammar.

Grammar type	Grammar accepted	Language accepted	Automaton
Type 0	unrestricted grammar	recursively enumerable language	Turing Machine
Type 1	context-sensitive grammar	context-sensitive language	linear-bounded automata
Type 2	Context-free grammar	Context-free language	Push down automata
Type 3	regular grammar	regular language	finite state automaton

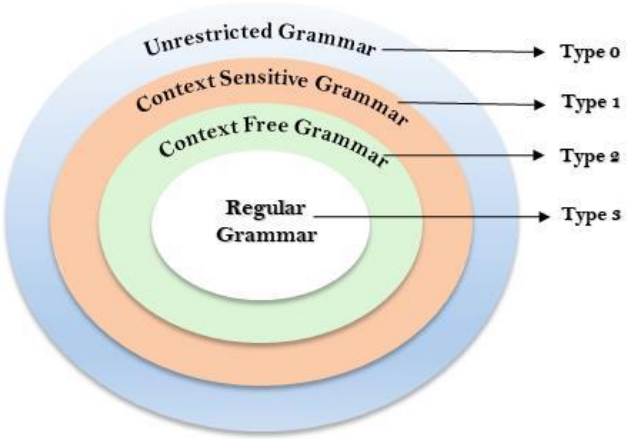


Fig: Chomsky Hierarchy

This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc.

Type 0 Grammar:

Type-0 grammars include all formal grammar. Type 0 grammar is known as unrestricted grammar. There is no restriction on the grammar rules of these types of languages. These languages can be efficiently modelled by Turing machines. These languages are also known as the Recursively Enumerable languages.

For example:

bAa → aa

S → s

Type 1 Grammar:

Type 1 grammar is known as Context Sensitive Grammar. The context sensitive grammar is used to represent context sensitive language. The language generated by the grammar is recognized by the Linear Bound Automata The context sensitive grammar follows the following rules:

- The context sensitive grammar may have more than one symbol on the left hand side of their production rules.
- The number of symbols on the left-hand side must not exceed the number of symbols on the right-hand side.
- The rule of the form $A \rightarrow \varepsilon$ is not allowed unless A is a start symbol. It does not occur on the right-hand side of any rule.
- The Type 1 grammar should be Type 0. In type 1, Production is in the form of $V \rightarrow T$ Where the count of symbol in V is less than or equal to T.

For example:

$S \rightarrow AT$

$T \rightarrow xy$

$A \rightarrow a$

Type 2 Grammar:

Type 2 Grammar is known as Context Free Grammar. Context free languages are the languages which can be represented by the context free grammar (CFG). The language generated by the grammar is recognized by Pushdown automata. Type 2 should be type 1. The production rule is of the form $A \rightarrow \alpha$

Where A is any single non-terminal and α is any combination of terminals and non-terminals.

For example:

$A \rightarrow aBb$

$A \rightarrow b$

$B \rightarrow a$

Type 3 Grammar:

Type 3 Grammar is known as Regular Grammar. Regular languages are those languages which can be described using regular expressions. These languages are exactly all languages that can be accepted by a finite-state automaton. Type 3 is the most restricted form of grammar. These languages can be modeled by NFA or DFA.

Type 3 is most restricted form of grammar. The Type 3 grammar should be Type 2 and Type 1.

Type 3 should be in the given form only:

V --> VT / T (left-regular grammar)

(or)

V --> TV /T (right-regular grammar)

---

2. Type checking:

Type checking is the process of verifying and enforcing constraints of types in values. A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language.

It allows the programmer to limit what types may be used in certain circumstances and assigns types to values. The type-checker determines whether these values are used appropriately or not.

It checks the type of objects and reports a type error in the case of a violation, and incorrect types are corrected. Whatever the compiler we use, while it is compiling the program, it has to follow the type rules of the language. Every language has its own set of type rules for the language. We know that the information about data types is maintained and computed by the compiler.

The information about data types like INTEGER, FLOAT, CHARACTER, and all the other data types is maintained and computed by the compiler. The compiler contains modules, where the type checker is a module of a compiler and its task is type checking.

Coercion/ Implicit:

Conversion from one type to another type is known as implicit if it is to be done automatically by the compiler. Implicit type conversions are also called Coercion and coercion is limited in many languages.

Example: An integer may be converted to a real but real is not converted to an integer.

Explicit:

Conversion is said to be Explicit if the programmer writes something to do the Conversion.

Types of Type Checking:

There are **two kinds** of type checking:

1. Static Type Checking.
2. Dynamic Type Checking.

In Static type checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variable is known at the compile time. It generally examines the program text during the translation of the program.

Some examples of static checks:

1. **Type checks** - A compiler should report an error if an operator is applied to an incompatible operand. **Example**: If an array variable and function variable are added together.
2. **Flow-of-control checks** - Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. **Example**: An enclosing statement, such as break, does not exist in switch statement.
3. **Name-related checks** - Sometimes the same name may appear two or more times. For **example** in Ada, a loop may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

The Benefits of Static Type Checking:

1. Runtime Error Protection.
2. It catches syntactic errors like spurious words or extra punctuation.
3. It catches wrong names like Math and Predefined Naming.
4. Detects incorrect argument types.
5. It catches the wrong number of arguments.
6. It catches wrong return types, like return “70”, from a function that’s declared to return an int.

In Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables. Implementations of dynamically

type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible. A static type system always restricts what can be conveniently expressed. A dynamic typing result in more compact programs since it is more flexible. Programming with a static type system often requires more design and implementation effort.

Languages like Pascal and C have static type checking. Type checking is used to check the correctness of the program before its execution. The main purpose of type-checking is to check the correctness and data type assignments and type-casting of the data types, whether it is syntactically correct or not before their execution. Static Type-Checking is also used to determine the amount of memory needed to store the variable.

3. Type conversions:-

The type conversion is an operation that takes a data object of one type and creates the equivalent data objects of multiple types. The signature of a type conversion operation is given as

$$conversion\_op : type1 \rightarrow type2$$

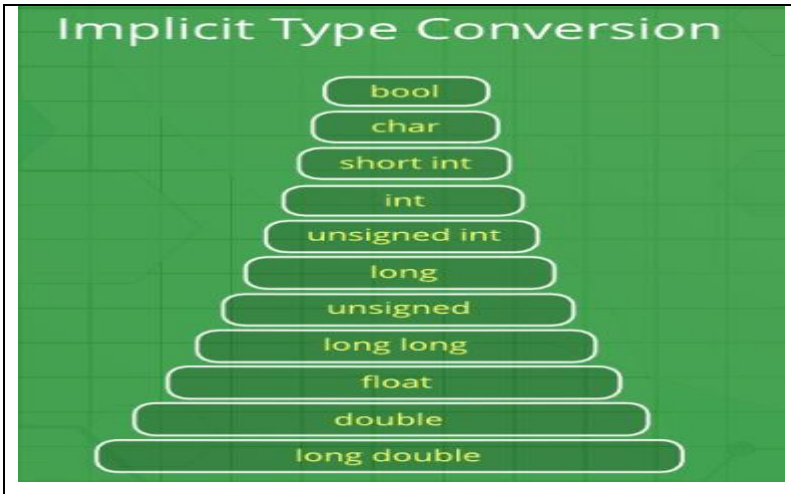
There are **two types** of type conversions which are as follows –

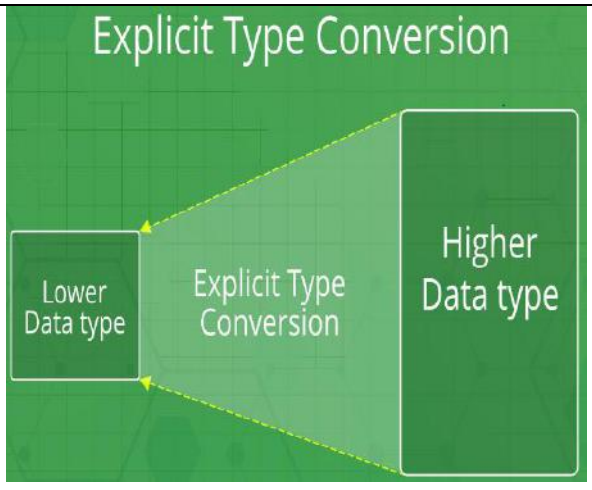
Implicit type conversion (Coercions) − The programming languages that enable mixed-mode expressions should describe conventions for implicit operand type conversions.

- Coercion is defined as an automatic conversion between types.
- For **example** in Pascal, if the operands for the addition operation are of integer type and other real types, one of then the integer data object is implicitly changed to type real earlier the addition is implemented.

Explicit type conversion − Some languages support few efficiencies for doing explicit conversions, both widening and narrowing. In some cases, warning messages are created when an explicit narrowing conversion results in a meaningful change to the value of the object being modified.

- For example, Pascal supports a built-in function round that changes a real-number data object to an integer data object with a value similar to the rounded value of the real. In C-based languages, explicit type conversions are known as casts. The desired type is located in parentheses only before the expression to be modified, as shown in (int) X for float X converts the value of X to type integer.





Advantages of Type Conversion

There are the following advantages of type conversion which are as follows –

- If during type checking, a mismatch appears between the actual type of an argument and the expected type for that operation, then type conversion easily converts the data object implicitly and prevents the error.
- In some languages such as C, type conversion is a built-in function, which implicitly casts an expression to convert it to the correct type.
- With dynamic type checking, conversions or coercions are built at the point that the type mismatch is recognized during execution. For static type checking, more code is added to the compiled program.

4. Equivalence of Type Expressions:

- If two type expressions are equal then **return** a certain type else return type error.
- **Key Ideas:**
  - The main difficulty arises from the fact that most modern languages allow the naming of user-defined types.
  - For instance, in C and C++ this is achieved by the typedef statement.
  - When checking equivalence of named types, we have two possibilities.
    - Structural Equivalence
    - Names Equivalence

Structural Equivalence of Type Expressions:

- Type expressions are built from basic types and constructors, a natural concept of equivalence between two type expressions is structural equivalence. i.e., two expressions are either the same basic type or formed by applying the same constructor to structurally equivalent types. That is, two type expressions are structurally equivalent if and only if they are identical.
- For example, the type expression **integer** is equivalent only to integer because they are the same basic type.
- Similarly, pointer (integer) is equivalent only to pointer (integer) because the two are formed by applying the same constructor pointer to equivalent types.
- The algorithm recursively compares the structure of type expressions without checking for cycles so it can be applied to a tree representation. It assumes that the only type constructors are for arrays, products, pointers, and functions.
- The constructed type array**(n1 , t1)** and array**(n2 , t2)** are equivalent if **n1 = n2** and **t1 =t2**

```
sequiv(s,t)
   if (s and t are same basic type) then
      return true
   else if ( s = array(s1 , s2) and t = array(t1 , t2)) then
      return (sequiv(s1 ,t1) and sequiv(s2 ,t2))
   else if ( s = s1 x s2 and t = t1 x t2 ) then
      return (sequiv(s1,t1) and sequiv(s2,t2))
   else if ( s = pointer (s1) and t= pointer (t1) then
      return (sequiv(s1 ,t1))
   else if ( s = s1<s2 and t = t1<t2 ) then
      return (sequiv(s1 ,t1) and sequiv(s2,t2))
   else return false
```

Names for Type Expressions:

- In some languages, types can be given names (Data type name).
- For **example**, in the Pascal program fragment.

```
Type    link  = ↑ cell;
Var     next  : link;
        last  : link;
        p     : ↑ cell;
        q, r  : ↑ cell;
```

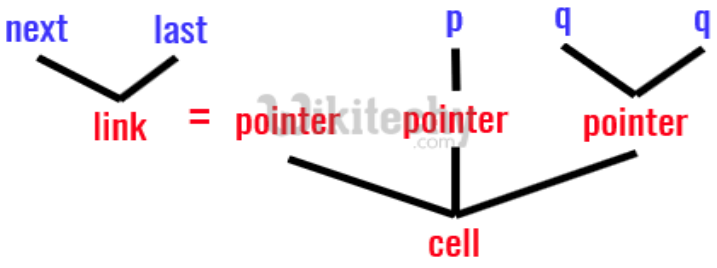
- The identifier link is declared to be a name for the type cell. The variables next, last, p, q, r are not identical type, because the type depends on the implementation.
- Type graph is constructed to check the name equivalence.
    - Every time a type constructor or basic type is seen, a new node is created.
    - Every time a new type name is seen, a leaf is created.
    - Two type expressions are equivalent if they are represented by the same node in the type graph.

Example: Consider Pascal program fragment

```
Type    link  = ↑ cell;
        np    : ↑ cell;
        nqr   : ↑ cell;
```

```
Var     next  : link;
        last  : link;
        p     : np;
        q     : nqr;
        r     : nqr;
```

- The identifier link is declared to be a name for the type ‡cell. new type names np and nqr have been introduced.
- since next and last are declared with the same type name, they are treated as having equivalent types. Similarly, q and r are treated as having equivalent types because the same implicit type name is associated with them.
- However, p, q, and next do not have equivalent types, since they all have types with different names.



Note that type name cel1 has three parents. All labeled pointer. An equal sign appears between the type name link and the node in the type graph to which it refers.

5. Function and operator overloading:

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

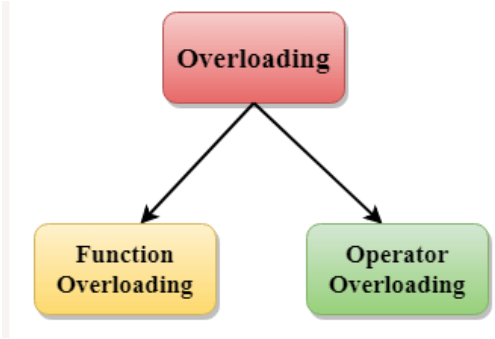
- methods,
- constructors, and

- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



Function Overloading:

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Function Overloading Example:

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
       return a + b;
   }
static int add(int a, int b, int c)
   {
       return a + b + c;
   }
};
int main(void) {
   Cal C;                                  //    class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Output:

```
30
55
```

// Program of function overloading with different types of arguments.

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   std::cout << "r1 is : " <<r1<< std::endl;
   std::cout <<"r2 is : "  <<r2<< std::endl;
   return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

Operators Overloading:

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operators that cannot be overloaded are as follows:

1. Scope operator (::)
2. Sizeof
3. member selector(.)
4. member pointer selector(*)
5. ternary operator(?:)

Syntax of Operator Overloading:

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()       {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output:

```
The Count is: 10
```

UNIT-IV

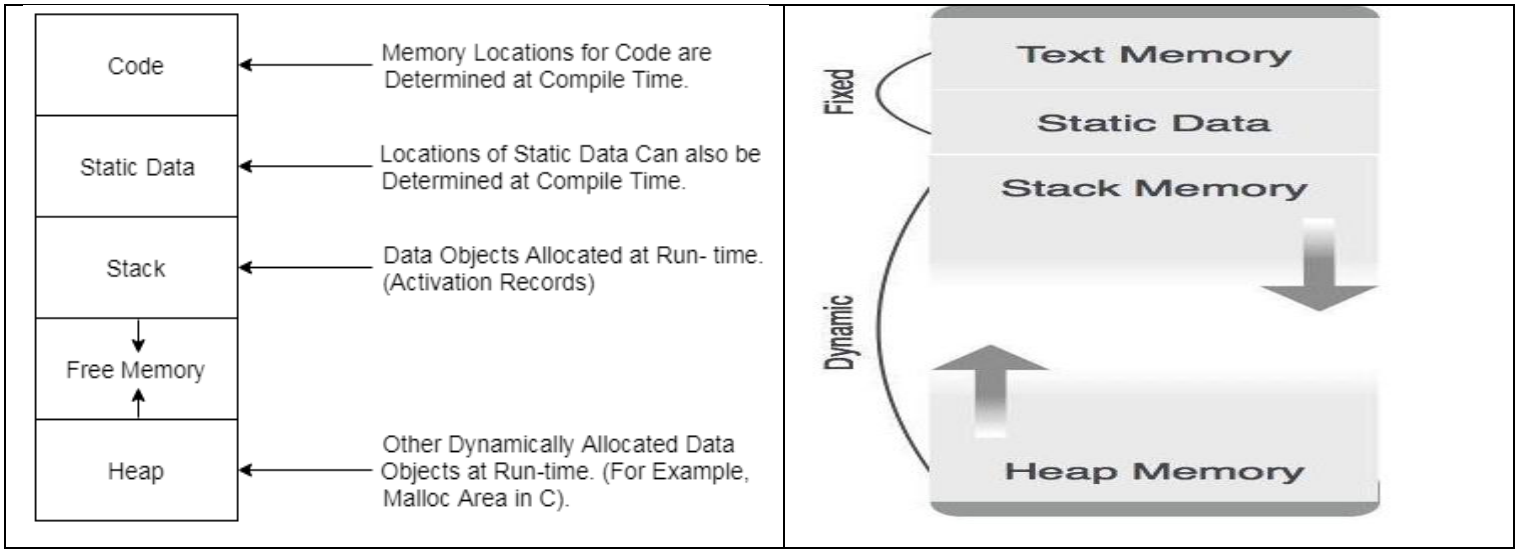
Run time storage: Storage organization, storage allocation strategies scope access to now local names, parameters, language facilities for dynamics storage allocation. **Code optimization**: Principal sources of optimization, optimization of basic blocks, peephole optimization, flow graphs, Data flow analysis of flow graphs.

---

1. Storage Organization:

- When the target program executes then it runs in its own logical address space in which the value of each program has a location.
- The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

Subdivision of Run-time Memory:



- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multi byte is stored in consecutive bytes and gives the first byte address.
- Run-time storage can be subdivide to hold the different components of an executing program:

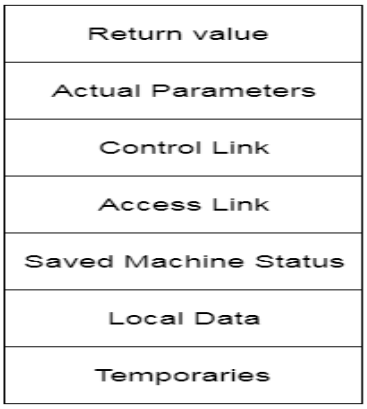
1. Generated executable code
2. Static data objects
3. Dynamic data-object- heap
4. Automatic data objects- stack

Activation Record

- Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.
- When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
- Activation record is used to manage the information needed by a single execution of a procedure.

- An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

The diagram below shows the contents of activation records:



Return Value: It is used by calling procedure to return a value to calling procedure.

Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.

Control Link: It points to activation record of the caller.

Access Link: It is used to refer to non-local data held in other activation records.

Saved Machine Status: It holds the information about status of machine before the procedure is called.

Local Data: It holds the data that is local to the execution of the procedure.

Temporaries: It stores the value that arises in the evaluation of an expression.

2. Storage allocation strategies:

The different ways to allocate memory are:

1. Static storage allocation: It is for all the data objects at compile time
2. Stack storage allocation: In this a stack is used to manage the run time storage (recursive calls make use)
3. Heap storage allocation:In this a heap is used to manage the dynamic memory allocation

Static storage allocation

- The size of data objects is known at compile time.
- The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.
- The binding of name with the amount of storage allocated do not change at run time. Hence the name of this allocation is static allocation.
- In this the compiler can determine the amount of storage required by each data object and therefore it becomes easy for a compiler to find the address of these data in the activation record.
- At compiler time compiler can fill the address at which the target code can find the data it operates on.
- Recursive procedures are not supported by this type of allocation.
- FORTRAN uses the static allocation.

Stack Storage Allocation

- In this the storage is organized as stack .This stack is also called control stack.

- As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped, when the activation end.
- The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation. The value of locals is deleted when the activation ends.
- The data structures can be created dynamically for stack allocation.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.
- The memory addressing can be done using pointers an index registers. Hence this type of allocation is slower than static allocation.

Heap Storage Allocation

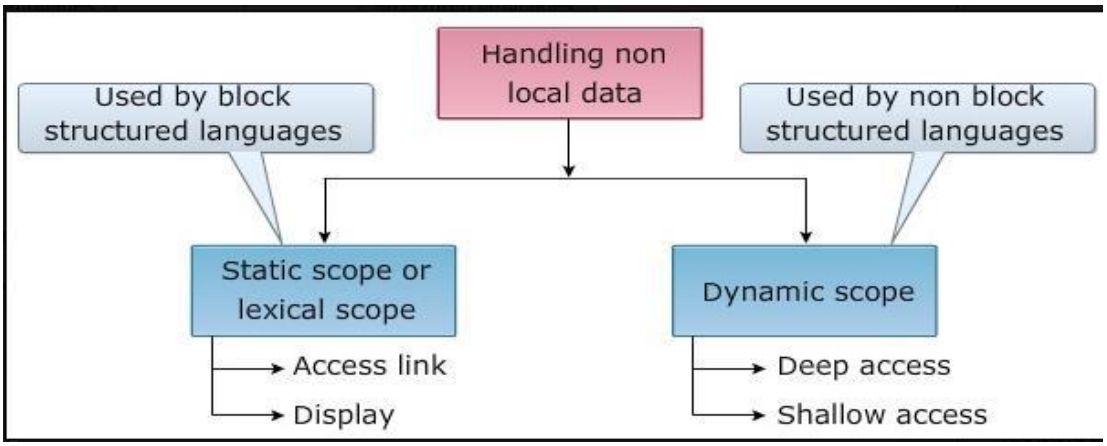
- If the values of non local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its Last in First Out nature. For retaining of such local variables heap allocation strategy is used.
- The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object, this allocated memory can be deallocated when activation ends. This deallocated space can be further reused by heap manager.
- The efficient heap management can be done by creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.
- Allocate the most suitable block of memory from the linked list i.e. use best fit technique for allocation of block.
- Free space can be further reused by heap manager
- It supports for recursion and data structures can be created at runtime
- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
- Heap storage allocation supports the recursion process.

3. Scope Access to Non-local Names:

- In some cases, when a procedure refer to variables that are not local to it, then such variables are called nonlocal variables
- There are two types of scope rules, for the non-local names. They are

  Static scope

  Dynamic scope



Static Scope or Lexical Scope

- Lexical scope is also called static scope. In this type of scope, the scope is verified by examining the text of the program.
- Examples: PASCAL, C and ADA are the languages that use the static scope rule.
- These languages are also called block structured languages

Block

- A block defines a new scope with a sequence of statements that contains the local data declarations. It is enclosed within the delimiters.

Example:

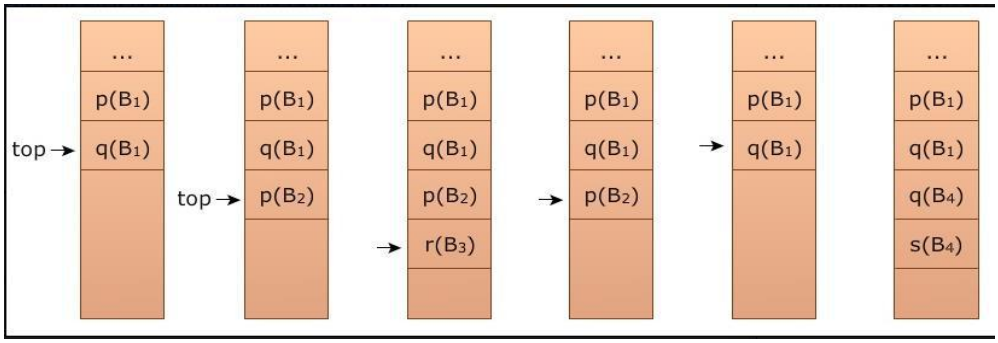
{

Declaration statements

…….

}

- The beginning and end of the block are specified by the delimiter. The blocks can be in nesting fashion that means block $_{B2}$ completely can be inside the block B_1
- In a block structured language, scope declaration is given by static rule or most closely nested loop
- At a program point, declarations are visible
  - The declarations that are made inside the procedure
  - The names of all enclosing procedures
  - The declarations of names made immediately within such procedures
- The displayed image on the screen shows the storage for the names corresponding to particular block
- Thus, block structure storage allocation can be done by stack



Lexical Scope for Nested Procedure

- If a procedure is declared inside another procedure then that procedure is known as nested procedure
- A procedure pi, can call any procedure, i.e., its direct ancestor or older siblings of its direct ancestor

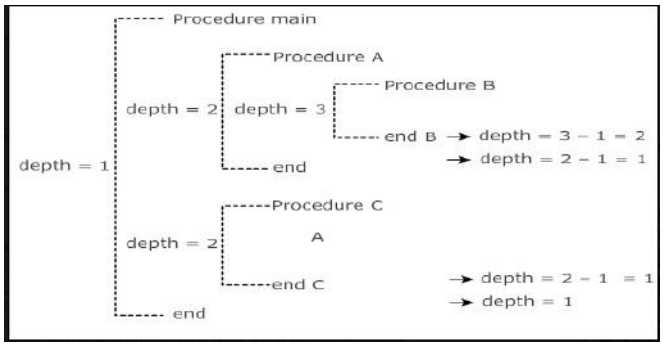
Procedure main

Procedure P1

Procedure P2

Procedure P3

Procedure P4



Nesting Depth:

- Lexical scope can be implemented by using nesting depth of a procedure. The procedure of calculating nesting depth is as follows:

The main programs nesting depth is '1'

When a new procedure begins, add '1' to nesting depth each time

When you exit from a nested procedure, subtract '1' from depth each time

The variable declared in specific procedure is associated with nesting depth

Static Scope or Lexical Scope

- The lexical scope can be implemented using access link and displays.

Access Link:

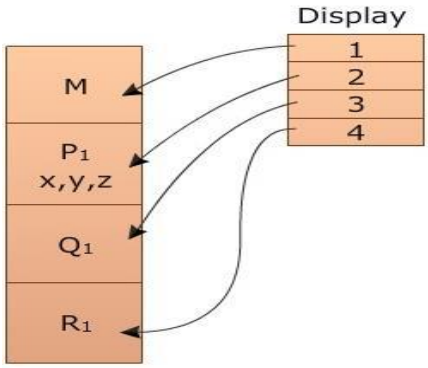
- Access links are the pointers used in the implementation of lexical scope which is obtained by using pointer to each activation record
- If procedure p is nested within a procedure q then access link of p points to access link or most recent activation record of procedure q

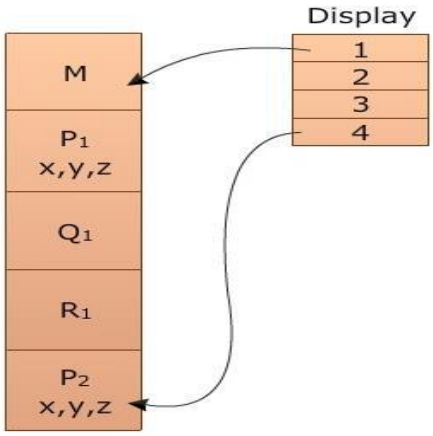
Example: Consider the following piece of code and the runtime stack during execution of the program

```
program test;
var a: int;
procedure A;
var d: int;
{
        a := 1;
}
procedure B(i: int);
var b : int;
procedure C;
var k : int;
{
        A;
}
{
        if(i<>0) then B(i-1)
        else C;
}
{
        B(1);
}
```

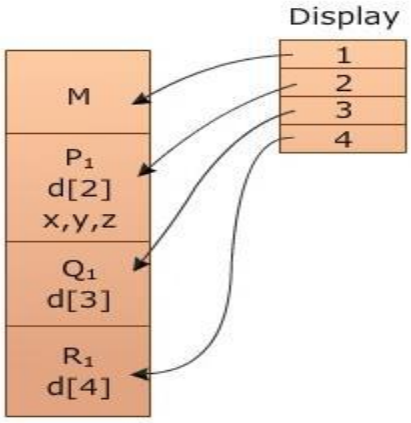
Displays:

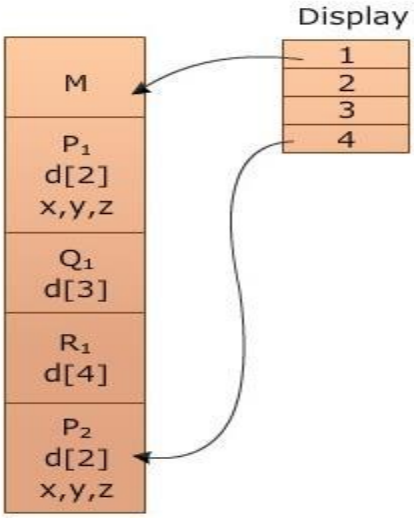
- If access links are used in the search, then the search can be slow
- So, optimization is used to access an activation record from the direct location of the variable without any search
- Display is a global array d of pointers to activation records, indexed by lexical nesting depth. The number of display elements can be known at compiler time
- d[i] is an array element which points to the most recent activation of the block at nesting depth (or lexical level)
- A nonlocal X is found in the following manner:
- Use one array access to find the activation record containing X. if the most-closely nested declaration of X is at nesting depth I, the d[i] points to the activation record containing the location for X
- Use relative address within the activation record





How to maintain display information?





- When a procedure is called, a procedure ‘p’ at nesting depth ‘i’ is setup:

  Save value of d[i] in activation record for ‘p’

  ‘I’ set d[i] to point to new activation record

- When a ‘p’ returns:

  Reset d[i] to display value stored

4. Parameter:

Among Procedures or Functions, the communication will be done by using parameters or arguments through values/references

Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value:

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

l-value:

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

Types of Parameters:

a. Actual Parameters
b. Formal Parameters

In **Formal Parameters**,Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

In **Actual Parameters**, Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

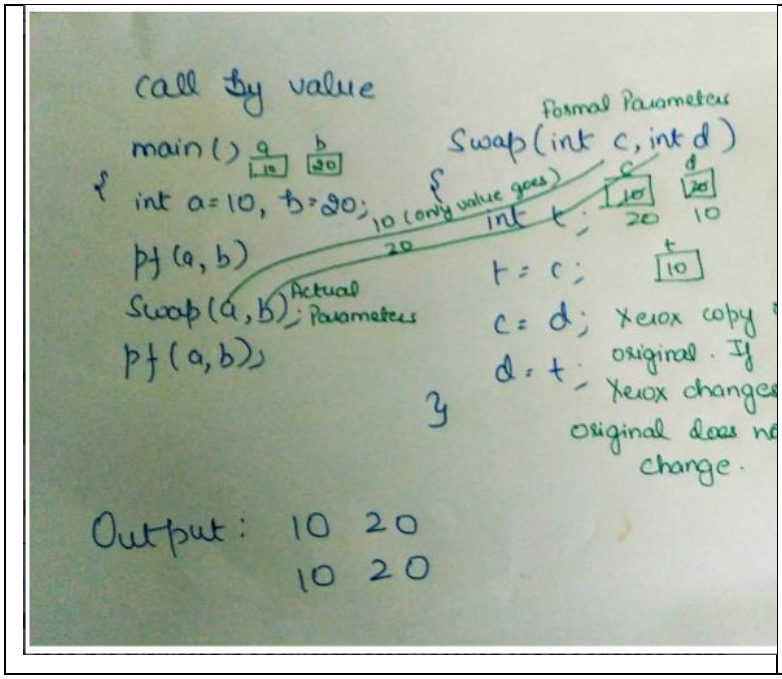
```
fun_one()
{
   int actual_parameter = 10;
   call fun_two(int actual_parameter);
}
   fun_two(int formal_parameter)
{
   print formal_parameter;
}
```

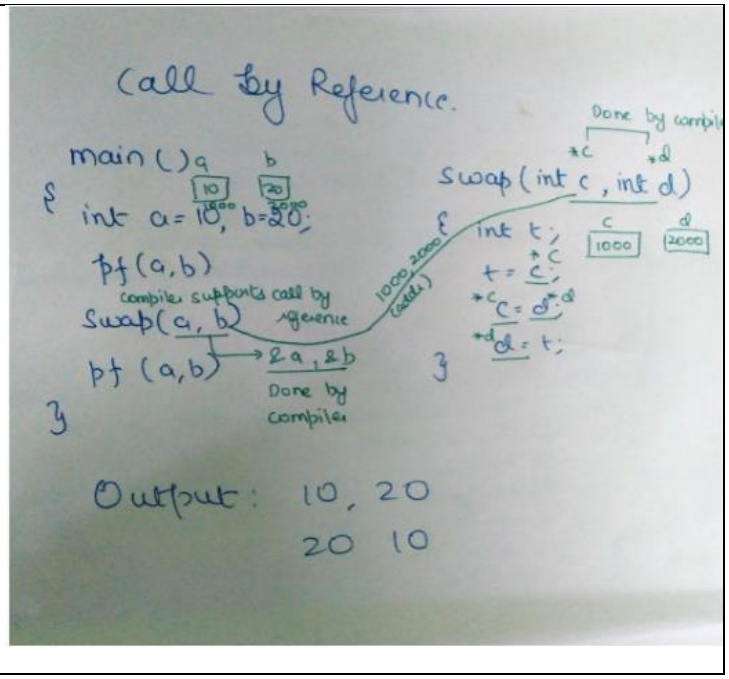
Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass by Value or Call by value:

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

1. This is the simplest method of parameter passing.
2. The actual parameters are evaluated and their r-values are passed to called procedure.
3. The operations on formal parameters do not changes the values of actual parameter.





Pass by Reference or Call by reference:

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

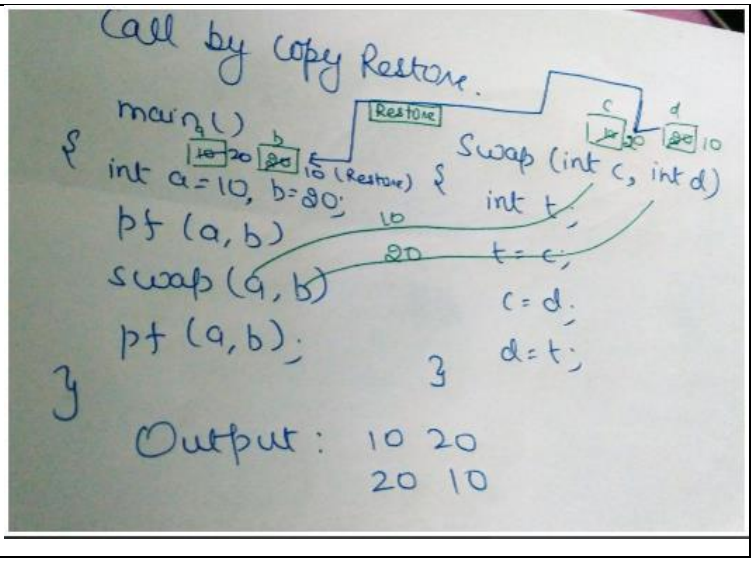
1. This method is also called as call by address or call by location.
2. The L-value, the address of actual parameter is passed to the called routines activation record.
3. The value of actual parameters can be changed.
4. The actual parameter should have an L-value.

Pass by Copy-restore:

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

1. This method is a hybrid between call by value and call by reference.
2. This is also known as copy-in-copy-out or values result.
3. The calling procedure calculates the value of actual parameters and it is then copied to activation record for the called procedure.
4. During execution of called procedure, the actual parameters value is not affected.
5. If the actual parameters have L-value then at return the value of formal parameter is copied to actual parameter.

```
int y;
calling_procedure()
{
   y = 10;
   copy_restore(y); //l-value of y is passed
   printf y; //prints 99
}
copy_restore(int x)
{
   x = 99; // y still has value 10 (unaffected)
   y = 0; // y is now 0
}
```



When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

1. This is less popular method of parameter passing.
2. Procedure is treated like macro.
3. The procedure body is substituted for call in caller with actual parameters substituted for formals.
4. The actual parameters can be surrounded by parenthesis to preserve their integrity.
5. The locals name of called procedure and names of calling procedure are distinct.

Language Facilities for Dynamics Storage Allocation:

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()

4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

Malloc() function in C:

1. The malloc () function allocates single block of requested memory.
2. It doesn't initialize memory at execution time, so it has garbage value initially.
3. It returns NULL if memory is not sufficient.

The **syntax of malloc()** function is given below:

```
ptr=(cast-type*)malloc(byte-size)
```

Let's see the **example** of malloc() function.

```c
#include<stdlib.h>
int main(){
  int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

calloc() function in C:

1. The calloc() function allocates multiple block of requested memory.
2. It initially initialize all bytes to zero.
3. It returns NULL if memory is not sufficient.

The **syntax of calloc()** function is given below:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Let's see the **example** of calloc() function.

```c
int main(){
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
   if(ptr==NULL)
   {
      printf("Sorry! unable to allocate memory");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc() function in C:

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the **syntax of realloc()** function.

```
ptr=realloc(ptr, new-size)
```

free() function in C:

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

```
free(ptr)
```

realloc()

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
        // This pointer will hold the
        // base address of the block created
        int* ptr;
```

free()

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
        // This pointer will hold the
        // base address of the block created
        int *ptr, *ptr1;
```

```
        int n, i;
// Get the number of elements for the array
        n = 5;
        printf("Enter number of elements: %d\n", n);
        // Dynamically allocate memory using calloc()
        ptr = (int*)calloc(n, sizeof(int));
        // Check if the memory has been successfully
        // allocated by malloc or not
        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else {
// Memory has been successfully allocated
printf("Memory successfully allocated using calloc.\n");
                // Get the elements of the array
                for (i = 0; i < n; ++i) {
                        ptr[i] = i + 1;
                }
                // Print the elements of the array
                printf("The elements of the array are: ");
                for (i = 0; i < n; ++i) {
                        printf("%d, ", ptr[i]);
                }
                // Get the new size for the array
                n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);
        // Dynamically re-allocate memory using realloc()
                ptr = realloc(ptr, n * sizeof(int));
                // Memory has been successfully allocated
printf("Memory successfully re-allocated using
realloc.\n");
                // Get the new elements of the array
                for (i = 5; i < n; ++i) {
                        ptr[i] = i + 1;
                }// Print the elements of the array
```

```
        int n, i;

        // Get the number of elements for the array
        n = 5;
        printf("Enter number of elements: %d\n", n);
        // Dynamically allocate memory using malloc()
        ptr = (int*)malloc(n * sizeof(int));
        // Dynamically allocate memory using calloc()
        ptr1 = (int*)calloc(n, sizeof(int));
        // Check if the memory has been successfully
        // allocated by malloc or not
        if (ptr == NULL || ptr1 == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else {
                // Memory has been successfully
allocated
printf("Memory successfully allocated using
malloc.\n");

                // Free the memory
                free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
printf("\nMemory successfully allocated using
calloc.\n");

                // Free the memory
                free(ptr1);
printf("Calloc Memory successfully freed.\n");
        }

        return 0;
}
```

```
            printf("The elements of the array are: ");
            for (i = 0; i < n; ++i) {
                printf("%d, ", ptr[i]);
            }   free(ptr);
        }   return 0;
}
```

Code Optimization:

Optimization is a program transformation technique, which tries to improve the code by making it consume fewer resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into **two types**:

- Machine independent
- Machine dependent.

6. Principle source of optimization / Code optimization techniques:

1. **Machine Independent**: are program transformations that improve target code, without taking into consideration any properties of target machine.

2. **Machine Dependent**: This optimization is based on register allocation and utilization of special machine instruction sequence.

A transformation of a program is called local if it is applied within a basic block and global if applied across basic blocks.

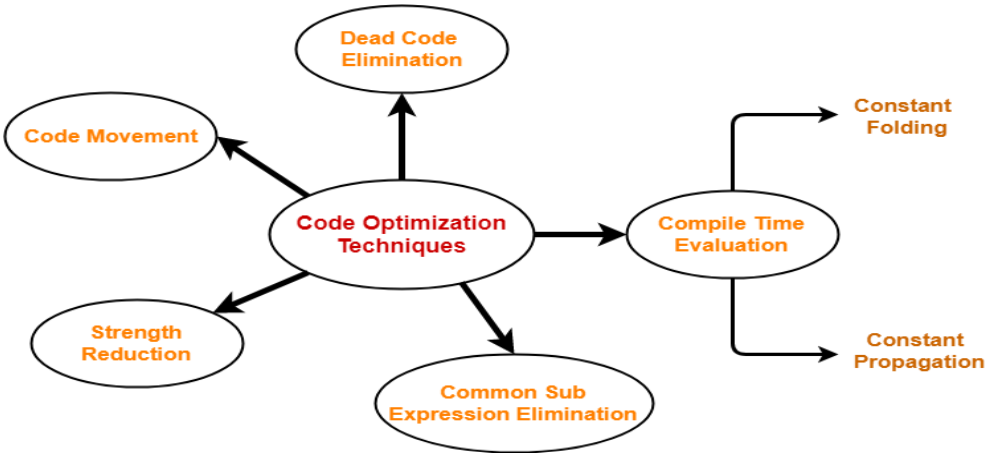
There are different types of transformations to improve the code and these transformations depend on the kind of optimization required.

1. Function-preserving transformations are those transformations that are performed without changing the function it computes. These are primarily used when global optimizations are performed.

2. Structure-preserving transformations are those that are performed without changing the set of expressions computed by the block. Many of these transformations are applied locally.

3. Algebraic transformations are used to simplify the computation of expression set using algebraic identities. These can replace expensive operations by cheaper ones, for instance, multiplication by 2 can be replaced by left shift.

Code optimization Techniques:



1. Compile Time Evaluation
    a. Constant Folding
    b. Constant Propagation
2. Common sub-expression elimination
3. Dead Code Elimination
4. Code Movement / Code Motion / Loop Invariant(Frequency reduction)
5. Strength Reduction

1. Compile time evaluation:

Compile time evaluation means shifting of computations from run time to compile time evaluation.

Two techniques that falls under compile time evaluation are-

- **Constant Folding**

In this technique,

As the name suggests, it involves folding the constants.

The expressions that contain the operands having constant values at compile time are evaluated.

Those expressions are then replaced with their respective results.

Example

Circumference of Circle = (22/7) x Diameter

Here, This technique evaluates the expression 22/7 at compile time.

The expression is then replaced with its result 3.14.

This saves the time at run time.

- **Constant Propagation**

In this technique,

If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.

The condition is that the value of variable must not get alter in between.

Example

pi = 3.14

radius = 10

Area of circle = pi x radius x radius

Here, This technique substitutes the value of variables 'pi' and 'radius' at compile time.

It then evaluates the expression 3.14 x 10 x 10.

The expression is then replaced with its result 314. This saves the time at run time.

2. Common Sub-Expression Elimination

The expression that has been already computed before and appears again in the code for computation. is called as **Common Sub-Expression**.

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Example

Code Before Optimization	**Code After Optimization**
S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i // **Redundant Expression** S5 = n S6 = b[S4] + S5	S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

3. Code Movement / Code Motion

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.
- There are 2 basic goals of code movement:
    1. To reduce the size of the code.
    2. To reduce the frequency of execution of code.

Example

Code Before Optimization	Code After Optimization

for (i = 1 ; i <=100 ; i ++)
{
z = i
x = 25*a ;
y=x+z ;
}
	x = 25*a ;
	for (i=1 ; i<=100 ; i ++)
	{
	z=i;
	y=x+z;
	}

Here, x=25*a; is loop invariant. Hence in the optimized program it is computed only once before entering for loop. y=x+z; is not loop invariant. Hence it cannot be subjected to frequency reduction.

4. Dead Code Elimination

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.
- The code which can be omitted from a program without affecting its results is called dead code.

Example

Code Before Optimization	Code After Optimization

i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
	i = 0 ;

5. Strength Reduction

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Here,

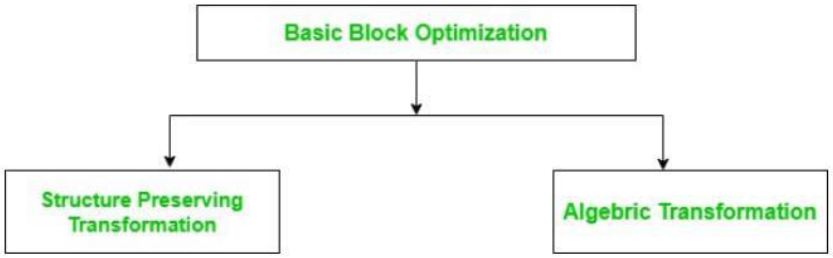
- The expression “A x 2” is replaced with the expression “A + A”.
- This is because the cost of multiplication operator is higher than that of addition operator.

7. OPTIMIZATION OF BASIC BLOCKS:

Optimization is applied to the basic blocks after the intermediate code generation phase of the compiler. Optimization is the process of transforming a program that improves the code by consuming fewer resources and delivering high speed. In optimization, high-level codes are replaced by their equivalent efficient low-level codes. Optimization of basic blocks can be machine-dependent or machine-independent. These transformations are useful for improving the quality of code that will be ultimately generated from basic block.

There are two types of basic block optimizations:

1. Structure preserving transformations
2. Algebraic transformations



The primary Structure-Preserving Transformation on basic blocks are:

1. Common sub-expression elimination
2. Dead code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced.

Example:

a: =b+c

b: =a-d

c: =b+c

d: =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: = b+c

b: = a-d

c: = a

d: = b

Dead code elimination:

It is possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program - once

declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u. In this a basic block is transformed to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

• Two statements

t1:=b+c

t2:=x+y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Algebraic Transformations:

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions. Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

a :=b+c

e :=c+d+b

the following intermediate code may be generated: **a :=b+c**

t :=c+d e :=t+b

Example:

x:=x+0 can be removed

x:=y**2 can be replaced by a cheaper statement x:=y*y

The compiler writer should examine the language specification carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

8. Peephole Optimization:-

Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.

A statement-by-statement code-generations strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

The **objective of peephole** optimization is as follows:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Characteristics of peephole optimizations:

1. Redundant-instructions elimination
2. Flow-of-control optimizations
3. Algebraic simplifications
4. Use of machine idioms
5. Unreachable

Redundant-instructions elimination: In this technique, redundancy is eliminated

If we see the instructions sequence

(1) MOV R0,a

(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….
 -----
If ( debug ) {
Print debugging information
 }
```

Flows-Of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

….

L1: gotoL2 (d)

by the sequence

goto L2

….

L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

….

L1: goto L2 (e)

can be replaced by

If a < b goto L2

….

L1: goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

L1: if a < b goto L2 (f) L3:

may be replaced by

If a < b goto L2

goto L3

…….

L3:

While the number of instructions in(e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) in execution time

Algebraic Simplification: There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 or

x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 → i++

i:=i-1 → i- -

9. **Basic Blocks and Flow Graphs:-**

Basic Block:

The basic block is a set of statements. The basic blocks do not have any in and out branches except entry and exit. It means the flow of control enters at the beginning and will leave at the end without any halt. The set of instructions of basic block executes in sequence.

Here, the first task is to partition a set of three-address code into the basic block. The new basic block always starts from the first instruction and keep adding instructions until a jump or a label is met. If no jumps or labels are found, the control will flow in sequence from one instruction to another.

The algorithm for the construction of basic blocks is given below:

Algorithm: Partitioning three-address code into basic blocks.

Input: The input for the basic blocks will be a sequence of three-address code.

Output: The output is a list of basic blocks with each three address statements in exactly one block.

METHOD: First, we will identify the leaders in the intermediate code. There are some rules for finding leaders, which are given below:

1. The first instruction in the intermediate code will always be a leader.
2. The instructions that target a conditional or unconditional jump statement are termed as a leader.
3. Any instructions that are just after a conditional or unconditional jump statement will be a leader.

Each leader's basic block will have all the instructions from the leader itself until the instruction, which is just before the starting of the next leader.

Example:

Consider the following source code for a 10 x 10 matrix to an identity matrix.

```
for i from 1 to 10 do
        for j from 1 to 10 do
           a [ i, j ] = 0.0;
 for i from 1 to 10 do
           a [ i,i ] = 1.0;
```

The three address code for the above source program is given below:

```
1) i = 1
 2) j = 1
 3) t1 = 10 * i
 4) t2 = t1 + j
 5) t3 = 8 * t2
 6) t4 = t3 - 88
 7) a[t4] = 0.0
 8) j = j + 1
 9) if j <= 10 goto (3)
 10) i = i + 1
 11) if i <= 10 goto (2)
 12) i = 1
 13) t5 = i - 1
 14) t6 = 88 * t5
 15) a[t6] = 1.0
 16) i = i + 1
 17) if i <= 10 goto (13)
```

- According to the given algorithm, instruction 1 is a leader.
- Instruction 2 is also a leader because this instruction is the target for instruction 11.
- Instruction 3 is also a leader because this instruction is the target for instruction 9.
- Instruction 10 is also a leader because it immediately follows the conditional goto statement.
- Similar to step 4, instruction 12 is also a leader.
- Instruction 13 is also a leader because this instruction is the target for instruction 17.

So there are six basic blocks for the above code, which are given below:

B1 for statement 1

B2 for statement 2

B3 for statement 3-9

B4 for statement 10-11

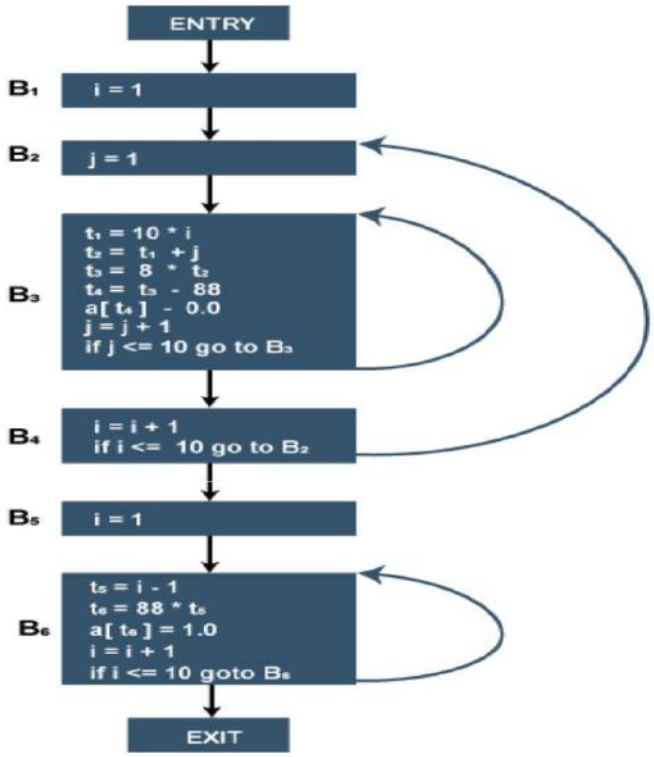
B5 for statement 12

B6 for statement 13-17.

10. Flow Graph:-

It is a directed graph. After partitioning an intermediate code into basic blocks, the flow of control among basic blocks is represented by a flow graph. An edge can flow from one block X to another block Y in such a case when the Y block's first instruction immediately follows the X block's last instruction. The following ways will describe the edge:

- There is a conditional or unconditional jump from the end of X to the starting of Y.
- Y immediately follows X in the original order of the three-address code, and X does not end in an unconditional jump.



Flow graph for the 10 x 10 matrix to an identity matrix.

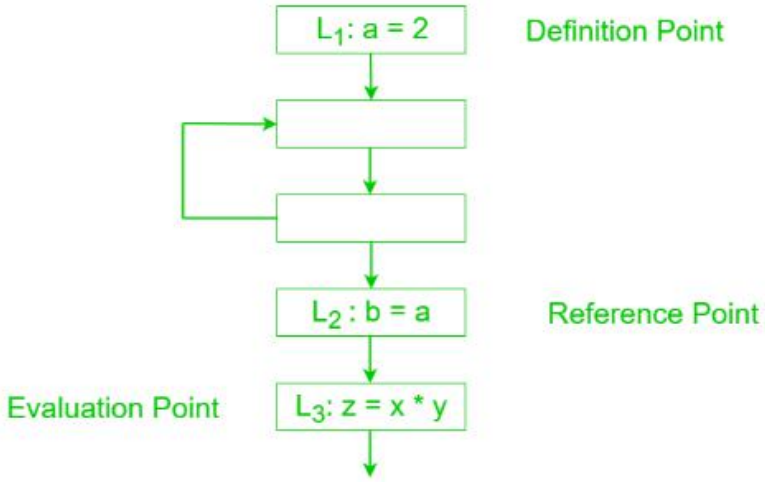
- Block B1 is the entry point for the flow graph because B1 contains starting instruction.
- B2 is the only successor of B1 because B1 doesn't end with unconditional jumps, and the B2 block's leader immediately follows the B1 block's leader.
- B3 block has two successors. One is a block B3 itself because the first instruction of the B3 block is the target for the conditional jump in the last instruction of block B3. Another successor is block B4 due to conditional jump at the end of B3 block.
- B6 block is the exit point of the flow graph.

11. Data Flow Analysis Of Flow Graphs:-

It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis, optimization can be done. In general, its process in which values are computed using data flow analysis. The data flow property represents information that can be used for optimization.

Basic Terminologies –

- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.

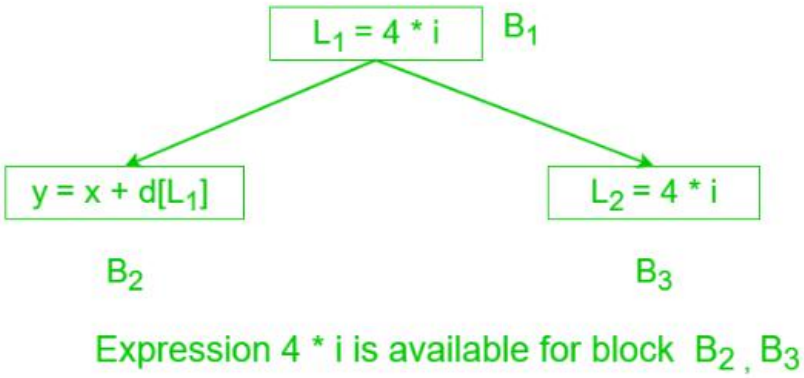


Data Flow Properties –

Available Expression – A expression is said to be available at a program point x if along paths its reaching to x. A Expression is available at its evaluation point.

An expression a+b is said to be available if none of the operands gets modified before their use.

Example –

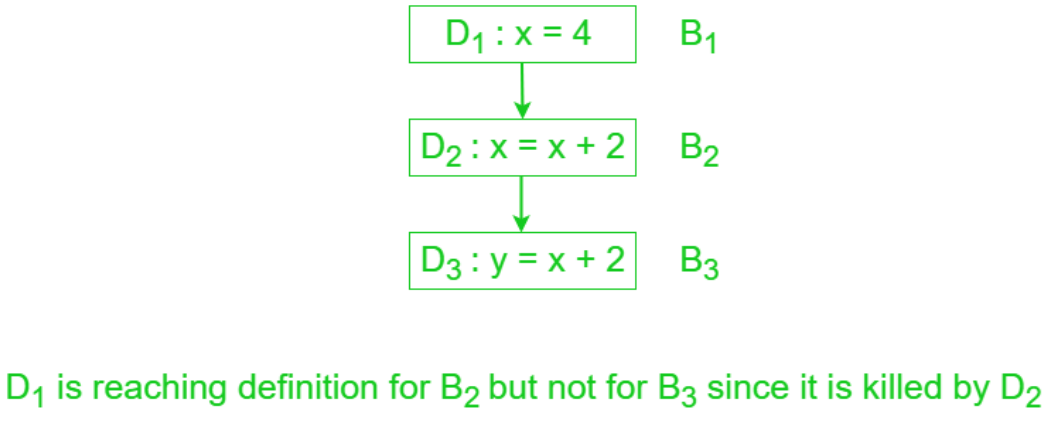


- **Advantage –**

  It is used to eliminate common sub expressions.

- **Reaching Definition –** A definition D is reaches a point x if there is path from D to x in which D is not killed, i.e., not redefined.

Example –

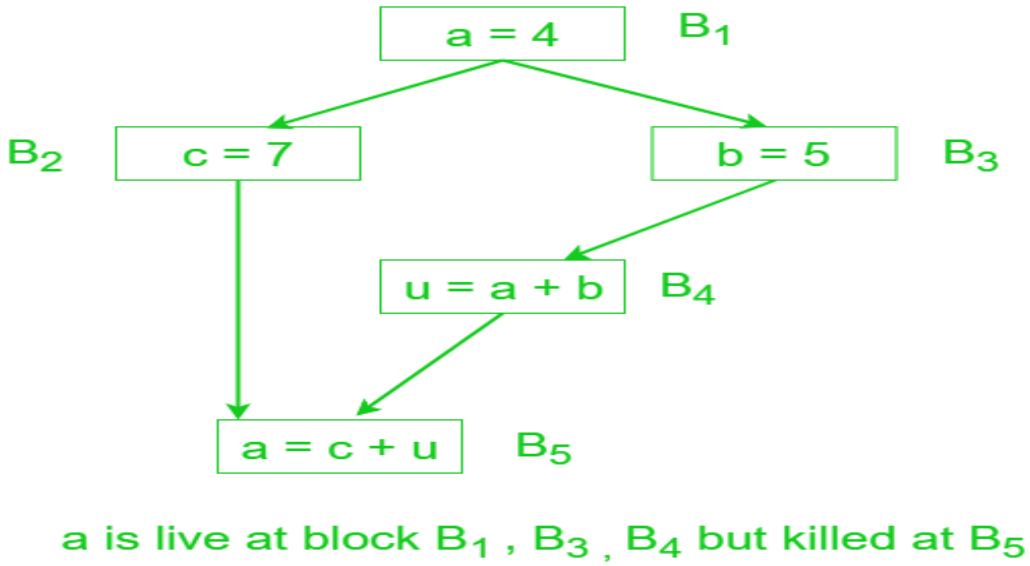


- **Advantage –**

  It is used in constant and variable propagation.

- **Live variable –** A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.

Example –



- **Advantage –**
  1. It is useful for register allocation.
  2. It is used in dead code elimination.
- **Busy Expression –** An expression is busy along a path if its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.

Advantage –

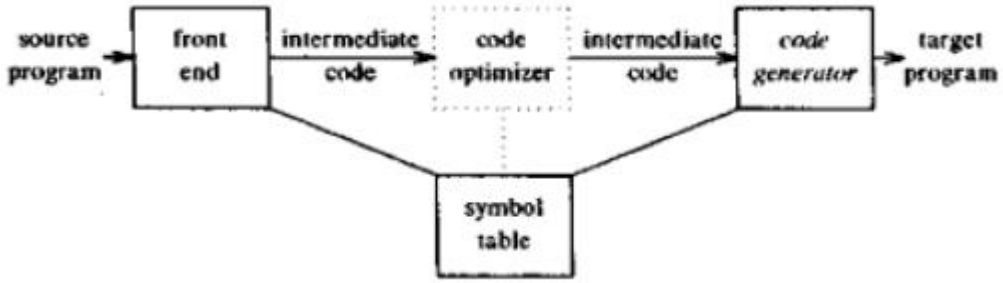
It is used for performing code movement optimization.

UNIT-V

Code generation: Machine dependent code generation, object code forms, generic code generation algorithm, Register allocation and assignment. Using DAG representation of Blocks

---

Code Generation: The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.



- The code generated by the compiler is an object code of some lower-level programming language, for **example**, assembly language.
- The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties
    - It should carry the exact meaning of the source code.
    - It should be efficient in terms of CPU usage and memory management.

1. Issues in the Design of a Code Generator:

The following issues arise during the code generation phase :

1) Input to code generator
2) Target program
3) Memory management
4) Instruction selection
5) Register allocation
6) Evaluation order

1. **Input to code generator**: The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be :

1) Linear representation such as postfix notation

2) Three address representation such as Quadruples

3) Virtual machine representation such as stack machine code

4) Graphical representations such as syntax trees and DAGS. Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program: The output of the code generator is the target program. The output may be:

a. **Absolute machine language**: It can be placed in a fixed memory location and can be executed immediately.

b. **Relocatable machine language**: It allows subprograms to be compiled separately.

c**. Assembly language**: Code generation is made easier.

3. Memory management: Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator. It makes use of symbol table, that is, a name in a threeaddress statement refers to a symbol-table entry for the name.

Labels in three-address statements have to be converted to addresses of instructions.

For example:

j : goto i generates jump instruction as follows : if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.

if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

1. The instructions of target machine should be complete and uniform.

2. Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

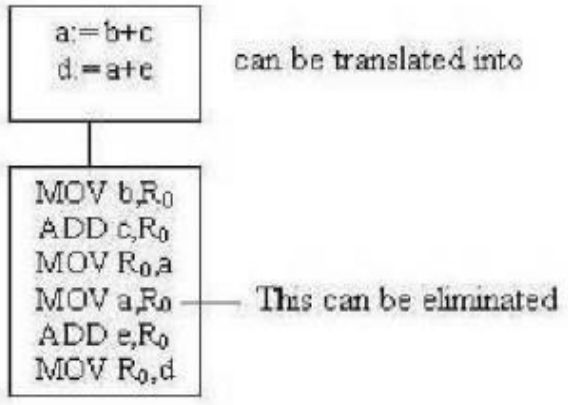
3. The quality of the generated code is determined by its speed and size.

For example: Every three-address statement of the form x=y+z where x, y and z are statically allocated.

Code sequence generated is shown as:

```
MOV y,R0 /* load y into register R0 */
ADD z,R0 /* add z to R0 */
MOV R0,x /* store R0 into x */
```

Unfortunately, this kind of statement-by-statement code generation often produces poor code. For example, the sequence of statements,



The quality of the generated code is determined by its speed and size. A target machine with a rich instruction set may provide several ways of implementing a given operation.

For example: If the target machine has an "increment" instruction (INC), then the three address statement

a=a+1

may be implemented more efficiently by the single instruction,

INC a

instead of ,

```
MOV a,R0
ADD #1,R0
MOV R0,a
```

5. Register allocation: Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two sub problems:

- **Register allocation** – the set of variables that will reside in registers in the program is selected.
- **Register assignment** -the specific register that a variable will reside is selected. Certain machine requires even-odd register pairs for some operands and results. For example consider the division instruction of the form :

  Div x, y where, x – dividend in even register in even/odd register pair, y – divisor in even register holds the remainder odd register holds the quotient

6. Evaluation order: At last, the code generator decides the order in which the instruction will be executed. The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

- Picking the best order is a difficult task.
- Initially avoid this problem by generating code for the three address statements in the order in which they have been produced by the intermediate code generator.
- It creates schedules for instructions to execute them. When instructions are independent, their evaluation order can be changed to utilize registers and save on instruction cost.
  - Consider the following instruction: a+b-(c+d)*e The three-address code, the corresponding code and its reordered instruction are given below:

Three-address code	Code	Reordered three-address code	Code	Inference

t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
	MOV a,R0
	ADD b,R0
	MOV R0,t1
	MOV c,R1
	ADD d,R1
	MOV e,R0
	MUL R1,R0
	MOV t1,R1
	SUB R0,R1
	MOV R1,t4
		t2:=c+d
		t3:=e*t2
		t1:=a+b
		t4:=t1-t3
			MOV c,R0
			ADD d,R0
			MOV e,R1
			MUL R0,R1
			MOV a,R0
			ADD b,R0
			SUB R1,R0
			MOV R0,t4
				The reordered instructions reduced the number of final code by 2 and thus saved in cost. The three-address code is reordered so that t1 is computed after computing t2 and t3. This reordering has saved in the instruction cost.

2. Register Allocation and Assignment:

Register allocation is an important method in the final phase of the compiler. Registers are faster to access than cache memory. Registers are available in small size up to few hundred Kb. Thus it is necessary to use minimum number of registers for variable allocation.

The register allocator determines which values will reside in the register and which register will hold each of those values.

There are two approaches, which are

Local allocators: These are based on usage counts

Global allocators: These are based on the concept of graph coloring

Local register allocation:

– Register allocation is only within a basic block. It follows top-down approach.

– Assign registers to the most heavily used variables

– Traverse the block

– Count uses

– Use count as a priority function

– Assign registers to higher priority variables first

Need of global register allocation:

- Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.
- To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

The method is also very simple and involves **two passes**:

- In the first pass, the target machine instructions or intermediate code instructions are selected as though there are infinite number of symbolic registers
- In the second pass, for each procedure, a register inference graph is constructed in which the nodes are symbolic registers and an edge connects to nodes if one is live at a point where the other is defined, i.e., if both are live at the same time

Register allocation depends on:

– Size of live range

– Number of uses/definitions

– Frequency of execution

– Number of loads/stores needed.

– Cost of loads/stores needed.

Usage Counts:

A simple method of determining the savings to be realized by keeping variable x in a register for the duration of loop L is to recognize that in our machine model we save one unit of cost for each reference to x if x is in a register.

An approximate formula for the benefit to be realized from allocating a register to x within a loop L is:

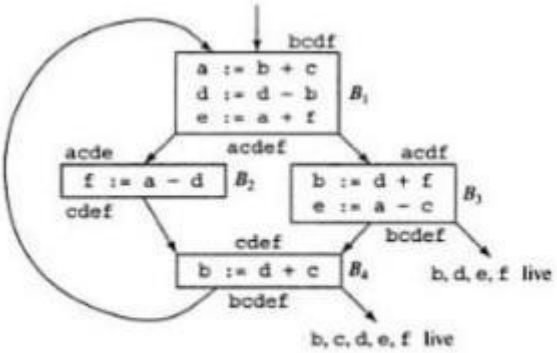
$$\sum_{blocks\ B\ in\ L} (us(x, B) + 2 * live(x, B))$$

where,

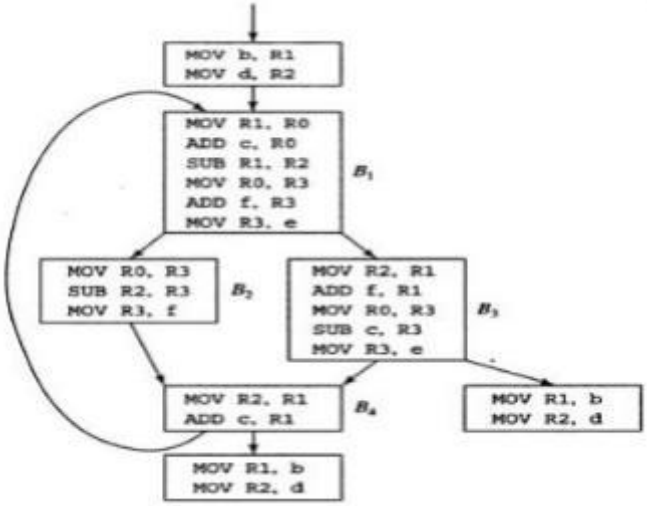
-use(x, B) is the number of times x is used in B prior to any definition of x;

-live(x,B) is 1 if x is live on exit from B and is assigned a value in B and 0 otherwise.

Example: Consider the basic block in the inner loop in Fig 5.3 where jump and conditional jumps have been omitted. Assume R0, R1 and R2 are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in the figure.



Flow graph of an inner loop



Code sequence using global register assignment

Now, an attempt is made to color the graph using k colors, where k is the number of available registers. We know that a graph is said to be colored if each node has been assigned a color in such a way that no two adjacent nodes have the same color. Once the coloring is over, register allocation can be made

Is Coloring a Graph Easy?

Determining whether a graph is k-colorable or not is NP-complete. Generally, the following strategy is followed to color a graph quickly:

Step 1: if a node n in the graph has less than k neighbors, n and its edges from the graph G are removed to obtain a graph G'

Step 2: now if G' is k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any register

Step 1 & Step 2 are repeated (if G' is not k-colorable) using G' as G, until

- An empty graph is obtained, in which a k-coloring of the original graph can be produced by coloring the nodes in the reverse order un which they were removed
- A graph in which each node has k or more adjacent node is obtained. In this case k-coloring is not possible. At this point a node is spilled by introducing code to store and reload the register

3. Using DAG representation of Blocks:

- A DAG for a basic block is a directed acyclic graph with the following labels on nodes:
  1. Leaves are labelled by unique identifiers, either variable names or constants.
  2. Interior nodes are labelled by an operator symbol.
  3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent Statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node (z) for case (i).

Step 2: For the **case (i),** create a node (OP) whose left child is node(y) and right child is node(z).

(Checking for common sub expression). Let n be this node.

For **case(ii),** determine whether there is node(OP) with one child node(y). If not create such a node.

For **case(iii)**, node n will be node(y).

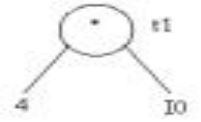
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

1. $t_1 := 4* i$
2. $t_2 := a[t_1]$
3. $t_3 := 4* i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2*t_4$
6. $t_6 := prod+t_5$
7. $prod := t_6$
8. $t_7 := i+1$
9. $i := t_7$
10. if i<=20 goto (1)

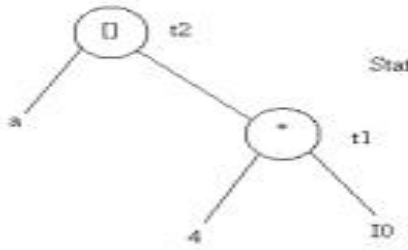
Stages in DAG Construction

(a)



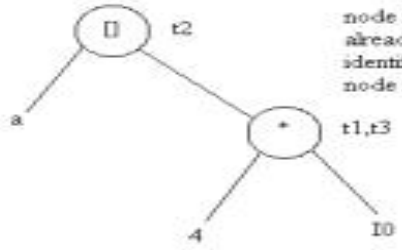
Statement (1)

(b)



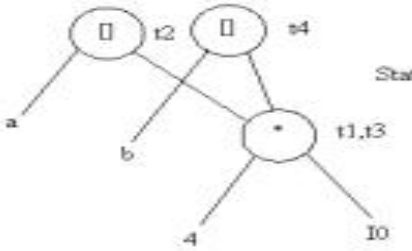
Statement (2)

(c)



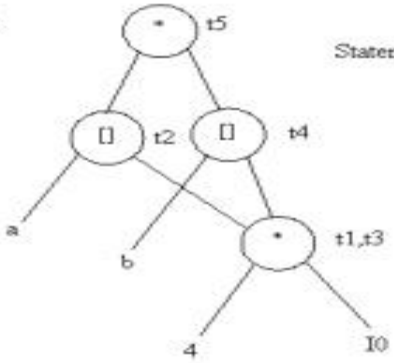
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)



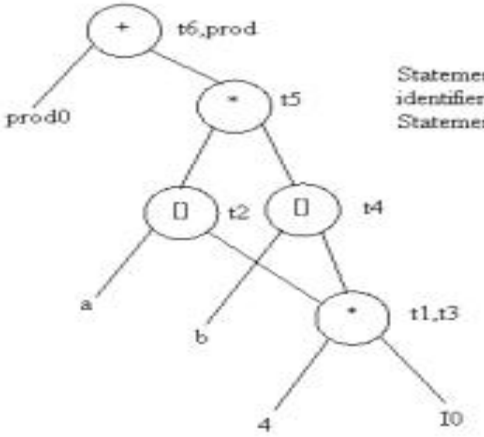
Statement (4)

(e)

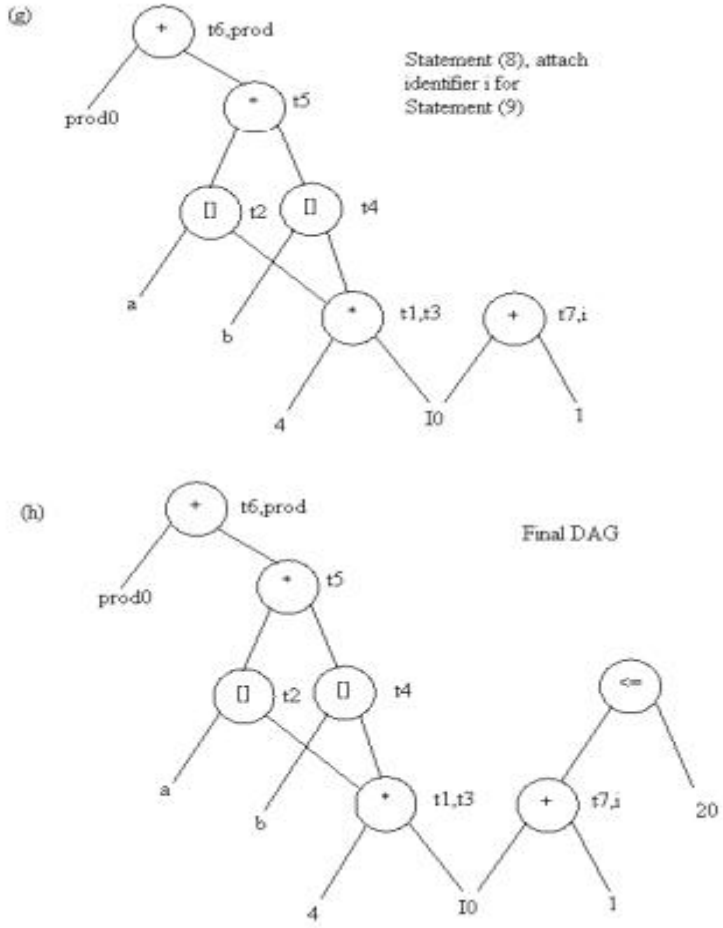


Statement (5)

(f)



Statement (6), attach identifier prod for Statement (7)



Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

4. Generic Code Generation Algorithm:

Assume that for each operator in the statement, there is a corresponding target language operator. The computed results can be left in registers as long as possible, storing them only if the register is needed for another computation or just before a procedure call, jump or labeled statement

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

For example: consider the three-address statement a: = b+c

It can have the following sequence of codes:

```
ADD Rj, Ri          Cost = 1    // if Ri contains b and Rj contains c
                (or)
ADD c, Ri           Cost = 2    // if c is in a memory location
                (or)
MOV c, Rj           Cost = 3    // move c from memory to Rj and add
ADD Rj, Ri
```

Register and Address Descriptors:

A **register descriptor** is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

An **address descriptor** stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block.

For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.
2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following threeaddress code sequence:

t : = a – b

u : = a – c

v : = t + u

d : = v + u with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating Code for Indexed Assignments:

The table shows the code sequences generated for the indexed assignment statements

a : = b [i] and a [i] : = b

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments:

The table shows the code sequences generated for the pointer assignments a : = *p and *p : = a

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

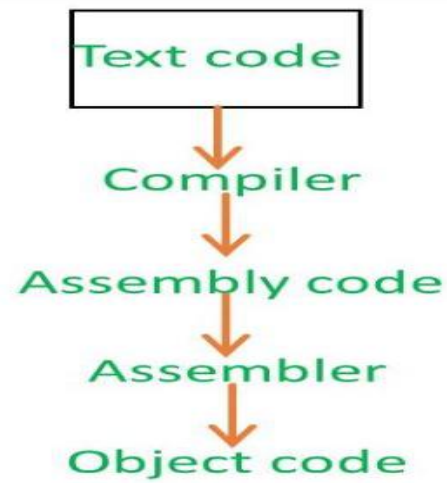
Generating Code for Conditional Statements:

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x : = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z

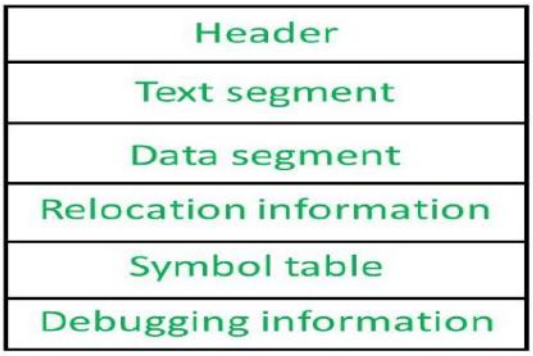
5. Object code forms

Let assume that, you have a c program, then you give the C program to compiler and compiler will produce the output in assembly code. Now, that assembly language code will give to the assembler and assembler is going to produce you some code. That is known as Object Code.



But, when you compile a program, then you are not going to use both compiler and assembler.You just take the program and give it to the compiler and compiler will give you the directly executable code. The compiler is actually combined inside the assembler along with loader and linker.So all the module kept together in the compiler software itself. So when you calling gcc, you are actually not just calling the compiler, you are calling the compiler, then assembler, then linker and loader.

Once you call the compiler, then your object code is going to present in Hard-disk. This object code contains various part –



Header –
The header will say what are the various parts present in this object code and then point that parts. So header will say where the text segment is going to start and a pointer to it and where the data segment going to start and it say where the relocation information and symbol information there.

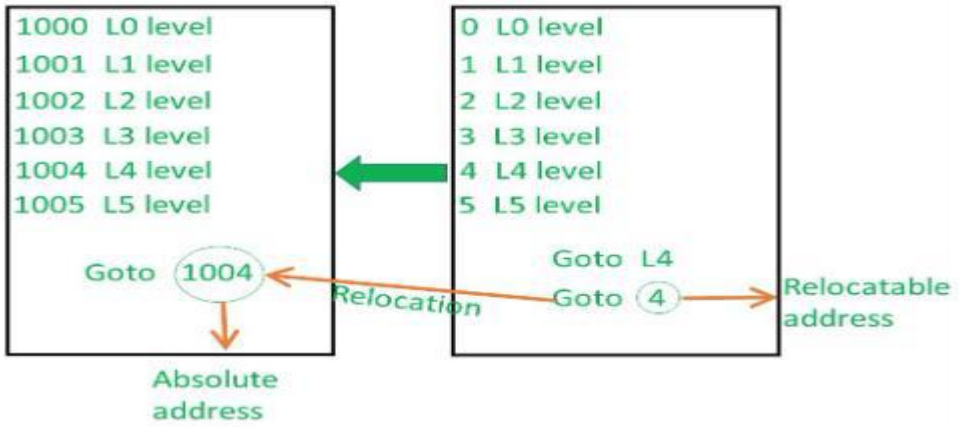
It is nothing but like an index, like you have a textbook, there an index page will contains at what page number each topic present. Similarly, the header will tell you, what are the palaces at which each information is present.So that later for other software it will be useful to directly go into that segment.

Text segment –
It is nothing but the set of instruction.

Data segment –
Data segment will contain whatever data you have used.For example, you might have used something constraint, then that going to be present in the data segment.

Relocation Information –
Whenever you try to write a program, we generally use symbol to specify anything.Let us assume you have instruction 1, instruction 2, instruction 3, instruction 4,….

Now if you say somewhere Goto L4 (Even if you don’t write Goto statement in the high-level language, the output of the compiler will write it), then that code will be converted into object code and L4 will be replaced by Goto 4. Now Goto 4 for the level L4 is going to work fine, as long as the program is going to be loaded starting at address no 0. But most of the cases, the initial part of the RAM is going to be dedicated to the operating system. Even if it is not dedicated to the operating system, then might be some other process which will already be running at address no 0. So, when you are going to load the program into memory, means if the program has to be load in the main memory, it might be loaded anywhere.Let us say 1000 is the new starting address, then all the addresses has to be changed, that is known as **Reallocation**.



The original address is known as **Relocatable address** and the final address which we get after loading the program into main memory is known as the **Absolute address**.

Symbol table –

It contains every symbol that you have in your program.for example, int a, b, c then, a, b, c are the symbol.it will show what the variables that your program contains are.

Debugging information –

It will help to find how a variable is keeping on changing.