Compiler Design

UNIT 1

1 What is compiler?

Ans:- A compiler is a program that translates human readable source code into computer executable machine code. To do this successfully the human readable code must comply with the syntax rules of whichever programming language it is written in. The compiler is only a program and cannot fix your programs for you. If you make a mistake, you have to correct the syntax or it won't compile

2 compare NFA and DFA ?

Ans:-

DFA- single value transition function is represented
- it take more memory space
- number of state is more
- it not support null move
- backtracking is possible
- dfa is faster recognizer

NFA- multi value transition function is represented
- it take less memory space
- number of state is less
- it support null moves
- backtrakin is possible some cases
- nfa is slower recognizer

3 Define bootstrap and cross compiler?

Ans:- Bootstrap:-

In computing, **bootstrapping** (from an old expression "to pull oneself up by one's bootstraps") is a technique by which a simple computer program activates a more complicated system of programs. In the start up process of a computer system, a small program (such as BIOS) initializes and tests that a basic requirement of hardware, peripherals and external memory devices are connected.

Cross compiler:-

A compiler that runs on one computer but produces object code for a different type of computer. Cross compilers are used to generate software that can run on computers with a new architecture or on special-purpose devices that cannot host their own compilers.

4 Define token lexeme pattern ?

Ans:- Token-A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, num, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Example of non-tokens:

- Comments, preprocessor directive, macros, blanks, tabs, newline, . . .

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

For example, the pattern for the RELOP token contains six lexemes (=, < >, <, < =, >, >=) so the lexical analyzer should return a RELOP token to parser whenever it sees any one of the six.

Pattern-There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
Regular expressions are an important notation for specifying patterns.

For example, the pattern for the Pascal identifier token, id, is: id → letter (letter | digit)*.

5 What is compilation ? Describe phase of compiler with diagram .

- **Compilation :-** 1 the action or process of compiling.
  2 a thing, especially a book or record, compiled from different sources.

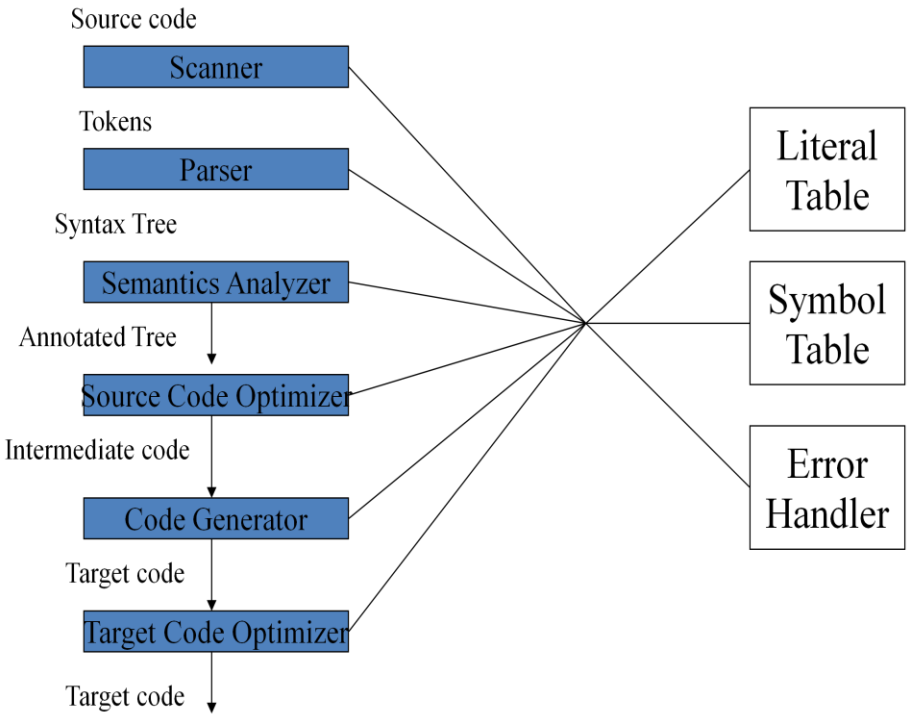
Phase of compiler:- There are 6 phases a typical compiler will implement. The major reason why separate a compiler into 6 phases is probably simplicity. Compiler design is thus full of ``Divide and Conquer'' strategy, component-level design, reusabililty and performance optimization.

- Lexical Analysis
- Syntax Analysis
- Error Recovery
- Scope Analysis
- Type Analysis
- Code Generation

The Phases of a Compiler

Phase	**Output**	**Sample**
Programmer (source code producer)	Source string	**A=B+C;**
Scanner (performs lexical analysis)	Token string	**'A', '=', 'B', '+', 'C', ';'** And symbol table with names
Parser (performs syntax analysis based on the grammar of the programming language)	Parse tree or abstract syntax tree	**;** **|** **=** **/** **A +** **/** **B C**
Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	**int2fp B t1** **+ t1 C t2** **:= t2 A**
Optimizer	Three-address code, quads, or RTL	**int2fp B t1** **+ t1 #2.3 A**
Code generator	Assembly code	**MOVF #2.3,r1** **ADDF2 r1,r2** **MOVF r2,A**
Peephole optimizer	Assembly code	**ADDF2 #2.3,r2** **MOVF r2,A**



6 Compare single pass and multi pass complier ?
Ans:-

- **Single pass**

  - Creates a table of Jump Instructions

  - Forward Jump Locations are generated incompletely

  - Jump Addresses entered into a fix-up table along with the label they are jumping to

  - As label destinations encountered, it is entered into the table of labels

  - After all inputs are read, CG revisits all of these problematic jump instructions

- **Multiple pass**

  - No Fix-Up table

  - In the first pass through the inputs, CG does nothing but generate table of labels.

  - Since all labels are now defined, whenever a jump is encountered, all labels already have pre-defined memory location.

  - Possible problem: In first pass, CG needs to know how many MLI correspond to a label.

  - Major Drawback-Speed

7 Explain symbol table manager and error handler ?
Ans:- Symbol table manager:-
– "bookkeeper"
– Maintains names used in program and information about them
• Type
•Kind : variable, array, constant, literal, procedure, function, record…
•Dimensions (arrays)
•Number of parameters and type (functions, procedures)
•Return type (functions)
Error Handler:-

–Control passed here on error
– Provides information about type
and location of error
–Called from any of the modules of
the front end of the compiler
•Lexical errors e.g. illegal character
•Syntax errors
•Semantic errors e.g. illegal type

8 Define finite automata ? Define the term alphabet, string and language ?
Ans:- **Finite Automata**

- A recognizer for a language is a program that takes a string x as an input and answers "yes" if x is a sentence of the language and "no" otherwise.
- One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automation.
- A finite automation can be deterministic means that more than one transition out of a state may be possible on a same input symbol.
- Both automata are capable of recognizing what regular expression can denote.

Strings
A string over some alphabet is a finite sequence of symbol taken from that alphabet.
For example, banana is a sequence of six symbols (i.e., string of length six) taken from ASCII computer alphabet. The empty string denoted by $\in$, is a special string with zero symbols (i.e., string length is 0).
If x and y are two strings, then the concatenation of x and y, written xy, is the string formed by appending y to x.
For example, If x = dog and y = house, then xy = doghouse. For empty string, $\in$, we have S$\in$ = $\in$S = S.
String exponentiation concatenates a string with itself a given number of times:
S^2 = SS or S.S
S^3 = SSS or S.S.S
S^4 = SSSS or S.S.S.S and so on

By definition S^0 is an empty string, $\in$, and S` = S. For example, if x =ba and na then xy^2 = banana.

Languages
A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.
Let L and M be two languages where L = {dog, ba, na} and M = {house, ba} then

- Union: LUM = {dog, ba, na, house}
- Concatenation: LM = {doghouse, dogba, bahouse, baba, nahouse, naba}
- Expontentiation: L2 = LL
- By definition: L^0 ={$\in$} and L` = L

The kleene closure of language L, denoted by L*, is "zero or more Concatenation of" L.
$L^* = L^0 \cup L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$
For example, If L = {a, b}, then

L* = {$\in$, a, b, aa, ab, ab, ba, bb, aaa, aba, baa, . . . }
The positive closure of Language L, denoted by L^+, is "one or more Concatenation of" L.

$L^+ = L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$

For example, If L = {a, b}, then

L^+ = {a, b, aa, ba, bb, aaa, aba, . . . }

UNIT-2

1 Define context free grammar ?

Ans:- Precise and easy way to specify the syntactical structure of a programming language

■ Efficient recognition methods exist

■ Natural specification of many "recursive"

constructs:

■ expr -> expr + expr | term

2 Write component of context free grammar ?

Ana:- Terminals T-

■ Symbols which form strings of L(G), G a CFG (= tokens in the scanner), e.g. if, else, id

Nonterminals N-

■ Syntactic variables denoting sets of strings of L(G)

■ Impose hierarchical structure (e.g., precedence rules)

Start symbol S (∈ N)-

■ Denotes the set of strings of L(G)

Productions P-

■ Rules that determine how strings are formed

■ N -> (N|T)*

3 How will you eliminate the left recursion problem ?

Ans:- Simple case: immediate left recursion:

Replace A -> A α | β with

A -> β A'

A' -> αA' | ε

Example:-

S -> Aa | b

A -> Ac | Sd | ε

Order: S,A

for i := 1 to n do begin

for j := 1 to i-1 do begin

replace each production of the

form

Ai -> Aj

γ by the productions

Ai-> δ1γ | δ2γ |...| δkγ

where Ai-> δ1 | δ2 | ... | δk are all current Aj productions end eliminate immediate left recursion among the Ai productions end

i=2,j=1:

Eliminate A->S γ

Replace A->Sd with

A->Ac|Aad|bd|ε

Eliminate immediate left recursion:

S->Aa|b

A -> bdA'|A'

A' ->cA' | adA' | ε

4 What do you mean by ambiguous grammar ? draw parse tree for Id+id*id.

Ans:- Ambiguous grammar-

■ >=2 different parse trees for some sentence ⇔ >= 2 leftmost/rightmost derivations

■ Usually want to have unambiguous grammars

■ E.g. want to just one evaluation order:

<id> + <id> * <id> to be parsed as <id> +

(<id> * <id>) not (<id>+<id>)*<id>

■ To keep grammars simple accept ambiguity and resolve separately (outside of grammar)

5 What are different error recovery technique used in parsing ?

Ans:- **Error-Recovery Strategies-**

The goals are clear, but difficult.

- Report errors clearly and accurately. One difficulty is that one error can mask another and can cause correct code to look faulty.
- Recover quickly enough to not miss other errors.
- Add minimal overhead.

Trivial Approach: No Recovery:-
Print an error message when parsing cannot continue and then terminate parsing.

Panic-Mode Recovery:-
The first level improvement. The parser discards input until it encounters a synchronizing token. These tokens are chosen so that the parser can make a fresh beginning. Good examples for C/Java are ; and }.

Phrase-Level Recovery:-
Locally replace some prefix of the remaining input by some string. Simple cases are exchanging ; with , and = with ==. Difficulties occur when the real error occurred long before an error was detected.

Error Productions:-
Include productions for common errors.

Global Correction:-
Change the input I to the closest correct input I' and produce the parse tree for I'.

6 Short note on parameter parsing ?
Ans :- Prarameter parsing- Parameter passing methods
– pass by value
– pass by result
– pass by value-result
– pass by reference
• aliasing
– pass by name
• Procedures/functions as argument
Procedures
• Modularize program structure
– Argument: information passed from caller to callee (actual parameter)
– Parameter: local variable whose value (sometimes) is received from caller (formal parameter**)**
• Procedure declaration
– name, formal parameters, procedure body with local declarations and statement list, optional result type
void translateX(point *p, int dx)
Parameter Association
• Positional association
– Arguments associated with formals one-by-one
• E.g., C, Pascal, Scheme, Java
• Keyword association
• E.g., Ada uses a mixture
procedure plot (x,y: in real; penup: in boolean)
…. plot (0.0, 0.0, penup=> true)
….plot (penup=>true, x=>0.0, y=>0.0)
Parameter Passing Modes
• pass by value
– C, Pascal, Ada, Scheme, Algol68
• pass by result
– Ada
• pass by value-result (copy-in, copy-out)
– Fortran, sometimes Ada
• pass by reference
– Fortran, Pascal var params, sometimes Cobol
• pass by name (outmoded)
– Algol60

7 Describe working of shift reduce parsing ? Define handle and pruning handle ?
Ans:-Shift Reduce Parssing-

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string ➔ the starting symbol

  reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation: $S \Rightarrow \omega$

Shift-Reduce Parser finds: $\omega \Leftarrow \ldots \Leftarrow S$

Shift-Reduce Parsing -- Example

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

input string: aaabb

aaAbb

aAbb ⇓ reduction

aABb

S

$S \Rightarrow aABb \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb$

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

Handle :

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle

- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is
  a production rule $A \rightarrow \beta$ and a position of γ
  where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \Rightarrow \alpha A\omega \Rightarrow \alpha\beta\omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ω is a string of terminals.

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$$

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

8 Construct LALR parsing table ?
Ans:

- **LALR** stands for **LookAhead LR.**

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALr Parsing table

Canonical LR(1) Parser → LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet =R,\$$ ➔ $S \rightarrow L \bullet =R$ Core
$R \rightarrow L \bullet ,\$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1: $L \rightarrow id \bullet ,=$ A new state: I_{12}: $L \rightarrow id \bullet ,=$
➔ $L \rightarrow id \bullet ,\$$
I_2: $L \rightarrow id \bullet ,\$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

$C=\{I_0,...,I_n\}$ ➔ $C'=\{J_1,...,J_m\}$ where $m \leq n$

- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores
    ➔ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

UNIT-3

1 Define syntax directed definition ?

- In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

  $b=f(c_1,c_2,\ldots,c_n)$ where f is a function,

  and b can be one of the followings:

  ➔ b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

  OR

  ➔ b is an inherited attribute one of the grammar symbols in α (on the

  right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

3 What is syntax directed translation ?

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - a string, a number, a memory location, a complex record.

3 What is done in syntax directed translation scheme ? figure .
Ans:-

Syntax-Directed Definitions and Translation Schemes

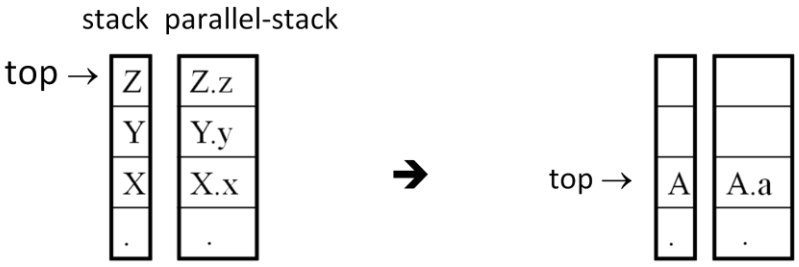
- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

2 Explain bottom up and top down evaluation of attributes ?
Ans:-

Bottom-Up Evaluation of S-Attributed Definitions

- We put the values of the synthesized attributes of the grammar symbols into a parallel stack.
  - When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.
- We evaluate the values of the attributes during reductions.

$A \rightarrow XYZ$ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized.



Bottom-Up Eval. of S-Attributed Definitions (cont.)

Production	Semantic Rules
L → E **return**	print(val[top-1])
E → E_1 + T	val[ntop] = val[top-2] + val[top]
E → T	
T → T_1 * F	val[ntop] = val[top-2] * val[top]
T → F	
F → (E)	val[ntop] = val[top-1]
F → **digit**	

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Bottom-Up Evaluation -- Example

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.

stack	val-stack	input	action	semantic rule
0		5+3*4r	s6	d.lexval(5) into val-stack
0d6	5	+3*4r	F→d	F.val=d.lexval – do nothing
0F4	5	+3*4r	T→F	T.val=F.val – do nothing
0T3	5	+3*4r	E→T	E.val=T.val – do nothing
0E2	5	+3*4r	s8	push empty slot into val-stack
0E2+8	5-	3*4r	s6	d.lexval(3) into val-stack
0E2+8d6	5-3	*4r	F→d	F.val=d.lexval – do nothing
0E2+8F4	5-3	*4r	T→F	T.val=F.val – do nothing
0E2+8T11	5-3	*4r	s9	push empty slot into val-stack
0E2+8T11*9	5-3-	4r	s6	d.lexval(4) into val-stack
0E2+8T11*9d6	5-3-4	r	F→d	F.val=d.lexval – do nothing
0E2+8T11*9F12	5-3-4	r	T→T*F	T.val=T_1.val*F.val
0E2+8T11	5-12	r	E→E+T	E.val=E_1.val*T.val
0E2	17	r	s7	push empty slot into val-stack
0E2r7	17-	$	L→Er	print(17), pop empty slot from val-stack
0L1	17	$	acc	

Top-Down Evaluation (of S-Attributed Definitions)

Productions	**Semantic Rules**
$A \rightarrow B$	print(B.n0), print(B.n1)
$B \rightarrow \mathbf{0}\ B_1$	$B.n0=B_1.n0+1$, $B.n1=B_1.n1$
$B \rightarrow \mathbf{1}\ B_1$	$B.n0=B_1.n0$, $B.n1=B_1.n1+1$
$B \rightarrow \varepsilon$	B.n0=0, B.n1=0

where B has two synthesized attributes (n0 and n1).

- Remember that: In a recursive predicate parser, each non-terminal corresponds to a procedure.

```
procedure A() {
    call B();                                              A → B
}
procedure B() {
    if (currtoken=0)  { consume 0; call B(); }             B → 0 B
    else if (currtoken=1) { consume 1; call B(); }         B → 1 B
    else if (currtoken=$)  {}    // $ is end-marker        B → ε
    else error("unexpected token");
}
```

```
procedure A() {
    int n0,n1;                          Synthesized attributes of non-terminal B
    call B(&n0,&n1);                    are the output parameters of procedure B.
    print(n0);   print(n1);
}                                                  All the semantic rules can be evaluated
procedure B(int *n0, int *n1) {                    at the end of parsing of production rules
    if (currtoken=0)
        {int a,b; consume 0; call B(&a,&b); *n0=a+1; *n1=b;}
    else if (currtoken=1)
        { int a,b; consume 1; call B(&a,&b); *n0=a; *n1=b+1; }
    else if (currtoken=$) {*n0=0; *n1=0; }    // $ is end-marker
    else error("unexpected token");
}
```

3 Explain syntax directed translation scheme for the Boolean expression ? example .
Ans:-

Syntax-Directed Definitions and Translation Schemes

- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

Boolean Expressions

- Boolean expressions have different translations depending on their context
  - Compute logical values – code can be generated in analogy to arithmetic expressions for the logical operators
  - Alter the flow of control – boolean expressions can be used as conditional expressions in statements: if, for and while.
- Control Flow Boolean expressions have two inherited attributes:
  - **B.true**, the label to which control flows if B is true
  - **B.false**, the label to which control flows if B is false
  - B.false = S.next means:

    if B is false, Goto whatever address comes after instruction S is completed.

    This would be used for $S \longrightarrow$ if (B) S1 expansion

    (in this case, we also have S1.next = S.next)
- Some language semantics decree that boolean expressions have so-called short-circuit semantics.
  - In this case, computing boolean operations may also have flow-of-control

    Example:

    ```
    if ( x < 100 || x > 200 &&  x != y )  x = 0;
    ```

    Translation:

    ```
    if x < 100 goto L2
    ifFalse x >200 goto L1
    ifFalse x != y goto L1
    L2:  x = 0
    L1:  …
    ```

4 obtain prefix and postfix notation ?

Ans:- We normally learn mathematics via **in**fix notation. That is, math teachers write binary operators (such as '+') surrounded by two operands like this: **A + B**. We compute the expression by knowing that somehow the values A and B must be added (isn't it obvious?) This is only natural because of our math development.

In some areas of computer science (such as compiler design), it makes much more sense to order expressions with prefix notation. In that case, the earlier example would be written as: **+AB**. The operator, +, precedes (comes before) the operands hence it is called **pre**fix notation. Doesn't it really make sense to know first what operation that you will be computing so that you can sort of get ready for

the right kind of problem and then to be told what the two values are. Prefix notation is sometimes called Polish notation because it was created by Jan Lukasiwicz.

Sometimes computer scientists use **post**fix notation. With postfix notation, you place the operator AFTER the operands so the example above would be written as: **AB+**. Postfix notation is sometimes called Reverse Polish notation. Note that prefix (or Polish notation) is sometimes called RPN which stands for Registered Parameter Numbers (see the Kent City High School Computer Club for details).

Examples:

Infix	**Prefix**	**Postfix**
(A + B) / D	/ + A B D	A B + D /
(A + B) / (D + E)	/ + A B + D E	A B + D E + /
(A - B / C + E)/(A + B)	/ + - A / B C E + A B	A B C / - E + A B + /
B ^ 2 - 4 * A * C	- ^ B 2 * * 4 A C	B 2 ^ 4 A * C * -

Let A = 2, B = 3, D = 4, and E = 5. Answer the following:

1. What is the final value of the prefix expression: + * A B - C D

2. What is the final value of the postfix expression: B C D A D - + - +

3. Which of the following is a valid expression (either postfix or prefix)?
   B C * D - +
   * A B C -
   B B B * *

5 Three address, quadruple direct and indirect triple ?
Ans:- **Quadruple Representation**

Using quadruple representation, the three-address statement x = y op z is represented by placing op in the operator field, y in the operand1 field, z in the operand2 field, and x in the result field. The statement x = op y, where op is a unary operator, is represented by placing op in the operator field, y in the operand1 field, and x in the result field; the operand2 field is not used. A statement like param t1 is represented by placing param in the operator field and t1 in the operand1 field; neither operand2 nor the result field are used. Unconditional and conditional jump statements are represented by placing the target labels in the result field. For example, a quadruple representation of the three-address code for the statement x = (a + b) * - c/d is shown in Table 1. The numbers in parentheses represent the pointers to the triple structure.

Table 1: Quadruple Representation of x = (a +b) * – c/d

**  **	**Operator**	**Operand1**	**Operand2**	**Result**
(1)	+	a	b	t1
(2)	–	c		t2
(3)	*	t1	t2	t3
(4)	/	t3	d	t4
(5)	=	t4		x

Triple Representation
The contents of the operand1, operand2, and result fields are therefore normally the pointers to the symbol records for the names represented by these fields. Hence, it becomes necessary to enter temporary names into the symbol table as they are created. This can be avoided by using the position of the statement to refer to a temporary value. If this is done, then a record structure with three fields is

enough to represent the three-address statements: the first holds the operator value, and the next two holding values for the operand1 and operand2, respectively. Such a representation is called a "triple representation". The contents of the operand1 and operand2 fields are either pointers to the symbol table records, or they are pointers to records (for temporary names) within the triple representation itself. For example, a triple representation of the three-address code for the statement x= (a+b)*−c/d is shown in Table 2.

Table 2: Triple Representation of x = (a + b) * − c/d

	Operator	**Operand1**	**Operand2**
(1)	+	a	b
(2)	−	c	
(3)	*	(1)	(2)
(4)	/	(3)	d
(5)	=	x	(4)

Indirect Triple Representation

Another representation uses an additional array to list the pointers to the triples in the desired order. This is called an indirect triple representation. For example, a triple representation of the three-address code for the statement x= (a+b)*−c/d is shown in Table 3.

Table 3: Indirect Triple Representation of x= (a + b) * − c/d

		Operator	**Operand1**	**Operand2**
(1)	(1)	+	a	b
(2)	(2)	−	c	
(3)	(3)	*	(1)	(2)
(4)	(4)	/	(3)	d
(5)	(5)	=	x	(4)

Comparison

By using quadruples, we can move a statement that computes A without requiring any changes in the statements using A, because the result field is explicit. However, in a triple representation, if we want to move a statement that defines a temporary value, then we must change all of the pointers in the operand1 and operand2 fields of the records in which this temporary value is used. Thus, quadruple representation is easier to work with when using an optimizing compiler, which entails a lot of code movement. Indirect triple representation presents no such problems, because a separate list of pointers to the triple structure is maintained. When statements are moved, this list is reordered, and no change in the triple structure is necessary; hence, the utility of indirect triples is almost the same as that of quadruples.

5 Define three address ? Write alteast six different type of three address ?
Ans:-

Three Address Code

- Consists of a sequence of instructions, each instruction may have up to three addresses, prototypically
  t1 = t2 op t3
- Addresses may be one of:
  - A name. Each name is a symbol table index. For convenience, we write the names as the identifier.
  - A constant.
  - A compiler-generated temporary. Each time a temporary address is needed, the compiler generates another name from the stream t1, t2, t3, etc.
    - Temporary names allow for code optimization to easily move instructions
    - At target-code generation time, these names will be allocated to registers or to memory.

6 TAC generation ?
Ans:-

Three Address Code Instructions

- Symbolic labels will be used as instruction addresses for instructions that alter the flow of control. The instruction addresses of labels will be filled in later.
  L: t1 = t2 op t3
- Assignment instructions: x = y op z
  - Includes binary arithmetic and logical operations
- Unary assignments: x = op y
  - Includes unary arithmetic op (-) and logical op (!) and type conversion
- Copy instructions: x = y
  - These may be optimized later.

- Unconditional jump: goto L
  - L is a symbolic label of an instruction
- Conditional jumps:
  if x goto L and ifFalse x goto L
  - Left: If x is true, execute instruction L next
  - Right: If x is false, execute instruction L next
- Conditional jumps:
  if x relop y goto L
- Procedure calls. For a procedure call p(x1, …, xn)
  param x1
  …
  param xn
  call p, n

- Indexed copy instructions: x = y[i] and x[i] = y
  - Left: sets x to the value in the location [i memory units beyond y] (in C)
  - Right: sets the contents of the location [i memory units beyond y] to x
- Address and pointer instructions:
  - x = &y sets the value of x to be the location (address) of y.
  - x = *y, presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.
  - *x = y sets the value of the object pointed to by x to the value of y.
- In Java, all object variables store references (pointers), and Strings and arrays are implicit objects:
  - Object o = "some string object", sets the reference o to hold the address of this string. The String object itself is shared, not copied by value.
  - x = y[i], uses the implicit length-aware array object y; there is full object here, not just array contents.

Three Address Code Representation

- Representations include quadruples (used here), triples and indirect triples.
- In the quadruple representation, there are four fields for each instruction: op, arg1, arg2 and result.
  - Binary ops have the obvious representation
  - Unary ops don't use arg2
  - Operators like param don't use either arg2 or result
  - Jumps put the target label into result

UNIT-4

1 Define activation tree ? What do you mean by life time of an activation of a procedure ?
Ans:

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
  - The root represents the activation of main program
  - Each node represents an activation of procedure
  - The node **a** is parent of **b** if control flows from **a** to **b**
  - The node **a** is to the left of node **b** if lifetime of **a** occurs before **b**

1 Dynamic storage allocation ?
Ans:-

Dynamic Storage Allocation

```
new(p);     p^.key:=k;     p^.info:=i;
```

Garbage : unreachable cells
- Lisp does garbage collection
- Pascal and C do not

```
head^.next := nil;
```

Dangling reference

```
dispose(head^.next )
```

1 What is activation record ? Explain different field .
Ans:- Activation records

Temporaries
local data
machine status
Access links
Control links

Parameters
Return value

- **temporaries:** used in expression evaluation
- **local data:** field for local data
- **saved machine status:** holds info about machine status before procedure call
- **access link :** to access non local data
- **control link :**points to activation record of caller
- **actual parameters:** field to hold actual parameters
- **returned value**: field for holding value to be returned

2 What the use of symbol table ? Explain various field and way to implement the symbol table .
Ans:- syntax tree have been constructed, the compiler must check whether the input program is typecorrect
(called type checking and part of the semantic analysis). During type checking, a compiler
checks whether the use of names (such as variables, functions, type names) is consistent with their definition in the program. Consequently, it is necessary to remember declarations so that we can detect inconsistencies and misuses during type checking. This is the task of **a symbol table**.

UNIT-5

1 Describe local and global optimization ?

Ans:- Local optimizations

These only consider information local to a function definition. This reduces the amount of analysis that needs to be performed (saving time and reducing storage requirements) but means that worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

Global optimization

The most common form is the minimization of one real-valued function f in the parameter-space $\vec{x} \in P$. There may be several constraints on the solution vectors $\vec{x}_{min}$.

In real-life problems, functions of many variables have a large number of local minima and maxima. Finding an arbitrary local optimum is relatively straightforward by using local optimisation methods. Finding the global maximum or minimum of a function is much more challenging and has been practically impossible for many problems so far.

The maximization of a real-valued function g(x) can be regarded as the minimization of the transformed function $f(x) := (-1) \cdot g(x)$.

.

2 Explain blocks?

Ans:- A **basic block** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block: t1 := a*a
t2 := a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4+t5

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

3 Define directed acyclic graph?

Ans:- DAGs may be used to model several different kinds of structure in mathematics and computer science. A collection of tasks that must be ordered into a sequence, subject to constraints that certain tasks must be performed earlier than others, may be represented as a DAG with a vertex for each task and an edge for each constraint; algorithms for topological ordering may be used to generate a valid sequence. DAGs may also be used to model processes in which information flows in a consistent direction through a network of processors. The reachability relation in a DAG forms a partial order, and any finite partial order may be represented by a DAG using reachability. Additionally, DAGs may be used as a space-efficient representation of a collection of sequences with overlapping subsequences.

The corresponding concept for undirected graphs is a forest, an undirected graph without cycles. Choosing an orientation for a forest produces a special kind of directed acyclic graph called a polytree. However there are many other kinds of directed acyclic graph that are not formed by orienting the edges of an undirected acyclic graph. For this reason it may be more accurate to call directed acyclic graphs **acyclic directed graphs** or **acyclic digraphs**.

4 Differentiate stack allocation and heap allocation ?

Ans:

Static allocation

-

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- In such a case de-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records

1 Short note on

(a) Global data flow analysis

(b) loop optimization

(c) Error recovery technique used in parsing

(d) YACC

(e) LEX

Ans:- (a) Global data flow analysis- Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

Forward Analysis

The reaching definition analysis calculates for each program point the set of definitions that may potentially reach this program point.

```
1: if b==4 then
2:   a = 5;
3: else
```

The reaching definition of variable "a" at line 7 is the set of assignments a=5 at line 2 and a=3 at line 4.

```
4:   a = 3;
5: endif
6:
7: if   a  <  4
then
8:   ...
```

Backward Analysis

The live variable analysis calculates for each program point the variables that may be potentially read afterwards before their next write update. The result is typically used by dead code elimination to remove statements that assign to a variable whose value is not used afterwards.

The in-state of a block is the set of variables that are live at the end of the block. Its out-state is the set of variable that is live at the start of it. The in-state is the union of the out-states of the blocks successors. The transfer function of a statement is applied by making the variables that are written dead, then making the variables that are read live.

```
 // out: {}
b1: a = 3;
   b = 5;
   d = 4;
   if a > b then
 // in: {a,b,d}

 // out: {a,b}
b2:   c = a + b;
     d = 2;
 // in: {b,d}

 // out: {b,d}
b3: endif
   c = 4;
   return b * d + c;
 // in:{}
```

The out-state of b3 only contains b and d, since c has been written. The in-state of b1 is the union of the out-states of b2 and b3. The definition of c in b2 can be removed, since c is not live immediately after the statement.

Solving the data-flow equations starts with initializing all in-states and out-states to the empty set. The work list is initialized by inserting the exit point (b3) in the work list (typical for backward flow). Its computed out-state differs from the previous one, so its predecessors b1 and b2 are inserted and the process continues. The progress is summarized in the table below.

processing	in-state	old out-state	new out-state	work list
b3	{}	{}	{b,d}	(b1,b2)
b1	{b,d}	{}	{}	(b2)
b2	{b,d}	{}	{a,b}	(b1)
b1	{a,b,d}	{}	{}	()

Note that b1 was entered in the list before b2, which forced processing b1 twice (b1 was re-entered as predecessor of b2). Inserting b2 before b1 would have allowed earlier completion.

Initializing with the empty set is an optimistic initialization: all variables start out as dead. Note that the out-states cannot shrink from one iteration to the next, although the out-state can be smaller that the in-state. This can be seen from the fact that after the first iteration the out-state can only change by a change of the in-state. Since the in-state starts as the empty set, it can only grow in further iterations.

(b) loop optimization –**loop optimization** plays an important role in improving cache performance, making effective use of parallel processing capabilities, and reducing overheads associated with executing loops. Most execution time of a scientific program is spent on loops. Thus a lot of compiler analysis and compiler optimization techniques have been developed to make the execution of loops faster.

(c) Error recovery technique used in parsing -We concentrated on error reporting in the previous section; in this section, we discuss the problem of error recovery. When an error is detected, the bison parser is left in an ambiguous position. It is unlikely that meaningful processing can continue without some adjustment to the existing parser stack.

Depending on the environment in which you'll be using your parser, error recovery may not always be necessary if the environment makes it easy to correct the error and rerun the parser. In other environments such as a compiler, it may be possible to recover from the error enough to continue parsing and look for additional errors, stopping the compiler at the end of the parse stage. This technique can improve the productivity of the programmer by shortening the edit-compile-test cycle, since several errors can be repaired in each iteration of the cycle.

(d) YACC-Yet Another Compiler-Compiler

omputer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ``input language'' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that han- dles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

(e) LEX -**A Lexical Analyzer Generator**

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

2 Explain register allocation ?

Ans:-

- Register allocation
  - What computational results are kept in registers
  - Possibly not enough registers for all values
- Simple case
  - Formals and locals not given registers (always spilled on stack)
  - Register allocation for expression evaluation
- Some registers have restrictions
  - $a0 used to return values etc.
  - See slides of previous classes

Weighted Register Allocation

- Two-phase solution
  - Dynamic Programming [Sethi & Ullman]
  - Optimal under certain conditions
    - ☒ uniform instruction cost
    - ☒ `symbolic' trees
- Bottom-up (labeling)
  - Compute weight for every subtree
    - ☒ the minimal number of registers needed
- Top-Down
  - Generate the code using labeling by preferring "heavier" subtrees (larger labeling)
  - W(e1 + e2) ← W(e1) if W(e1) > W(e2)
  - W(e1 + e2) ← W(e2) if W(e2) > W(e1)
  - W(e1 + e2) ← W(e1)+ 1 o/w

3 What is Loop invariant ? How will you detect loop invariant computation ,give algorithm ?
Ans:- Loop-Invariant code motion is a form of partial redundancy elimination (PRE) whose purpose it is to find code in a loop body that produces the same value in every iteration of the loop. This code can be moved out of the loop so that it is not computed over and over again. It is an undecidable question if a fragment of a loop body has the same effect with each iteration, but we can easily come up with a reasonable conservative approximation.
The computation d = a b (for an operator 2 f+; ; ——————————————————g) is loopinvariant for a loop if
1. a; b are numerical constants,
2. a; b are defined outside the loop
(for non-SSA this means that all reaching definitions of a; b are outside the loop), or
3. a; b are loop invariants
(for non-SSA this means that there is only one reaching definition of a; b and that is loop-invariant). Loop-invariant computations can easily be found by collecting computations according to the three steps above repeatedly until nothing changes anymore.

If we find a loop-invariant computation in SSA form, then we just move it out of the loop to a block before the loop. When moving a (side-effectfree) loop-invariant computation to a previous position, nothing can go wrong, because the value it computes cannot be overwritten later and the values it depends on cannot have been changed before (and either are already or can be placed outside the loop by the loop-invariance condition). In fact, it's part of the whole point of SSA do be able to do simple global code motion and have the required dataflow analysis be trivial. In order to make sure we do not needlessly compute the loop-invariant expression in the case when the loop is not entered, we can add an extra basic block like for critical edges. This essentially turns

```
j = loopinv
while (e) f
S
g
```

into

```
if (e) f
j = loopinv
while (e) f
S
g
g
```

The transformation is often more efficient on the intermediate representation level. This, of course, depends on e being side-effect free, otherwise extra precautions have to be done.

For non-SSA form, we have to be much more careful when moving a loop-invariant computation. See Figure 1.

Moving a loop-invariant computation d = a b before the loop is still okay on non-SSA if

1. that computation d = a b dominates all loop exists after which d is still live (violated in Figure 1b),
2. and d is only defined once in the loop body (violated in Figure 1c),
3. and d is not live after the block before the loop (violated in Figure 1d)

a Good:

```
L0: d = 0
L1: i = i + 1
d = a b
M[i] = d
if (i<N) goto L1
L2: x = d
```

b Bad:

```
L0: d = 0
L1: if (i>=N) goto L2
i = i + 1
d = a b
M[i] = d
goto L1
L2: x = d
```

c Bad:

```
L0: d = 0
L1: i = i + 1
d = a b
M[i] = d
d = 0
M[j] = d
if (i<N) goto L1
L2:
```

d Bad:

```
L0: d = 0
L1: M[j] = d
i = i + 1
d = a b
```

M[i] = d
if (i<N) goto L1
L2: x = d
Figure 1: Good and bad examples for code motion of the loop-invariant computation d=ab. a: good. b: bad, because d used after loop, yet should not be changed if loop iterates 0 times c: bad, because d reassigned in loop body, thus would be killed. d: bad, because initial d used in loop body before computing d=ab.

Detecting Loop Invariant Computation

Input: Basic Blocks and CFG, Dominator Relation, Loop Information
Output: Instructions that are Loop Invariant

```
InvSet = ∅
repeat
 for each instruction i ∉ InvSet
   if operands are constants, or
have definitions outside the loop, or
have exactly one definition d ∈ InvSet;
   then
     InvSet = InvSet U {i};
 until no changes in InvSet;
```