

Question: Explain the term Function Chunking

Answer: Function chunking is a compiler optimisation for improving code locality. Profiling information is used to move rarely executed code outside of the main function body.

Question: Explain the three (3) main techniques for loop optimization

Question: List and explain three loop optimization technique

Answer:

- **Strength Reduction** which replaces an expensive (time consuming) operator by a faster one;
- **Induction Variable Elimination** which eliminates variables from inner loops;
- **Code Motion** which moves pieces of code outside loops.

Question: Mention the semantic errors that can be recognized by the semantic analyzer

Answer:

1. "identifier not declared"; either omitted declaration or misspelt an identifier.
2. incompatible use of types e.g assigning boolean to strings
3. Parameter or subscript miscount

Question: Consider the grammar below:

G: $E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / i$

Find FIRST(E)

Answer:

Elements of FIRST(E) = { (, i }

Elements of FIRST(E') = { +, ϵ }

Elements of FIRST T') = { * , ϵ }

Question: Define target programme and give three examples

Answer: Target Programmes

The output of the code generator is the target programme. It may take on a variety of forms:

- a. absolute machine language,
- b. relocatable machine language, or
- c. Assembly language.

Question: Mention three ways of removing ambiguity from the grammar

Answer:

when there is a ambiguity; scanner generators chooses a rule that matches the maximum number of characters. This disambiguation rule is called the longest match

rule. If there are more than one rule that match the same maximum number of characters, the rule listed first is chosen. This is the rule priority disambiguation rule.

Question: Enumerate the five (5) transformation techniques used by most optimisation algorithms

Question: write down the various most frequently applied transformation techniques in common optimization algorithm

Answer:

- i. Function-preserving transformations
- ii. common sub-expressions identification/elimination
- iii. copy propagation
- iv. dead-code elimination
- v. Loop optimisations
- vi. induction variables and reduction in strength
- vii. code motion
- viii. Function Chunking.

Question: What are the benefits of LR parsing

Question: Enumerate three (3) benefits of LR parsing.

Question: Outline the benefits and the drawback of LR parsing

Answer:

Benefits:

- i. LR parsing can handle a larger range of languages than LL parsing
- ii. LR parsers can be constructed to recognise virtually all programming language constructs for which context-free grammars can be written
- iii. It is more general than operator precedence or any other common shift-reduce techniques, yet can be implemented with the same efficiency as other methods
- iv. the most general nonbacktracking parsing method.

Drawback:

It is too much work to implement an LR parser by hand. However, LR parser generator are now readily available

Question: What is the major role and the categories of optimization within a compiler

Answer: Optimisation within a compiler is concerned with improving in some way the generated object code while ensuring the result is identical.

Optimisations fall into three categories:

- a. **Speed:** improving the runtime performance of the generated object code.
- b. **Space:** reducing the size of the generated object code
- c. **Safety:** reducing the possibility of data structures becoming corrupted

Question: Suppose we have a grammar G:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow a$

Find the left parse of the sentence $a * (a + a)$

Answer:

$E \rightarrow T$ (2)
 $T \rightarrow T * F$ (3)
 $F \rightarrow T$ (4)
 $a \rightarrow F$ (6)
 $a \rightarrow (E)$ (5)
 $a \rightarrow (T + T)$ (1)
 $a \rightarrow (T + T)$ (2)
 $a \rightarrow (F + T)$ (4)
 $a \rightarrow (a + T)$ (6)
 $a \rightarrow (a + F)$ (4)
 $a \rightarrow (a + a)$ (6)
Sequence of derivation is 23465124646

Question: What are the different areas that causes problems during the measurement on the performance of an actual compiler

Question: Outline the problematic areas when considering the performance measurement of an actual compiler

Answer:

- i. **multiple routine** calls during lexical analysis for each and every source character
- ii. skipping **white space** during lexical analysis
- iii. skipping over a **comment**
- iv. decoding a **tightly packed parse table** during syntax analysis
- v. **looking** things up in the **name table** during semantic analysis
- vi. determining whether some name is a **reserved keyword** or a user definable identifier.

Question: Describe any three (3) reasons for studying LR grammars

Question: What are the reasons for studying LR grammars:

Answer:

- i. LR (1) grammars are often used to construct parsers
- ii. virtually all context-free programming language constructs can be expressed in an LR(1) form
- iii. LR grammars are the most general grammars parse-able by a deterministic, bottom-up parser

- iv. efficient parsers can be implemented for LR(1) grammars
- v. LR parsers detect an error as soon as possible in a left-to-right scan of the input
- vi. • LR grammars describe a proper superset of the languages recognised by predictive (i.e., LL) parsers

Question: Outline the steps in Lex implementation.

Question: List out the steps in Lex implementation

Answer:

1. Read input lang. spec
2. Construct NFA with epsilon-moves
3. Convert NFA to DFA
4. Optimise the DFA
5. Generate parsing tables & code

Question: Outline the properties provides by optimizing compilers

Answer:

- i. the transformations should preserve the semantics of the programmes,
- ii. the transformations should speed up considerably the compiler on the average
- iii. the transformation should be worth the intellectual effort.

Question: Given the following simple source program:

```
while a < b do
  if c < d then
    x := y + z
  else
    x := y - z
```

Generate an equivalent intermediate code

Answer:

```
L1: if (a < b) goto L2
    goto Lnext
L2: if (c < d) goto L3
    goto L4
L3: t1 := y + z
    x := t1
    goto L1
L4: t2 := y - z
    x := t2
    goto L1
Lnext:
```

Question: Outline the four actions adopted when implementing the shift reduce algorithm

Answer:

Actions

- a. **Shift** - push token onto stack
- b. **Reduce** - remove handle from stack and push on corresponding nonterminal
- c. **Accept** - recognise sentence when stack contains only the distinguished symbol and input is empty
- d. **Error** - happens when none of the above is possible; means original input was not a sentence.

Question: Define parse tree

Answer:

Parse tree is a graphical representation for derivation that filters out the choice regarding replacement. It has the important purpose of making explicit the hierarchical syntactic structure of sentences that is implied by the grammar.

Question: With sample representations, describe the various types of parse trees.

Answer:

Question: List and discuss two (2) types of grammar

Question: Outline the four basic types of grammars

Answer:

- a. **Type-0** grammars (unrestricted grammars) include all formal grammars.
- b. **Type-1** grammars (context-sensitive grammars) generate the context-sensitive languages.
- c. **Type-2** grammars (context-free grammars) generate the context-free languages.
- d. **Type-3** grammars (regular grammars) generate the regular languages.

Question: With examples, define a lexeme

Answer:

A lexeme is a sequence of characters that matches the pattern for a token, e.g.,

- **identifiers:** count, x1, i, position
- **keywords:** if
- **operators:** =, ==, !=, +=

Question: What are the roles of a parser

Answer:

1. The parser reads the sequence of tokens generated by the lexical analyser
2. It verifies that the sequence of tokens obeys the correct syntactic structure of the programming language.
3. It enters information about tokens into the symbol table
4. It reports syntactic errors to the user.

Question: Given the expression

$$X = (a + (b * c)) / (a - (b * c))$$

Generate the corresponding parse tree

Answer:

Question: With diagrammatic representation, give a description of the directed acyclic graph

Answer: A directed acyclic graph depicts the natural hierarchical structure of a source programme in a compact way, because sub-expressions are identified

Question: How is directed acyclic graph different from a parse tree

Answer:

Question: what is the output from an intermediate phase of a compiler

Answer: Target program. The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language

Question: Given the following source code statement:

$c = a + b * S$

show the phases of compilation process and their results from the initial phase upto the intermediate code level

Answer:

Question: Given the expression

$d := (a - b) + (a - c) + (a - c)$

Write the three address code

Answer:

$t1 := a - b$

$t2 := a - c$

$t3 := t2 + t2$

$t4 := t1 + t3$

$d := t4$

Question: Generate the corresponding target machine code for the expression in prev

Answer:

LOAD R1, a // Step 1: Load a into R1

LOAD R2, b // Step 1: Load b into R2

SUB R1, R2 // Step 1: $R1 = a - b$ (t1)

LOAD R2, a // Step 2: Load a into R2

LOAD R3, c // Step 2: Load c into R3

SUB R2, R3 // Step 2: $R2 = a - c$ (t2)

ADD R2, R2 // Step 3: $R2 = t2 + t2$ (t3)

ADD R1, R2 // Step 4: $R1 = t1 + t3$ (t4)

STORE R1, d // Step 5: Store t4 into d

Question: Complete the table by writing out the corresponding semantic rules for each of the production rule

Answer:

Question: What are the drawbacks of syntax analyzer that makes semantic analysis phase very important

Answer:

1. Lack of Meaningful Interpretation
2. No Type Checking
3. No Type Checking
4. No Control Flow Analysis
5. No Enforcement of Language-Specific Constraints

Question: Discuss Nondeterministic Finite Automaton (NFA)

Answer: An NFA accepts an input string x if there is a path in the transition graph from the initial state to a final state that spells out x . The language defined by an NFA is the set of strings accepted by the NFA.

Question: What does a nondeterministic finite automaton (NFA) consist of

Answer:

- a finite set of **states** S
- an **input alphabet** consisting of a finite set of symbols Σ
- a **transition function** δ that maps $S \times (\Sigma \cup \{\epsilon\})$ to subsets of S .
- an **initial state** s_0 in S
- F , a subset of S , called the **final** (or accepting) **states**.

Question: Differentiate between Deterministic and Non-deterministic Finite automata

Answer: An NFA is similar to a DFA but it also permits multiple transitions over the same character and transitions over ϵ

Aspect	Deterministic Finite Automata (DFA)	Non-deterministic Finite Automata (NFA)
Definition	A finite automaton where for each state and input symbol, there is exactly one possible next state.	A finite automaton where for each state and input symbol, there may be multiple possible next states or none.
Transitions	Each state has exactly one outgoing transition for every input symbol.	A state can have multiple transitions for the same input symbol, or even no transition.

Epsilon (ϵ) Transitions	Does not allow epsilon transitions (i.e., moves without consuming input).	Allows epsilon transitions, where the automaton can change states without reading any input.
Determinism	Fully deterministic: given the current state and input symbol, the next state is uniquely determined.	Non-deterministic: multiple paths may exist for the same input symbol and state.
Representation	Easier to implement in software or hardware because of its deterministic nature.	Conceptually simpler, but requires transformation to a DFA for practical implementation.
Acceptance	Accepts input if it reaches a final state after processing all input symbols.	Accepts input if any possible path leads to a final state.
State Count	May require more states than an equivalent NFA.	Often requires fewer states than an equivalent DFA.
Conversion	Cannot be directly converted to an NFA (already deterministic).	Can be converted to a DFA using the subset construction algorithm.
Execution Complexity	Processes input in linear time relative to the input length (one active state at a time).	May involve exploring multiple paths simultaneously, but equivalent DFA processes in linear time.

Question: How is finite automaton different from the transition tree

Answer:

Question: Write down what is involved when considering code generation task

Answer:

Question: differentiate between Kleene star and Kleene plus closure

Answer: The Kleene Star applies to a set or expression and represents zero or more repetitions of the pattern. The Kleene Plus is similar to the Kleene Star but represents one or more repetitions of the pattern. The kleene plus closure excludes the empty string as valid match

Question: Explain the Dead Code Elimination Concept

Question: Define dead code elimination and give example with explanation

Answer: Dead code elimination is a size optimisation that aims to remove logically impossible statements from the generated object code.

Consider the following programme:

```

a = 5
if (a != 5) {
// some complicated calculation
}

```

...

Since the body of an if(false) statement will never execute - it is dead code we can rewrite the code:

```

a = 5
// some statements

```

Question: Define Formal Grammar

Question: Explain the term: Formal Grammar.

Answer: A formal grammar (sometimes called a grammar) is a set of rules of a specific kind, for forming strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar describes only the form of the strings and not the meaning or what can be done with them in any context.

Question: Enumerate three (3) functions of a lexical analyser

Answer:

1. keeping track of line numbers
2. producing an output listing if necessary,
3. stripping out white space (such as redundant blanks and tabs), and
4. Deleting comments.

Question: Outline the implementation of a shift-reduce parser using a stack.

- i. start with an empty stack
- ii. a "shift" action corresponds to pushing the current input symbol onto the stack
- iii. a "reduce" action occurs when we have a handle on top of the stack. To perform the reduction, we pop the handle off the stack and replace it with the nonterminal on the LHS of the corresponding rule. This is referred to as **handle pruning**.

Question: Consider the grammar below:

Sentence $\rightarrow$ NounPhrase VerbPhrase

NounPhrase $\rightarrow$ Art Noun

VerbPhrase $\rightarrow$ Verb | Adverb Verb

Art $\rightarrow$ the | a | ...

Verb $\rightarrow$ jumps | sings | ...

Noun $\rightarrow$ dog | cat | ...

For the input: **the dog jumps**, show the implementation of bottom up parsing by completing the table below.

Stack	Input Sequence	Action
	<i>the dog jumps</i>	Shift <i>the</i>

<i>the</i>	<i>dog jumps</i>	Reduce using <i>Art</i> → <i>the</i>
<i>Art</i>	<i>dog jumps</i>	Shift <i>dog</i>
<i>Art dog</i>	<i>jumps</i>	Reduce using <i>Noun</i> → <i>dog</i>
<i>Art Noun</i>	<i>Jumps</i>	Reduce using <i>NounPhrase</i> → <i>Art Noun</i>
<i>Nounphrase</i>	<i>Nounphrase</i>	Shift <i>jumps</i>
<i>Nounphrase Jumps</i>		Reduce using <i>Verb</i> → <i>jumps</i>
<i>Nounphrase Verb</i>		Reduce using <i>Verbphrase</i> → <i>verb</i>
<i>Nounphrase Verbphrase</i>		Reduce using <i>Sentence</i> → <i>Nounphrase Verbphrase</i>
<i>Sentence</i>		Parsing Complete

Question: Outline the algorithm for shift-reduce parsing

Answer:

1. **Step 1:** Start with the sentence to be parsed
2. **Step 2:** Until the sentential form is the start symbol do:
 - a) Scan through the input until we recognise something that corresponds to the RHS of one of the production rules (this is called a handle)
 - b) replace the RHS of the rule which appears in the sentential form with the LHS of the rule. (an action known as a reduction)

Question: State any three difficulties in top-down parsing

Question: Describe three (3) difficulties with top-down parsing

- i. Which production will you apply?
- ii. A left recursive production, can lead to continue cycling without getting to the answer
- iii. There is a particular sequence that will lead you to the input string.
- iv. A wrong production will require starting all over again.

Question: Enumerate the errors which can be detected during lexical analysis

Question: Discuss the following (include examples where possible)

Answer:

- i. Errors during Lexical Analysis:
 - a) Strange characters: Some programming languages do not use all possible characters, so any strange ones which appear can be reported.
 - b) Long quoted strings I: Many programming languages do not allow quoted strings to extend over more than one line; in such cases a missing quote can be detected.
 - c) Invalid numbers: A number such as 123.45.67 could be detected as invalid during lexical analysis

- ii. Errors during Syntax Analysis: during syntax analysis the compiler automatically generate a useful error message just by listing the tokens which would be acceptable at that point.
- iii. Errors during Semantic Analysis: One of the most common errors reported during semantic analysis is
 - a) "identifier not declared",
 - b) others include incompatible type, or
 - c) parameter miscount

Question: Enumerate five (5) common run-time errors which can be detected by most IDEs with its debugging option

Answer:

- attempt to divide by 0
- overflow (and possibly underflow) during arithmetic operations.
- attempt to use a variable before it has been set (undefined variable)
- attempt to refer to a non-existent array element (invalid subscript).
- attempt to set a variable to some value outside its range
- various errors related to pointers:
- Attempt to use a pointer before it has been set
- attempt to use a nil pointer
- attempt to use a pointer which points outside an array
- attempt to use a pointer after the memory it points to has been released.

Question: Describe any three (3) structures which can be used to implement symbol tables (include the computational complexity of each structure)

Answer:

a. Linear List

- a) $O(n)$ probes per lookup
- b) easy to expand — no fixed size
- c) one allocation per insertion

b. Ordered Linear List

- a) $O(\log_2 n)$ probes per lookup using binary search
- b) insertion is expensive (to reorganise list)

c. Binary Tree

- a) $O(n)$ probes per lookup, if the tree is unbalanced
- b) $O(\log_2 n)$ probes per lookup, if the tree is balanced
- c) easy to expand with no fixed size
- d) one allocation per insertion

d. Hash Table

- a) $O(1)$ probes per lookup on the average
- b) expansion costs vary with specific scheme

Question: Draw a syntax tree for the assignment statement $a := b * -c + a * -c$.

Answer:

Question: Enumerate three (3) examples of analytic grammar formalisms.

Answer:

- i. the Language Machine directly implements **unrestricted analytic grammars**.
- ii. **top-down parsing language (TDPL)**
- iii. **link grammars**: a form of analytic grammar designed for linguistics
- iv. **parsing expression grammars (PEGs)**

Question: Describe any three (3) phases of a compiler.

Answer:

1. **The Lexical Analyser:** this is the first phase and it is also referred to as the Scanner. It separates characters of the source language into groups that logically belong together; these groups are called **tokens**
2. **The Syntax Analyser:** This groups tokens together into syntactic structures. For example, the three tokens representing A+B might be grouped into a syntactic structure called an **expression**.
3. **The Intermediate Code Generator:** This uses the structure produced by the syntax analyser to create a stream of simple instructions.
4. **Code Optimisation:** This is an optional phase designed to improve the intermediate code so that the ultimate object programme runs faster and/or takes less space.
5. **Code Generation:** This is the final phase and it produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done.

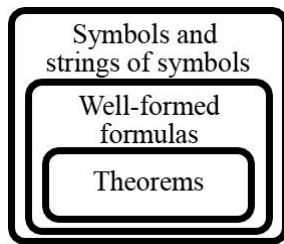
6. **Question:** Summarize the languages, automata and production rules of Chomsky's Four Types of Grammars by completing the table below:

Answer:

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restriction)
Type-1	Contextsensitive	Linear-bounded nondeterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Nondeterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow \alpha$ $A\beta \rightarrow \alpha\gamma\beta$

Question: Using a diagram, show the syntactic divisions within a Formal System.

Answer:



Symbols and strings of symbols may be broadly divided into nonsense and wellformed formulas. The set of well-formed formulas is divided into theorems and non-theorems. However, quite often, a formal system will simply define all of its well-formed formula as theorems.

Question: Consider the grammar, below

$$G: E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid i$$

Evaluate the augmented grammar.

Answer:

The augmented grammar for this grammar is grammar G' below:

$$G': E' \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid i$$

Question: Explain what is meant by top-down parsing technique

Answer: A Top-down parser builds a parse tree by starting at the root and working down towards the leaves.

Question: State the properties of LR parser

Question: List the types of LR parser.

Answer:

Simple LR (SLR) Parser

Canonical LR Parser

Lookahead LR (LALR) Parser

Question: Given the production, $A \rightarrow A\alpha$,

Answer:

i. Identify the problem with this production: The production rule is left recursive which can lead to continuous cycling without getting to the answer.

ii. Show how the problem can be removed

If you have a production like: $A \rightarrow A\alpha$. This is left recursive but it can be removed by having

$A \rightarrow A\alpha \mid \beta$

You can set

$A \rightarrow \beta A'$

Then set $A' \rightarrow \alpha A' \mid \varepsilon$

Question: In top-down parsing, order of trying the alternatives may influence getting the input or not.

Answer

i) State the method that can be used to overcome this.

factoring i.e left factoring is used to ensure that the right sequence/order is used.

ii. Hence given the grammar,

G: $A \rightarrow \alpha\beta$

$A \rightarrow \alpha\gamma$

How do we get rid of this dilemma using the method stated in (i) above?

We can get rid of

this dilemma by using left factoring to determine the right sequence.

That is the above becomes:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

Question: What do you understand by the term Viable Prefix?

Answer: Suppose $S \Rightarrow^* \alpha A \omega \Rightarrow \alpha \beta \omega$ in a rightmost derivation in grammar G. Then γ is a viable prefix of G if γ is a prefix of $\alpha\beta$.

Question: Consider the grammar,

$G: S \rightarrow L = R \mid R$

$L \rightarrow *R \mid i$

$R \rightarrow L$

- Compute all the LR(0) items for the above grammar
- Construct an NFA whose states are the LR(0) items from (a)

Question: Define the following for any given grammar?

Answer:

- FOLLOW A:** the set of terminals a , that can appear immediately to the right of A in some sentential form.
- FIRST(α):** the set of terminals that begin strings derived from α^* . If $\alpha \Rightarrow * \varepsilon$, then ε is also in FIRST(α).

Question: Given the grammar G with following production rules, $F \rightarrow c \mid cF \mid dF$, determine whether the string **ccdcdddc** can be generated by the grammar

Answer:

Question: What are the characteristics of Simple LR

Question: List the common techniques for building tables for an “LR” parser stating the characteristics of each?

I. simple LR (SLR (1) for short) characteristics:

- smallest class of grammars
- smallest tables (number of states)
- simple, fast construction

II. canonical LR (or LR (1)).

- full set of LR(1) grammars
- largest tables (number of states)
- slow, expensive and large construction

III. lookahead LR (LALR (1) for short).

- intermediate sized set of grammars
- same number of states as SLR(1)
- canonical construction is slow and large
- better construction techniques exist

Question: Consider the following grammar for list structure:

$S \rightarrow a \mid ^ \mid (T)$

$T \rightarrow T,S \mid S$

Find the rightmost derivations for:

Answer:

(i) $(a, (a, a))$: $S \Rightarrow (T) \Rightarrow (T,S) \Rightarrow (a,S) \Rightarrow (a,(T)) \Rightarrow (a,(S,S)) \Rightarrow (a,(a,a))$

(ii) $((a, a), ^, (a)), a$: $S \Rightarrow (T) \Rightarrow (T,S) \Rightarrow (((S,S), \wedge, S), S) \Rightarrow (((a,a), \wedge, (a)), a)$

Question: Consider the grammar G given below:

G: $E \rightarrow E - T \mid T$

$T \rightarrow T / F \mid F$

$F \rightarrow (E) \mid i$

Find all the first and last terminals in this grammar

Nonterminal	First terminal	Last terminal
E	-, /, (, i	/, -,), i
T	/, (, i	/,), i
F	(, i	), i

Question: To increase the compiler's portability and simplify retargeting, a typical real-world compiler usually has multiple phases, which are divided into front-end and back-end. Briefly describe each phase comprised by each division

Answer:

The front end consists of the following phases:

- **scanning:** a scanner groups input characters into tokens
- **parsing:** a parser recognises sequences of tokens according to some grammar and generates Abstract Syntax Trees (ASTs)
- **semantic analysis:** performs type checking and translates ASTs into IRs
- **optimisation:** optimises IRs.

The back end consists of the following phases:

- **instruction selection:** maps IRs into assembly code
- **code optimisation:** optimises the assembly code using control-flow and data-flow analyses, register allocation, etc
- **code emission:** generates machine code from assembly code.

Question: Briefly describe the utilities provided by the operating system to execute the object code generated by the compiler.

- linking:** A linker takes several object files and libraries as input and produces one executable object file.
- loading:** A loader loads an executable object file into memory, initialises the registers, heap, data, etc. and starts the execution of the programme.
- Relocatable shared libraries:** allow effective memory use when many different applications share the same code.

Question: Write short notes on:

Answer:

- Top-down Parsing:** A Top-down parser builds a parse tree by starting at the root and working down towards the leaves. It is easy to generate by hand. Examples are Recursive-descent parser and Predictive parser.
- Bottom-up Parsing:** A bottom-up parser builds a parser tree by starting at the leaves and working up towards the root. It is not easy to implement by hand, it handles a large class of grammar

Question: Differentiate between code optimization and code generation

Answer: The primary objective of the code generator, the final phase of the compiler, is to convert atoms or syntax trees to instructions. code generator takes as input an intermediate representation of the source programme and produces as output an equivalent target programme. **Code Optimisation** within a compiler is concerned with improving in some way the generated object code while ensuring the result is identical.

Question: State the most difficult part of a compiler designer

Answer: Designing a code generator that produces truly efficient object programmes is one of the most difficult parts of a compiler design, both practically and theoretically.

Question: State the characteristics of canonical LR parsing

Answer:

- full set of LR(1) grammars
- largest tables (number of states)
- slow, expensive and large construction

Question: Consider the grammar G given below:

G: $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a$

Generate the operator precedence passing table for this grammar

	+	*	(	)	a	\$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
a	>	>		>		>
\$	<	<	<		<	

Question: In the context of grammar, describe with examples, Terminals and Non-terminal notational Convention

Answer: Terminals are usually denoted by:

- lower-case letters early in the alphabet: a, b, c;
- operator symbols: +, -, *, etc.;
- punctuation symbols: (,), {, }, ; etc.;
- digits: 0, 1, 2, ..., 9;
- boldface strings: if, else, etc.;

Nonterminals are usually denoted by:

- upper-case letters early in the alphabet: A, B, C;

- the letter S representing the start symbol;
- lower-case italic names: *expr*, *stmt*, etc.;

Question: Briefly describe the concept of Transition Diagram

Answer:

A transition diagram is a flowchart for remembering characters for a lexical analyzer. In a TD, the boxes of the flowchart are drawn as circles and called states. The states are connected by arrows, called edges. The labels on the various edges leaving a state indicate the input characters that can appear after that state.

Question: Discuss the concept of LL (K) Grammars

Answer: LL (K) grammars are those for which the left parser can be made to work deterministically, if it is allowed to look at K-input symbols, to the right of its current input position. The left parser expands the leftmost non-terminal or variable.

Question: Describe the operations of the following concepts:

- Re-hashing:** Suppose we hash a symbol s to h and find that a different symbol already occupies the entry h . Then a collision has occurred. We then compare s against an entry $h + p_1$ (modulo the table size) for some integer p_1 . If a collision occurs again, we compare with an entry $h + p_2$ (modulo the table size) for some integer p_2 . This continues until $h = h + p_i$ (modulo table size) is empty, contains s or is again entry h . In other words $p_i = 0$. In the latter case, we stop the programme because the table is full. If $\{P_i\}$ is the set of natural numbers then it is a linear rehash otherwise it is non-linear.
- Chaining:** The hash table consists of buckets, each of which points to a chain (linked list) of symbols. Initially, all buckets are empty (point to nil). Symbols are hashed into buckets using a hash function. If a collision occurs (the bucket is already occupied), the new symbol is added to the chain at that bucket. Symbols are added in a first-come-first-served (FCFS) manner. Each symbol has a pointer field to connect it to the next symbol in the chain.