CS Exit Ethiopia

Questions & Worked Answers

Computer Science Exit Examination

200 complete questions

1 selected course | 18 November 2026

Selected courses

01. Compiler Design

https://csexitethiopia.com

abelasnake2003@gmail.com | Telegram: @Abechu7

Course contents

Each course begins on a new page. PDF bookmarks provide question, study-note section and revision-topic navigation.

Compiler Design 3

Reading the answers

Every question includes its answer, explanation, choice-by-choice contrasts and available evidence. Primary note proof quotes the study notes. Related reading and module passages are labelled separately; a module reference is not presented as primary note proof.

Compiler Design

Questions & Worked Answers

200 questions

Question 1 | Q01406

Which grammar gives multiple parse trees for the same string?

A. Unambiguous

B. Regular

C. Ambiguous

D. All of the above

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: An ambiguous grammar.

COMP10 defines ambiguity by the existence of one terminal string with two different parse trees. This is exactly the property asked for. An unambiguous grammar excludes it. Being regular describes a grammar class rather than guaranteeing multiple trees, so it does not make all the listed descriptions correct.

Why each choice is right or wrong

A - Incorrect: Unambiguous means each generated string has a unique parse tree, directly contradicting the property.

B - Incorrect: Regularity restricts grammar form; it does not require multiple parse trees for a string.

C - Correct: Ambiguity is precisely the existence of multiple parse trees for the same terminal string.

D - Incorrect: Unambiguous grammars do not have the stated property, so all cannot be correct.

Primary note proof

Study-note section COMP10

> A grammar is ambiguous if at least one terminal string has two different parse trees, equivalently two leftmost derivations. Merely changing derivation order on the same tree does not establish ambiguity. E→E+E|E*E|id allows different trees for id+id*id.

Study-note section COMP10

> Use E→E+T|T, T→T*F|F, F→(E)|id to give * higher precedence and both operators left associativity. For subtraction, a−b−c then groups (a−b)−c. The dangling-else grammar permits

else to attach to either nested if; a nearest-unmatched-if rule or grammar rewrite resolves it. Taking a leftmost derivation by itself does not resolve ambiguous interpretation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Compiler Design module | PDF page 9 | lines 22-27

Derivation Trees
• Derivations can be displayed as trees
• Internal nodes of the tree are all non-terminals
• Leaves are all terminals
• The yield of a derivation tree is the list of the labels of all the leaves read from left to right.

Question sources

Compiler Design review questions · Question 14

Question 2 | Q01416

Leaf nodes in a parse tree indicate?

A. sub-terminals

B. half-terminals

C. non-terminals

D. terminals

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: terminals

PDF p. 9 explicitly states that internal nodes are non-terminals and leaves are terminals. Thus the leaf labels in this module's derivation trees are terminals, not the grammar variables expanded at internal nodes.

Why each choice is right or wrong

A - Incorrect: Sub-terminals is not a category in the module's CFG or tree definitions (p. 9).

B - Incorrect: Half-terminals is not a category in those definitions (p. 9).

C - Incorrect: Non-terminals label internal nodes, not leaves (p. 9).

D - Correct: The module expressly assigns terminals to leaves (p. 9).

Primary note proof

Study-note section COMP09

> A context-free grammar G=(N,T,P,S) has nonterminals N, terminals T, productions P and start S. A production A→α rewrites one nonterminal. A sentential form is any string of terminals/nonterminals derivable from S; a sentence contains only terminals. A leftmost derivation rewrites the leftmost remaining nonterminal; a rightmost one rewrites the rightmost.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 22-27

> Derivation Trees
> • Derivations can be displayed as trees
> • Internal nodes of the tree are all non-terminals
> • Leaves are all terminals
> • The yield of a derivation tree is the list of the labels of all the leaves read from left to right.

Question sources

Compiler Design review questions · Question 24

Question 3 | Q00047

Which one of the following is not the task of lexical analyzer?

A. Group characters into lexemes

B. Output a sequence of tokens

C. Read the input characters

D. Build syntax tree

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Build syntax tree

Scanning reads characters and represents matched lexemes as tokens (PDF pp. 5, 8). Syntax analysis then groups those tokens and produces a parse tree. Building that tree is therefore the non-lexical task.

Why each choice is right or wrong

A - Incorrect: Grouping characters into matched lexemes belongs to scanning (pp. 5, 8).

B - Incorrect: A sequence of tokens is the scanner's stated output (p. 8).

C - Incorrect: Reading the stream of source characters is explicitly lexical analysis.

D - Correct: Constructing the syntactic tree belongs to syntax analysis.

Primary note proof

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters
> - Represent lexemes in the form of tokens as:
> <token-name, attribute-value>
> - Token

• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

MOE Computer Science Exit Exam · 2016 · Question 47

Question 4 | Q00054

Which statement is false about a compiler parser that constructs a parse tree and supports syntax-error diagnosis and recovery?

A. Parser builds the parse tree

B. Parser attempts to correct or recover few errors

C. Parser cannot perform context free syntax analysis

D. Parser produces appropriate error messages

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: Parser cannot perform context free syntax analysis

A parser analyzes the token stream against the context-free syntax and builds its structural representation. In a compiler with syntax diagnostics and recovery, it can report an unexpected token and try to resume at a useful boundary. Therefore the claim that it cannot perform context-free syntax analysis reverses its defining role. Building a parse tree, reporting syntax errors and attempting the specified recovery behavior are compatible responsibilities.

Why each choice is right or wrong

A - Incorrect: Building a parse tree represents the grammar structure recognized by syntax analysis.

B - Incorrect: Error recovery may skip to a synchronization token or make a limited local repair.

C - Correct: Context-free syntax analysis is the parser's defining task, so denying it is false.

D - Incorrect: Syntax diagnostics identify unexpected or missing tokens with useful source locations.

Primary note proof

Study-note section COMP09

> A context-free grammar G=(N,T,P,S) has nonterminals N, terminals T, productions P and start S. A production A→α rewrites one nonterminal.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Study-note section COMP39

> Panic-mode recovery skips input until a synchronizing token such as ; or } permits parsing to resume. Phrase-level recovery makes a local insertion/deletion. Report the location and cause while avoiding cascades of misleading diagnostics. Lexical, syntax and semantic phases each detect their own errors; error handling is a shared service.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

> Grammars
> • Every programming language has grammar rules
> • Parsers or syntax analyzers are generated for a particular grammar
> • CFG are used for syntax specification of programming languages
> Context Free Grammar (CFG)
> • Is denoted as G = (N, T , P, S)
> • N : finite set of non-terminals
> • T : finite set of terminals
> • S ∈ N: The start symbol

• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

MOE Computer Science Exit Exam · 2016 · Question 54

Question 5 | Q00058

For an infix source expression targeting a postfix stack evaluator, which notation conversion prepares it for evaluation?

A. Convert infix notation to postfix notation

B. Convert postfix notation to prefix notation

C. Evaluate infix notation to prefix notation

D. Convert postfix to infix notation

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer A: Convert infix notation to postfix notation

A postfix evaluator reads operands before the operator that combines them. Converting an infix expression to postfix therefore supplies the required input order for the specified stack target. For example, the tree for a−4+c yields a 4 − c + in postorder. This is a choice of representation for this evaluator, rather than a mandatory strategy for every compiler.

Why each choice is right or wrong

A - Correct: Infix-to-postfix supplies the required target order.

B - Incorrect: Neither source nor target matches this direction.

C - Incorrect: Prefix is not the specified target.

D - Incorrect: This reverses the specified translation direction.

Primary note proof

Study-note section COMP25

A syntax-directed translation scheme embeds executable actions in productions; the actions run while translating a source program, not while generating the compiler. Attributes may compute types, values, AST pointers or generated code. The scheme defines an input-to-output translation, not merely an input language.

Study-note section COMP25

Create an identifier leaf with mkleaf(id,entry), a number with mkleaf(num,value), and an operator node with mknode(op,left,right). For a−4+c, construct +(−(a,4),c). Postorder traversal prints postfix: a 4 − c +. A stack evaluator pushes operands and pops right then left operands for an operator. Infix-to-postfix is useful for stack machines but not mandatory for all compilers.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 17-25

Syntax Directed Definition- Example
• Production Semantic Rules
• L → E return print(E.val)
• E → E1 + T E.val = E1.val + T.val
• E → T E.val = T.val
• T → T1 * F T.val = T1.val * F.val
• T → F T.val = F.val
• T → (E) F.val = E.val
• F → digit F.val = digit.lexval

Compiler Design module | PDF page 16 | lines 2-8

Three Address Code
• Is a generic form and can be implemented as:
- Quadruples
- Triples
- Indirect Triples
- Tree
- DAG

Question sources

MOE Computer Science Exit Exam · 2016 · Question 58

Question 6 | Q00335

Which named transformation directly replaces an expression whose operands are compile-time constants with its computed value?

A. Pee hole optimization

B. Constant folding

C. Invariant computation

D. Code hoisting

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B: Constant folding

Constant folding evaluates an expression from operands known at compilation and substitutes its value, such as replacing 3*4 with 12. This removes that expression's run-time arithmetic. A peephole pass describes a small inspection region and may perform several transformations; it is not the particular evaluation rule. Loop-invariant computation can remain unknown until execution, and hoisting moves safe invariant work rather than necessarily computing a compile-time value.

Why each choice is right or wrong

A - Incorrect: Peephole identifies a short-window optimization method, not specifically constant-expression evaluation.

B - Correct: Constant folding computes a known constant expression during compilation.

C - Incorrect: An invariant may depend on a run-time input even though it stays unchanged across a loop.

D - Incorrect: Hoisting moves a computation to a preheader; it need not evaluate it at compilation.

Primary note proof

Study-note section COMP30

> Constant folding evaluates constant expressions during compilation: 3*4→12.

Study-note section COMP32

> A loop-invariant expression has unchanged inputs during a loop. Hoist it to a preheader only when evaluation is safe on the paths where it will newly run.

Study-note section COMP31

> Peephole optimization examines short instruction windows.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 23-34

Peephole Optimization
• Transforming to optimal sequence of instructions
Common Techniques:
• Elimination of redundant loads and stores
- Eg.
- r2 = r1 + 5
- I = r2
- r3 = I
- r4 = r3 * 3

• Constant folding
- Eg.
- R2 = 3 * 2

Compiler Design module | PDF page 18 | lines 2-10

Techniques
• Common sub-expression elimination
- Repeated appearance computed previously

• Strength reduction
- Replacement of expensive expressions with simple ones

• Code movement
- Moving a block of code outside a loop

• Dead code elimination
- Eliminated code statements that are either never executed or unreachable

Question sources

Computer Science model exam · Question 66

University of Gondar model exam · Question 66

Question 7 | Q00337

Which listed deterministic parser families can evaluate synthesized attributes for suitable grammars and semantic actions? Select all that apply.

A. Ambiguous grammar

B. LR grammar

C. LL grammar

D. None

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer B, C: LR grammar; LL grammar

Synthesized attributes flow from child nodes to their parent. LR parsers naturally compute these during reduction, after the right-hand-side children have been recognized. Suitable LL implementations can also compute them after their recursive child calls return. Ambiguity is a property of grammar interpretation, not a deterministic parser family or attribute-evaluation method. Thus both LR and LL are valid families for this task.

Why each choice is right or wrong

A - Incorrect: Ambiguity is not a deterministic parser family or evaluation mechanism.

B - Correct: LR reduction actions can compute parent values from children.

C - Correct: Suitable LL actions can compute values after child processing.

D - Incorrect: Both LR and LL provide supported implementations.

Primary note proof

Study-note section COMP24

> A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an inherited or synthesized attribute. Synthesized attributes flow from children to a parent; inherited attributes carry context from parent or siblings.

Study-note section COMP24

> For E→E1+T, E.val=E1.val+T.val synthesizes an expression value. A declaration's type environment passed into its body is inherited. S-attributed definitions use only synthesized attributes and fit LR reduction-time evaluation naturally; suitable LL parsers can also compute them. L-attributed definitions restrict dependencies so a left-to-right traversal can evaluate inherited context. Attribute dependency cycles can prevent evaluation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 4-16

> Syntax Directed Translation
> • Attaching actions to the grammar rules(productions).
> • Actions are executed during the compilation
> ● Not during the generation of the compiler
> • Actions are executed according to the parsing mechanism.
> Syntax Directed Definitions
> • Is a generalization of a context free grammar
> • Is a CFG with attributes and rules
> • Attributes are associated with grammar symbols and rules with productions
> • Attributes may be:
> ● Numbers
> ● Types
> ● Strings etc

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science model exam · Question 68

University of Gondar model exam · Question 68

Question 8 | Q00338

Identifiers consist of one Letter followed by zero or more Letters or Digits. Which expressions denote exactly that language? Select all that apply. Here + means one-or-more, | alternation, and _ a literal underscore.

A. Letter(Letter+|Digit*)*

B. Letter(Letter|Digit)+

C. Letter(Letter|Digit)*

D. Letter(_Digit)*

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer A, C: Letter(Letter+|Digit*)*; Letter(Letter|Digit)*

C directly expresses a leading letter followed by any sequence of letters and digits. A is equivalent: its outer star may take zero blocks, and repeated letter or digit blocks can generate every such suffix. B excludes a one-letter identifier because its suffix uses plus. D permits literal underscores and restricted digit blocks, so it does not denote the specified identifier language.

Why each choice is right or wrong

A - Correct: Its repeated blocks generate every allowed suffix, including the empty suffix.

B - Incorrect: The trailing plus excludes a one-letter identifier.

C - Correct: This directly states the required pattern.

D - Incorrect: Its literal underscores violate the specified alphabet and its restricted suffix misses identifiers.

Primary note proof

Study-note section COMP05

> Regular expressions use concatenation, alternation r|s, Kleene star r* for zero or more, and r+ for one or more. In a simplified alphabet, identifier = Letter(Letter|Digit)*; it permits a single letter and forbids a leading digit.

Study-note section COMP05

> Letter(Letter+|Digit*)* denotes the same language as Letter(Letter|Digit)* under ordinary regular-expression semantics: repeated blocks can contain any trailing letter/digit sequence. These two regular expressions are equivalent. Regular patterns handle ordinary tokens; arbitrary balanced nesting generally needs grammar/stack structure.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 10-18

> Tokens, Patterns, and Lexemes
> • Token
> ● A string of characters which logically belong together
> ● Classes of similar lexemes
> l identifier, keywords, constants etc.
> • Pattern
> ● A rule which describes a token
> • Lexeme
> ● The sequence of characters matched by a pattern to form the token

Compiler Design module | PDF page 8 | lines 19-24

> • Classes of Tokens
> ● Identifiers: names chosen by the programmer
> ● Keywords: names already in the programming language
> ● Separators: punctuation characters
> ● Operators: symbols that operate on arguments and produce results
> ● Literals: numeric, textual literals

Question sources

Computer Science model exam · Question 69

University of Gondar model exam · Question 69

Question 9 | Q00339

What is the process of finding a parse tree for a string of tokens called?

A. Recognizing

B. Parsing

C. Tokenizing

D. Analyzing

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Parsing

Parsing is explicitly the process of constructing a parse tree for a sentence generated by a grammar (PDF p. 10). That is the operation named in the question; token production precedes it (p. 8).

Why each choice is right or wrong

A - Incorrect: The module does not use Recognizing as the name of this construction (p. 10).

B - Correct: Parsing matches the express definition (p. 10).

C - Incorrect: Tokenizing produces the token stream, rather than its parse tree (p. 8).

D - Incorrect: Analyzing is a broad description; the defined specific process is Parsing (p. 10).

Primary note proof

Study-note section COMP09

> A context-free grammar G=(N,T,P,S) has nonterminals N, terminals T, productions P and start S. A production A→α rewrites one nonterminal. A sentential form is any string of terminals/nonterminals derivable from S; a sentence contains only terminals. A leftmost derivation rewrites the leftmost remaining nonterminal; a rightmost one rewrites the rightmost.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

> Parsing
> • Process of constructing parse tree for a sentence generated by a given grammar.

- 2 types of parsers
  - Top down parsing (predictive parsers)
- LL(1)
  - Bottom up parsing (SR parsers)
- LR(1)

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
- The first phase of a compiler
- The input is a high level language program
- The output is a sequence of tokens
- Strips off blanks, tabs, newlines, and comments from the source program
- Keeps track of line numbers

Question sources

Computer Science model exam · Question 70

University of Gondar model exam · Question 70

Question 10 | Q00340

Which compiler phase groups source characters into lexemes and classifies them as identifier, keyword, literal or operator tokens?

A. Interpretation analysis

B. Lexical analysis

C. Syntax analysis

D. General syntax analysis

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Lexical analysis

Lexical analysis groups the source character sequence into lexemes and emits token categories with attributes. Syntax analysis instead combines already recognized tokens into larger grammatical structures such as expressions and statements. Specifying characters and token categories resolves the original mixture of lexical classification and parsing terminology. The generic interpretation and syntax labels do not describe the requested character-to-token phase.

Why each choice is right or wrong

A - Incorrect: Interpretation executes a representation rather than classifying lexemes.

B - Correct: Lexical analysis turns character sequences into tokens.

C - Incorrect: Syntax analysis groups tokens into grammatical structures.

D - Incorrect: This syntax label concerns parsing rather than lexical classification.

Primary note proof

Study-note section COMP41

> Grouping characters into token classes is lexical analysis; grouping tokens into expressions and statements is syntax analysis.

Study-note section COMP41

> Trees store hierarchical constructs, while a symbol table stores declaration metadata.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters
> - Represent lexemes in the form of tokens as:
> <token-name, attribute-value>
> - Token
> • smallest meaningful element that a compiler understands.
> - Eg.
> • Identifiers, Keywords, Literals, Operators and Special symbols.
> - Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 5 | lines 12-18

> 2. Syntax / Hierarchical Analysis - Parsing
> • Tokens are grouped hierarchically into nested collections with collective meaning.
> • The result is generally a parse tree.
> • expressions, statements, declarations etc... are identified by using the results of lexical analysis.
> • Most syntactic errors in the source program are caught in this phase.
> • Syntactic rules of the source language are given via a Grammar.

Question sources

Computer Science model exam · Question 71

University of Gondar model exam · Question 71

Question 11 | Q00532

For a register-based target compiler, which choice most completely lists general code-generator tasks?

A. Instruction selection

B. Register allocation and assignment

C. Instruction ordering

D. All

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: All

COMP34 separates selecting target operations, deciding storage in registers, assigning physical registers, and scheduling instructions. The three concrete choices name these standard code-generation tasks. Since the question offers an aggregate covering them, that aggregate is the complete answer; each individual choice supplies only part of the requested task list.

Why each choice is right or wrong

A - Incorrect: True as one component, but incomplete as the requested full selection. Instruction selection chooses which target instructions implement the intermediate operations.

B - Incorrect: True as one component, but incomplete as the requested full selection. Register allocation and physical assignment decide where values reside and which registers hold them.

C - Incorrect: True as one component, but incomplete as the requested full selection. Instruction ordering schedules computations while preserving data and control dependencies.

D - Correct: All three listed activities belong to the code-generator task set described in COMP34.

Primary note proof

Study-note section COMP34

> Code generation lowers checked IR to a target form. Instruction selection chooses target operations; register allocation chooses where values reside; scheduling orders instructions while respecting dependencies; storage layout assigns offsets and follows ABI requirements. Register assignment names actual physical registers after allocation decisions.

Study-note section COMP34

> For t=a+b, a simple target could emit LOAD R1,a; LOAD R2,b; ADD R3,R1,R2; STORE t,R3. Other targets may combine loads with arithmetic or keep t in a register. Machine idioms, register availability and instruction order affect execution time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Compiler Design module | PDF page 18 | lines 11-22

Register Allocation
• Registers hold values
• Example
- a = c + d
- e = a + b
- f = e - 1

• With the assumption that a and e die after use
• Temporary a can be reused after e=a+b, same wz a
• Can allocate a,e and f all to one register(r1)
- r1 = r2 + r3
- r1 = r1 + r4
- r1 = r1 - 1

Question sources

Computer Science model exam · 2015 · Question 1

Question 12 | Q00537

What is the term for an actual occurrence of a character sequence in source text that matches a token pattern, such as total or 42?

A. Pattern

B. Lexemes

C. Tokens

D. All

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Lexemes

A lexeme is the actual character sequence occurring in source text. A token is the category and optional attribute emitted for that occurrence, while a pattern defines which spellings are members of the category. For example, total is a lexeme classified as an identifier token. These are related but distinct objects, so the aggregate choice cannot name the concrete occurrence.

Why each choice is right or wrong

A - Incorrect: A pattern describes possible spellings.

B - Correct: Lexeme names the actual matched source sequence.

C - Incorrect: Token names the category and optional attribute.

D - Incorrect: The three concepts are distinct, not synonymous.

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP41

Grouping characters into token classes is lexical analysis; grouping tokens into expressions and statements is syntax analysis.

Study-note section COMP41

Trees store hierarchical constructs, while a symbol table stores declaration metadata.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
- A string of characters which logically belong together
- Classes of similar lexemes
l identifier, keywords, constants etc.
• Pattern

- A rule which describes a token
- Lexeme
- The sequence of characters matched by a pattern to form the token

Question sources

Computer Science model exam · 2015 · Question 6

Question 13 | Q00561

Which one of the following are the specific character strings that make up a token?

A. Token

B. Lexemes

C. Lexical units

D. Pattern

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Lexemes

A lexeme is the sequence of characters matched by a pattern to form a token (PDF p. 8). A token instead denotes a class of similar lexemes, and a pattern specifies the rule. “Specific character strings” therefore identifies lexemes.

Why each choice is right or wrong

A - Incorrect: Token names the class in the module’s token/lexeme distinction (p. 8).

B - Correct: Lexemes are the actual matched sequences (p. 8).

C - Incorrect: Lexical units is not the defined name for specific strings in this passage (p. 8).

D - Incorrect: A pattern is the matching rule, not its particular match (p. 8).

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive

or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
- A string of characters which logically belong together
- Classes of similar lexemes

l identifier, keywords, constants etc.
• Pattern
- A rule which describes a token

• Lexeme
- The sequence of characters matched by a pattern to form the token

Question sources

Computer Science model exam · 2015 · Question 30

Question 14 | Q00578

Which one of the following are the program properties that can be checked at run time?

A. Static checking

B. Dynamic semantics

C. Other Static Checks

D. None

Hint

Ask whether the required information is a declaration’s identity and attributes, program structure or the ordered source tokens.

Answer B: Dynamic semantics

Semantic checks are static when done during compilation and dynamic when done during run-time (PDF p. 14). Interpreting “Dynamic semantics” as the option’s name for run-time semantic checking, B fits; static checking and other static checks occur during compilation.

Why each choice is right or wrong

A - Incorrect: Static checking is done at compilation, not run-time (p. 14).

B - Correct: B corresponds to the module’s run-time dynamic semantic checks (p. 14).

C - Incorrect: Other static checks retain the compile-time timing (p. 14).

D - Incorrect: None does not fit because the dynamic check category is present (p. 14).

Primary note proof

Study-note section COMP21

Static checks occur before execution: type compatibility, duplicate declarations within a restricted scope, valid names, argument counts, return compatibility and control-flow placement. A break is valid only in constructs allowed by the language; in C/C++ it can exit a loop or switch. A compiler may annotate an AST or create typed IR; there is no mandatory uniquely named output tree.

Study-note section COMP21

Dynamic checks depend on runtime values, such as some bounds and null checks. Static analysis can prove some accesses safe or faulty; “all memory errors cannot be statically checked” is too strong. Unreachable code is not universally a semantic error: some languages reject it in specified cases, others merely warn or optimize it away.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 2-15

• Type Errors:
- Improper or inconsistent operations during program execution
• Type-safety:
- Absence of type errors

Type Checking
• Semantic checks to enforce the type safety of the program
• Semantic Checks
- Static - done during compilation
- Dynamic - done during run-time
• Examples
- Unary and binary operators
- Number and type of arguments
- Return statement with return type
- Compatible assignment

Compiler Design module | PDF page 14 | lines 16-23

Static Checking
• The compiler must check the semantic conventions of the source language
• Static Checking: ensures that certain kind of errors are detected and reported
• Example
• Type Checks: incompatible operands
• Flow Control Check
• Uniqueness Check
• Name Related Check

Question sources

Computer Science model exam · 2015 · Question 47

Question 15 | Q00610

For a language with insignificant whitespace and comments, which choice most completely lists the scanner’s secondary tasks?

A. Stripping out blanks, tabs, new lines

B. Stripping out comments

C. Error reporting

D. All

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: All

COMP04 says scanners usually skip whitespace and comments while retaining source positions. COMP39 adds that lexical analysis detects its own errors and that error handling is shared. These establish all three listed secondary scanner duties under the ordinary language convention. Layout-sensitive languages are an explicit exception in COMP04, not evidence that scanners never perform these tasks.

Why each choice is right or wrong

A - Incorrect: True as one component, but incomplete as the requested full selection. Ordinary scanners skip blanks, tabs and newlines while retaining positions; layout-sensitive languages need special handling.

B - Incorrect: True as one component, but incomplete as the requested full selection. Comments are normally discarded instead of being delivered as ordinary parser tokens.

C - Incorrect: True as one component, but incomplete as the requested full selection. Lexical errors require reporting, with positions retained to locate the offending source text.

D - Correct: Together the choices cover normal whitespace removal, comment removal and lexical error reporting.

Primary note proof

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP39

> Report the location and cause while avoiding cascades of misleading diagnostics. Lexical, syntax and semantic phases each detect their own errors; error handling is a shared service.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

Computer Science model exam · 2015 · Question 79

Question 16 | Q00613

Which one of the following is the parse tree showing the values of attributes at each node?

A. Inherited attribute

B. Synthesized attribute

C. Annotated parse tree

D. A&B

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer C: An annotated parse tree.

COMP24 directly calls a parse tree displaying computed attribute values an annotated parse tree. Inherited and synthesized classify how attribute information flows; neither is the name of the entire displayed tree. Combining those attribute categories still does not name the representation described in the question.

Why each choice is right or wrong

A - Incorrect: An inherited attribute carries contextual information; it is not a parse tree displaying values.

B - Incorrect: A synthesized attribute carries a value upward, rather than naming the annotated tree.

C - Correct: This is the note's precise name for a parse tree showing attribute values at nodes.

D - Incorrect: Combining two attribute categories does not give the name of the displayed tree.

Primary note proof

Study-note section COMP24

> A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an inherited or synthesized attribute. Synthesized attributes flow from children to a parent; inherited attributes carry context from parent or siblings.

Study-note section COMP24

> For E→E1+T, E.val=E1.val+T.val synthesizes an expression value. A declaration's type environment passed into its body is inherited. S-attributed definitions use only synthesized attributes and fit LR reduction-time evaluation naturally; suitable LL parsers can also compute them. L-attributed definitions restrict dependencies so a left-to-right traversal can evaluate inherited context. Attribute dependency cycles can prevent evaluation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 4-16

> Syntax Directed Translation
> • Attaching actions to the grammar rules(productions).
> • Actions are executed during the compilation
> ● Not during the generation of the compiler
> • Actions are executed according to the parsing mechanism.
> Syntax Directed Definitions
> • Is a generalization of a context free grammar
> • Is a CFG with attributes and rules
> • Attributes are associated with grammar symbols and rules with productions
> • Attributes may be:
> ● Numbers
> ● Types
> ● Strings etc

Compiler Design module | PDF page 12 | lines 17-25

> Syntax Directed Definition- Example
> • Production Semantic Rules
> • L → E return print(E.val)
> • E → E1 + T E.val = E1.val + T.val
> • E → T E.val = T.val
> • T → T1 * F T.val = T1.val * F.val
> • T → F T.val = F.val

- T → (E) F.val = E.val
- F → digit F.val = digit.lexval

Question sources

Computer Science model exam · 2015 · Question 82

Question 17 | Q00631

Which one of the following can be applied directly after intermediate code generation to improve the intermediate representation?

A. Peephole optimization

B. Constant Propagation

C. Dead code elimination

D. All

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer D: All

Optimization changes intermediate code to remove inefficiencies (PDF p. 7). The module lists peephole transformations and constant propagation (p. 18), and elimination of never-executed or unreachable statements (p. 18). Each can improve an intermediate representation before target code is emitted; no exclusive ordering among those transformations is specified.

Why each choice is right or wrong

A - Incorrect: Peephole transformation improves instruction sequences (p. 18).

B - Incorrect: Constant propagation is a listed optimization technique (p. 18).

C - Incorrect: Removing unreachable intermediate statements is dead code elimination (p. 18).

D - Correct: All three fit the optimization of intermediate code described on p. 7.

Primary note proof

Study-note section COMP30

Constant folding evaluates constant expressions during compilation: 3*4→12. Constant propagation substitutes known values: x=3; y=x+2 becomes y=3+2 before folding. Copy propagation substitutes a copied name while its source remains unchanged. Dead-code elimination removes unreachable code or computations whose values are unused and whose evaluation has no necessary side effects.

Study-note section COMP31

Common-subexpression elimination reuses an earlier value only if operand values and relevant memory remain unchanged. A DAG can represent sharing. Algebraic simplification removes identities when valid: integer x+0→x. Strength reduction replaces an operation with an equivalent cheaper form, such as unsigned x*8→x<<3 for a valid width and shift count. Target costs determine whether it is actually cheaper.

Study-note section COMP31

Peephole optimization examines short instruction windows. It can remove redundant loads/stores, replace machine idioms and shorten jump chains: jump L1 where L1 immediately jumps L2 can become jump L2. Local optimization is a wider category; peephole optimization is a concrete method within the relevant scope.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 2-6

5. Code Optimization
Changes the IC by removing such inefficiencies
Improve the code
a. Improvement may be time, space, or power consumption.
It changes the structure of programs,

Compiler Design module | PDF page 18 | lines 23-34

Peephole Optimization
• Transforming to optimal sequence of instructions
Common Techniques:
• Elimination of redundant loads and stores
- Eg.
- r2 = r1 + 5
- I = r2
- r3 = I
- r4 = r3 * 3

• Constant folding
- Eg.
- R2 = 3 * 2

Compiler Design module | PDF page 18 | lines 32-41

• Constant folding
- Eg.
- R2 = 3 * 2

• Constant Propagation
- Eg.
- r1 = 3
- r2 = r1 * 2

• Copy Propagation
- Eg.
- r2 = r1

Compiler Design module | PDF page 18 | lines 2-10

Techniques
• Common sub-expression elimination
  • Repeated appearance computed previously
• Strength reduction
  • Replacement of expensive expressions with simple ones
• Code movement
  • Moving a block of code outside a loop
• Dead code elimination
  • Eliminated code statements that are either never executed or unreachable

Question sources

Computer Science model exam · 2015 · Question 100

Question 18 | Q00652

Which compiler structure associates declared identifier names or identities with attributes such as type, scope and storage location, irrespective of its container implementation?

A. Literal Table

B. Symbol table

C. Hash table

D. All

Hint

Ask whether the required information is a declaration’s identity and attributes, program structure or the ordered source tokens.

Answer B: Symbol table

A symbol table maps identifier names or declaration identities to attributes such as type, scope and storage location. A hash table is one possible implementation of that mapping, not its semantic role. Literal tables concern literal values, and punctuation need not have symbol records at all. Restricting the question to identifier metadata avoids claiming that every token kind uses the same table.

Why each choice is right or wrong

A - Incorrect: A literal table concerns literal values.

B - Correct: The symbol table provides this mapping.

C - Incorrect: Hashing is an implementation technique rather than the semantic role.

D - Incorrect: The structures have different roles.

Primary note proof

Study-note section COMP19

A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage. A hash table gives expected O(1) insertion/lookup with suitable hashing; a stack of scope tables supports lexical nesting. Search the nearest scope first, insert declarations in the current scope, and remove visibility when leaving it.

Study-note section COMP19

Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal. Not every punctuation token needs a symbol-table entry; literals may use a separate literal table. A hash table is an implementation, not the information's semantic role.

Study-note section COMP41

Grouping characters into token classes is lexical analysis; grouping tokens into expressions and statements is syntax analysis.

Study-note section COMP41

Trees store hierarchical constructs, while a symbol table stores declaration metadata.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float

id.width=4
offset = offset +id.width { offset = 6 }

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
● A string of characters which logically belong together
● Classes of similar lexemes
l identifier, keywords, constants etc.
• Pattern
● A rule which describes a token
• Lexeme
● The sequence of characters matched by a pattern to form the token

Question sources

Computer Science model exam · 2015 · Question 121

Question 19 | Q00754

The output of lexical analyzer is*

A. Syntax tree

B. Strings of characters

C. A set of tokens

D. set of regular expressions

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: A set of tokens

The output of lexical analysis is expressly a sequence of tokens (PDF p. 8). C is the only option naming that output, although "set" should be read informally: order matters, so the module's precise word is sequence.

Why each choice is right or wrong

A - Incorrect: The parse tree is produced by syntax analysis.

B - Incorrect: Characters are the scanner input, not its classified output.

C - Correct: Tokens are the output elements (p. 8).

D - Incorrect: Patterns describe tokens; they are not the source-specific output sequence (p. 8).

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

Computer Science model exam 2 · Question 95

Question 20 | Q00755

Which of the following is NOT the primary role of code generator in compiler design? *

A. instruction selection

B. Declaration of variables

C. instruction ordering

D. A Register allocation

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer B: Declaration of variables

Code generation lowers checked intermediate code into target operations. It selects instructions, allocates registers and orders instructions subject to dependencies. Declarations instead establish source names, types and storage information during analysis; generating code may use that information without declaring the variables again. Thus the declaration activity is outside the primary code-generator task set, while the other three choices describe target implementation decisions.

Why each choice is right or wrong

A - Incorrect: Instruction selection chooses target operations implementing the IR.

B - Correct: Processing source declarations belongs to analysis and symbol-table/type handling.

C - Incorrect: Instruction ordering schedules target computations while preserving dependencies.

D - Incorrect: Register allocation decides where intermediate values reside.

Primary note proof

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Study-note section COMP34

> Code generation lowers checked IR to a target form. Instruction selection chooses target operations; register allocation chooses where values reside; scheduling orders instructions while respecting dependencies; storage layout assigns offsets and follows ABI requirements. Register assignment names actual physical registers after allocation decisions.

Study-note section COMP34

> For t=a+b, a simple target could emit LOAD R1,a; LOAD R2,b; ADD R3,R1,R2; STORE t,R3. Other targets may combine loads with arithmetic or keep t in a register. Machine idioms, register availability and instruction order affect execution time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Question sources

Computer Science model exam 2 · Question 96

Question 21 | Q00756

Which one of the following is different from the others?

A. quadruples

B. triples

C. tree

D. indirect triples

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer C: tree

Quadruples, triples, indirect triples, trees and DAGs all implement three-address code (PDF p. 16). Quadruples and triples have numbered record fields, and indirect triples use pointers; tree is the hierarchical representation. C is the natural odd one out under that structural distinction.

Why each choice is right or wrong

A - Incorrect: Quadruples are instruction records with four fields (p. 16).

B - Incorrect: Triples are instruction records with three fields (p. 16).

C - Correct: Tree is the hierarchical structure among the options (p. 16).

D - Incorrect: Indirect triples retain a pointer-based triple representation (p. 16).

Primary note proof

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 16 | lines 2-8

> Three Address Code
> • Is a generic form and can be implemented as:
> - Quadruples
> - Triples
> - Indirect Triples
> - Tree
> - DAG

Compiler Design module | PDF page 16 | lines 13-23

> Three Address Code
> • Quadruples:
> - Each instruction is divided into four fields
> - Operator, arg1, arg2, and result
> • Triples:
> - Has three fields
> - Operator, arg1 and arg2
> • DAG and Tree
> - Similar presentation of expression to triples
> • Indirect Triples
> - Uses pointers instead of position to store results

Question sources

Computer Science model exam 2 · Question 97

Question 22 | Q00757

A compiler computes expression-type attributes and checks compatibility as grammar productions are processed. Which offered description most specifically identifies this mechanism?

A. During lexical analysis

B. During syntax analysis

C. During syntax directed translation

D. During code optimization

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: During syntax directed translation

The compiler in the scenario evaluates type attributes and applies compatibility checks as grammar productions are processed. This is syntax-directed translation: semantic rules or actions are attached to grammatical structure. Pure lexical analysis only recognizes tokens, and pure syntax recognition alone does not establish operand-type compatibility. Optimization transforms an already analyzed representation. Parsing can invoke these semantic actions, so the mechanism is more precise than naming syntax analysis alone.

Why each choice is right or wrong

A - Incorrect: Token recognition alone does not compute full expression-type compatibility.

B - Incorrect: Pure syntax recognition checks grammar; the scenario additionally attaches semantic computations.

C - Correct: Syntax-directed semantic actions compute and check attributes associated with productions.

D - Incorrect: Optimization transforms analyzed code rather than naming the stated checking mechanism.

Primary note proof

Study-note section COMP24

> A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions.

Study-note section COMP25

> A syntax-directed translation scheme embeds executable actions in productions; the actions run while translating a source program, not while generating the compiler. Attributes may compute types, values, AST pointers or generated code.

Study-note section COMP22

A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 12 | lines 4-16

Syntax Directed Translation
• Attaching actions to the grammar rules(productions).
• Actions are executed during the compilation
- Not during the generation of the compiler
• Actions are executed according to the parsing mechanism.
Syntax Directed Definitions
• Is a generalization of a context free grammar
• Is a CFG with attributes and rules
• Attributes are associated with grammar symbols and rules with productions
• Attributes may be:
- Numbers
- Types
- Strings etc

Question sources

Computer Science model exam 2 · Question 98

Question 23 | Q00758

In a conventional pipeline that lowers TAC further to native code or VM instructions, which representation is intermediate rather than the final emitted target?

A. Three address code

B. Assembly language

C. Absolute machine code

D. Byte Code

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer A: Three address code

Three-address code is intermediate in the specified pipeline: analysis and optimization transform it before code generation lowers it further. Assembly, absolute machine code and VM bytecode can each be final target forms for an appropriate compiler. This pipeline-specific distinction does not assert that TAC could never be another translator's target; it identifies the expressly stated intermediate role here.

Why each choice is right or wrong

A - Correct: TAC is lowered further in this pipeline.

B - Incorrect: Assembly can be a final emitted target.

C - Incorrect: Absolute machine code can be a final target.

D - Incorrect: VM bytecode can be a final target.

Primary note proof

Study-note section COMP34

> Code generation lowers checked IR to a target form. Instruction selection chooses target operations; register allocation chooses where values reside; scheduling orders instructions while respecting dependencies; storage layout assigns offsets and follows ABI requirements. Register assignment names actual physical registers after allocation decisions.

Study-note section COMP34

> For t=a+b, a simple target could emit LOAD R1,a; LOAD R2,b; ADD R3,R1,R2; STORE t,R3. Other targets may combine loads with arithmetic or keep t in a register. Machine idioms, register availability and instruction order affect execution time.

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 15 | lines 13-21

Three Address Code
• Is a sequence of statements of the form
- X = Y op Z
- X,Y and Z are names, constants or compiler generated temporaries
- Op is operator (arithmetic, logical)

• Example:
- a = b + c , x = -y , if a > b goto L1

• LHS is the target
• RHS has at most two sources and one operator

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Question sources

Computer Science model exam 2 · Question 99

Question 24 | Q00759

Which of the following is NOT important property of intermediate code representation?

A. Ease of generation

B. Ease of manipulation

C. Register allocation

D. Level of abstraction

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer C: Register allocation

An intermediate representation should be convenient to generate, analyze and transform, and its abstraction should suit optimization and target lowering. Those are design properties of the representation. Register allocation is an algorithmic backend task that chooses locations for values; it is not itself a property such as ease of generation or abstraction level. An IR can carry information useful to that task without making allocation a representation property.

Why each choice is right or wrong

A - Incorrect: Ease of generation is a useful IR design property.

B - Incorrect: Ease of manipulation supports analysis and transformation passes.

C - Correct: Register allocation is a backend decision over values rather than an IR property.

D - Incorrect: Abstraction level determines which source and target details the representation exposes.

Primary note proof

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Study-note section COMP34

> Code generation lowers checked IR to a target form. Instruction selection chooses target operations; register allocation chooses where values reside; scheduling orders instructions while respecting dependencies; storage layout assigns offsets and follows ABI requirements. Register assignment names actual physical registers after allocation decisions.

Study-note section COMP34

> For t=a+b, a simple target could emit LOAD R1,a; LOAD R2,b; ADD R3,R1,R2; STORE t,R3. Other targets may combine loads with arithmetic or keep t in a register. Machine idioms, register availability and instruction order affect execution time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 11-12

> 4. Intermediate Code Generation
> Easy to produce and easy to translate to machine code

Compiler Design module | PDF page 7 | lines 7-11

> 6. Code Generation
> Converts intermediate code to machine code.
> Must handle all aspects of machine architecture
> Storage allocation decisions are made
> a. Register allocation and assignment

Compiler Design module | PDF page 16 | lines 2-8

> Three Address Code
> • Is a generic form and can be implemented as:
> • Quadruples

- Triples
- Indirect Triples
- Tree
- DAG

Question sources

Computer Science model exam 2 · Question 100

Question 25 | Q00903

What is the main purpose of syntax-directed translation in a compiler?

A. To interleave semantic analysis with syntax analysis using attributes attached to grammar symbols

B. To generate optimized machine code directly

C. .
To tokenize the source code

D. To replace the parser with a finite automaton

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer A: To interleave semantic analysis with syntax analysis using attributes attached to grammar symbols

A syntax-directed definition is a CFG with attributes and rules; attributes attach to symbols and rules to productions (PDF p. 12). Actions execute according to the parsing mechanism. This is the grammar-based integration of semantic processing described in A, rather than a direct optimized-machine-code replacement for the remaining phases.

Why each choice is right or wrong

A - Correct: A matches the attribute/rule and parsing-action definitions (p. 12).

B - Incorrect: Machine-code conversion is a later code-generation task (p. 7), not the definition of SDT.

C - Incorrect: Tokenization is lexical analysis (p. 8).

D - Incorrect: The module retains the grammar/parsing mechanism; it does not replace it with an automaton (p. 12).

Primary note proof

Study-note section COMP24

A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an

inherited or synthesized attribute. Synthesized attributes flow from children to a parent; inherited attributes carry context from parent or siblings.

Study-note section COMP24

For E→E1+T, E.val=E1.val+T.val synthesizes an expression value. A declaration's type environment passed into its body is inherited. S-attributed definitions use only synthesized attributes and fit LR reduction-time evaluation naturally; suitable LL parsers can also compute them. L-attributed definitions restrict dependencies so a left-to-right traversal can evaluate inherited context. Attribute dependency cycles can prevent evaluation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 4-16

Syntax Directed Translation
• Attaching actions to the grammar rules(productions).
• Actions are executed during the compilation
- Not during the generation of the compiler
• Actions are executed according to the parsing mechanism.
Syntax Directed Definitions
• Is a generalization of a context free grammar
• Is a CFG with attributes and rules
• Attributes are associated with grammar symbols and rules with productions
• Attributes may be:
- Numbers
- Types
- Strings etc

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science Exit Exam · 2018 · Question 16

Question 26 | Q00908

Straight directed translation uses:

A. Purely lexical rules

B. Backtracking algorithms

C. ¢.
Grammar without attributes

D. Grammar with sematic rule

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer D: Grammar with sematic rule

Interpreting the apparent typo “Straight directed translation” as syntax-directed translation, the module defines it through actions on productions and a CFG with attributes and semantic rules (PDF p. 12). D is the only option that retains this grammar-plus-rule structure.

Why each choice is right or wrong

A - Incorrect: Pure lexical rules omit the production-associated semantic actions (p. 12).

B - Incorrect: Backtracking is not the defining mechanism; actions follow the chosen parser (p. 12).

C - Incorrect: A grammar without attributes omits the defining additions in a syntax-directed definition (p. 12).

D - Correct: A grammar with semantic rules matches the definition (p. 12).

Primary note proof

Study-note section COMP24

> A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an inherited or synthesized attribute. Synthesized attributes flow from children to a parent; inherited attributes carry context from parent or siblings.

Study-note section COMP24

> For E→E1+T, E.val=E1.val+T.val synthesizes an expression value. A declaration’s type environment passed into its body is inherited. S-attributed definitions use only synthesized attributes and fit LR reduction-time evaluation naturally; suitable LL parsers can also compute them. L-attributed definitions restrict dependencies so a left-to-right traversal can evaluate inherited context. Attribute dependency cycles can prevent evaluation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 4-16

Syntax Directed Translation
• Attaching actions to the grammar rules(productions).
• Actions are executed during the compilation
- Not during the generation of the compiler
• Actions are executed according to the parsing mechanism.
Syntax Directed Definitions
• Is a generalization of a context free grammar
• Is a CFG with attributes and rules
• Attributes are associated with grammar symbols and rules with productions
• Attributes may be:
- Numbers
- Types
- Strings etc

Question sources

Computer Science Exit Exam · 2018 · Question 21

Question 27 | Q00911

Which one of the following is a top-down parser without backtracking?

A. LR parser

B. Brute force parser

C. Operator precedence parser

D. Predictive parser

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer D: A predictive parser.

COMP11 defines predictive parsing by choosing productions with lookahead without backtracking. This supplies both requirements in the stem: top-down direction and absence of backtracking. LR and operator-precedence parsing belong to bottom-up approaches; brute-force top-down trial does not provide the no-backtracking guarantee.

Why each choice is right or wrong

A - Incorrect: LR parsing reduces input bottom-up rather than expanding the start symbol top-down.

B - Incorrect: Brute-force production trials can require backtracking and do not meet the defining predictive constraint.

C - Incorrect: Operator-precedence parsing is a bottom-up technique rather than predictive top-down expansion.

D - Correct: Predictive parsing uses lookahead to choose productions without backtracking, exactly as COMP11 states.

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse.

Study-note section COMP-S01

> Operator-precedence parsing is a restricted bottom-up expression-parsing method using a stack and precedence relations. With a suitable expression grammar, the stack retains unfinished surrounding expressions while nested parentheses are processed.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

> Parsing
> • Process of constructing parse tree for a sentence generated by a given grammar.
> • 2 types of parsers
> ● Top down parsing (predictive parsers)
> • LL(1)
> ● Bottom up parsing (SR parsers)
> • LR(1)

Compiler Design module | PDF page 10 | lines 18-23

> Top Down Parsing
> • The parse tree is created top to bottom
> • Starts from the start symbol and transform it to the input
> Bottom Up Parsing
> • Starts with the input symbols and tries to construct the parse tree up to the start symbol.
> • One way of reducing a sentence is to follow the right most derivation in reverse

Compiler Design module | PDF page 11 | lines 2-17

> LL(1) Grammar
> • L - left to right
> • L - left most derivation
> • 1 - number of look ahead
> • First( ) and Follow( )

- the first terminal in a string and the terminal that follows a variable respectively.

LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)

LL	LR
Leftmost derivation	Rightmost derivation in reverse
Starts with root non-terminal on stack	Ends with root non-terminal on the stack
Builds the parse tree top-down	Builds the parse tree bottom-up
Expands the non-terminals	Reduces the non-terminals

Question sources

Computer Science Exit Exam · 2018 · Question 24

Question 28 | Q00940

If input is "a" is for syntax tree, which function is used to create tree?

A. mknode(num, value)

B. mkleaf(num, value)

C. mkleaf(id, entry)

D. mknode(op, left, right)

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer C: mkleaf(id, entry)

For the expression a-4+c the first construction is P1=mkleaf(id,entrya) (PDF p. 13). Since a is an identifier, its node stores a pointer to the identifier’s symbol-table entry. Numeric leaf construction is instead used for 4; operator nodes connect child nodes.

Why each choice is right or wrong

A - Incorrect: The module uses mkleaf, not mknode, for a numeric constant, and a is not a number (p. 13).

B - Incorrect: This creates a numeric leaf, suitable for 4 rather than identifier a (p. 13).

C - Correct: This is exactly the identifier-leaf constructor used for a (p. 13).

D - Incorrect: This makes an operator node with two children, not an identifier leaf (pp. 12-13).

Primary note proof

Study-note section COMP25

A syntax-directed translation scheme embeds executable actions in productions; the actions run while translating a source program, not while generating the compiler. Attributes may compute types, values, AST pointers or generated code. The scheme defines an input-to-output translation, not merely an input language.

Study-note section COMP25

Create an identifier leaf with mkleaf(id,entry), a number with mkleaf(num,value), and an operator node with mknode(op,left,right). For a−4+c, construct +(−(a,4),c). Postorder traversal prints postfix: a 4 − c +. A stack evaluator pushes operands and pops right then left operands for an operator. Infix-to-postfix is useful for stack machines but not mandatory for all compilers.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 26-28

Functions for Syntax Tree Nodes
• mknode (op, left, right)
- Creates an operator node with label op &

Compiler Design module | PDF page 13 | lines 2-14

- Two fields containing pointers to left and right
• mkleaf(id, entry)
- Creates an identifier node with label id &
- A field containing entry, ptr to symbol table entry for the identifier
• mkleaf(num, val)
- Create a number node with label num &
- A field containing val, the value for the number

Syntax tree for expression a-4+c
• P1=mkleaf(id,entrya);
• P2=mkleaf(num, 4);
• P3=mknode('-',p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode('+',p3,p4);

Question sources

Computer Science Exit Exam · 2018 · Question 53

Question 29 | Q00945

Which statement is WRONGLY described about code generation?

A. Code generation can be considered as the start phase of compilation

B. The target program is the output of the code generator

C. Checked intermediate code is supplied as input to the code generator.

D. It used to produce the target code for three-address statements.

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Code generation can be considered as the start phase of compilation

Lexical analysis begins the standard pipeline by recognizing tokens. Code generation occurs after analysis and IR construction, lowering checked IR to a target form. Its output is target code, which may be assembly or an object form, and it can translate three-address statements. Therefore calling code generation the start phase is the wrong description. The input option is read as checked IR suitable for lowering, without claiming a compiler can prove the absence of every program defect.

Why each choice is right or wrong

A - Correct: Code generation follows the frontend and IR; it is not the initial scanning phase.

B - Incorrect: The generator produces code in a target representation.

C - Incorrect: Checked intermediate code supplies the operations and metadata to lower.

D - Incorrect: Three-address statements can be lowered into target instructions.

Primary note proof

Study-note section COMP03

> The conventional sequence is lexical analysis → syntax analysis → semantic analysis → intermediate-code generation → optimization → target-code generation. Scanning turns characters into tokens; parsing builds structure; semantics checks declarations and types; synthesis produces implementation code. A phase is a logical activity; a pass is a traversal that can combine activities.

Study-note section COMP34

> Code generation lowers checked IR to a target form.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

> 6. Code Generation
> Converts intermediate code to machine code.
> Must handle all aspects of machine architecture
> Storage allocation decisions are made
> a. Register allocation and assignment

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters
> - Represent lexemes in the form of tokens as:
> <token-name, attribute-value>

- Token
- smallest meaningful element that a compiler understands.
- Eg.
- Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Question sources

Computer Science Exit Exam · 2018 · Question 58

Question 30 | Q00973

Which of the following is NOT a task of the lexical analyzer?

A. Generating parse trees

B. Correlating errors with line numbers

C. Entering identifiers into the symbol table

D. Stripping whitespace and comments

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Generating parse trees

Scanning turns characters into tokens, usually skips insignificant whitespace and comments, and retains source positions for diagnostics. A scanner may also intern identifier spellings in a symbol table; later semantic processing adds scope and declaration information. Parse-tree construction belongs to syntax analysis because it arranges the token stream according to grammar productions. It is consequently the activity outside the scanner's listed responsibilities.

Why each choice is right or wrong

A - Correct: Parse trees represent grammar structure built by parsing.

B - Incorrect: Tracking token positions and line numbers supports scanner diagnostics.

C - Incorrect: Scanner-side interning can enter identifier spellings, distinct from later declaration analysis.

D - Incorrect: Ordinary scanners discard insignificant whitespace and comments.

Primary note proof

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Question sources

Computer Science Exit Exam · 2018 · Question 86

Question 31 | Q01165

Which analysis used to read the sequence of the characters from the source program?

A. a | Lexical Analysis A

B. i? Syntax Analysis ><

C. /@| Semantic Analysis x

D. | All of the above x

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: a | Lexical Analysis A

PDF p. 5 explicitly says lexical analysis scans source code as a stream of characters; syntax groups tokens, and semantics checks meaningful compatibility (p. 6).

Why each choice is right or wrong

A - Correct: Lexical analysis explicitly reads the character stream.

B - Incorrect: Syntax analysis works on tokens grouped hierarchically.

C - Incorrect: Semantic analysis performs meaning/type checks (p. 6).

D - Incorrect: The raw-character scanning responsibility is assigned specifically to lexical analysis, not all three phases (pp. 5-6).

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters

- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science Exit Exam practice questions · Question 26

Question 32 | Q01193

What is the bottom-up parsing method also known as?

A. Recursive descent parsing

B. Predictive parsing

C. Shift-reduce parsing

D. All of them

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer C: Shift-reduce parsing

Bottom-up parsing starts with input symbols and reduces recognized right-hand sides until it reaches the start symbol. A shift-reduce parser implements this by shifting input onto a stack and reducing handles. Recursive descent expands from a start-symbol procedure and is top-down; predictive parsing is a top-down approach that uses lookahead to choose productions. Therefore the aggregate alternative cannot apply to methods with these different construction directions.

Why each choice is right or wrong

A - Incorrect: Recursive descent expands productions from nonterminals and is top-down.

B - Incorrect: Predictive parsing chooses top-down productions using lookahead.

C - Correct: Shifting input and reducing handles is the bottom-up shift-reduce method.

D - Incorrect: The list includes top-down methods, so not all are bottom-up.

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

> Parsing
> • Process of constructing parse tree for a sentence generated by a given grammar.
> • 2 types of parsers
> ● Top down parsing (predictive parsers)
> • LL(1)
> ● Bottom up parsing (SR parsers)
> • LR(1)

Compiler Design module | PDF page 10 | lines 18-23

> Top Down Parsing
> • The parse tree is created top to bottom
> • Starts from the start symbol and transform it to the input
> Bottom Up Parsing
> • Starts with the input symbols and tries to construct the parse tree up to the start symbol.
> • One way of reducing a sentence is to follow the right most derivation in reverse

Question sources

Computer Science model exam 2 · Question 14

Question 33 | Q01222

How many tokens are in the C++ loop header for (int i = 0; i <= 10; i++)? Ignore whitespace and count <= and ++ as single operator tokens.

A. 4

B. 5

C. 9

D. 14

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer D: 14

The tokens are for, (, int, i, =, 0, ;, i, <=, 10, ;, i, ++,). There are fourteen occurrences. Whitespace contributes nothing, but both semicolons and both parentheses count. Longest match makes <= and ++ one token each; it does not merge neighboring identifiers or discard repeated tokens. The numerical option therefore gives the unique count.

Why each choice is right or wrong

A - Incorrect: Four omits most token occurrences.

B - Incorrect: Five misses operators, punctuation and operands.

C - Incorrect: Nine is below the fourteen-token count.

D - Correct: The complete sequence contains fourteen tokens.

Primary note proof

Study-note section COMP07

> At each input position, choose the longest matching lexeme; if equal lengths match multiple flex rules, choose the earliest rule. Thus ifx is an identifier rather than if followed by x; <= and ++ are each one operator token.

Study-note section COMP07

> There are 14 tokens, irrespective of spacing. The increment operator is one token.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
- A string of characters which logically belong together
- Classes of similar lexemes

l identifier, keywords, constants etc.
• Pattern
- A rule which describes a token

• Lexeme
- The sequence of characters matched by a pattern to form the token

Compiler Design module | PDF page 8 | lines 19-24

• Classes of Tokens
- Identifiers: names chosen by the programmer
- Keywords: names already in the programming language
- Separators: punctuation characters
- Operators: symbols that operate on arguments and produce results
- Literals: numeric, textual literals

Question sources

Computer Science model exam 2 · Question 43

Question 34 | Q01243

Identify the correct definition of lexical analysis.

A. Breaking a sequence of characters into packets.

B. Breaking a sequence of characters into groups.

C. Breaking a sequence of characters into tokens.

D. Breaking a sequence of characters into lines.

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: Breaking a sequence of characters into tokens.

Lexical analysis takes a source program and outputs a sequence of tokens (PDF p. 8). Thus its relevant grouping units are tokens, rather than network packets, arbitrary groups, or source lines.

Why each choice is right or wrong

A - Incorrect: Packets are not the compiler units described on p. 8.

B - Incorrect: Groups is too unspecific; the defined output units are tokens (p. 8).

C - Correct: Breaking the character sequence into tokens matches scanning (pp. 5, 8).

D - Incorrect: Lines are tracked and whitespace stripped; they are not the primary output units (p. 8).

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
- A string of characters which logically belong together
- Classes of similar lexemes

l identifier, keywords, constants etc.
• Pattern
- A rule which describes a token

• Lexeme
- The sequence of characters matched by a pattern to form the token

Question sources

Computer Science model exam 2 · Question 64

Question 35 | Q01254

What does a top-down parser generate?

A. Rightmost derivation in reverse.

B. Rightmost derivation.

C. Leftmost derivation in reverse.

D. Leftmost derivation.

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer D: Leftmost derivation.

The module says top-down parsing builds from the start symbol (PDF p. 10), and the LL/LR comparison pairs top-down LL with leftmost derivation and bottom-up LR with rightmost derivation in reverse (p. 11). D matches the top-down direction.

Why each choice is right or wrong

A - Incorrect: Rightmost derivation in reverse is the LR/bottom-up entry (p. 11).

B - Incorrect: Rightmost derivation alone is not the listed top-down entry (p. 11).

C - Incorrect: The top-down derivation is leftmost in the forward direction, not reversed (p. 11).

D - Correct: Leftmost derivation is explicitly the LL/top-down entry (p. 11).

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP11

> A nonrecursive predictive parser stores grammar symbols on an explicit stack and consults a table. It is another implementation of predictive parsing, not a wholly different language class. Table-driven and recursive predictive implementations can both be linear for a fixed suitable grammar. Efficiency cannot be ranked from their names alone.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 18-23

> Top Down Parsing
> • The parse tree is created top to bottom

• Starts from the start symbol and transform it to the input
Bottom Up Parsing
• Starts with the input symbols and tries to construct the parse tree up to the start symbol.
• One way of reducing a sentence is to follow the right most derivation in reverse

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science model exam 2 · Question 75

Question 36 | Q01257

For line 3, fro ((I = o, I;, in the displayed C program, classify only lexical and syntactic well-formedness. Treat fro and o as legal identifier spellings and leave declaration/type diagnostics outside this question.

A. Syntax error, with no lexical error established.

B. No compilation error.

C. Only lexical error.

D. Both lexical and syntax error.

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Syntax error, with no lexical error established.

The characters fro and o form valid identifier lexemes, so their spellings do not establish a lexical error. The displayed line nevertheless has mismatched parentheses and an invalid token arrangement, establishing a syntax error. Under this restricted lexical/syntactic classification, A is correct. Declaration or type diagnostics may arise in a real compiler, but are outside the expressly stated classification scope.

Why each choice is right or wrong

A - Correct: The arrangement is syntactically invalid without an established lexical fault.

B - Incorrect: Mismatched delimiters prevent syntactic validity.

C - Incorrect: Legal identifier spellings do not establish a lexical error.

D - Incorrect: A syntax error exists but no lexical error is established.

Primary note proof

Study-note section COMP08

> A lexical error prevents forming a legal token, such as an invalid character or unterminated literal. A syntax error is an illegal token arrangement, such as a missing delimiter. A semantic error violates name, scope or type rules after grammatical recognition.

Study-note section COMP08

> In the malformed C line fro ((I = o, I;, fro and lowercase o are legal identifier lexemes. Misspelling for does not turn fro into a lexical error; mismatched syntax is definite, and undeclared o can also produce a semantic diagnostic.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters
> - Represent lexemes in the form of tokens as:
> <token-name, attribute-value>
> - Token
> • smallest meaningful element that a compiler understands.
> - Eg.
> • Identifiers, Keywords, Literals, Operators and Special symbols.
> - Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 5 | lines 12-18

> 2. Syntax / Hierarchical Analysis - Parsing
> • Tokens are grouped hierarchically into nested collections with collective meaning.
> • The result is generally a parse tree.
> • expressions, statements, declarations etc... are identified by using the results of lexical analysis.

• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
● A string of characters which logically belong together
● Classes of similar lexemes
l identifier, keywords, constants etc.
• Pattern
● A rule which describes a token
• Lexeme
● The sequence of characters matched by a pattern to form the token

Question sources

Computer Science model exam 2 · Question 78

Question 37 | Q01397

Which tool is used for grouping of characters in tokens in the compiler?

A. Parser

B. Code optimizer

C. Code generator

D. Scanner

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Scanner

Lexical analysis is called scanning on PDF p. 5 and outputs tokens on p. 8. Hence the scanner is the component that groups source characters into tokens, before the parser handles token structure.

Why each choice is right or wrong

A - Incorrect: The parser groups tokens into syntactic constructs.

B - Incorrect: The optimizer improves intermediate code (p. 7).

C - Incorrect: The generator converts intermediate code to machine code (p. 7).

D - Correct: Scanner is the component associated with lexical analysis (pp. 5, 8).

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Question sources

Compiler Design review questions · Question 5

Question 38 | Q01399

When parsing techniques are classified by the direction in which they construct the parse tree, how many broad categories are there?

A. three types

B. four types

C. two types

D. five types

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer C: two types

Construction direction gives two broad categories. Top-down parsing expands from the start symbol toward the input, while bottom-up parsing reduces input toward the start symbol. Recursive descent, predictive parsing and LR are algorithms or families within that classification rather than additional directions. Counting named algorithms would give a different and open-ended taxonomy, so the revised question explicitly counts only the broad construction directions.

Why each choice is right or wrong

A - Incorrect: Three is not the number of construction directions in this two-way taxonomy.

B - Incorrect: Four may count selected algorithms, but it does not count the two broad directions.

C - Correct: The directions are top-down and bottom-up.

D - Incorrect: Five is not the number of construction directions.

Primary note proof

Study-note section COMP-S01

> The common two-way classification of parsing is by construction direction: top-down parsing expands from the start symbol, while bottom-up parsing reduces the input toward that symbol. This counts broad directions, not the number of parser algorithms or implementations.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

> Parsing
> • Process of constructing parse tree for a sentence generated by a given grammar.
> • 2 types of parsers
> ● Top down parsing (predictive parsers)
> • LL(1)
> ● Bottom up parsing (SR parsers)
> • LR(1)

Question sources

Compiler Design review questions · Question 7

Question 39 | Q01400

In which parsing, the parser constructs the parse tree from the start symbol and transforms it into the input symbol.

A. Bottom-up parsing

B. Top-down parsing

C. None of the above

D. Both a and b

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Top-down parsing

Top-down parsing starts from the start symbol and transforms it into the input (PDF p. 10). Bottom-up parsing starts with input symbols and constructs toward the start symbol. The stem is an almost verbatim description of the first direction.

Why each choice is right or wrong

A - Incorrect: Bottom-up starts from input symbols, the opposite endpoint (p. 10).

B - Correct: Top-down starts from the start symbol as requested (p. 10).

C - Incorrect: The described category is explicitly present, so none is false (p. 10).

D - Incorrect: Both is false because the two categories have opposite starting points (p. 10).

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP11

> A nonrecursive predictive parser stores grammar symbols on an explicit stack and consults a table. It is another implementation of predictive parsing, not a wholly different language class. Table-driven and recursive predictive implementations can both be linear for a fixed suitable grammar. Efficiency cannot be ranked from their names alone.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 18-23

Top Down Parsing
• The parse tree is created top to bottom
• Starts from the start symbol and transform it to the input
Bottom Up Parsing
• Starts with the input symbols and tries to construct the parse tree up to the start symbol.
• One way of reducing a sentence is to follow the right most derivation in reverse

Question sources

Compiler Design review questions · Question 8

Question 40 | Q01401

Which derivation is generated by the top-down parser?

A. Right-most derivation in reverse

B. Left-most derivation in reverse

C. Right-most derivation

D. Left-most derivation

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer D: Left-most derivation

The LL/top-down column uses leftmost derivation, while the LR/bottom-up column uses rightmost derivation in reverse (PDF p. 11). Consequently a top-down parser generates a forward leftmost derivation.

Why each choice is right or wrong

A - Incorrect: This is the bottom-up LR direction (p. 11).

B - Incorrect: The LL entry is leftmost without reversal (p. 11).

C - Incorrect: Rightmost is not the listed top-down derivation (p. 11).

D - Correct: Leftmost is explicitly the top-down/LL entry (p. 11).

Primary note proof

Study-note section COMP11

Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP11

A nonrecursive predictive parser stores grammar symbols on an explicit stack and consults a table. It is another implementation of predictive parsing, not a wholly different language class. Table-driven and recursive predictive implementations can both be linear for a fixed suitable grammar. Efficiency cannot be ranked from their names alone.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Compiler Design module | PDF page 10 | lines 18-23

Top Down Parsing
• The parse tree is created top to bottom
• Starts from the start symbol and transform it to the input
Bottom Up Parsing
• Starts with the input symbols and tries to construct the parse tree up to the start symbol.
• One way of reducing a sentence is to follow the right most derivation in reverse

Question sources

Compiler Design review questions · Question 9

Question 41 | Q01402

Which derivation is generated by the bottom-up parser?

A. Right-most derivation in reverse

B. Left-most derivation in reverse

C. Right-most derivation

D. Left-most derivation

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A: Right-most derivation in reverse

PDF p. 10 describes bottom-up parsing as reducing input symbols toward the start symbol, following a rightmost derivation in reverse. PDF p. 11 repeats this as the definition of LR parsing. Thus A gives both the correct derivation kind and direction.

Why each choice is right or wrong

A - Correct: Rightmost derivation in reverse is explicitly stated (pp. 10-11).

B - Incorrect: Leftmost in reverse is not the module's bottom-up rule (p. 11).

C - Incorrect: Rightmost without reversal would expand rather than describe the stated reductions (pp. 10-11).

D - Incorrect: Leftmost is paired with LL/top-down parsing (p. 11).

Primary note proof

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse. A handle is a production RHS occurrence whose reduction is the next step of that reverse derivation. Not every substring matching a RHS is a handle in the current context.

Study-note section COMP15

> Shift pushes the next input token and a state; reduce replaces a recognized RHS by its LHS and consults GOTO; accept recognizes completion; error reports an invalid situation. For A→XYZ, pop three grammar symbols and their states, expose state q, then push A and GOTO[q,A]. "Go" is not a separate standard shift-reduce ACTION, although GOTO is essential table machinery.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 18-23

> Top Down Parsing
> • The parse tree is created top to bottom
> • Starts from the start symbol and transform it to the input
> Bottom Up Parsing
> • Starts with the input symbols and tries to construct the parse tree up to the start symbol.
> • One way of reducing a sentence is to follow the right most derivation in reverse

Compiler Design module | PDF page 11 | lines 2-17

> LL(1) Grammar
> • L - left to right
> • L - left most derivation
> • 1 - number of look ahead

• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Compiler Design review questions · Question 10

Question 42 | Q01403

Among these named deterministic parser methods, which has the broadest grammar coverage under the standard SLR(1), LALR(1) and canonical LR(1) comparison?

A. Operator Precedence

B. SLR

C. Canonical LR

D. LALR

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer C: Canonical LR

Canonical LR(1) keeps context-specific lookahead and covers at least the grammars handled by SLR(1) and LALR(1), with strictly greater coverage in the standard comparison.
Operator-precedence parsing is a more restricted expression-parsing method. This is a comparison of grammar coverage, not speed, implementation simplicity or table size. Canonical LR may require larger tables and still does not parse every unambiguous context-free grammar.

Why each choice is right or wrong

A - Incorrect: Operator-precedence parsing applies to a more restricted grammar family.

B - Incorrect: SLR(1) uses less context-specific reduction information and has narrower grammar coverage.

C - Correct: Canonical LR(1) has the broadest coverage among the named methods.

D - Incorrect: Merging LR(1) cores for LALR(1) can lose distinctions needed by some LR(1) grammars.

Primary note proof

Study-note section COMP-S01

> In the standard comparison by grammar coverage, canonical LR(1) is more general than the restricted operator-precedence method. Together with SLR(1)⊆LALR(1)⊆LR(1), this makes canonical LR(1) the broadest of those named deterministic parsing methods. Broader coverage does not mean faster execution or smaller tables.

Study-note section COMP17

> The standard grammar-class hierarchy is LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1), with increasing capability and often table size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 11 | lines 2-17

> LL(1) Grammar
> • L - left to right
> • L - left most derivation
> • 1 - number of look ahead
> • First( ) and Follow( )
> ● the first terminal in a string and the terminal that follows a variable respectively.
> LR Parsing
> • LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
> Types of LR Parsers
> • LR (0) , SLR (1) , LALR (1) , CLR (1)
> LL LR
> Leftmost derivation Rightmost derivation in reverse
> Starts with root non-terminal on stack Ends with root non-terminal on the stack
> Builds the parse tree top-down Builds the parse tree bottom-up
> Expands the non-terminals Reduces the non-terminals

Question sources

Compiler Design review questions · Question 11

Question 43 | Q01404

The output of the lexical analyzer is _____

A. string character

B. a syntax tree

C. a set of RE

D. a set of tokens

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: a set of tokens

The lexical analyzer's output is a sequence of tokens (PDF p. 8). Among the options only D identifies these elements; "set" is informal wording for the ordered stream.

Why each choice is right or wrong

A - Incorrect: Source characters are scanned as input.

B - Incorrect: The syntax tree belongs to parsing.

C - Incorrect: Regular expressions are not the listed scanner output; patterns are rules (p. 8).

D - Correct: Tokens are explicitly the scanner output (p. 8).

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

> What is Lexical Analysis
> • The first phase of a compiler
> • The input is a high level language program
> • The output is a sequence of tokens
> • Strips off blanks, tabs, newlines, and comments from the source program
> • Keeps track of line numbers

Compiler Design module | PDF page 8 | lines 10-18

> Tokens, Patterns, and Lexemes
> • Token
> • A string of characters which logically belong together

- Classes of similar lexemes
l identifier, keywords, constants etc.
• Pattern
- A rule which describes a token
• Lexeme
- The sequence of characters matched by a pattern to form the token

Question sources

Compiler Design review questions · Question 12

Question 44 | Q01405

Which is the least sufficient Chomsky grammar family for ordinary token patterns specified by classical regular expressions, rather than a purpose-based grammar name?

A. Lexical Grammar

B. Context-free Grammar

C. Syntactic Grammar

D. Regular Grammar

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer D: Regular Grammar

Ordinary regular-expression token patterns describe regular languages, whose least sufficient Chomsky family is regular grammar. A lexical grammar names a specification by purpose, whereas this question asks for its formal family. Context-free grammars are more expressive than required and typically model nested syntax. Syntactic grammar is another purpose label rather than the specific formal family matching the regular token patterns.

Why each choice is right or wrong

A - Incorrect: Lexical grammar is a purpose label, not the requested family.

B - Incorrect: CFGs are stronger than these regular patterns require.

C - Incorrect: Syntactic grammar is a purpose label.

D - Correct: Regular grammars generate exactly regular token languages.

Primary note proof

Study-note section COMP-S03

Classical regular expressions, consistently right-linear regular grammars and finite automata describe the same regular-language family. Thus an ordinary token pattern specified by a regular expression also has a regular grammar and an equivalent finite-state recognizer.

Study-note section COMP-S03

This equivalence concerns expressive power, not spelling of the specification or implementation size. Context-free grammars can also describe regular languages, but their larger family is needed for syntax with unbounded nested structure rather than ordinary regular token patterns.

Study-note section COMP18

Regular patterns are chiefly lexical specifications; CFGs chiefly describe nested syntax.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
- A string of characters which logically belong together
- Classes of similar lexemes
l identifier, keywords, constants etc.
• Pattern
- A rule which describes a token
• Lexeme
- The sequence of characters matched by a pattern to form the token

Question sources

Compiler Design review questions · Question 13

MOE Computer Science Exit Exam · 2016 · Question 93

Question 45 | Q01407

In Compiler lexical analyzer is used for?

A. removing comments

B. removing whitespace

C. breaking the syntaxes in the set of tokens

D. All of the mentioned

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: All of the mentioned

Lexical analysis scans source characters into tokens and removes blanks/newlines/comments (PDF pp. 5, 8). Thus comment removal, whitespace removal, and tokenization all belong to the scanner in this module, making D the inclusive choice.

Why each choice is right or wrong

A - Incorrect: Comment removal is expressly stated (p. 8).

B - Incorrect: Whitespace stripping is expressly stated (p. 8).

C - Incorrect: Interpreting "syntaxes" as source text, tokenization is the lexical role.

D - Correct: All combines the three documented responsibilities (pp. 5, 8).

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters
> - Represent lexemes in the form of tokens as:

<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Question sources

Compiler Design review questions · Question 15

Question 46 | Q01408

Which is considered as the sequence of characters in a token?

A. Mexeme

B. Lexeme

C. Texeme

D. Pattern

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Lexeme

PDF p. 8 calls the matched character sequence a Lexeme. A pattern is the rule that matches that sequence, rather than the concrete spelling itself.

Why each choice is right or wrong

A - Incorrect: Mexeme is not a defined tokenization term (p. 8).

B - Correct: Lexeme is the exact defined term (p. 8).

C - Incorrect: Texeme is not a defined tokenization term (p. 8).

D - Incorrect: Pattern is a rule describing a token, not the matched sequence (p. 8).

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 10-18

Tokens, Patterns, and Lexemes
• Token
- A string of characters which logically belong together
- Classes of similar lexemes

l identifier, keywords, constants etc.
• Pattern
- A rule which describes a token

• Lexeme
- The sequence of characters matched by a pattern to form the token

Question sources

Compiler Design review questions · Question 16

Question 47 | Q01409

Which part of the compiler highly used the grammar concept?

A. Code optimization

B. Code generation

C. Parser

D. Lexical Analysis

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: Parser

PDF p. 9 explicitly states that parsers/syntax analyzers are generated for a particular grammar and CFGs specify programming-language syntax. This directly connects the parser to the grammar concept asked about.

Why each choice is right or wrong

A - Incorrect: Optimization is defined through removal of inefficiencies, not grammar-based recognition (p. 7).

B - Incorrect: Generation converts intermediate code to machine code (p. 7).

C - Correct: Parser generation and syntax specification are explicitly grammar-based (p. 9).

D - Incorrect: Lexical patterns classify tokens (p. 8); the cited grammar emphasis is the parser (p. 9).

Primary note proof

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

Compiler Design review questions · Question 17

Question 48 | Q01410

Which phase of the compiler checks the grammar of the programming?

A. Code Optimization

B. Semantic Analysis

C. Code Generation

D. Syntax Analysis

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Syntax Analysis

Syntax analysis groups tokens into nested constructs, catches syntactic errors, and uses source-language grammar rules (PDF p. 5). Therefore it checks grammatical program structure; semantic checks, optimization and target-code generation perform different work.

Why each choice is right or wrong

A - Incorrect: Optimization removes intermediate-code inefficiencies (p. 7).

B - Incorrect: Semantic analysis checks meaningful compatibility and types (p. 6).

C - Incorrect: Code generation emits machine code from intermediate code (p. 7).

D - Correct: Syntax analysis is explicitly governed by grammar rules.

Primary note proof

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Compiler Design module | PDF page 7 | lines 2-6

5. Code Optimization
Changes the IC by removing such inefficiencies
Improve the code
a. Improvement may be time, space, or power consumption.
It changes the structure of programs,

Question sources

Compiler Design review questions · Question 18

Question 49 | Q01411

Which listed data structure supplies declaration names, scopes and types to semantic analysis?

A. Yacc

B. Lex

C. Symbol Table

D. Type Checking

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Symbol Table

A symbol table supplies the identities and attributes needed to resolve names and check types during semantic analysis. Type checking is an important activity using that information, but is not itself the requested data structure. Yacc and Lex generate parsers and scanners respectively. This narrower role makes the symbol table unique without denying that several components contribute to semantic processing.

Why each choice is right or wrong

A - Incorrect: Yacc is a parser generator.

B - Incorrect: Lex is a scanner generator.

C - Correct: The symbol table stores declaration and type metadata.

D - Incorrect: Type checking is an activity, not the information structure.

Primary note proof

Study-note section COMP19

> A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage.

Study-note section COMP39

> Lex/flex generates scanners from patterns and actions. Yacc means Yet Another Compiler Compiler; Yacc/Bison generates parsers from grammar rules and semantic actions.

Study-note section COMP22

> A type checker verifies that operations are defined for their operands.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

> 3. Semantic Analysis
> • Certain checks are performed to make sure that the components of the program fit together meaningfully.
> • Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
> - Type checking of various programming language constructs is one of the most important tasks.
> • Stores type information in the symbol table or the syntax tree.
> - Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 14 | lines 2-15

• Type Errors:
- Improper or inconsistent operations during program execution

• Type-safety:
- Absence of type errors

Type Checking
• Semantic checks to enforce the type safety of the program
• Semantic Checks
- Static - done during compilation
- Dynamic - done during run-time

• Examples
- Unary and binary operators
- Number and type of arguments
- Return statement with return type
- Compatible assignment

Question sources

Compiler Design review questions · Question 19

Question 50 | Q01412

Which phase of the compiler is also known as Scanner?

A. Syntax Analysis

B. Lexical Analysis

C. Semantic Analysis

D. Code generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Lexical Analysis

The module labels lexical/linear analysis "scanning" (PDF p. 5). Its scanner reads characters into tokens (p. 8), so B names the phase associated with Scanner.

Why each choice is right or wrong

A - Incorrect: Syntax analysis is called parsing.

B - Correct: Lexical analysis is explicitly scanning.

C - Incorrect: Semantic analysis checks types/meaning (p. 6).

D - Incorrect: Code generation converts intermediate code to machine code (p. 7).

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Question sources

Compiler Design review questions · Question 20

Question 51 | Q01413

Which phase of the compiler is also known as Parser?

A. Code Optimization

B. Semantic Analysis

C. Syntax Analysis

D. Lexical Analysis

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: Syntax Analysis

Syntax/hierarchical analysis is explicitly labeled Parsing (PDF p. 5); p. 9 equates parsers with syntax analyzers. Thus the phase known as Parser is syntax analysis.

Why each choice is right or wrong

A - Incorrect: Optimization improves intermediate code (p. 7).

B - Incorrect: Semantic analysis checks meaningful compatibility (p. 6).

C - Correct: Syntax analysis is expressly Parsing (pp. 5, 9).

D - Incorrect: Lexical analysis is scanning, preceding parsing.

Primary note proof

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

> 2. Syntax / Hierarchical Analysis - Parsing
> • Tokens are grouped hierarchically into nested collections with collective meaning.
> • The result is generally a parse tree.
> • expressions, statements, declarations etc... are identified by using the results of lexical analysis.
> • Most syntactic errors in the source program are caught in this phase.
> • Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 9 | lines 6-16

> Grammars
> • Every programming language has grammar rules
> • Parsers or syntax analyzers are generated for a particular grammar
> • CFG are used for syntax specification of programming languages
> Context Free Grammar (CFG)
> • Is denoted as G = (N, T , P, S)

• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Question sources

Compiler Design review questions · Question 21

Question 52 | Q01414

Which of the following parser is a top-down parser?

A. An LALR parser

B. A LR parser

C. Operator precedence parser

D. Recursive descent parser

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer D: Recursive descent.

COMP11 introduces recursive descent as a top-down implementation, normally with one procedure per nonterminal. It expands the start symbol toward the input. LR and LALR instead use bottom-up reductions, and operator-precedence parsing is also bottom-up. Therefore only recursive descent names the top-down family among these options.

Why each choice is right or wrong

A - Incorrect: LALR is an LR bottom-up family, using merged LR item states.

B - Incorrect: LR recognizes input through bottom-up shifts and reductions.

C - Incorrect: Operator-precedence parsing reduces expressions bottom-up rather than expanding productions top-down.

D - Correct: Recursive descent implements top-down parsing through procedures corresponding to nonterminals.

Primary note proof

Study-note section COMP11

Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using

lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP15

Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse.

Study-note section COMP-S01

Operator-precedence parsing is a restricted bottom-up expression-parsing method using a stack and precedence relations. With a suitable expression grammar, the stack retains unfinished surrounding expressions while nested parentheses are processed.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
- Top down parsing (predictive parsers)
• LL(1)
- Bottom up parsing (SR parsers)
• LR(1)

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
- the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Compiler Design review questions · Question 22

Question 53 | Q01415

Keywords are recognized in a compiler during -

A. the code generation

B. the data flow analysis

C. the lexical analysis of the program

D. the program parsing

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: the lexical analysis of the program

Keywords are listed among the token categories identified during lexical analysis (PDF pp. 5, 8). Recognizing a reserved spelling therefore occurs in the lexical classification of the program rather than later target generation or parsing.

Why each choice is right or wrong

A - Incorrect: Generation acts on intermediate code, not raw keyword spellings (p. 7).

B - Incorrect: Data-flow analysis is not identified as keyword recognition.

C - Correct: Keywords are explicitly lexical token classes (pp. 5, 8).

D - Incorrect: Parsing groups the already-produced tokens.

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters
> - Represent lexemes in the form of tokens as:

<token-name, attribute-value>
- Token
- smallest meaningful element that a compiler understands.
- Eg.
- Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 19-24

- Classes of Tokens
- Identifiers: names chosen by the programmer
- Keywords: names already in the programming language
- Separators: punctuation characters
- Operators: symbols that operate on arguments and produce results
- Literals: numeric, textual literals

Question sources

Compiler Design review questions · Question 23

Question 54 | Q01417

Which names denote the graph whose vertices are basic blocks and edges are possible next-block control transfers? Select all that apply. In A, control graph abbreviates control-flow graph.

A. Control graph

B. DAG

C. Flow graph

D. Hamilton graph

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer A, C: Control graph; Flow graph

A control-flow graph represents basic blocks as vertices and possible execution transfers, including fall-through, as edges. With the stated abbreviation, both control graph and flow graph name that representation. An expression DAG instead represents shared computations and value dependencies. A Hamiltonian-path label concerns a graph property rather than the compiler representation of block successors. Thus A and C are both valid names.

Why each choice is right or wrong

A - Correct: The declared abbreviation denotes the specified graph.

B - Incorrect: An expression DAG records value dependencies, not block successors.

C - Correct: Flow graph conventionally names this representation.

D - Incorrect: Hamiltonian paths do not define the block-successor representation.

Primary note proof

Study-note section COMP28

> A basic block is a maximal straight-line instruction sequence entered at its beginning and exited at its end. Leaders are the first instruction, every jump target, and each instruction after a branch. Start a block at each leader and continue until the next leader.

Study-note section COMP28

> A control-flow graph has blocks as vertices and possible next-block transfers as edges, including fall-through. A node D dominates N when every entry-to-N path passes through D. For an edge N→H where H dominates N, H is a loop header of a natural loop. This structure supports loop analysis. An expression DAG records value dependencies, not arbitrary control flow.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 2-10

> Techniques
> • Common sub-expression elimination
> ● Repeated appearance computed previously
> • Strength reduction
> ● Replacement of expensive expressions with simple ones
> • Code movement
> ● Moving a block of code outside a loop
> • Dead code elimination
> ● Eliminated code statements that are either never executed or unreachable

Compiler Design module | PDF page 16 | lines 2-8

> Three Address Code
> • Is a generic form and can be implemented as:
> ● Quadruples
> ● Triples
> ● Indirect Triples
> ● Tree
> ● DAG

Question sources

Compiler Design review questions · Question 25

Question 55 | Q01420

In the compiler, the function of using intermediate code is:

A. to improve the register allocation

B. to increase the error reporting & recovery.

C. to make semantic analysis easier.

D. to increase the chances of re-using the machine-independent code optimizer in other compilers.

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer D: Reuse the machine-independent optimizer across compilers.

COMP26 explains that IR separates source analysis from target implementation, making optimizers and backends reusable. This directly supports reusing a machine-independent optimizer in different compilers. An IR can support allocation and analysis, but none of the other proposed improvements is its defining architectural function, nor does merely introducing IR guarantee them.

Why each choice is right or wrong

A - Incorrect: Allocation may analyze IR, but improved allocation is not the portability function identified here.

B - Incorrect: Error reporting and recovery are shared compiler services, not the defining purpose of introducing IR.

C - Incorrect: Semantic analysis normally establishes source correctness before lowering; IR is not introduced chiefly to simplify it.

D - Correct: Separating source and target through IR enables reuse of machine-independent optimization passes.

Primary note proof

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 3 | lines 25-29

Phases of a Compiler
• Analysis
● Machine Independent/Language Dependent
• Synthesis
● Machine Dependent/Language independent

Compiler Design module | PDF page 6 | lines 11-12

4. Intermediate Code Generation
Easy to produce and easy to translate to machine code

Compiler Design module | PDF page 7 | lines 2-6

5. Code Optimization
Changes the IC by removing such inefficiencies
Improve the code
a. Improvement may be time, space, or power consumption.
It changes the structure of programs,

Question sources

Compiler Design review questions · Question 28

Question 56 | Q01421

How many categories result when optimizations are divided solely by whether they depend on the target machine?

A. two types

B. three types

C. four types

D. five types

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer A: two types

The stated criterion divides optimizations into machine-dependent and machine-independent categories, giving two categories. Other taxonomies use different criteria, such as local, global and loop scope, and need not have the same count. Individual transformations are not additional categories under this binary division. Naming the criterion fixes the count without claiming a universal number of optimization types across all classifications.

Why each choice is right or wrong

A - Correct: The two categories are machine-dependent and machine-independent.

B - Incorrect: Three fits another possible scope taxonomy.

C - Incorrect: This binary division does not yield four categories.

D - Incorrect: This binary division does not yield five categories.

Primary note proof

Study-note section COMP32

> A loop-invariant expression has unchanged inputs during a loop. Hoist it to a preheader only when evaluation is safe on the paths where it will newly run. Induction variables change by a loop-invariant increment, such as i=i+1. For repeated t=4*i, maintain a running t increased by 4 when equivalent, reducing repeated multiplication.

Study-note section COMP32

> Loop fusion or jamming combines loops with compatible bounds and safe dependence order. Unrolling replicates a body to reduce branch overhead and expose parallelism; handle remaining iterations and increased code size. Loop-invariant code motion, fusion, unrolling and strength reduction are different transformations. A finite taxonomy needs a criterion: machine dependent/independent versus local/global/loop are different classifications.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 2-6

> 5. Code Optimization
> Changes the IC by removing such inefficiencies
> Improve the code
> a. Improvement may be time, space, or power consumption.
> It changes the structure of programs,

Compiler Design module | PDF page 17 | lines 16-22

> Chapter 8
> Introduction to Code Optimization
> Goals of Code Optimization
> • Remove redundant code without changing the meaning of program
> • Executes faster
> • Efficient memory usage
> • Better performance

Question sources

Compiler Design review questions · Question 29

Question 57 | Q01422

The value of which variable is updated inside the loop by a loop-invariant value?

A. loop

B. strength

C. induction

D. invariable

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer C: An induction variable.

COMP32 defines induction variables as values changed by a loop-invariant increment, such as i=i+1. This matches an update inside a loop by an invariant amount. A loop-invariant value itself stays unchanged; strength is part of the term strength reduction, and loop alone names a construct rather than this variable class.

Why each choice is right or wrong

A - Incorrect: Loop names the repeated control construct rather than the variable category defined by invariant increments.

B - Incorrect: Strength refers to replacing costly operations, not to this variable classification.

C - Correct: An induction variable is updated by a loop-invariant increment during loop iterations.

D - Incorrect: An invariant value remains unchanged, unlike the updated variable described in the stem.

Primary note proof

Study-note section COMP32

> A loop-invariant expression has unchanged inputs during a loop. Hoist it to a preheader only when evaluation is safe on the paths where it will newly run. Induction variables change by a loop-invariant increment, such as i=i+1. For repeated t=4*i, maintain a running t increased by 4 when equivalent, reducing repeated multiplication.

Study-note section COMP32

> Loop fusion or jamming combines loops with compatible bounds and safe dependence order. Unrolling replicates a body to reduce branch overhead and expose parallelism; handle remaining iterations and increased code size. Loop-invariant code motion, fusion, unrolling and strength reduction are different transformations. A finite taxonomy needs a criterion: machine dependent/independent versus local/global/loop are different classifications.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 2-10

Techniques
• Common sub-expression elimination
  • Repeated appearance computed previously
• Strength reduction
  • Replacement of expensive expressions with simple ones
• Code movement
  • Moving a block of code outside a loop
• Dead code elimination
  • Eliminated code statements that are either never executed or unreachable

Question sources

Compiler Design review questions · Question 30

Question 58 | Q01423

Which compiler runs on one machine and generates code for multiple machines?

A. Multipass compiler

B. Cross compiler

C. Optimizing compiler

D. Onepass compiler

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Cross compiler

A cross-compiler runs on platform A and generates executable code for another platform B (PDF p. 3). B matches the host/target distinction. The module's definition concerns a different target platform; it does not require one executable to support several targets simultaneously.

Why each choice is right or wrong

A - Incorrect: Multipass is not the host/target distinction described on p. 3.

B - Correct: Cross compiler is explicitly the platform-A to platform-B translator (p. 3).

C - Incorrect: Optimizing refers to improving code (p. 7), not the host/target relationship.

D - Incorrect: Onepass is not the documented host/target category.

Primary note proof

Study-note section COMP02

A preprocessor handles directives such as file inclusion and macros. An assembler translates assembly to object code. A linker combines object files and libraries, resolves external symbols and performs relocation. A loader maps an executable into memory and starts it. Compilation alone does not necessarily produce a ready executable.

Study-note section COMP02

A cross-compiler runs on a host platform and generates code for a different target. Multiple targets require appropriate backends, not a special definition of cross-compilation. A retargetable compiler supports changing target descriptions. Incremental compilation recompiles affected fragments or dependent units after edits; it is more than literally replacing changed lines. One-pass versus multipass counts traversals, not supported targets.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 3 | lines 20-24

• Cross-Compiler
● compiler that runs on platform (A) and generates executable code for another platform (B).
• Source-to-source Compiler
● compiler that translates source code of one programming language to another

Question sources

Compiler Design review questions · Question 31

Question 59 | Q01425

Which phenomenon happens when the non-terminal on the left side is repeated as the first symbol on the right side?

A. Left-most derivation

B. Left recursion

C. Left factoring

D. Left parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Immediate left recursion.

COMP12 writes immediate left recursion as A→Aα: the production's left-hand nonterminal occurs first on its right-hand side. This is precisely the pattern in the question. Leftmost derivation chooses which nonterminal to expand; left factoring extracts a shared prefix. Neither is the repeated-leftmost-self-reference property.

Why each choice is right or wrong

A - Incorrect: Leftmost derivation chooses the leftmost remaining nonterminal to rewrite, regardless of self-reference.

B - Correct: A production beginning with its own left-hand nonterminal is immediate left recursion.

C - Incorrect: Left factoring extracts common prefixes from alternatives and does not define recursive self-reference.

D - Incorrect: Left parsing is not the note's name for the production pattern described.

Primary note proof

Study-note section COMP12

Immediate left recursion has A→Aα. For A→Aα|β, where β does not start with A, introduce a new nonterminal A′ and use:
A→βA′
A′→αA′|ε
This preserves the generated language; tree shape changes, so semantic actions must preserve intended associativity. Indirect left recursion follows a cycle through other nonterminals and also needs removal for ordinary predictive descent.

Study-note section COMP12

Left factoring rewrites A→αβ|αγ using:
A→αA′
A′→β|γ
It postpones a choice until after the common prefix. It does not by itself remove ambiguity or guarantee LL(1). LR parsers can handle suitable left-recursive grammars; ordinary unmodified recursive descent can recurse forever before consuming input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Compiler Design module | PDF page 10 | lines 18-23

Top Down Parsing
• The parse tree is created top to bottom
• Starts from the start symbol and transform it to the input
Bottom Up Parsing

• Starts with the input symbols and tries to construct the parse tree up to the start symbol.
• One way of reducing a sentence is to follow the right most derivation in reverse

Question sources

Compiler Design review questions · Question 33

Question 60 | Q01428

The compiler can detect what type of errors?

A. neither logical nor grammatical error

B. logical errors only

C. grammatical errors only

D. both grammatical and logical errors

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: grammatical errors only

PDF p. 5 says syntax analysis catches syntactic errors, so grammar violations are detectable. PDF p. 6 adds semantic type mismatches; it does not claim detection of arbitrary mistakes in the intended algorithm. Among the offered grammatical-versus-logical choices, C fits if "logical" means wrong intended algorithm rather than all semantic errors.

Why each choice is right or wrong

A - Incorrect: Neither conflicts with the documented detection of syntactic errors.

B - Incorrect: Logical-only omits the explicitly detected grammatical errors.

C - Correct: Grammatical errors are expressly detected; the intended-algorithm interpretation of logical makes this the matching choice.

D - Incorrect: The module does not claim that general intended-algorithm errors are detected.

Primary note proof

Study-note section COMP08

A lexical error prevents forming a legal token, such as an invalid character or unterminated literal. A syntax error is an illegal token arrangement, such as a missing delimiter. A semantic error violates name, scope or type rules after grammatical recognition. A wrong algorithm can compile successfully; compilers do not detect all logical mistakes.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Compiler Design review questions · Question 36

Question 61 | Q01430

Which method merges the multiple loops into the single one?

A. Constant Folding

B. Loop rolling

C. Loop fusion or jamming

D. None of the above

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer C: Loop fusion, also called jamming.

COMP32 explicitly defines loop fusion or jamming as combining loops with compatible bounds and safe dependence order. That is the requested merging transformation. Constant folding evaluates constant expressions, while loop rolling does not name this fusion of separate loops. Fusion must preserve dependence order; it is not automatically legal for arbitrary loops.

Why each choice is right or wrong

A - Incorrect: Constant folding evaluates constant expressions at compile time; it does not combine loop bodies.

B - Incorrect: Loop rolling is not the note's named transformation for merging separate compatible loops.

C - Correct: Fusion or jamming combines compatible loops into one while preserving required dependence order.

D - Incorrect: The explicit fusion/jamming option already names the requested transformation.

Primary note proof

Study-note section COMP32

> A loop-invariant expression has unchanged inputs during a loop. Hoist it to a preheader only when evaluation is safe on the paths where it will newly run. Induction variables change by a loop-invariant increment, such as i=i+1. For repeated t=4*i, maintain a running t increased by 4 when equivalent, reducing repeated multiplication.

Study-note section COMP32

> Loop fusion or jamming combines loops with compatible bounds and safe dependence order. Unrolling replicates a body to reduce branch overhead and expose parallelism; handle remaining iterations and increased code size. Loop-invariant code motion, fusion, unrolling and strength reduction are different transformations. A finite taxonomy needs a criterion: machine dependent/independent versus local/global/loop are different classifications.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 2-10

> Techniques
> • Common sub-expression elimination
> ● Repeated appearance computed previously
> • Strength reduction
> ● Replacement of expensive expressions with simple ones
> • Code movement
> ● Moving a block of code outside a loop
> • Dead code elimination
> ● Eliminated code statements that are either never executed or unreachable

Question sources

Compiler Design review questions · Question 38

Question 62 | Q01431

Which parser is known as the shift-reduce parser?

A. Bottom-up parser

B. Top-down parser

C. Both Top-down and bottom-up

D. None of the Above

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A: Bottom-up parser

PDF p. 10 names bottom-up parsing as SR parsing and describes reductions from the input to the start symbol. Interpreting SR as shift-reduce, A is the matching category; top-down is instead predictive.

Why each choice is right or wrong

A - Correct: Bottom-up is explicitly the SR category (p. 10).

B - Incorrect: Top-down is labeled predictive, not SR (p. 10).

C - Incorrect: Both collapses two categories the module distinguishes (p. 10).

D - Incorrect: The bottom-up category is present (p. 10), so none is false.

Primary note proof

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse. A handle is a production RHS occurrence whose reduction is the next step of that reverse derivation. Not every substring matching a RHS is a handle in the current context.

Study-note section COMP15

> Shift pushes the next input token and a state; reduce replaces a recognized RHS by its LHS and consults GOTO; accept recognizes completion; error reports an invalid situation. For A→XYZ, pop three grammar symbols and their states, expose state q, then push A and GOTO[q,A]. “Go” is not a separate standard shift-reduce ACTION, although GOTO is essential table machinery.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

> Parsing
> • Process of constructing parse tree for a sentence generated by a given grammar.
> • 2 types of parsers
> ● Top down parsing (predictive parsers)
> • LL(1)
> ● Bottom up parsing (SR parsers)
> • LR(1)

Compiler Design module | PDF page 10 | lines 18-23

> Top Down Parsing
> • The parse tree is created top to bottom

• Starts from the start symbol and transform it to the input
Bottom Up Parsing
• Starts with the input symbols and tries to construct the parse tree up to the start symbol.
• One way of reducing a sentence is to follow the right most derivation in reverse

Question sources

Compiler Design review questions · Question 39

Question 63 | Q01433

The full form of YACC is:

A. Yet Another Computer Computer

B. Yet Another Computer Compiler

C. Yet Another Compiler Computer

D. Yet Another Compiler Compiler

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Yet Another Compiler Compiler.

COMP39 expands Yacc as Yet Another Compiler Compiler and describes Yacc/Bison as generating parsers from grammar rules and semantic actions. The other expansions interchange computer and compiler and do not match the note. The name refers to a parser generator, distinct from Lex/flex scanner generation.

Why each choice is right or wrong

A - Incorrect: Computer Computer replaces both required occurrences of Compiler and is not the stated expansion.

B - Incorrect: The first noun must be Compiler, not Computer.

C - Incorrect: The final noun must also be Compiler, not Computer.

D - Correct: This matches the exact Yacc expansion given in COMP39.

Primary note proof

Study-note section COMP39

Panic-mode recovery skips input until a synchronizing token such as ; or } permits parsing to resume. Phrase-level recovery makes a local insertion/deletion. Report the location and cause while avoiding cascades of misleading diagnostics. Lexical, syntax and semantic phases each detect their own errors; error handling is a shared service.

Study-note section COMP39

Lex/flex generates scanners from patterns and actions. Yacc means Yet Another Compiler Compiler; Yacc/Bison generates parsers from grammar rules and semantic actions. Typical specification files have definitions, %% rules, and %% user code. Classic Yacc uses LALR; Bison supports additional LR and generalized modes. Scanner generation does not replace the CFG parser.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Question sources

Compiler Design review questions · Question 41

Question 64 | Q01434

What type of conflicts can occur in the shift-reduce parsing? Select the most complete choice.

A. reduce/reduce

B. shift/reduce

C. Both shift/reduce and reduce/reduce

D. None of the above

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer C: Both shift/reduce and reduce/reduce

COMP17 distinguishes a shift/reduce conflict, where shifting competes with reduction, from reduce/reduce, where two reductions compete. Both can arise in shift-reduce table construction. The combined option therefore answers the question completely; either individual conflict is a valid example but omits the other named kind.

Why each choice is right or wrong

A - Incorrect: True as one component, but incomplete as the requested full selection. Two competing reductions form a reduce/reduce conflict, one of the listed possible types.

B - Incorrect: True as one component, but incomplete as the requested full selection. A shift competing with a reduction forms the other standard conflict type.

C - Correct: This aggregate includes both conflict types explicitly defined in COMP17.

D - Incorrect: Both listed kinds can occur, so denying every listed type is false.

Primary note proof

Study-note section COMP17

> SLR(1) uses LR(0) states and FOLLOW(A) for reduction by a completed A production. Canonical LR(1) attaches context-specific lookahead to items. LALR(1) merges canonical states with the same LR(0) core, combining lookaheads; merging can introduce reduce/reduce conflicts. The standard grammar-class hierarchy is LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1), with increasing capability and often table size.

Study-note section COMP17

> A shift/reduce conflict offers both consuming a token and reducing; reduce/reduce offers two reductions. An SLR or LALR conflict does not by itself prove the grammar ambiguous. Canonical LR(1) handles LR(1) grammars, not every unambiguous CFG. Precedence and associativity can resolve suitable expression conflicts; changing parser names alone does not repair all ambiguity.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

> Parsing
> • Process of constructing parse tree for a sentence generated by a given grammar.
> • 2 types of parsers
> ● Top down parsing (predictive parsers)
> • LL(1)
> ● Bottom up parsing (SR parsers)
> • LR(1)

Compiler Design module | PDF page 11 | lines 2-17

> LL(1) Grammar
> • L - left to right
> • L - left most derivation
> • 1 - number of look ahead
> • First( ) and Follow( )
> ● the first terminal in a string and the terminal that follows a variable respectively.
> LR Parsing
> • LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
> Types of LR Parsers

• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Compiler Design review questions · Question 42

Question 65 | Q01436

Which algorithm invokes a function GETREG()?

A. Code motion algorithm

B. Code optimization algorithm

C. Intermediate Code

D. Code generation algorithm

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer D: A code-generation algorithm.

COMP35 directly associates getReg with a simple code-generation algorithm. It selects operand/result registers for a TAC instruction before loads and the target operation are emitted and descriptors are updated. Code motion and general optimization serve other transformation purposes; intermediate code is the input representation here rather than the algorithm making the register choice.

Why each choice is right or wrong

A - Incorrect: Code motion relocates computations; it is not the described getReg-driven emission algorithm.

B - Incorrect: Optimization is broader, but COMP35 specifically locates getReg in the target code-generation algorithm.

C - Incorrect: Intermediate code supplies instructions to the generator; it is not the algorithm invoking the register selector.

D - Correct: The note explicitly describes getReg as part of simple target code generation.

Primary note proof

Study-note section COMP35

A register descriptor records which current values are held in a register. An address descriptor for a name records all current locations containing its value, possibly memory and multiple registers. A memory copy can become stale after a register-only assignment. These are code-generator bookkeeping structures, not source-language types.

Study-note section COMP35

A simple code-generation algorithm calls getReg for a TAC instruction to select operand/result registers, emits necessary loads, emits the operation, then updates descriptors. Prefer an already suitable register or a reusable dead operand; otherwise choose a spill victim and store any live value lacking another current copy. At a block exit, preserve values needed by successors.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Compiler Design module | PDF page 18 | lines 11-22

Register Allocation
• Registers hold values
• Example
- a = c + d
- e = a + b
- f = e - 1

• With the assumption that a and e die after use
• Temporary a can be reused after e=a+b, same wz a
• Can allocate a,e and f all to one register(r1)
- r1 = r2 + r3
- r1 = r1 + r4
- r1 = r1 - 1

Question sources

Compiler Design review questions · Question 44

Question 66 | Q01438

The execution time of the code depends on? Select the most complete choice.

A. the usage of machine idioms

B. the way the registers are used

C. the orders in which the computations are performed

D. All of the mentioned

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: All of the mentioned

COMP34 states that machine idioms, register availability and instruction order affect execution time. These correspond to all three concrete choices. Better register use can reduce loads/spills; target idioms affect instruction cost, and legal scheduling changes execution efficiency. The aggregate is therefore the complete response rather than any one factor in isolation.

Why each choice is right or wrong

A - Incorrect: True as one component, but incomplete as the requested full selection. Choosing target machine idioms changes the operations emitted and their target costs.

B - Incorrect: True as one component, but incomplete as the requested full selection. Register use affects memory traffic and spills, which can change execution time.

C - Incorrect: True as one component, but incomplete as the requested full selection. The scheduling of computations affects execution efficiency while dependencies constrain legal orders.

D - Correct: COMP34 explicitly includes all three factors, so the aggregate is the complete answer.

Primary note proof

Study-note section COMP34

> Code generation lowers checked IR to a target form. Instruction selection chooses target operations; register allocation chooses where values reside; scheduling orders instructions while respecting dependencies; storage layout assigns offsets and follows ABI requirements. Register assignment names actual physical registers after allocation decisions.

Study-note section COMP34

> For t=a+b, a simple target could emit LOAD R1,a; LOAD R2,b; ADD R3,R1,R2; STORE t,R3. Other targets may combine loads with arithmetic or keep t in a register. Machine idioms, register availability and instruction order affect execution time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

> 6. Code Generation
> Converts intermediate code to machine code.
> Must handle all aspects of machine architecture
> Storage allocation decisions are made
> a. Register allocation and assignment

Compiler Design module | PDF page 18 | lines 11-22

Register Allocation
• Registers hold values
• Example
- a = c + d
- e = a + b
- f = e - 1

• With the assumption that a and e die after use
• Temporary a can be reused after e=a+b, same wz a
• Can allocate a,e and f all to one register(r1)
- r1 = r2 + r3
- r1 = r1 + r4
- r1 = r1 - 1

Compiler Design module | PDF page 17 | lines 16-22

Chapter 8
Introduction to Code Optimization
Goals of Code Optimization
• Remove redundant code without changing the meaning of program
• Executes faster
• Efficient memory usage
• Better performance

Question sources

Compiler Design review questions · Question 46

Question 67 | Q01439

Which is the most specific named method for examining a short instruction window and replacing jump L1 with jump L2 when L1 immediately jumps to L2?

A. Latter optimization technique

B. Peephole optimization technique

C. Local optimization technique

D. Code optimization technique

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B: Peephole optimization technique

Peephole optimization examines a short instruction window and replaces it with an equivalent, preferable sequence. The described jump-chain rewrite is a direct example. Local optimization and code optimization are broader classifications that may include the transformation, but do not name the specific short-window method requested. Latter optimization is not a method defined in the supplied note.

Why each choice is right or wrong

A - Incorrect: Latter optimization does not name the specified method.

B - Correct: Peephole explicitly performs short-window jump-chain rewrites.

C - Incorrect: Local optimization is broader than the requested method.

D - Incorrect: Code optimization is broader than the requested method.

Primary note proof

Study-note section COMP31

> Peephole optimization examines short instruction windows. It can remove redundant loads/stores, replace machine idioms and shorten jump chains: jump L1 where L1 immediately jumps L2 can become jump L2.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 23-34

> Peephole Optimization
> • Transforming to optimal sequence of instructions
> Common Techniques:
> • Elimination of redundant loads and stores
> - Eg.
> - r2 = r1 + 5
> - I = r2
> - r3 = I
> - r4 = r3 * 3
>
> • Constant folding
> - Eg.
> - R2 = 3 * 2

Compiler Design module | PDF page 18 | lines 2-10

> Techniques
> • Common sub-expression elimination
> - Repeated appearance computed previously
>
> • Strength reduction
> - Replacement of expensive expressions with simple ones
>
> • Code movement
> - Moving a block of code outside a loop
>
> • Dead code elimination
> - Eliminated code statements that are either never executed or unreachable

Question sources

Compiler Design review questions · Question 47

Question 68 | Q01440

Which of the following term is used to keep track of the location where the current values of the name are stored?

A. Register descriptor

B. Address descriptor

C. Allocation descriptor

D. Flag register

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer B: An address descriptor.

COMP35 defines an address descriptor for a name as recording all current locations that contain its value, including memory and registers. That matches tracking where a name's current value is stored. A register descriptor instead records values held in a particular register, reversing the direction of the association.

Why each choice is right or wrong

A - Incorrect: A register descriptor maps a register to its current values rather than a name to all locations.

B - Correct: An address descriptor records the current memory and register locations holding a name's value.

C - Incorrect: Allocation descriptor is not the note's defined bookkeeping structure for name-to-location tracking.

D - Incorrect: A flag register stores machine status flags rather than all locations of a named value.

Primary note proof

Study-note section COMP35

> A register descriptor records which current values are held in a register. An address descriptor for a name records all current locations containing its value, possibly memory and multiple registers. A memory copy can become stale after a register-only assignment.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 11-22

> Register Allocation
> • Registers hold values
> • Example
> - a = c + d
> - e = a + b

- f = e - 1
• With the assumption that a and e die after use
• Temporary a can be reused after e=a+b, same wz a
• Can allocate a,e and f all to one register(r1)
- r1 = r2 + r3
- r1 = r1 + r4
- r1 = r1 - 1

Question sources

Compiler Design review questions · Question 48

Question 69 | Q01441

In a nontrivial expression AST whose leaves are operands, which complete category of nodes carries operator symbols, including operators in nested subexpressions?

A. Root

B. Interior nodes

C. Leaves

D. Nodes

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Interior nodes

In an expression AST, leaves carry operands such as identifiers and constants, while nonleaf nodes carry the operators that combine child expressions. A nontrivial tree’s root can also be an operator, but naming only the root omits nested operators. Naming all nodes includes operand leaves. The complete category requested is therefore the interior nodes, which includes an operator root and any other nonleaf operator nodes.

Why each choice is right or wrong

A - Incorrect: The root alone omits operators in nested subexpressions.

B - Correct: Interior nodes are the complete nonleaf category carrying operators in this expression AST.

C - Incorrect: Leaves carry operands rather than combining operators.

D - Incorrect: All nodes includes operand leaves and is too broad.

Primary note proof

Study-note section COMP09

A context-free grammar G=(N,T,P,S) has nonterminals N, terminals T, productions P and start S. A production A→α rewrites one nonterminal. A sentential form is any string of terminals/nonterminals derivable from S; a sentence contains only terminals. A leftmost derivation rewrites the leftmost remaining nonterminal; a rightmost one rewrites the rightmost.

Study-note section COMP09

A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 26-28

Functions for Syntax Tree Nodes
• mknode (op, left, right)
- Creates an operator node with label op &

Compiler Design module | PDF page 13 | lines 2-14

- Two fields containing pointers to left and right
• mkleaf(id, entry)
- Creates an identifier node with label id &
- A field containing entry, ptr to symbol table entry for the identifier
• mkleaf(num, val)
- Create a number node with label num &
- A field containing val, the value for the number

Syntax tree for expression a-4+c
• P1=mkleaf(id,entrya);
• P2=mkleaf(num, 4);
• P3=mknode('-',p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode('+',p3,p4);

Question sources

Compiler Design review questions · Question 49

Question 70 | Q01443

Which of the following is used in various stages or phases of the compiler?

A. Records

B. Program

C. Symbol Table

D. Table

Hint

Ask whether the required information is a declaration's identity and attributes, program structure or the ordered source tokens.

Answer C: Symbol Table

The module stores semantic type information in the symbol table (PDF p. 6), puts identifier-entry pointers in syntax-tree leaves (p. 13), and enters names/types during declarations (p. 17). These uses support the symbol table as a shared structure across compiler work.

Why each choice is right or wrong

A - Incorrect: Records is only a generic storage term, not the shared named structure in these passages.

B - Incorrect: Program is the object being compiled, not the metadata table.

C - Correct: Symbol Table is explicitly used in several contexts (pp. 6, 13, 17).

D - Incorrect: Table is nonspecific; Symbol Table is the documented structure.

Primary note proof

Study-note section COMP19

> A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage.

Study-note section COMP03

> Symbol-table management and error reporting support multiple phases rather than forming one mandatory seventh final phase.

Study-note section COMP19

> Scanning may intern a spelling first; semantic analysis determines the declaration and type.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

> 3. Semantic Analysis
> • Certain checks are performed to make sure that the components of the program fit together meaningfully.
> • Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
> - Type checking of various programming language constructs is one of the most important tasks.
> • Stores type information in the symbol table or the syntax tree.
> - Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 13 | lines 2-14

- Two fields containing pointers to left and right
• mkleaf(id, entry)
- Creates an identifier node with label id &
- A field containing entry, ptr to symbol table entry for the identifier
• mkleaf(num, val)
- Create a number node with label num &
- A field containing val, the value for the number

Syntax tree for expression a-4+c
• P1=mkleaf(id,entrya);
• P2=mkleaf(num, 4);
• P3=mknode('-',p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode('+',p3,p4);

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Question sources

Compiler Design review questions · Question 51

Question 71 | Q01444

Which of the following structure has four fields?

A. Parse tree

B. Triples

C. Indirect Triples

D. Quadruples

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer D: Quadruples

PDF p16 explicitly divides a quadruple into operator, arg1, arg2, and result: four fields. It contrasts triples with three fields.

Why each choice is right or wrong

A - Incorrect: A parse tree is a hierarchical representation, not the four-field instruction defined on p16.

B - Incorrect: P16 explicitly gives triples three fields.

C - Incorrect: P16 describes indirect triples through pointers, not a fourth result field.

D - Correct: P16 explicitly lists the four quadruple fields.

Primary note proof

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 16 | lines 13-23

> Three Address Code
> • Quadruples:
> - Each instruction is divided into four fields
> - Operator, arg1, arg2, and result
> • Triples:
> - Has three fields
> - Operator, arg1 and arg2
> • DAG and Tree
> - Similar presentation of expression to triples
> • Indirect Triples
> - Uses pointers instead of position to store results

Question sources

Compiler Design review questions · Question 52

Question 72 | Q01445

What is the conventional name for the compact expression representation with operand leaves and operator interior nodes?

A. Syntax tree

B. Parser tree

C. Structured tree

D. Sematic tree

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Syntax tree

The conventional expression AST places operands at leaves and operators at interior nodes, removing grammar scaffolding and punctuation. A parse tree instead records grammar productions: its internal nodes are nonterminals, and its yield includes the terminals of the source phrase. The other two names do not identify the conventional expression representation in this comparison. The syntax-tree choice supplies the specific structure described by the question.

Why each choice is right or wrong

A - Correct: An expression syntax tree places operands at leaves and operators at nonleaf nodes.

B - Incorrect: A parse tree's internal labels are grammar nonterminals, not generally expression operators.

C - Incorrect: This generic phrase does not specify the conventional expression AST representation requested.

D - Incorrect: Semantic annotation can be attached to representations, but this is not the conventional name for the described AST.

Primary note proof

Study-note section COMP09

> A context-free grammar G=(N,T,P,S) has nonterminals N, terminals T, productions P and start S. A production A→α rewrites one nonterminal. A sentential form is any string of terminals/nonterminals derivable from S; a sentence contains only terminals. A leftmost derivation rewrites the leftmost remaining nonterminal; a rightmost one rewrites the rightmost.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 26-28

Functions for Syntax Tree Nodes
• mknode (op, left, right)
● Creates an operator node with label op &

Compiler Design module | PDF page 13 | lines 2-14

● Two fields containing pointers to left and right
• mkleaf(id, entry)
● Creates an identifier node with label id &
● A field containing entry, ptr to symbol table entry for the identifier
• mkleaf(num, val)
● Create a number node with label num &
● A field containing val, the value for the number
Syntax tree for expression a-4+c
• P1=mkleaf(id,entrya);
• P2=mkleaf(num, 4);
• P3=mknode('-',p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode('+',p3,p4);

Compiler Design module | PDF page 9 | lines 22-27

Derivation Trees
• Derivations can be displayed as trees
• Internal nodes of the tree are all non-terminals
• Leaves are all terminals
• The yield of a derivation tree is the list of the labels of all the leaves read from left to right.

Question sources

Compiler Design review questions · Question 53

Question 73 | Q01446

Which statement is an abstract form of intermediate code?

A. 3- address

B. 2-address

C. address

D. Intermediate code

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer A: 3- address

PDF pp15-16 defines three-address code as a generic intermediate representation, X = Y op Z. Thus A names the supplied abstract statement form; D only repeats the general category.

Why each choice is right or wrong

A - Correct: P15 gives precisely this statement form.

B - Incorrect: No two-address intermediate statement form is supplied; it is not the form defined on p15.

C - Incorrect: A bare address is not the statement X = Y op Z.

D - Incorrect: This names the category rather than the particular statement form requested.

Primary note proof

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 15 | lines 13-21

> Three Address Code
> • Is a sequence of statements of the form
> - X = Y op Z
> - X,Y and Z are names, constants or compiler generated temporaries
> - Op is operator (arithmetic, logical)
>
> • Example:
> - a = b + c , x = -y , if a > b goto L1
>
> • LHS is the target
> • RHS has at most two sources and one operator

Compiler Design module | PDF page 16 | lines 2-8

> Three Address Code
> • Is a generic form and can be implemented as:

- Quadruples
- Triples
- Indirect Triples
- Tree
- DAG

Question sources

Compiler Design review questions · Question 54

Question 74 | Q01447

Which mapping is described by the implementation of the syntax-directed translator?

A. Parse table

B. Input

C. Output

D. Input-Output

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer D: An input-to-output mapping.

COMP25 distinguishes translation from recognition: semantic actions compute an output, such as values, ASTs or code, from an input program. Accordingly the implemented translator defines an input-to-output mapping. A parse table may drive an implementation, but neither the table nor either endpoint alone names that mapping.

Why each choice is right or wrong

A - Incorrect: A parse table can guide recognition; it is not the translation mapping itself.

B - Incorrect: Input is only the domain side and omits the translated result.

C - Incorrect: Output is only the result side and omits the mapping from source inputs.

D - Correct: Syntax-directed translation relates each processed input to its computed translation output.

Primary note proof

Study-note section COMP25

A syntax-directed translation scheme embeds executable actions in productions; the actions run while translating a source program, not while generating the compiler. Attributes may compute types, values, AST pointers or generated code. The scheme defines an input-to-output translation, not merely an input language.

Study-note section COMP25

Create an identifier leaf with mkleaf(id,entry), a number with mkleaf(num,value), and an operator node with mknode(op,left,right). For a−4+c, construct +(−(a,4),c). Postorder traversal prints postfix: a 4 − c +. A stack evaluator pushes operands and pops right then left operands for an operator. Infix-to-postfix is useful for stack machines but not mandatory for all compilers.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 4-16

Syntax Directed Translation
• Attaching actions to the grammar rules(productions).
• Actions are executed during the compilation
- Not during the generation of the compiler
• Actions are executed according to the parsing mechanism.
Syntax Directed Definitions
• Is a generalization of a context free grammar
• Is a CFG with attributes and rules
• Attributes are associated with grammar symbols and rules with productions
• Attributes may be:
- Numbers
- Types
- Strings etc

Compiler Design module | PDF page 12 | lines 17-25

Syntax Directed Definition- Example
• Production Semantic Rules
• L → E return print(E.val)
• E → E1 + T E.val = E1.val + T.val
• E → T E.val = T.val
• T → T1 * F T.val = T1.val * F.val
• T → F T.val = F.val
• T → (E) F.val = E.val
• F → digit F.val = digit.lexval

Question sources

Compiler Design review questions · Question 55

Question 75 | Q01448

Which transition function moves the dot over a grammar symbol and takes closure to build successor states in the canonical LR(0) item collection?

A. FIRST

B. GOTO

C. COMPUTE

D. FOLLOW

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: GOTO

GOTO(I,X) advances the dot over X in eligible items and then takes closure, producing a successor item state. Repeating this over grammar symbols builds the canonical collection. The collection is the resulting set of item sets, not the function itself. FIRST and FOLLOW compute terminal sets for different parsing purposes; COMPUTE does not name the defined LR successor operation.

Why each choice is right or wrong

A - Incorrect: FIRST finds possible beginning terminals.

B - Correct: GOTO produces the successor item state.

C - Incorrect: COMPUTE is not the defined successor-state function.

D - Incorrect: FOLLOW finds possible following terminals.

Primary note proof

Study-note section COMP16

> Augment the grammar with S′→S. An LR(0) item A→α·β records progress through a production. CLOSURE(I) repeatedly adds B→·γ whenever an item has its dot before nonterminal B and B→γ is a production.

Study-note section COMP16

> GOTO(I,X) moves the dot past X in every eligible item and takes closure. Start with CLOSURE({S′→·S}); repeatedly compute nonempty GOTO sets for all grammar symbols until no new state appears. The resulting set of item sets is the canonical collection, not GOTO itself.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 11 | lines 2-17

> LL(1) Grammar
> • L - left to right
> • L - left most derivation

• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Compiler Design review questions · Question 56

Question 76 | Q01449

Which of the following option is not a function of the shift-reduce parser?

A. Reduce

B. Accept

C. Go

D. Shift

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer C: “Go” is not a standard shift-reduce ACTION.

COMP15 lists shift, reduce, accept and error as standard shift-reduce actions. GOTO is essential after a reduction but is separate table machinery, not a distinct action called Go. Reading function as the standard action category yields the offered Go choice; the three other choices are explicitly standard actions.

Why each choice is right or wrong

A - Incorrect: Reduce is a standard action that replaces a recognized right-hand side by its left-hand nonterminal.

B - Incorrect: Accept is a standard action indicating successful completion of recognition.

C - Correct: Go is not one of the standard ACTION names; GOTO is separate transition-table machinery.

D - Incorrect: Shift is a standard action that consumes input and pushes a token with its state.

Primary note proof

Study-note section COMP15

Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse. A handle is a production RHS occurrence whose reduction is the next step of that reverse derivation. Not every substring matching a RHS is a handle in the current context.

Study-note section COMP15

Shift pushes the next input token and a state; reduce replaces a recognized RHS by its LHS and consults GOTO; accept recognizes completion; error reports an invalid situation. For A→XYZ, pop three grammar symbols and their states, expose state q, then push A and GOTO[q,A]. “Go” is not a separate standard shift-reduce ACTION, although GOTO is essential table machinery.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
- Top down parsing (predictive parsers)
• LL(1)
- Bottom up parsing (SR parsers)
• LR(1)

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
- the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Compiler Design review questions · Question 57

Question 77 | Q01450

Which listed grammar classes guarantee no adjacent nonterminals on a production right-hand side? Select all that apply. Regular means consistently right-linear; irregular means arbitrary unrestricted.

A. Arbitrary unrestricted grammar

B. Right-linear regular grammar

C. Operator grammar

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B, C: Right-linear regular grammar; Operator grammar

An operator grammar explicitly forbids adjacent nonterminals on a right-hand side. A consistently right-linear regular grammar has at most one nonterminal in such a rule, so it also guarantees the restriction. Arbitrary unrestricted grammars have no comparable adjacency guarantee. The property is therefore shared by the two specified classes and cannot be used to identify only operator grammar.

Why each choice is right or wrong

A - Incorrect: An unrestricted grammar can contain adjacent nonterminals.

B - Correct: A right-linear regular rule has at most one nonterminal.

C - Correct: Operator grammars forbid adjacent nonterminals by definition.

Primary note proof

Study-note section COMP18

> LL, LR and recursive-descent parsers can all handle nested parentheses for suitable grammars. An explicit or implicit stack, not the LR label alone, supports unbounded nesting. Both LL and LR can use parsing tables. Deterministic LL/LR grammars must be unambiguous; a general ambiguous grammar requires rewriting, disambiguation policy or a generalized method such as GLR.

Study-note section COMP18

> An operator grammar conventionally forbids adjacent nonterminals on a RHS and usually ε productions. Operator-precedence parsing additionally requires conflict-free precedence relations. Ordinary right-linear regular grammars also lack adjacent nonterminals, so that property alone does not uniquely identify the operator-grammar category. Regular patterns are chiefly lexical specifications; CFGs chiefly describe nested syntax.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Question sources

Compiler Design review questions · Question 58

Question 78 | Q01488

In an implementation where the scanner interns identifier spellings in a symbol table before returning their tokens, which phase first interacts with that table?

A. Syntax analysis

B. Parsing

C. Scanning

D. Semantic analysis

Hint

Ask whether the required information is a declaration’s identity and attributes, program structure or the ordered source tokens.

Answer C: Scanning

Interning associates an identifier spelling with a table entry while scanning its characters and returning the identifier token. Under this explicit implementation convention, scanning is the first phase to interact with the table. Parsing and semantic analysis use or enrich this information later; semantic analysis resolves declaration identity and type. This establishes an ordering for the stated implementation rather than a universal compiler requirement.

Why each choice is right or wrong

A - Incorrect: Syntax analysis receives tokens after this scanner has interned identifiers.

B - Incorrect: Parsing is syntax analysis and occurs after the stated interning.

C - Correct: Scanning performs the stipulated initial interning.

D - Incorrect: Semantic analysis enriches name/type information after that initial interaction.

Primary note proof

Study-note section COMP19

Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal.

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Mock exam 1 · Question 4

Question 79 | Q01489

What is the name given to the sequence of alphabets that are identified as a token?

A. Lexeme

B. Prefix

C. Pattern

D. Grammar

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Lexeme

PDF p8 calls the sequence of characters matched by a pattern to form a token a lexeme. Interpreting alphabets as characters makes A exact.

Why each choice is right or wrong

A - Correct: Matches the p8 definition.

B - Incorrect: Prefix is not the defined name for the matched sequence.

C - Incorrect: P8 defines a pattern as a rule, not the matched sequence.

D - Incorrect: P9 uses grammar for syntactic production rules, not a token occurrence.

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 10-18

> Tokens, Patterns, and Lexemes
> • Token
> ● A string of characters which logically belong together
> ● Classes of similar lexemes
> l identifier, keywords, constants etc.
> • Pattern
> ● A rule which describes a token
> • Lexeme
> ● The sequence of characters matched by a pattern to form the token

Question sources

Mock exam 1 · Question 5

Question 80 | Q01490

In a compiler where semantic analysis annotates an existing AST with resolved declarations and checked types, what is its resulting program representation?

A. Language constructs

B. Unannotated concrete syntax tree

C. AST annotated with resolved declarations and checked types

D. Token

Hint

Ask whether the required information is a declaration's identity and attributes, program structure or the ordered source tokens.

Answer C: AST annotated with resolved declarations and checked types

The stated implementation resolves names and checks types on an existing abstract syntax tree, so its result is that AST with semantic annotations. An unannotated syntax tree omits the newly checked information; language constructs is too generic to identify the representation. Tokens are the scanner's earlier output. Other compilers may produce typed IR, but the question specifies the retained-tree implementation.

Why each choice is right or wrong

A - Incorrect: Language constructs does not identify the specified annotated representation.

B - Incorrect: An unannotated concrete syntax tree is not the specified result.

C - Correct: The result is the AST enriched with checked semantic information.

D - Incorrect: Tokens precede syntax and semantic analysis.

Primary note proof

Study-note section COMP21

> Static checks occur before execution: type compatibility, duplicate declarations within a restricted scope, valid names, argument counts, return compatibility and control-flow placement. A break is valid only in constructs allowed by the language; in C/C++ it can exit a loop or switch. A compiler may annotate an AST or create typed IR; there is no mandatory uniquely named output tree.

Study-note section COMP21

> Dynamic checks depend on runtime values, such as some bounds and null checks. Static analysis can prove some accesses safe or faulty; "all memory errors cannot be statically checked" is too strong. Unreachable code is not universally a semantic error: some languages reject it in specified cases, others merely warn or optimize it away.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

Mock exam 1 · Question 6

Question 81 | SM0026

For const int x=8; const int y=16; if(x>y){int z=x+y;}, which techniques directly apply to evaluating the condition from known constants and removing its unreachable branch? Select all that apply. Distinguish constant evaluation from algebraic identity simplification.

A. Arithmetic simplification

B. Dead code elimination

C. Code motion

D. Compile time evaluation

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B, D: Dead code elimination; Compile time evaluation

With x=8 and y=16, the condition x>y evaluates to false during compilation. The branch body is consequently unreachable and can be removed. These are compile-time evaluation and dead-code elimination, respectively. No computation needs to move to another program location, and the decisive evaluation is not an algebraic identity such as x+0→x. Both direct transformations belong in the answer.

Why each choice is right or wrong

A - Incorrect: No algebraic identity rewrite is the decisive step.

B - Correct: The constant-false branch is unreachable and removable.

C - Incorrect: Code motion does not describe evaluating or deleting the branch.

D - Correct: Known constants permit compile-time evaluation of the condition.

Primary note proof

Study-note section COMP30

> For const int x=8,y=16; if(x>y){int z=x+y;}, propagate the constants, fold 8>16 to false, then remove the unreachable branch. Both compile-time evaluation and dead-code elimination apply.

Study-note section COMP31

> Algebraic simplification removes identities when valid: integer x+0→x.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 2-10

> Techniques
> • Common sub-expression elimination
> ● Repeated appearance computed previously
> • Strength reduction
> ● Replacement of expensive expressions with simple ones
> • Code movement
> ● Moving a block of code outside a loop
> • Dead code elimination
> ● Eliminated code statements that are either never executed or unreachable

Compiler Design module | PDF page 18 | lines 32-41

> • Constant folding
> ● Eg.
> ● R2 = 3 * 2
> • Constant Propagation
> ● Eg.
> ● r1 = 3
> ● r2 = r1 * 2
> • Copy Propagation
> ● Eg.
> ● r2 = r1

Question sources

MOE Computer Science Exit Exam · Question 8-1

National Computer Science Exit Exam · Question 5

National Computer Science Exit Exam · Question 6

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 263

Question 82 | SM0041

Which one of the following phases is done BEFORE the semantic analysis phase of the compiling process?

A. Syntax analysis

B. Intermediate code generation

C. Code generation

D. Code optimization

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Syntax analysis

PDF pp5-7 orders syntax analysis before semantic analysis, followed by intermediate generation, optimization, and code generation.

Why each choice is right or wrong

A - Correct: Syntax analysis appears on p5 before semantics on p6.

B - Incorrect: Intermediate generation follows semantic analysis on p6.

C - Incorrect: Code generation is phase 6 on p7.

D - Incorrect: Optimization is phase 5 on p7, after semantics.

Primary note proof

Study-note section COMP03

> The conventional sequence is lexical analysis → syntax analysis → semantic analysis → intermediate-code generation → optimization → target-code generation. Scanning turns characters into tokens; parsing builds structure; semantics checks declarations and types; synthesis produces implementation code. A phase is a logical activity; a pass is a traversal that can combine activities.

Study-note section COMP03

> The front end handles source-language rules; a middle end analyzes and transforms IR; the backend handles target details. Symbol-table management and error reporting support multiple phases rather than forming one mandatory seventh final phase. Optimization can occur at several levels, including after instruction selection. “Final phase” means code generation in the standard six-phase teaching pipeline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 6 | lines 11-12

4. Intermediate Code Generation
Easy to produce and easy to translate to machine code

Compiler Design module | PDF page 7 | lines 2-6

5. Code Optimization
Changes the IC by removing such inefficiencies
Improve the code
a. Improvement may be time, space, or power consumption.
It changes the structure of programs,

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Question sources

MOE Computer Science Exit Exam · Question 12-3

National Computer Science Exit Exam · Question 87

National Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 264

Question 83 | SM0052

Which phase of the compiling process is considered the FINAL phase?

A. Intermediate code generation

B. Semantic analysis

C. Code optimization

D. Code generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Code generation in the conventional pipeline.

COMP03 calls code generation the final phase of the standard six-phase teaching pipeline. Intermediate generation, semantic analysis and optimization precede it there. Some implementations perform later target optimization, but that does not change the conventional phase answer requested here; the note explicitly records that distinction.

Why each choice is right or wrong

A - Incorrect: Intermediate-code generation precedes optimization and target-code generation in the conventional sequence.

B - Incorrect: Semantic analysis checks names and types before intermediate and target synthesis.

C - Incorrect: Optimization precedes final code generation in the simplified six-phase teaching pipeline.

D - Correct: COMP03 explicitly names code generation as final under this standard pipeline convention.

Primary note proof

Study-note section COMP03

> The conventional sequence is lexical analysis → syntax analysis → semantic analysis → intermediate-code generation → optimization → target-code generation. Scanning turns characters into tokens; parsing builds structure; semantics checks declarations and types; synthesis produces implementation code. A phase is a logical activity; a pass is a traversal that can combine activities.

Study-note section COMP03

> The front end handles source-language rules; a middle end analyzes and transforms IR; the backend handles target details. Symbol-table management and error reporting support multiple phases rather than forming one mandatory seventh final phase. Optimization can occur at

several levels, including after instruction selection. “Final phase” means code generation in the standard six-phase teaching pipeline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Compiler Design module | PDF page 7 | lines 2-6

5. Code Optimization
Changes the IC by removing such inefficiencies
Improve the code
a. Improvement may be time, space, or power consumption.
It changes the structure of programs,

Question sources

MOE Computer Science Exit Exam · Question 15-3

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 265

Question 84 | SM0093

Which one of the following is used for the purpose of syntax analysis?

A. Loader

B. Linker

C. Parser

D. Scanner

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: Parser

PDF p10 defines parsing as constructing a parse tree, and p5 labels syntax analysis Parsing. P3 assigns distinct object-code roles to loader and linker.

Why each choice is right or wrong

A - Incorrect: P3 loads an executable into memory.

B - Incorrect: P3 merges object files.

C - Correct: Parsing is syntax analysis (p5).

D - Incorrect: Scanning produces tokens (p8), prior to parsing.

Primary note proof

Study-note section COMP02

A preprocessor handles directives such as file inclusion and macros. An assembler translates assembly to object code. A linker combines object files and libraries, resolves external symbols and performs relocation. A loader maps an executable into memory and starts it.

Study-note section COMP03

Scanning turns characters into tokens; parsing builds structure; semantics checks declarations and types; synthesis produces implementation code.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
● Top down parsing (predictive parsers)
• LL(1)
● Bottom up parsing (SR parsers)
• LR(1)

Compiler Design module | PDF page 3 | lines 9-16

• Assembler
● assembly language into machine code
● output of an assembler is called an object file
• Linker
● links and merges various object files to make an executable file.
● determine the memory location where these codes will be loaded
• Loader
● loading executable files into memory and execute them.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Question sources

MOE Computer Science Exit Exam · Question 26-3

Computer Science Exit Exam practice collection · Question 266

Question 85 | SM0216

Which output does syntax analysis produce when it retains every grammar production and terminal occurrence, including punctuation, without semantic attribute annotation?

A. Concrete syntax tree (parse tree)

B. Annotated parse tree

C. Abstract syntax tree

D. Pattern

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer A: Concrete syntax tree (parse tree)

A concrete syntax or parse tree records the grammar productions used, with nonterminal interiors and terminal leaves. Because the specified representation retains grammar scaffolding and punctuation, it is not the abstract syntax tree that omits those details. Semantic attribute values would make an annotated tree, which this question excludes. A lexical pattern is a specification for tokens rather than the parsed program representation.

Why each choice is right or wrong

A - Correct: A full syntax/parse tree records the retained grammar detail.

B - Incorrect: Semantic attribute annotation is expressly excluded.

C - Incorrect: An AST omits the scaffolding and punctuation retained here.

D - Incorrect: A pattern specifies lexical membership rather than parsed program structure.

Primary note proof

Study-note section COMP24

A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an inherited or synthesized attribute.

Study-note section COMP09

A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 9 | lines 22-27

Derivation Trees
• Derivations can be displayed as trees
• Internal nodes of the tree are all non-terminals
• Leaves are all terminals
• The yield of a derivation tree is the list of the labels of all the leaves read from left to right.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 5-3

Computer Science Exit Exam practice collection · Question 270

Question 86 | SM0219

For a fixed suitable grammar, which listed approaches can be efficient top-down parsers using prediction without backtracking? Select all that apply.

A. Recursive-descent parser

B. Shift-reduce parser

C. Non-recursive predictive parser

D. LR parser

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A, C: Recursive-descent parser; Non-recursive predictive parser

Recursive descent can use lookahead to predict productions without backtracking. Nonrecursive predictive parsing implements the same top-down strategy with an explicit stack and table. Both can run linearly for a fixed suitable grammar. Shift-reduce and LR instead work bottom-up, reducing input toward the start symbol, so they do not meet the specified parsing direction even if they are efficient.

Why each choice is right or wrong

A - Correct: Predictive recursive descent is a supported efficient top-down implementation.

B - Incorrect: Shift-reduce is bottom-up.

C - Correct: Nonrecursive prediction is a supported efficient top-down implementation.

D - Incorrect: LR is bottom-up.

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP11

> A nonrecursive predictive parser stores grammar symbols on an explicit stack and consults a table. It is another implementation of predictive parsing, not a wholly different language class. Table-driven and recursive predictive implementations can both be linear for a fixed suitable grammar. Efficiency cannot be ranked from their names alone.

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 18-23

> Top Down Parsing
> • The parse tree is created top to bottom
> • Starts from the start symbol and transform it to the input
> Bottom Up Parsing
> • Starts with the input symbols and tries to construct the parse tree up to the start symbol.
> • One way of reducing a sentence is to follow the right most derivation in reverse

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science Exit Exam · Sene 2016 · Question 6-3

Question 87 | SM0224

Which abstract data structure naturally represents hierarchical source-program constructs with parent/child relationships, as in a parse tree or AST?

A. 2D-array

B. Stack

C. Tree

D. Queue

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: Tree

A tree represents the hierarchy of program constructs through parent/child relationships. Parse trees preserve grammar detail, while ASTs retain the essential expression and statement structure. Arrays may implement node storage, and stacks or queues may support traversal, but those container roles do not make them the requested hierarchical model. The structure being represented, rather than the storage allocation technique, determines this answer.

Why each choice is right or wrong

A - Incorrect: A 2D array is not the requested abstract hierarchical model.

B - Incorrect: A stack supports processing order rather than the full parent/child model.

C - Correct: A tree directly models hierarchical constructs.

D - Incorrect: A queue supports work ordering rather than the hierarchy itself.

Primary note proof

Study-note section COMP41

> Grouping characters into token classes is lexical analysis; grouping tokens into expressions and statements is syntax analysis.

Study-note section COMP41

> Trees store hierarchical constructs, while a symbol table stores declaration metadata.

Study-note section COMP09

> A context-free grammar G=(N,T,P,S) has nonterminals N, terminals T, productions P and start S. A production A→α rewrites one nonterminal. A sentential form is any string of terminals/nonterminals derivable from S; a sentence contains only terminals. A leftmost derivation rewrites the leftmost remaining nonterminal; a rightmost one rewrites the rightmost.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

> 3. Semantic Analysis
> • Certain checks are performed to make sure that the components of the program fit together meaningfully.
> • Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
> - Type checking of various programming language constructs is one of the most important tasks.
> • Stores type information in the symbol table or the syntax tree.
> - Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 13 | lines 2-14

> ● Two fields containing pointers to left and right
> • mkleaf(id, entry)
> ● Creates an identifier node with label id &
> ● A field containing entry, ptr to symbol table entry for the identifier
> • mkleaf(num, val)
> ● Create a number node with label num &
> ● A field containing val, the value for the number
> Syntax tree for expression a-4+c

• P1=mkleaf(id,entrya);
• P2=mkleaf(num, 4);
• P3=mknode('-',p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode('+',p3,p4);

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Question sources

Computer Science Exit Exam · Sene 2016 · Question 8-2

Computer Science Exit Exam practice collection · Question 142

Question 88 | SM0238

In which phase of the compilation process does the source program get broken into its constituent parts?

A. Semantic Analysis

B. Syntax Analysis

C. Synthesis

D. Lexical Analysis

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Lexical Analysis

PDF p5 scans characters into lexemes represented as tokens. These constituent lexical units precede hierarchical grouping during parsing.

Why each choice is right or wrong

A - Incorrect: P6 checks meaningful compatibility after lexical decomposition.

B - Incorrect: Parsing groups already recognized tokens (p5).

C - Incorrect: P3 calls synthesis machine dependent; it is not the scanning stage.

D - Correct: P5 decomposes the character stream into tokens.

Primary note proof

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 12-3

Computer Science Exit Exam practice collection · Question 269

Question 89 | SM0265

Which described check requires runtime values rather than being solely a static check? In D, the index depends on input and no static proof of its bounds is available.

A. Type checking for compatibility

B. Uniqueness checks like redefined variables

C. Checking that break is inside an allowed loop or switch

D. Checking an input-dependent array index against runtime bounds, with no static proof available

Hint

Ask whether the required information is a declaration's identity and attributes, program structure or the ordered source tokens.

Answer D: Checking an input-dependent array index against runtime bounds, with no static proof available

Type compatibility, duplicate declarations and valid placement of break can be checked from the source program and its declarations. The specified array index depends on runtime input, and no static bounds proof is available, so its validity requires a runtime check. This does not claim that every memory access fault is inherently undetectable statically; the explicit value-dependent case supplies the distinction.

Why each choice is right or wrong

A - Incorrect: Type compatibility is a static semantic check.

B - Incorrect: Duplicate declarations can be checked statically within the relevant scope.

C - Incorrect: Placement in an allowed loop or switch is a static check.

D - Correct: The stipulated input-dependent bounds need runtime validation.

Primary note proof

Study-note section COMP21

> Static checks occur before execution: type compatibility, duplicate declarations within a restricted scope, valid names, argument counts, return compatibility and control-flow placement. A break is valid only in constructs allowed by the language; in C/C++ it can exit a loop or switch. A compiler may annotate an AST or create typed IR; there is no mandatory uniquely named output tree.

Study-note section COMP21

> Dynamic checks depend on runtime values, such as some bounds and null checks. Static analysis can prove some accesses safe or faulty; "all memory errors cannot be statically checked" is too

strong. Unreachable code is not universally a semantic error: some languages reject it in specified cases, others merely warn or optimize it away.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 16-23

Static Checking
• The compiler must check the semantic conventions of the source language
• Static Checking: ensures that certain kind of errors are detected and reported
• Example
• Type Checks: incompatible operands
• Flow Control Check
• Uniqueness Check
• Name Related Check

Compiler Design module | PDF page 14 | lines 2-15

• Type Errors:
- Improper or inconsistent operations during program execution
• Type-safety:
- Absence of type errors

Type Checking
• Semantic checks to enforce the type safety of the program
• Semantic Checks
- Static - done during compilation
- Dynamic - done during run-time
• Examples
- Unary and binary operators
- Number and type of arguments
- Return statement with return type
- Compatible assignment

Question sources

Computer Science Exit Exam · Sene 2016 · Question 21-2

Computer Science Exit Exam practice collection · Question 271

Question 90 | SM0285

Which tool is used to collect codes separately compiled or assembled in different object files into a file that is directly executable?

A. Linker

B. Loader

C. Assembler

D. Preprocessor

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Linker

A linker combines separately compiled object files and required libraries, resolves their external symbol references and relocates addresses to produce the executable form. An assembler translates assembly into object code before this combination. A loader places the resulting executable in memory and starts it. A preprocessor handles source directives before compilation. These tool roles distinguish object-file combination from translation, source expansion and program loading.

Why each choice is right or wrong

A - Correct: The linker combines object files, resolves symbols and performs relocation.

B - Incorrect: The loader maps an already produced executable into memory.

C - Incorrect: An assembler translates assembly source into object code.

D - Incorrect: A preprocessor handles source directives rather than combining object files.

Primary note proof

Study-note section COMP02

> A preprocessor handles directives such as file inclusion and macros. An assembler translates assembly to object code. A linker combines object files and libraries, resolves external symbols and performs relocation. A loader maps an executable into memory and starts it. Compilation alone does not necessarily produce a ready executable.

Study-note section COMP02

> A cross-compiler runs on a host platform and generates code for a different target. Multiple targets require appropriate backends, not a special definition of cross-compilation. A retargetable compiler supports changing target descriptions. Incremental compilation recompiles affected fragments or dependent units after edits; it is more than literally replacing changed lines. One-pass versus multipass counts traversals, not supported targets.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 3 | lines 9-16

> • Assembler
> - assembly language into machine code
> - output of an assembler is called an object file
> • Linker
> - links and merges various object files to make an executable file.
> - determine the memory location where these codes will be loaded
> • Loader
> - loading executable files into memory and execute them.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 28-1

Computer Science Exit Exam practice collection · Question 272

Question 91 | SS059

Among phases of compiling process, in which phase a sequence of characters are converted into a tokens?

A. Code optimization

B. Syntax analysis

C. Semantic analysis

D. Lexical analysis

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Lexical analysis

PDF pp5,8 gives lexical analysis a character-stream input and a sequence-of-tokens output.

Why each choice is right or wrong

A - Incorrect: Optimization changes intermediate code (p7).

B - Incorrect: Syntax analysis groups tokens already recognized (p5).

C - Incorrect: Semantic analysis checks meaningful fit (p6).

D - Correct: The exact input-output description is on p8.

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:

<token-name, attribute-value>
- Token

• smallest meaningful element that a compiler understands.
- Eg.

• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Question sources

National Computer Science Exit Exam · Question 61

National Computer Science Exit Exam

Question 92 | AM0779

Which phase of the compiler is responsible for recognizing tokens and building a symbol table?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Ask whether the required information is a declaration’s identity and attributes, program structure or the ordered source tokens.

Answer A: Lexical Analysis

Token recognition uniquely identifies lexical analysis on p5. However, p6 and p17 assign type/name entries to semantic and declaration processing, so building the symbol table must mean contributing lexical identifier entries, not constructing every attribute exclusively.

Why each choice is right or wrong

A - Correct: P5 explicitly recognizes tokens; A is supported for that decisive responsibility.

B - Incorrect: Parsing groups tokens rather than recognizing the character stream (p5).

C - Incorrect: P6 stores types in the symbol table but does not perform lexical token recognition.

D - Incorrect: P7 converts intermediate code into machine code.

Primary note proof

Study-note section COMP19

Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal.

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address

```
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }
```

Question sources

Computer Science MCQ practice collection · Question 779

Question 93 | AM0780

What is the primary ordered output of lexical analysis passed to the parser?

A. Abstract Syntax Tree (AST)

B. Ordered token stream

C. Intermediate Code

D. Assembly Code

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Ordered token stream

Lexical analysis groups characters into lexemes and emits an ordered token stream, including token categories and relevant attributes. The parser consumes that stream to build grammatical structure. A symbol table may be updated as an auxiliary service, but it is not the primary ordered output. ASTs, intermediate code and assembly belong to later processing stages in the conventional pipeline.

Why each choice is right or wrong

A - Incorrect: AST construction follows scanning during parsing or syntax-directed work.

B - Correct: The ordered token stream is the parser's input.

C - Incorrect: Intermediate code follows syntactic and semantic processing.

D - Incorrect: Assembly is a possible later target output.

Primary note proof

Study-note section COMP41

Grouping characters into token classes is lexical analysis; grouping tokens into expressions and statements is syntax analysis.

Study-note section COMP41

Trees store hierarchical constructs, while a symbol table stores declaration metadata.

Study-note section COMP04

A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Question sources

Computer Science MCQ practice collection · Question 780

Computer Science MCQ practice collection · Question 790

Question 94 | AM0781

Which phase of the compiler is responsible for checking the correctness of the program's structure?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Syntax Analysis

PDF p5 groups tokens by grammar and catches most syntactic errors. Program structure here means syntactic structure, so B applies.

Why each choice is right or wrong

A - Incorrect: Recognizes lexical units rather than their grammatical arrangement (p5).

B - Correct: Checks grammar-governed hierarchy and syntactic errors (p5).

C - Incorrect: Checks meaning and type mismatch (p6).

D - Incorrect: Translates intermediate code into machine code (p7).

Primary note proof

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

> 2. Syntax / Hierarchical Analysis - Parsing
> • Tokens are grouped hierarchically into nested collections with collective meaning.
> • The result is generally a parse tree.
> • expressions, statements, declarations etc... are identified by using the results of lexical analysis.
> • Most syntactic errors in the source program are caught in this phase.
> • Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 2-10

> 3. Semantic Analysis
> • Certain checks are performed to make sure that the components of the program fit together meaningfully.
> • Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
> - Type checking of various programming language constructs is one of the most

important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science MCQ practice collection · Question 781

Question 95 | AM0782

Which component of the compiler generates an Abstract Syntax Tree (AST)?

A. Lexical Analyzer

B. Parser

C. Semantic Analyzer

D. Code Generator

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Parser

PDF p5 gives parsing the tree-building role. P12 says syntax-directed actions execute according to parsing, and pp12-13 give operator/operand syntax-tree constructors. Together these support parser as the component building the abstract structural tree.

Why each choice is right or wrong

A - Incorrect: Lexical analysis emits tokens (p8).

B - Correct: Parsing builds a tree (p5), with syntax-directed tree constructors on pp12-13.

C - Incorrect: Semantic analysis may annotate the existing tree (p6); it is not the phase assigned general syntactic tree construction.

D - Incorrect: Code generation translates intermediate code (p7).

Primary note proof

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 12 | lines 4-16

Syntax Directed Translation
• Attaching actions to the grammar rules(productions).
• Actions are executed during the compilation
- Not during the generation of the compiler
• Actions are executed according to the parsing mechanism.
Syntax Directed Definitions
• Is a generalization of a context free grammar
• Is a CFG with attributes and rules
• Attributes are associated with grammar symbols and rules with productions
• Attributes may be:
- Numbers
- Types
- Strings etc

Compiler Design module | PDF page 12 | lines 26-28

Functions for Syntax Tree Nodes
• mknode (op, left, right)
- Creates an operator node with label op &

Compiler Design module | PDF page 13 | lines 2-14

- Two fields containing pointers to left and right
• mkleaf(id, entry)
- Creates an identifier node with label id &
- A field containing entry, ptr to symbol table entry for the identifier
• mkleaf(num, val)
- Create a number node with label num &
- A field containing val, the value for the number
Syntax tree for expression a-4+c
• P1=mkleaf(id,entrya);
• P2=mkleaf(num, 4);
• P3=mknode('-',p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode('+',p3,p4);

Question sources

Computer Science MCQ practice collection · Question 782

Computer Science MCQ practice collection · Question 792

Question 96 | AM0783

What does the semantic analysis phase of the compiler focus on?

A. Identifying syntax errors

B. Analyzing the meaning of program statements

C. Generating intermediate code

D. Translating source code to machine code

Hint

Ask whether the required information is a declaration's identity and attributes, program structure or the ordered source tokens.

Answer B: Analyzing the meaning of program statements

PDF p6 defines semantic analysis through meaningful fit and type-mismatch checks, directly supporting B.

Why each choice is right or wrong

A - Incorrect: Most syntactic errors are assigned to parsing on p5.

B - Correct: Meaningful compatibility is the explicit semantic task (p6).

C - Incorrect: Intermediate generation is separately headed on p6.

D - Incorrect: Machine-code translation is code generation (p7).

Primary note proof

Study-note section COMP21

> Static checks occur before execution: type compatibility, duplicate declarations within a restricted scope, valid names, argument counts, return compatibility and control-flow placement. A break is valid only in constructs allowed by the language; in C/C++ it can exit a loop or switch. A compiler may annotate an AST or create typed IR; there is no mandatory uniquely named output tree.

Study-note section COMP21

> Dynamic checks depend on runtime values, such as some bounds and null checks. Static analysis can prove some accesses safe or faulty; "all memory errors cannot be statically checked" is too strong. Unreachable code is not universally a semantic error: some languages reject it in specified cases, others merely warn or optimize it away.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 11-12

4. Intermediate Code Generation
Easy to produce and easy to translate to machine code

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Question sources

Computer Science MCQ practice collection · Question 783

Question 97 | AM0784

Which phase of the compiler performs type checking and enforces semantic rules?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Semantic Analysis

PDF p6 explicitly names type checking as one of the most important semantic-analysis tasks.

Why each choice is right or wrong

A - Incorrect: Scanning recognizes tokens (p5).

B - Incorrect: Parsing checks grammatical hierarchy (p5).

C - Correct: Type checking is explicitly here (p6).

D - Incorrect: Code generation converts intermediate code (p7).

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

> 3. Semantic Analysis
> • Certain checks are performed to make sure that the components of the program fit together meaningfully.
> • Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
> - Type checking of various programming language constructs is one of the most important tasks.
> • Stores type information in the symbol table or the syntax tree.
> - Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science MCQ practice collection · Question 784

Question 98 | AM0785

Which phase of the compiler translates the intermediate representation into the target machine code?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Code Generation

PDF p7 states that code generation converts intermediate code to machine code.

Why each choice is right or wrong

A - Incorrect: Scanning works on source characters (p5).

B - Incorrect: Parsing groups tokens (p5).

C - Incorrect: Semantic analysis checks types and meaning (p6).

D - Correct: Matches the explicit conversion on p7.

Primary note proof

Study-note section COMP34

> Code generation lowers checked IR to a target form. Instruction selection chooses target operations; register allocation chooses where values reside; scheduling orders instructions while respecting dependencies; storage layout assigns offsets and follows ABI requirements. Register assignment names actual physical registers after allocation decisions.

Study-note section COMP34

> For t=a+b, a simple target could emit LOAD R1,a; LOAD R2,b; ADD R3,R1,R2; STORE t,R3. Other targets may combine loads with arithmetic or keep t in a register. Machine idioms, register availability and instruction order affect execution time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

> 6. Code Generation
> Converts intermediate code to machine code.
> Must handle all aspects of machine architecture
> Storage allocation decisions are made
> a. Register allocation and assignment

Question sources

Computer Science MCQ practice collection · Question 785

Computer Science MCQ practice collection · Question 794

Question 99 | AM0786

What is the primary goal of the code optimization phase in compiler design?

A. To reduce the size of the source code

B. To eliminate syntax errors

C. To improve the performance of the generated code

D. To generate human-readable code

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer C: To improve the performance of the generated code

PDF p7 names improvement in time, space, or power; p17 adds faster execution, efficient memory, and preserving meaning. Thus improving generated-code performance is the best statement.

Why each choice is right or wrong

A - Incorrect: Source text length is not the optimization goal.

B - Incorrect: Syntax errors are caught by parsing (p5); optimization assumes meaningful code to improve.

C - Correct: Matches the performance goal on p17.

D - Incorrect: Human readability is not the goal listed on pp7,17.

Primary note proof

Study-note section COMP30

> Constant folding evaluates constant expressions during compilation: 3*4→12. Constant propagation substitutes known values: x=3; y=x+2 becomes y=3+2 before folding. Copy propagation substitutes a copied name while its source remains unchanged. Dead-code elimination removes unreachable code or computations whose values are unused and whose evaluation has no necessary side effects.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 2-6

5. Code Optimization
Changes the IC by removing such inefficiencies
Improve the code
a. Improvement may be time, space, or power consumption.
It changes the structure of programs,

Compiler Design module | PDF page 17 | lines 16-22

Chapter 8
Introduction to Code Optimization
Goals of Code Optimization
• Remove redundant code without changing the meaning of program
• Executes faster
• Efficient memory usage
• Better performance

Question sources

Computer Science MCQ practice collection · Question 786

Question 100 | AM0787

For E→E+E|E*E|id, which technique resolves the ambiguity by requiring * to bind more tightly than + and both operators to associate left?

A. Parsing with explicit operator precedence and left-associativity rules

B. Lexical analysis

C. Semantic analysis

D. Leftmost derivation

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Parsing with explicit operator precedence and left-associativity rules

The grammar permits multiple groupings of expressions such as id+id*id. An explicit precedence and associativity policy selects multiplication before addition and left association within each level. A parser may enforce this through declarations or a grammar rewrite. Lexical analysis and generic semantic checking do not themselves establish that policy, and taking a leftmost derivation does not remove the competing parse trees.

Why each choice is right or wrong

A - Correct: The stated policy selects the intended grouping.

B - Incorrect: Token recognition does not determine operator grouping.

C - Incorrect: Generic semantic checking supplies no stated grouping policy.

D - Incorrect: An ambiguous grammar can have multiple leftmost derivations.

Primary note proof

Study-note section COMP10

A grammar is ambiguous if at least one terminal string has two different parse trees, equivalently two leftmost derivations. Merely changing derivation order on the same tree does not establish ambiguity. E→E+E|E*E|id allows different trees for id+id*id.

Study-note section COMP10

Use E→E+T|T, T→T*F|F, F→(E)|id to give * higher precedence and both operators left associativity. For subtraction, a−b−c then groups (a−b)−c. The dangling-else grammar permits else to attach to either nested if; a nearest-unmatched-if rule or grammar rewrite resolves it. Taking a leftmost derivation by itself does not resolve ambiguous interpretation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
● Top down parsing (predictive parsers)
• LL(1)
● Bottom up parsing (SR parsers)
• LR(1)

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number

of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science MCQ practice collection · Question 787

Question 101 | AM0788

What is the purpose of semantic rules in a programming language?

A. To define the syntax of the language

B. To specify the order of execution of statements

C. To enforce the correct use of language constructs

D. To optimize the generated code

Hint

Ask whether the required information is a declaration’s identity and attributes, program structure or the ordered source tokens.

Answer C: To enforce the correct use of language constructs

PDF p6 checks that program components fit together meaningfully and p14 requires semantic conventions to hold. C captures those constraints on use of language constructs.

Why each choice is right or wrong

A - Incorrect: Grammatical structure is assigned to syntax analysis (p5).

B - Incorrect: Execution order alone does not capture the type/compatibility constraints on pp6,14.

C - Correct: Correct semantic use follows the cited compatibility and convention checks.

D - Incorrect: Optimization is separately concerned with efficiency (p7).

Primary note proof

Study-note section COMP21

Static checks occur before execution: type compatibility, duplicate declarations within a restricted scope, valid names, argument counts, return compatibility and control-flow placement. A break is valid only in constructs allowed by the language; in C/C++ it can exit a loop

or switch. A compiler may annotate an AST or create typed IR; there is no mandatory uniquely named output tree.

Study-note section COMP21

Dynamic checks depend on runtime values, such as some bounds and null checks. Static analysis can prove some accesses safe or faulty; “all memory errors cannot be statically checked” is too strong. Unreachable code is not universally a semantic error: some languages reject it in specified cases, others merely warn or optimize it away.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 14 | lines 16-23

Static Checking
• The compiler must check the semantic conventions of the source language
• Static Checking: ensures that certain kind of errors are detected and reported
• Example
• Type Checks: incompatible operands
• Flow Control Check
• Uniqueness Check
• Name Related Check

Question sources

Computer Science MCQ practice collection · Question 788

Question 102 | AM0789

Which of the following is responsible for recognizing keywords and identifiers in a programming language?

A. Lexical Analyzer

B. Parser

C. Semantic Analyzer

D. Code Generator

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Lexical Analyzer

PDF p5 lists identifiers and keywords among lexical tokens, and p8 explains their token classes.

Why each choice is right or wrong

A - Correct: Recognizes the token classes explicitly listed on pp5,8.

B - Incorrect: Parser consumes tokens already recognized (p9).

C - Incorrect: Semantic analysis checks type and meaning (p6).

D - Incorrect: Code generation produces machine code (p7).

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> - Scans the source code as a stream of characters
> - Represent lexemes in the form of tokens as:
> <token-name, attribute-value>
> - Token
> • smallest meaningful element that a compiler understands.
> - Eg.
> • Identifiers, Keywords, Literals, Operators and Special symbols.
> - Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 8 | lines 19-24

> • Classes of Tokens
> - Identifiers: names chosen by the programmer
> - Keywords: names already in the programming language

- Separators: punctuation characters
- Operators: symbols that operate on arguments and produce results
- Literals: numeric, textual literals

Question sources

Computer Science MCQ practice collection · Question 789

Question 103 | AM0791

Which phase of the compiler is responsible for checking the syntax of the program?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Syntax Analysis

PDF p5 names syntax analysis Parsing and assigns it grammatical hierarchy and most syntactic-error detection.

Why each choice is right or wrong

A - Incorrect: Scanning identifies lexemes, not grammar structure (p5).

B - Correct: Directly matches the syntax-checking phase on p5.

C - Incorrect: P6 concerns semantic errors such as type mismatch.

D - Incorrect: P7 concerns machine-code generation.

Primary note proof

Study-note section COMP04

Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

Computer Science MCQ practice collection · Question 791

Question 104 | AM0793

What is the main goal of the semantic analysis phase in compiler design?

A. To check the syntax of the program

B. To generate intermediate code

C. To perform type checking and enforce semantic rules

D. To translate the source code to machine code

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: To perform type checking and enforce semantic rules

PDF p6 checks meaningful fit, detects semantic errors, and explicitly emphasizes type checking.

Why each choice is right or wrong

A - Incorrect: Syntactic structure is checked in parsing (p5).

B - Incorrect: Intermediate generation is a separate phase (p6).

C - Correct: Matches both semantic compatibility and type checking on p6.

D - Incorrect: Machine-code conversion belongs to code generation (p7).

Primary note proof

Study-note section COMP22

A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science MCQ practice collection · Question 793

Question 105 | AM0795

What is the purpose of an Abstract Syntax Tree (AST)?

A. To store the program's symbol table

B. To generate assembly code

C. To visualize the program's control flow

D. To represent the hierarchical structure of the program

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: To represent the hierarchical structure of the program

PDF pp12-13 represents a-4+c through identifier/number leaves and operator nodes linked by child pointers. This encodes the expression hierarchy, supporting D.

Why each choice is right or wrong

A - Incorrect: P13 leaves may point to a symbol-table entry, but the whole tree is not the symbol table.

B - Incorrect: P7 assigns machine-code production to code generation; that is not the tree definition.

C - Incorrect: The illustrated tree encodes expression operators and operands, not control-flow paths.

D - Correct: Operator-child nesting on pp12-13 represents program syntax hierarchy.

Primary note proof

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

> 2. Syntax / Hierarchical Analysis - Parsing
> • Tokens are grouped hierarchically into nested collections with collective meaning.
> • The result is generally a parse tree.
> • expressions, statements, declarations etc... are identified by using the results of lexical analysis.
> • Most syntactic errors in the source program are caught in this phase.
> • Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 12 | lines 26-28

> Functions for Syntax Tree Nodes
> • mknode (op, left, right)
> ● Creates an operator node with label op &

Compiler Design module | PDF page 13 | lines 2-14

> ● Two fields containing pointers to left and right
> • mkleaf(id, entry)
> ● Creates an identifier node with label id &
> ● A field containing entry, ptr to symbol table entry for the identifier
> • mkleaf(num, val)
> ● Create a number node with label num &
> ● A field containing val, the value for the number
> Syntax tree for expression a-4+c
> • P1=mkleaf(id,entrya);
> • P2=mkleaf(num, 4);

• P3=mknode('-',p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode('+',p3,p4);

Question sources

Computer Science MCQ practice collection · Question 795

Question 106 | AM0796

What does the symbol table in a compiler store?

A. Intermediate code

B. Machine code

C. Source code

D. Information about identifiers and their attributes

Hint

Ask whether the required information is a declaration's identity and attributes, program structure or the ordered source tokens.

Answer D: Information about identifiers and their attributes

PDF p17 puts names and types in the symbol table; p6 adds variables, function parameters, and array dimensions. P13 points identifier leaves to table entries. This is identifier information and attributes.

Why each choice is right or wrong

A - Incorrect: Intermediate code has separate statement representations (pp15-16).

B - Incorrect: Machine code is output of code generation (p7).

C - Incorrect: The cited table entries are names/types, not the full source text.

D - Correct: Names and type attributes are explicitly stored (pp6,17).

Primary note proof

Study-note section COMP19

A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage. A hash table gives expected O(1) insertion/lookup with suitable hashing; a stack of scope tables supports lexical nesting. Search the nearest scope first, insert declarations in the current scope, and remove visibility when leaving it.

Study-note section COMP19

Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal. Not every

punctuation token needs a symbol-table entry; literals may use a separate literal table. A hash table is an implementation, not the information’s semantic role.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Compiler Design module | PDF page 13 | lines 2-14

- Two fields containing pointers to left and right
• mkleaf(id, entry)
- Creates an identifier node with label id &
- A field containing entry, ptr to symbol table entry for the identifier
• mkleaf(num, val)
- Create a number node with label num &
- A field containing val, the value for the number
Syntax tree for expression a-4+c
• P1=mkleaf(id,entrya);
• P2=mkleaf(num, 4);
• P3=mknode(‘-’,p1,p2);
• P4=mkleaf(id,entryc);
• P5=mknode(‘+’,p3,p4);

Question sources

Computer Science MCQ practice collection · Question 796

Question 107 | AM0797

Which phase of the compiler performs type checking?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Semantic Analysis

PDF p6 explicitly places type checking in semantic analysis.

Why each choice is right or wrong

A - Incorrect: Scanning turns characters into tokens (p5).

B - Incorrect: Parsing groups tokens and checks syntax (p5).

C - Correct: P6 explicitly names this task.

D - Incorrect: Generation converts intermediate to machine code (p7).

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science MCQ practice collection · Question 797

Question 108 | AM0798

What is the purpose of intermediate code in compiler design?

A. To optimize the program's execution

B. To generate assembly code

C. To represent the program's semantics in a platform-independent form

D. To visualize the program's control flow

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer C: Represent program meaning in a reusable, substantially platform-independent intermediate form.

COMP26 identifies separation of source analysis and target implementation as IR's architectural purpose, allowing reuse across targets. C expresses that purpose best. Optimization and assembly generation are operations enabled by the representation, while a control-flow graph is one possible representation aspect. Platform independence is a design benefit here, not a claim that every IR has no target details.

Why each choice is right or wrong

A - Incorrect: Optimization is enabled by IR, but it is not the general representational purpose distinguishing this choice.

B - Incorrect: Assembly is one possible later output; an IR need not be tied to assembly generation.

C - Correct: A reusable representation of program semantics separates front-end meaning from target-specific implementation.

D - Incorrect: Control-flow visualization is one possible use and does not exhaust the purpose of IR.

Primary note proof

Study-note section COMP26

IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 11-12

4. Intermediate Code Generation
Easy to produce and easy to translate to machine code

Compiler Design module | PDF page 3 | lines 25-29

Phases of a Compiler
• Analysis
- Machine Independent/Language Dependent
• Synthesis
- Machine Dependent/Language independent

Compiler Design module | PDF page 15 | lines 13-21

Three Address Code
• Is a sequence of statements of the form
- X = Y op Z
- X,Y and Z are names, constants or compiler generated temporaries
- Op is operator (arithmetic, logical)
• Example:
- a = b + c , x = -y , if a > b goto L1
• LHS is the target
• RHS has at most two sources and one operator

Question sources

Computer Science MCQ practice collection · Question 798

Question 109 | AM0799

Which of the following is used to generate a lexical analyzer for a programming language?

A. Finite Automaton

B. Context-Free Grammar

C. Pushdown Automaton

D. Turing Machine

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer A: A finite automaton.

COMP06 describes converting regular expressions to an NFA and then a DFA. This is the automaton construction used for ordinary generated token recognizers. CFGs and pushdown machinery address nested syntax rather than the regular token-pattern mechanism; a Turing machine is not the specific scanner model in the supplied construction.

Why each choice is right or wrong

A - Correct: NFA/DFA finite automata implement recognition of regular token patterns in the stated generator pipeline.

B - Incorrect: CFGs describe nested syntactic structure rather than the regular scanner-automaton construction.

C - Incorrect: Pushdown machinery adds a stack for context-free structure and is not the ordinary token-recognizer construction.

D - Incorrect: A Turing machine is more general but is not the specific model used in this scanner-generation pipeline.

Primary note proof

Study-note section COMP06

> A common generator pipeline is regular expressions → NFA using Thompson construction → DFA using subset construction; minimization is optional. For concatenation rs, connect r's final state to s's start by ε. Alternation introduces branching ε paths; star adds a way to skip and repeat the fragment.

Study-note section COMP06

> The ε-closure of states includes those reachable by zero or more ε edges. A DFA state is a set of NFA states; its a-transition is ε-closure(move(set,a)). Start with the NFA start closure, repeatedly add unseen subsets, and mark subsets containing an NFA final state as accepting. Token identity and rule priority annotate accepting states; the empty subset acts as a dead state.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

What is Lexical Analysis
• The first phase of a compiler
• The input is a high level language program
• The output is a sequence of tokens
• Strips off blanks, tabs, newlines, and comments from the source program
• Keeps track of line numbers

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Question sources

Computer Science MCQ practice collection · Question 799

Question 110 | AM0800

What is the purpose of a lexical analyzer in a compiler?

A. To check the syntax of the program

B. To generate intermediate code

C. To perform type checking

D. To recognize and tokenize the input source code

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: To recognize and tokenize the input source code

PDF p8 gives the scanner a high-level source-program input and sequence-of-tokens output.

Why each choice is right or wrong

A - Incorrect: Syntax checking belongs to parsing (p5).

B - Incorrect: Intermediate generation is separate (p6).

C - Incorrect: Type checking belongs to semantic analysis (p6).

D - Correct: Directly matches p8 input and output.

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 4-9

> What is Lexical Analysis
> • The first phase of a compiler
> • The input is a high level language program
> • The output is a sequence of tokens
> • Strips off blanks, tabs, newlines, and comments from the source program
> • Keeps track of line numbers

Question sources

Computer Science MCQ practice collection · Question 800

Question 111 | AM0801

Which listed notation directly specifies ordinary token patterns using alternation, concatenation and repetition, rather than a general production notation or a tree of one program?

A. Regular Expressions

B. Context-Free Grammars

C. Backus-Naur Form (BNF)

D. Abstract Syntax Trees (AST)

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer A: Regular Expressions

Classical regular expressions combine alternation, concatenation and repetition to specify ordinary token patterns. CFGs and BNF are production-based grammar descriptions and can also describe regular languages, so mere expressive capability would not distinguish them. An AST represents the analyzed structure of a particular program. The revised comparison asks for the direct pattern notation characterized by its operators, making regular expressions the specific answer.

Why each choice is right or wrong

A - Correct: Regular expressions directly use the stated pattern-combination operators.

B - Incorrect: A CFG is a general production grammar, not the direct expression notation described.

C - Incorrect: BNF is a notation for context-free productions.

D - Incorrect: An AST represents one analyzed program structure rather than the permitted token patterns.

Primary note proof

Study-note section COMP05

> Regular expressions use concatenation, alternation r|s, Kleene star r* for zero or more, and r+ for one or more. In a simplified alphabet, identifier = Letter(Letter|Digit)*; it permits a single letter and forbids a leading digit. A C-like ASCII rule commonly also allows underscore: [A-Za-z_][A-Za-z0-9_]*
> State the convention instead of claiming this covers every language's identifiers.

Study-note section COMP-S02

> Backus-Naur form, or BNF, is a notation for context-free grammar production rules. It describes how syntactic constructs are formed; it is different from an abstract syntax tree representing one particular program.

Study-note section COMP09

> An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 8 | lines 10-18

> Tokens, Patterns, and Lexemes
> • Token
> ● A string of characters which logically belong together
> ● Classes of similar lexemes
> l identifier, keywords, constants etc.
> • Pattern

- A rule which describes a token
- Lexeme
- The sequence of characters matched by a pattern to form the token

Compiler Design module | PDF page 9 | lines 6-16

Grammars
• Every programming language has grammar rules
• Parsers or syntax analyzers are generated for a particular grammar
• CFG are used for syntax specification of programming languages
Context Free Grammar (CFG)
• Is denoted as G = (N, T , P, S)
• N : finite set of non-terminals
• T : finite set of terminals
• S ∈ N: The start symbol
• P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Question sources

Computer Science MCQ practice collection · Question 801

Question 112 | AM0802

Which listed parsing approaches can handle arbitrarily nested parenthesized expressions for suitable grammars and stack-based implementations? Select all that apply.

A. LL Parsing

B. LR Parsing

C. Recursive Descent Parsing

D. Operator Precedence Parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A, B, C, D: LL Parsing; LR Parsing; Recursive Descent Parsing; Operator Precedence Parsing

Nested parentheses require remembering unfinished surrounding expressions. LL and LR parsers use a parsing stack, recursive descent uses the call stack, and suitable operator-precedence expression parsing also uses stack structure and precedence relations. Each can therefore handle the stipulated nested expressions for an appropriate grammar. The existence of nesting alone does not distinguish one of these parser families from the others.

Why each choice is right or wrong

A - Correct: LL can parse a suitable nested-expression grammar using its stack.

B - Correct: LR can recognize the nesting with bottom-up stack actions.

C - Correct: Recursive descent processes nested expressions through recursive calls.

D - Correct: A suitable operator-precedence parser supports parenthesized subexpressions.

Primary note proof

Study-note section COMP18

> LL, LR and recursive-descent parsers can all handle nested parentheses for suitable grammars. An explicit or implicit stack, not the LR label alone, supports unbounded nesting.

Study-note section COMP-S01

> Operator-precedence parsing is a restricted bottom-up expression-parsing method using a stack and precedence relations. With a suitable expression grammar, the stack retains unfinished surrounding expressions while nested parentheses are processed.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 18-23

> Top Down Parsing
> • The parse tree is created top to bottom
> • Starts from the start symbol and transform it to the input
> Bottom Up Parsing
> • Starts with the input symbols and tries to construct the parse tree up to the start symbol.
> • One way of reducing a sentence is to follow the right most derivation in reverse

Compiler Design module | PDF page 11 | lines 2-17

> LL(1) Grammar
> • L - left to right
> • L - left most derivation
> • 1 - number of look ahead
> • First( ) and Follow( )
> ● the first terminal in a string and the terminal that follows a variable respectively.
> LR Parsing
> • LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
> Types of LR Parsers
> • LR (0) , SLR (1) , LALR (1) , CLR (1)
> LL LR
> Leftmost derivation Rightmost derivation in reverse
> Starts with root non-terminal on stack Ends with root non-terminal on the stack
> Builds the parse tree top-down Builds the parse tree bottom-up
> Expands the non-terminals Reduces the non-terminals

Compiler Design module | PDF page 12 | lines 17-25

Syntax Directed Definition- Example
• Production Semantic Rules
• L → E return print(E.val)
• E → E1 + T E.val = E1.val + T.val
• E → T E.val = T.val
• T → T1 * F T.val = T1.val * F.val
• T → F T.val = F.val
• T → (E) F.val = E.val
• F → digit F.val = digit.lexval

Question sources

Computer Science MCQ practice collection · Question 802

Question 113 | AM0803

Which parsing technique builds the parse tree from the bottom-up?

A. LL Parsing

B. LR Parsing

C. Recursive Descent Parsing

D. Top-Down Parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: LR Parsing

LR parsing constructs the parse tree bottom-up, reducing handles that correspond to a rightmost derivation in reverse. LL methods construct from the start symbol toward the input and are top-down. Recursive descent normally implements one procedure per nonterminal, also expanding top-down. The generic top-down option states the opposite construction direction. These are implementation and derivation distinctions, not a ranking of their execution speed.

Why each choice is right or wrong

A - Incorrect: LL parsing expands a leftmost derivation from the start symbol.

B - Correct: LR parsing reduces input handles and builds bottom-up.

C - Incorrect: Recursive-descent procedures expand grammar nonterminals top-down.

D - Incorrect: Top-down construction starts from the root, the opposite direction.

Primary note proof

Study-note section COMP11

Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general

recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP15

Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
● Top down parsing (predictive parsers)
• LL(1)
● Bottom up parsing (SR parsers)
• LR(1)

Compiler Design module | PDF page 10 | lines 18-23

Top Down Parsing
• The parse tree is created top to bottom
• Starts from the start symbol and transform it to the input
Bottom Up Parsing
• Starts with the input symbols and tries to construct the parse tree up to the start symbol.
• One way of reducing a sentence is to follow the right most derivation in reverse

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science MCQ practice collection · Question 803

Question 114 | AM0804

Which is the most specific listed name for a stack-based expression parsing method using pairwise operator-precedence relations in a suitable conflict-free operator grammar?

A. Precedence Parsing

B. Operator Precedence Parsing

C. LR Parsing

D. LL Parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Operator Precedence Parsing

Operator-precedence parsing applies precedence relations to a suitable operator grammar to choose expression parsing actions. The question specifies that concrete method rather than asking for a universal cure for shift/reduce conflicts. Precedence parsing is a broader label, while LR and LL name larger parser families without supplying these particular relations. Therefore B is the most specific match to the described technique.

Why each choice is right or wrong

A - Incorrect: Precedence parsing is broader than the specific method requested.

B - Correct: Operator-precedence parsing directly names this method.

C - Incorrect: LR alone does not specify the operator-precedence method.

D - Incorrect: LL alone does not specify the operator-precedence method.

Primary note proof

Study-note section COMP18

> An operator grammar conventionally forbids adjacent nonterminals on a RHS and usually ε productions. Operator-precedence parsing additionally requires conflict-free precedence relations.

Study-note section COMP-S01

> Operator-precedence parsing is a restricted bottom-up expression-parsing method using a stack and precedence relations. With a suitable expression grammar, the stack retains unfinished surrounding expressions while nested parentheses are processed.

Study-note section COMP11

Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science MCQ practice collection · Question 804

Question 115 | AM0805

Which listed parsing categories have a standard table-driven implementation that uses lookahead to select the next parsing action? Select all that apply. For C, mean ordinary procedure-based recursive descent with choices encoded directly in code.

A. LL Parsing

B. LR Parsing

C. Recursive Descent Parsing

D. Top-Down Parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A, B, D: LL Parsing; LR Parsing; Top-Down Parsing

A nonrecursive LL parser consults a nonterminal/lookahead table to select productions. LR parsers use ACTION and GOTO tables for shifts, reductions and state changes. Table-driven predictive parsing is also a top-down implementation, so the broader D category applies. The stipulated ordinary recursive-descent alternative encodes decisions directly in procedures instead of consulting an explicit parsing table. Thus A, B and D are selected.

Why each choice is right or wrong

A - Correct: Table-driven LL selects productions from lookahead entries.

B - Correct: LR uses ACTION/GOTO tables.

C - Incorrect: The stipulated implementation encodes decisions in procedures rather than an explicit table.

D - Correct: Nonrecursive predictive parsing is a table-driven top-down method.

Primary note proof

Study-note section COMP18

> Both LL and LR can use parsing tables.

Study-note section COMP11

> A nonrecursive predictive parser stores grammar symbols on an explicit stack and consults a table.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 11 | lines 2-17

> LL(1) Grammar
> • L - left to right
> • L - left most derivation
> • 1 - number of look ahead
> • First( ) and Follow( )
> ● the first terminal in a string and the terminal that follows a variable respectively.
> LR Parsing
> • LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
> Types of LR Parsers
> • LR (0) , SLR (1) , LALR (1) , CLR (1)
> LL LR
> Leftmost derivation Rightmost derivation in reverse
> Starts with root non-terminal on stack Ends with root non-terminal on the stack
> Builds the parse tree top-down Builds the parse tree bottom-up
> Expands the non-terminals Reduces the non-terminals

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
● Top down parsing (predictive parsers)
• LL(1)
● Bottom up parsing (SR parsers)
• LR(1)

Question sources

Computer Science MCQ practice collection · Question 805

Question 116 | AM0806

Which parsing technique is more suitable for handling left-recursive grammar rules?

A. LL Parsing

B. LR Parsing

C. Recursive Descent Parsing

D. Top-Down Parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: LR parsing.

COMP12 states that suitable left-recursive grammars are handled by LR parsers, while ordinary unmodified recursive descent can recurse forever without consuming input. LL/top-down prediction generally needs left recursion removed first. This directly supports LR as the suitable choice, without claiming LR accepts every possible left-recursive grammar.

Why each choice is right or wrong

A - Incorrect: Ordinary LL prediction requires eliminating left recursion rather than directly exploiting it.

B - Correct: LR can directly handle suitable left-recursive grammar rules using bottom-up recognition.

C - Incorrect: Unmodified recursive descent can recurse indefinitely before consuming input on a left-recursive rule.

D - Incorrect: Ordinary top-down prediction needs recursion removal; it is not the suitable direct handling choice.

Primary note proof

Study-note section COMP12

Immediate left recursion has A→Aα. For A→Aα|β, where β does not start with A, introduce a new nonterminal A′ and use:
A→βA′
A′→αA′|ε
This preserves the generated language; tree shape changes, so semantic actions must preserve intended associativity. Indirect left recursion follows a cycle through other nonterminals and also needs removal for ordinary predictive descent.

Study-note section COMP12

Left factoring rewrites A→αβ|αγ using:
A→αA′
A′→β|γ
It postpones a choice until after the common prefix. It does not by itself remove ambiguity or guarantee LL(1). LR parsers can handle suitable left-recursive grammars; ordinary unmodified recursive descent can recurse forever before consuming input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 18-23

Top Down Parsing
• The parse tree is created top to bottom
• Starts from the start symbol and transform it to the input
Bottom Up Parsing
• Starts with the input symbols and tries to construct the parse tree up to the start symbol.
• One way of reducing a sentence is to follow the right most derivation in reverse

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science MCQ practice collection · Question 806

Question 117 | AM0807

Which of the following is not a top-down parsing algorithm?

A. Recursive Descent Parsing

B. LL Parsing

C. Predictive Parsing

D. LR Parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer D: LR Parsing

Recursive descent, LL parsing and predictive parsing all expand productions from the start symbol toward the input. Predictive parsing uses lookahead to avoid backtracking, while a general recursive-descent implementation may backtrack. LR parsing instead shifts input and reduces handles to build structure bottom-up. Its construction direction therefore excludes it from the top-down family, independent of whether either implementation uses a stack.

Why each choice is right or wrong

A - Incorrect: Recursive descent expands nonterminal procedures from the start symbol.

B - Incorrect: LL constructs a leftmost derivation top-down.

C - Incorrect: Prediction chooses a top-down production using lookahead.

D - Correct: LR is bottom-up, reducing handles from the input.

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
● Top down parsing (predictive parsers)
• LL(1)
● Bottom up parsing (SR parsers)
• LR(1)

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science MCQ practice collection · Question 807

Question 118 | AM0808

Which listed method can retain multiple parse possibilities for a general ambiguous CFG rather than requiring an ordinary deterministic LL/LR grammar?

A. LL Parsing

B. Generalized LR parsing (GLR)

C. Recursive Descent Parsing

D. Top-Down Parsing

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Generalized LR parsing (GLR)

A general ambiguous grammar has multiple parse trees for at least one input. Generalized LR parsing can handle those possibilities, whereas ordinary deterministic LL/LR grammars must be unambiguous. A separate disambiguation policy may still be needed to select intended meaning. Recursive descent and top-down labels alone do not provide a generalized multi-parse method, so the GLR alternative uniquely answers the question.

Why each choice is right or wrong

A - Incorrect: Ordinary deterministic LL requires an unambiguous predictive grammar.

B - Correct: GLR is the generalized method identified for ambiguous grammars.

C - Incorrect: Ordinary recursive descent alone does not specify a generalized multi-parse algorithm.

D - Incorrect: Top-down direction alone does not handle general ambiguity.

Primary note proof

Study-note section COMP18

> LL, LR and recursive-descent parsers can all handle nested parentheses for suitable grammars. An explicit or implicit stack, not the LR label alone, supports unbounded nesting. Both LL and LR can use parsing tables. Deterministic LL/LR grammars must be unambiguous; a general ambiguous grammar requires rewriting, disambiguation policy or a generalized method such as GLR.

Study-note section COMP18

> An operator grammar conventionally forbids adjacent nonterminals on a RHS and usually ε productions. Operator-precedence parsing additionally requires conflict-free precedence relations. Ordinary right-linear regular grammars also lack adjacent nonterminals, so that property alone does not uniquely identify the operator-grammar category. Regular patterns are chiefly lexical specifications; CFGs chiefly describe nested syntax.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 9 | lines 6-16

> Grammars
> • Every programming language has grammar rules
> • Parsers or syntax analyzers are generated for a particular grammar
> • CFG are used for syntax specification of programming languages
> Context Free Grammar (CFG)
> • Is denoted as G = (N, T , P, S)
> • N : finite set of non-terminals
> • T : finite set of terminals
> • S ∈ N: The start symbol
> • P : Finite set of productions, each of the form A→α, where A∈N and α ∈ (N U T)*

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science MCQ practice collection · Question 808

Question 119 | AM0809

What is the main goal of a compiler?

A. Execution of programs

B. Translation of programs

C. Optimization of programs

D. Debugging of programs

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Translation of programs

PDF p2 defines a compiler as reading a program in one language and translating it to an equivalent program in another.

Why each choice is right or wrong

A - Incorrect: Execution is not the defining translation task on p2.

B - Correct: Exactly matches the compiler definition on p2.

C - Incorrect: Optimization is an application/stage, not the defining goal (pp2,7).

D - Incorrect: Testing may use compiler technology (p2), but debugging is not its defining goal.

Primary note proof

Study-note section COMP01

> A compiler translates a source program into an equivalent target representation while preserving defined observable behavior. The target may be machine code, assembly, bytecode or another source language. A transpiler is a source-to-source compiler, not a fundamentally separate noncompiler technology. A compiler preserves language semantics, not the programmer's unstated intention.

Study-note section COMP01

> An interpreter executes a program representation; it need not work one source line at a time. Systems can compile to bytecode and then interpret or JIT-compile it. AOT compilation happens before execution; JIT compilation happens during execution and may use runtime profiles. Neither is universally faster; startup cost, optimization and workload matter.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 2 | lines 13-15

> What is Compiler
> • a program that reads a program written in one language and translates it into an equivalent program in another language.

Question sources

Computer Science MCQ practice collection · Question 809

Question 120 | AM0810

Which phase of the compiler analyzes the source code to identify the basic syntactic structures?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Syntax Analysis

PDF p5 assigns identifying expressions, statements, and declarations through grammatical grouping to syntax analysis. Here syntactic structures means those higher-level constructs.

Why each choice is right or wrong

A - Incorrect: Scanning recognizes tokens, the lexical units used by parsing (p5).

B - Correct: P5 explicitly identifies expressions, statements and declarations here.

C - Incorrect: Semantic analysis checks meaningful compatibility (p6).

D - Incorrect: Code generation translates IC to machine code (p7).

Primary note proof

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

> 2. Syntax / Hierarchical Analysis - Parsing
> • Tokens are grouped hierarchically into nested collections with collective meaning.
> • The result is generally a parse tree.
> • expressions, statements, declarations etc... are identified by using the results of lexical analysis.
> • Most syntactic errors in the source program are caught in this phase.
> • Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 5 | lines 3-11

> 1. Lexical / Linear Analysis (scanning)
> • Scans the source code as a stream of characters
> • Represent lexemes in the form of tokens as:
> <token-name, attribute-value>
> • Token
> • smallest meaningful element that a compiler understands.
> • Eg.
> • Identifiers, Keywords, Literals, Operators and Special symbols.
> • Blanks, new lines, comments will be removed from the source program.

Question sources

Computer Science MCQ practice collection · Question 810

Question 121 | AM0811

Which phase of the compiler assigns meaning to the syntactic structures and performs type checking?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Semantic Analysis

PDF p6 explicitly checks meaningful fit and performs type checking, which identifies semantic analysis.

Why each choice is right or wrong

A - Incorrect: Lexical analysis recognizes tokens (p5).

B - Incorrect: Syntax analysis checks grammar structure (p5).

C - Correct: Matches both meaningful compatibility and type checking on p6.

D - Incorrect: Code generation converts intermediate code (p7).

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

> 3. Semantic Analysis
> • Certain checks are performed to make sure that the components of the program fit

together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science MCQ practice collection · Question 811

Question 122 | AM0812

Which of the following represents the correct order of compiler phases?

A. Code Generation, Semantic Analysis, Lexical Analysis, Syntax Analysis

B. Lexical Analysis, Syntax Analysis, Semantic Analysis, Code Generation

C. Syntax Analysis, Lexical Analysis, Semantic Analysis, Code Generation

D. Semantic Analysis, Syntax Analysis, Lexical Analysis, Code Generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Lexical Analysis, Syntax Analysis, Semantic Analysis, Code Generation

PDF pp5-7 orders lexical, syntax, semantic, intermediate generation, optimization, and generation. B preserves the relative order of the four listed phases, though it omits two intervening stages.

Why each choice is right or wrong

A - Incorrect: Starts with generation before analysis, reversing the pp5-7 sequence.

B - Correct: Preserves the pp5-7 relative order: lexical, syntax, semantic, generation.

C - Incorrect: Syntax must consume lexical tokens (p5), so it cannot precede lexical analysis.

D - Incorrect: Semantics is phase 3 after lexical and syntax analysis (pp5-6).

Primary note proof

Study-note section COMP03

The conventional sequence is lexical analysis → syntax analysis → semantic analysis → intermediate-code generation → optimization → target-code generation. Scanning turns characters into tokens; parsing builds structure; semantics checks declarations and types; synthesis produces implementation code. A phase is a logical activity; a pass is a traversal that can combine activities.

Study-note section COMP03

The front end handles source-language rules; a middle end analyzes and transforms IR; the backend handles target details. Symbol-table management and error reporting support multiple phases rather than forming one mandatory seventh final phase. Optimization can occur at several levels, including after instruction selection. “Final phase” means code generation in the standard six-phase teaching pipeline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 3-11

1. Lexical / Linear Analysis (scanning)
- Scans the source code as a stream of characters
- Represent lexemes in the form of tokens as:
<token-name, attribute-value>
- Token
• smallest meaningful element that a compiler understands.
- Eg.
• Identifiers, Keywords, Literals, Operators and Special symbols.
- Blanks, new lines, comments will be removed from the source program.

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 6 | lines 11-12

4. Intermediate Code Generation
Easy to produce and easy to translate to machine code

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Question sources

Computer Science MCQ practice collection · Question 812

Question 123 | AM0813

Which data structure is commonly used in symbol tables during the compilation process?

A. Stack

B. Queue

C. Hash Table

D. Linked List

Hint

Ask whether the required information is a declaration's identity and attributes, program structure or the ordered source tokens.

Answer C: A hash table.

COMP19 explicitly says a hash table offers expected constant-time insertion and lookup with suitable hashing. That makes it the conventional structure for the name-to-attribute mapping. A stack can organize nested scope tables, and lists are possible implementation choices, but the common keyed lookup structure asked for here is the hash table.

Why each choice is right or wrong

A - Incorrect: A stack manages nested scope order, while the keyed mapping within a scope commonly uses hashing.

B - Incorrect: FIFO queue order does not directly provide the keyed lookup role of a symbol table.

C - Correct: Hashing commonly implements efficient name-to-attribute insertion and lookup.

D - Incorrect: A linked list is possible but does not provide the expected constant-time keyed lookup highlighted in COMP19.

Primary note proof

Study-note section COMP19

A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage. A hash table gives expected O(1) insertion/lookup with suitable hashing; a stack of scope tables supports lexical nesting. Search the nearest scope first, insert declarations in the current scope, and remove visibility when leaving it.

Study-note section COMP19

Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal. Not every punctuation token needs a symbol-table entry; literals may use a separate literal table. A hash table is an implementation, not the information's semantic role.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science MCQ practice collection · Question 813

Question 124 | AM0814

Which phase of the compiler performs optimization techniques to improve the efficiency of the generated code?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Optimization

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer D: Code Optimization

PDF p7 explicitly improves intermediate code by removing inefficiencies; p17 names faster execution and efficient memory as goals.

Why each choice is right or wrong

A - Incorrect: Lexical analysis tokenizes (p5).

B - Incorrect: Syntax analysis parses (p5).

C - Incorrect: Semantic analysis checks meaningful fit (p6).

D - Correct: The named improvement phase on pp7,17 is code optimization.

Primary note proof

Study-note section COMP30

> Constant folding evaluates constant expressions during compilation: 3*4→12. Constant propagation substitutes known values: x=3; y=x+2 becomes y=3+2 before folding. Copy propagation substitutes a copied name while its source remains unchanged. Dead-code elimination removes unreachable code or computations whose values are unused and whose evaluation has no necessary side effects.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 2-6

> 5. Code Optimization
> Changes the IC by removing such inefficiencies
> Improve the code
> a. Improvement may be time, space, or power consumption.
> It changes the structure of programs,

Compiler Design module | PDF page 17 | lines 16-22

Chapter 8
Introduction to Code Optimization
Goals of Code Optimization
• Remove redundant code without changing the meaning of program
• Executes faster
• Efficient memory usage
• Better performance

Question sources

Computer Science MCQ practice collection · Question 814

Question 125 | AM0815

Which of the following is not a typical optimization performed by a compiler?

A. Constant Folding

B. Loop Unrolling

C. Dead Code Elimination

D. Syntax Analysis

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer D: Syntax Analysis

Constant folding evaluates known expressions, loop unrolling replicates a loop body to reduce branch overhead, and dead-code elimination removes unnecessary computation when effects permit. Each transforms an existing program while preserving its required behavior. Syntax analysis instead checks and organizes grammatical structure in the frontend. It is necessary compiler work, but it is not an optimization transformation of the kind named by the other alternatives.

Why each choice is right or wrong

A - Incorrect: Constant folding replaces constant expressions with computed values.

B - Incorrect: Unrolling replicates loop iterations, handling remainders and code-size costs.

C - Incorrect: Dead-code elimination removes unreachable or unused effect-free computations.

D - Correct: Syntax analysis recognizes grammatical structure rather than optimizing code.

Primary note proof

Study-note section COMP30

Constant folding evaluates constant expressions during compilation: 3*4→12.

Study-note section COMP30

Dead-code elimination removes unreachable code or computations whose values are unused and whose evaluation has no necessary side effects.

Study-note section COMP32

Unrolling replicates a body to reduce branch overhead and expose parallelism; handle remaining iterations and increased code size.

Study-note section COMP03

The conventional sequence is lexical analysis → syntax analysis → semantic analysis → intermediate-code generation → optimization → target-code generation. Scanning turns characters into tokens; parsing builds structure; semantics checks declarations and types; synthesis produces implementation code.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 2-10

Techniques
• Common sub-expression elimination
- Repeated appearance computed previously
• Strength reduction
- Replacement of expensive expressions with simple ones
• Code movement
- Moving a block of code outside a loop
• Dead code elimination
- Eliminated code statements that are either never executed or unreachable

Compiler Design module | PDF page 18 | lines 32-41

• Constant folding
- Eg.
- R2 = 3 * 2
• Constant Propagation
- Eg.
- r1 = 3
- r2 = r1 * 2
• Copy Propagation
- Eg.
- r2 = r1

Compiler Design module | PDF page 5 | lines 12-18

2. Syntax / Hierarchical Analysis - Parsing
• Tokens are grouped hierarchically into nested collections with collective meaning.
• The result is generally a parse tree.
• expressions, statements, declarations etc... are identified by using the results of lexical analysis.
• Most syntactic errors in the source program are caught in this phase.
• Syntactic rules of the source language are given via a Grammar.

Question sources

Computer Science MCQ practice collection · Question 815

Question 126 | AM0816

Which phase of the compiler generates the target code?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Code Generation

PDF p7 states that code generation converts intermediate code to machine code, the target in this question.

Why each choice is right or wrong

A - Incorrect: Lexical analysis emits tokens (p8).

B - Incorrect: Syntax analysis emits a tree (p5).

C - Incorrect: Semantic analysis checks compatibility (p6).

D - Correct: Machine-code output is the explicit code-generation role (p7).

Primary note proof

Study-note section COMP34

> Code generation lowers checked IR to a target form. Instruction selection chooses target operations; register allocation chooses where values reside; scheduling orders instructions while respecting dependencies; storage layout assigns offsets and follows ABI requirements. Register assignment names actual physical registers after allocation decisions.

Study-note section COMP34

> For t=a+b, a simple target could emit LOAD R1,a; LOAD R2,b; ADD R3,R1,R2; STORE t,R3. Other targets may combine loads with arithmetic or keep t in a register. Machine idioms, register availability and instruction order affect execution time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Question sources

Computer Science MCQ practice collection · Question 816

Question 127 | AM0817

Which of the following is an advantage of using an intermediate representation during the compilation process?

A. Faster compilation speed

B. Improved error detection

C. Portability across different platforms

D. Reduced memory usage

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer C: Portability across platforms.

COMP26 explains that IR separates source analysis from target implementation, allowing optimizers and backends to be reused. This supports retargeting and portability across platforms. Introducing IR does not inherently promise faster compilation, better error detection or lower memory use; it can add processing stages and data structures while improving architecture.

Why each choice is right or wrong

A - Incorrect: An extra representation and processing stage do not inherently guarantee faster compilation.

B - Incorrect: Error detection depends on analyses and language rules; it is not the stated architectural advantage.

C - Correct: Separating source meaning from target lowering enables reuse across different platforms.

D - Incorrect: IR introduces storage and does not guarantee reduced compiler memory consumption.

Primary note proof

Study-note section COMP26

IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 11-12

4. Intermediate Code Generation
Easy to produce and easy to translate to machine code

Compiler Design module | PDF page 3 | lines 25-29

Phases of a Compiler
• Analysis
- Machine Independent/Language Dependent
• Synthesis
- Machine Dependent/Language independent

Compiler Design module | PDF page 7 | lines 7-11

6. Code Generation
Converts intermediate code to machine code.
Must handle all aspects of machine architecture
Storage allocation decisions are made
a. Register allocation and assignment

Question sources

Computer Science MCQ practice collection · Question 817

Question 128 | AM0818

What is the shared compiler service that supports detecting, recovering from and reporting lexical, syntax and semantic errors?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Error Handling

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer D: Error Handling

Error handling and reporting is a shared service supporting multiple compilation phases. Scanning detects malformed tokens, parsing detects invalid token arrangements, and semantic analysis detects name, scope or type violations. These are contributors rather than one phase responsible for every error category. Naming the shared service makes D the appropriate complete answer without denying the diagnostic responsibilities of the other phases.

Why each choice is right or wrong

A - Incorrect: Lexical analysis handles lexical errors but is not the complete shared service.

B - Incorrect: Syntax analysis handles parsing errors but is not the complete shared service.

C - Incorrect: Semantic analysis handles semantic errors but is not the complete shared service.

D - Correct: Error handling names the shared support service.

Primary note proof

Study-note section COMP39

> Panic-mode recovery skips input until a synchronizing token such as ; or } permits parsing to resume. Phrase-level recovery makes a local insertion/deletion. Report the location and cause while avoiding cascades of misleading diagnostics. Lexical, syntax and semantic phases each detect their own errors; error handling is a shared service.

Study-note section COMP39

> Lex/flex generates scanners from patterns and actions. Yacc means Yet Another Compiler Compiler; Yacc/Bison generates parsers from grammar rules and semantic actions. Typical specification files have definitions, %% rules, and %% user code. Classic Yacc uses LALR; Bison supports additional LR and generalized modes. Scanner generation does not replace the CFG parser.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 5 | lines 12-18

> 2. Syntax / Hierarchical Analysis - Parsing
> • Tokens are grouped hierarchically into nested collections with collective meaning.
> • The result is generally a parse tree.
> • expressions, statements, declarations etc... are identified by using the results of lexical analysis.
> • Most syntactic errors in the source program are caught in this phase.
> • Syntactic rules of the source language are given via a Grammar.

Compiler Design module | PDF page 6 | lines 2-10

> 3. Semantic Analysis
> • Certain checks are performed to make sure that the components of the program fit together meaningfully.
> • Unlike parsing, this phase checks for semantic errors in the source program (e.g. type

mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 14 | lines 16-23

Static Checking
• The compiler must check the semantic conventions of the source language
• Static Checking: ensures that certain kind of errors are detected and reported
• Example
• Type Checks: incompatible operands
• Flow Control Check
• Uniqueness Check
• Name Related Check

Question sources

Computer Science MCQ practice collection · Question 818

Question 129 | AM0819

Which of the following is true about syntax-directed translation?

A. It only involves the lexical analysis phase.

B. It involves the generation of intermediate code from the parse tree.

C. It is concerned with the optimization of the generated code.

D. It is performed during the code generation phase.

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer B: It can generate intermediate code using parse-tree structure and semantic actions.

COMP25 says syntax-directed actions may compute types, values, AST pointers or generated code. Generating IR from grammatical structure is therefore a supported use of syntax-directed translation. B describes that standard use without limiting it to lexical analysis or a single final phase. Optimization may use other analyses, and syntax-directed translation is not defined solely by optimization.

Why each choice is right or wrong

A - Incorrect: Syntax-directed translation is tied to grammatical productions and semantic actions, not solely tokenization.

B - Correct: Semantic actions associated with parsed structure can compute and emit an intermediate representation.

C - Incorrect: Optimization is not the defining task; translation actions can construct many forms of semantic output.

D - Incorrect: It is not restricted to final target-code generation; actions can run while parsing and constructing IR.

Primary note proof

Study-note section COMP25

A syntax-directed translation scheme embeds executable actions in productions; the actions run while translating a source program, not while generating the compiler. Attributes may compute types, values, AST pointers or generated code. The scheme defines an input-to-output translation, not merely an input language.

Study-note section COMP25

Create an identifier leaf with mkleaf(id,entry), a number with mkleaf(num,value), and an operator node with mknode(op,left,right). For a−4+c, construct +(−(a,4),c). Postorder traversal prints postfix: a 4 − c +. A stack evaluator pushes operands and pops right then left operands for an operator. Infix-to-postfix is useful for stack machines but not mandatory for all compilers.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 4-16

Syntax Directed Translation
• Attaching actions to the grammar rules(productions).
• Actions are executed during the compilation
- Not during the generation of the compiler
• Actions are executed according to the parsing mechanism.
Syntax Directed Definitions
• Is a generalization of a context free grammar
• Is a CFG with attributes and rules
• Attributes are associated with grammar symbols and rules with productions
• Attributes may be:
- Numbers
- Types
- Strings etc

Compiler Design module | PDF page 12 | lines 17-25

Syntax Directed Definition- Example
• Production Semantic Rules
• L → E return print(E.val)
• E → E1 + T E.val = E1.val + T.val
• E → T E.val = T.val
• T → T1 * F T.val = T1.val * F.val

• T → F T.val = F.val
• T → (E) F.val = E.val
• F → digit F.val = digit.lexval

Compiler Design module | PDF page 12 | lines 26-28

Functions for Syntax Tree Nodes
• mknode (op, left, right)
● Creates an operator node with label op &

Question sources

Computer Science MCQ practice collection · Question 819

Question 130 | AM0820

What is the general defining role of semantic rules attached to productions in a syntax-directed definition, rather than one possible specialized use?

A. To define the syntactic structure of the language.

B. To define computations of attributes such as types, values, environments and generated code.

C. To perform optimization techniques on the intermediate code.

D. To define the order of evaluation of expressions.

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer B: To define computations of attributes such as types, values, environments and generated code.

The grammar productions define syntactic structure, while semantic rules define computations of attributes associated with grammar symbols. Those attributes may represent types, values, environments or generated code. Assigning types and constraining evaluation dependencies are examples rather than exhaustive definitions. Optimization may occur in a particular translation, but is not the general defining role of all such rules.

Why each choice is right or wrong

A - Incorrect: Productions, rather than semantic rules, specify the grammatical structure.

B - Correct: Computing attributes is the general defining role.

C - Incorrect: Optimization is a possible specialized transformation, not the general definition.

D - Incorrect: Evaluation order is a specialized concern rather than the complete role.

Primary note proof

Study-note section COMP24

A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an inherited or synthesized attribute. Synthesized attributes flow from children to a parent; inherited attributes carry context from parent or siblings.

Study-note section COMP24

For E→E1+T, E.val=E1.val+T.val synthesizes an expression value. A declaration's type environment passed into its body is inherited. S-attributed definitions use only synthesized attributes and fit LR reduction-time evaluation naturally; suitable LL parsers can also compute them. L-attributed definitions restrict dependencies so a left-to-right traversal can evaluate inherited context. Attribute dependency cycles can prevent evaluation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 12 | lines 4-16

Syntax Directed Translation
• Attaching actions to the grammar rules(productions).
• Actions are executed during the compilation
- Not during the generation of the compiler
• Actions are executed according to the parsing mechanism.
Syntax Directed Definitions
• Is a generalization of a context free grammar
• Is a CFG with attributes and rules
• Attributes are associated with grammar symbols and rules with productions
• Attributes may be:
- Numbers
- Types
- Strings etc

Compiler Design module | PDF page 12 | lines 17-25

Syntax Directed Definition- Example
• Production Semantic Rules
• L → E return print(E.val)
• E → E1 + T E.val = E1.val + T.val
• E → T E.val = T.val
• T → T1 * F T.val = T1.val * F.val
• T → F T.val = F.val
• T → (E) F.val = E.val
• F → digit F.val = digit.lexval

Compiler Design module | PDF page 14 | lines 24-30

Type Checking of Expressions
E → literal { E.type = char }
E → num { E.type = int }
E → id { E.type = lookup(id.entry) }
E→E1 mod E2 { E.type=if E1.type=int and E2.type= int

then int
else type_error }

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Question sources

Computer Science MCQ practice collection · Question 820

Question 131 | AM0821

Which phase of the compiler is responsible for type checking?

A. Lexical Analysis

B. Syntax Analysis

C. Semantic Analysis

D. Code Generation

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Semantic Analysis

PDF p6 explicitly names type checking as a major task of semantic analysis.

Why each choice is right or wrong

A - Incorrect: Lexical analysis recognizes tokens (p5).

B - Incorrect: Syntax analysis checks grammatical hierarchy (p5).

C - Correct: Type checking is explicitly assigned here (p6).

D - Incorrect: Code generation converts IC to machine code (p7).

Primary note proof

Study-note section COMP22

A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Question sources

Computer Science MCQ practice collection · Question 821

Question 132 | AM0822

What is type checking in a compiler?

A. Verifying the syntax of the program.

B. Detecting lexical errors in the program.

C. Ensuring that the operations on variables are well-defined and compatible.

D. Generating the intermediate code for the program.

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Ensuring that the operations on variables are well-defined and compatible.

PDF p14 defines type checking as enforcing type safety and lists incompatible operands; p15 makes assignment produce type_error when the identifier and expression types differ. Thus C expresses compatible operations.

Why each choice is right or wrong

A - Incorrect: Grammatical syntax is handled by parsing (p5).

B - Incorrect: Token recognition is lexical analysis (p8), not type compatibility.

C - Correct: Compatible operations and assignments match pp14-15.

D - Incorrect: Intermediate generation is distinct from semantic checking (pp6,15).

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 2-15

> • Type Errors:
> - Improper or inconsistent operations during program execution
> • Type-safety:
> - Absence of type errors
> Type Checking
> • Semantic checks to enforce the type safety of the program
> • Semantic Checks
> - Static - done during compilation
> - Dynamic - done during run-time
> • Examples
> - Unary and binary operators
> - Number and type of arguments
> - Return statement with return type
> - Compatible assignment

Compiler Design module | PDF page 14 | lines 16-23

Static Checking
• The compiler must check the semantic conventions of the source language
• Static Checking: ensures that certain kind of errors are detected and reported
• Example
• Type Checks: incompatible operands
• Flow Control Check
• Uniqueness Check
• Name Related Check

Compiler Design module | PDF page 15 | lines 2-10

E→E1[E2] { E.type=if E2.type=int and
E1.type=array(s,t) then t else type_error }
Type Checking of Statements
S→id=E { S.type = if id.type=E.type then
void else type_error }
S→if E then S1 { S.type = if E.type=Boolean then
S1.type else type_error }
S→while E do S1 { S.type = if E.type = Boolean then
S1.type else type_error }

Question sources

Computer Science MCQ practice collection · Question 822

Question 133 | AM0823

Which of the following is an example of a static type checking technique?

A. Type inference

B. Dynamic typing

C. Run-time type checking

D. Operator overloading

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer A: Type inference.

COMP22 defines type inference as deriving types from constraints and explicitly calls it static even without explicit type annotations. Dynamic typing and runtime checks concern execution-time behavior. Operator overloading defines several meanings for an operation, but is not a replacement for the checking technique itself. Thus only type inference is the named static technique.

Why each choice is right or wrong

A - Correct: Type inference derives expression types from constraints before execution and is explicitly a static technique.

B - Incorrect: Dynamic typing does not name the requested before-execution type-inference technique.

C - Incorrect: Runtime type checking occurs during execution, so it is not static checking.

D - Incorrect: Overloading supplies operation meanings; COMP22 distinguishes it from type checking itself.

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 2-15

> • Type Errors:
> - Improper or inconsistent operations during program execution
> • Type-safety:
> - Absence of type errors
> Type Checking
> • Semantic checks to enforce the type safety of the program
> • Semantic Checks
> - Static - done during compilation
> - Dynamic - done during run-time
> • Examples
> - Unary and binary operators
> - Number and type of arguments
> - Return statement with return type
> - Compatible assignment

Compiler Design module | PDF page 14 | lines 24-30

> Type Checking of Expressions
> E → literal { E.type = char }

E → num { E.type = int }
E → id { E.type = lookup(id.entry) }
E→E1 mod E2 { E.type=if E1.type=int and E2.type= int
then int
else type_error }

Question sources

Computer Science MCQ practice collection · Question 823

Question 134 | AM0824

Which attribute is specifically recorded for each variable in a type-oriented symbol table and distinguishes its typing information from the name that identifies the entry?

A. To store the types of variables used in the program.

B. To store the names of variables used in the program.

C. To store the intermediate code generated during parsing.

D. To store the syntax rules of the programming language.

Hint

Ask whether the required information is a declaration's identity and attributes, program structure or the ordered source tokens.

Answer A: To store the types of variables used in the program.

A symbol table associates a name or declaration identity with attributes. In a type-oriented table, the relevant attribute is the variable's type, used to check whether operations and assignments are compatible. Names identify the records but do not themselves give typing information. Intermediate instructions and grammar rules belong to other representations and are not the per-variable typing attribute requested.

Why each choice is right or wrong

A - Correct: Variable types provide the requested typing attribute.

B - Incorrect: Names identify entries but are not their type attributes.

C - Incorrect: IR instructions are not the per-variable type attribute.

D - Incorrect: Syntax rules describe the language rather than variable types.

Primary note proof

Study-note section COMP19

A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage. A hash table gives expected O(1) insertion/lookup with suitable hashing; a stack of scope tables supports lexical nesting. Search

the nearest scope first, insert declarations in the current scope, and remove visibility when leaving it.

Study-note section COMP19

Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal. Not every punctuation token needs a symbol-table entry; literals may use a separate literal table. A hash table is an implementation, not the information's semantic role.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 6 | lines 2-10

3. Semantic Analysis
• Certain checks are performed to make sure that the components of the program fit together meaningfully.
• Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
- Type checking of various programming language constructs is one of the most important tasks.
• Stores type information in the symbol table or the syntax tree.
- Types of variables, function parameters, array dimensions, etc.

Compiler Design module | PDF page 17 | lines 2-15

Declarations
• Involves allocation of space in memory &
• Entry of type and name in symbol table
• Off set variable (Offset=0) is used to denote the base address
int a; float b;
Allocation process: { offset = 0 }
int a;
id.type = int
id.width = 2
offset = offset + id.width { offset = 2 }
float b;
id.type=float
id.width=4
offset = offset +id.width { offset = 6 }

Compiler Design module | PDF page 14 | lines 24-30

Type Checking of Expressions
E → literal { E.type = char }
E → num { E.type = int }
E → id { E.type = lookup(id.entry) }
E→E1 mod E2 { E.type=if E1.type=int and E2.type= int
then int
else type_error }

Question sources

Computer Science MCQ practice collection · Question 824

Question 135 | AM0825

In ordinary C++ with no custom conversions, which offered example is specifically a type mismatch rather than a syntax, value-domain or layout issue?

A. Syntax error in an if-else statement.

B. Division of a number by zero.

C. int x="hello";

D. Incorrect indentation in the program.

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: int x="hello";

An ordinary C++ declaration int x="hello" attempts to initialize an integer from a string literal and is a type mismatch under the stated rules. Division by zero can involve correctly typed operands and is instead a value-domain fault. Incorrect indentation is layout, while malformed if-else syntax concerns grammatical arrangement. The option illustrates an incompatible-type example.

Why each choice is right or wrong

A - Incorrect: This explicitly describes a syntax error rather than a type mismatch.

B - Incorrect: A zero divisor is a value issue even with compatible numeric types.

C - Correct: A string literal is not a compatible ordinary integer initializer.

D - Incorrect: Indentation does not establish an incompatible operand type.

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 2-15

• Type Errors:
- Improper or inconsistent operations during program execution

• Type-safety:
- Absence of type errors

Type Checking
• Semantic checks to enforce the type safety of the program
• Semantic Checks
- Static - done during compilation
- Dynamic - done during run-time

• Examples
- Unary and binary operators
- Number and type of arguments
- Return statement with return type
- Compatible assignment

Compiler Design module | PDF page 14 | lines 24-30

Type Checking of Expressions
E → literal { E.type = char }
E → num { E.type = int }
E → id { E.type = lookup(id.entry) }
E→E1 mod E2 { E.type=if E1.type=int and E2.type= int
then int
else type_error }

Compiler Design module | PDF page 15 | lines 2-10

E→E1[E2] { E.type=if E2.type=int and
E1.type=array(s,t) then t else type_error }
Type Checking of Statements
S→id=E { S.type = if id.type=E.type then
void else type_error }
S→if E then S1 { S.type = if E.type=Boolean then
S1.type else type_error }
S→while E do S1 { S.type = if E.type = Boolean then
S1.type else type_error }

Question sources

Computer Science MCQ practice collection · Question 825

Question 136 | AM0826

Which advantage of static type checking specifically concerns detecting incompatible operations before the program executes?

A. Improved program performance.

B. Enhanced code optimization.

C. Early detection of potential runtime errors.

D. Simplified debugging process.

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Early detection of potential runtime errors.

Static type checking can reject an incompatible operation during compilation, before that operation causes a run-time type error. This is the direct timing benefit asked for. Type information may also aid optimization, and diagnostics may make debugging easier, but those are additional consequences rather than the specific before-execution detection property. Type checking does not by itself guarantee faster code or eliminate all possible run-time errors.

Why each choice is right or wrong

A - Incorrect: Faster execution is not guaranteed and does not state detection before execution.

B - Incorrect: Optimization may use types but is a different benefit from the requested early rejection.

C - Correct: Compile-time rejection detects incompatible operations before execution.

D - Incorrect: Debugging may be easier, but that broad consequence does not specify the requested detection timing.

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 2-15

• Type Errors:
- Improper or inconsistent operations during program execution
• Type-safety:
- Absence of type errors

Type Checking
• Semantic checks to enforce the type safety of the program
• Semantic Checks
- Static - done during compilation
- Dynamic - done during run-time
• Examples
- Unary and binary operators
- Number and type of arguments
- Return statement with return type
- Compatible assignment

Compiler Design module | PDF page 14 | lines 16-23

Static Checking
• The compiler must check the semantic conventions of the source language
• Static Checking: ensures that certain kind of errors are detected and reported
• Example
• Type Checks: incompatible operands
• Flow Control Check
• Uniqueness Check
• Name Related Check

Question sources

Computer Science MCQ practice collection · Question 826

Question 137 | AM0827

What is the primary goal of type checking in a compiler?

A. To ensure syntactic correctness of the program.

B. To generate efficient machine code.

C. To enforce programming language rules and constraints.

D. To provide support for code reuse through libraries.

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: To enforce programming language rules and constraints.

PDF p14 defines type checking as semantic checks enforcing type safety and says the compiler checks source-language semantic conventions. C is the offered broad rule-enforcement description, limited here to type constraints.

Why each choice is right or wrong

A - Incorrect: Syntax correctness is the parsing task (p5).

B - Incorrect: Machine-code efficiency is an optimization goal (pp7,17).

C - Correct: Type-related semantic conventions and constraints match p14.

D - Incorrect: Library reuse is not a type-checking goal stated in the module.

Primary note proof

Study-note section COMP22

> A type checker verifies that operations are defined for their operands. In a simple strict language, integer mod requires two integers; array indexing requires an array and integer index; an if condition requires Boolean; assignment requires equal or convertible types. Function calls check parameter count and compatibility, and returns match result types.

Study-note section COMP22

> Type inference derives types from constraints: if x is integer and + expects numeric operands, infer the expression type from numeric promotion rules. It is a static technique even without explicit annotations. Operator overloading supplies multiple operation meanings, not a substitute for type checking. int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 2-15

> • Type Errors:
> - Improper or inconsistent operations during program execution
> • Type-safety:
> - Absence of type errors
> Type Checking
> • Semantic checks to enforce the type safety of the program
> • Semantic Checks
> - Static - done during compilation
> - Dynamic - done during run-time
> • Examples
> - Unary and binary operators
> - Number and type of arguments
> - Return statement with return type
> - Compatible assignment

Compiler Design module | PDF page 14 | lines 16-23

Static Checking
• The compiler must check the semantic conventions of the source language
• Static Checking: ensures that certain kind of errors are detected and reported
• Example
• Type Checks: incompatible operands
• Flow Control Check
• Uniqueness Check
• Name Related Check

Question sources

Computer Science MCQ practice collection · Question 827

Question 138 | AM0828

Which of the following is an example of a type coercion or type conversion?

A. Assigning a floating-point value to an integer variable.

B. Assigning a value to an uninitialized variable.

C. Declaring a variable without specifying its type.

D. Using an undeclared variable in the program.

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer A: Converting a floating-point value when assigning to an integer variable, where the language permits it.

COMP23 gives C++ int x=3.14 as an ordinary conversion yielding 3 when representable. That is the listed example of conversion, with the language caveat that Java needs a permitted cast and C++ brace initialization rejects narrowing. The remaining choices concern initialization, inference or undeclared names rather than converting one value type to another.

Why each choice is right or wrong

A - Correct: Floating-to-integer assignment performs conversion where permitted; exact legality and precision loss depend on the language.

B - Incorrect: First assigning to a variable is initialization or assignment, not inherently a change of value type.

C - Incorrect: Omitting an explicit declaration type may invoke inference; it does not itself convert a value.

D - Incorrect: An undeclared name is a name-resolution problem rather than an example of value conversion.

Primary note proof

Study-note section COMP23

Coercion is an implicit conversion permitted by the language; a cast requests an explicit conversion. Widening often increases representable range, but can still lose precision: Java int→float is widening although large integers round. Java int→long is exact. Narrowing can lose range or fractional information.

Study-note section COMP23

In C++ ordinary int x=3.14 converts to 3 when representable; it is not inherently a type error. int x{3.14} is rejected as narrowing. C++ floating-to-integer conversion discards the fraction and has undefined behavior if the truncated value is unrepresentable. Java ordinary double→int assignment needs a permitted explicit cast. Mixing char and int is ordinarily valid after promotion in C++/Java; do not import a fictitious universal ban.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 14 | lines 2-15

• Type Errors:
- Improper or inconsistent operations during program execution
• Type-safety:
- Absence of type errors

Type Checking
• Semantic checks to enforce the type safety of the program
• Semantic Checks
- Static - done during compilation
- Dynamic - done during run-time
• Examples
- Unary and binary operators
- Number and type of arguments
- Return statement with return type
- Compatible assignment

Compiler Design module | PDF page 15 | lines 2-10

E→E1[E2] { E.type=if E2.type=int and
E1.type=array(s,t) then t else type_error }
Type Checking of Statements
S→id=E { S.type = if id.type=E.type then
void else type_error }
S→if E then S1 { S.type = if E.type=Boolean then
S1.type else type_error }
S→while E do S1 { S.type = if E.type = Boolean then
S1.type else type_error }

Question sources

Computer Science MCQ practice collection · Question 828

Question 139 | CS0121

Which listed approaches are bottom-up and therefore are not top-down parsers? Select all that apply. Assume predictive recursive descent for A.

A. Recursive-descent parser

B. Shift-reduce parser

C. Non-recursive predictive parser

D. LR parser

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B, D: Shift-reduce parser; LR parser

Shift-reduce parsing starts from the input and reduces recognized handles toward the start symbol. LR is a bottom-up family using that strategy. Recursive descent and nonrecursive prediction instead expand productions top-down for suitable grammars. Therefore both B and D fail the top-down description. This directional distinction avoids an unsupported efficiency ranking between recursive and table-driven predictive implementations.

Why each choice is right or wrong

A - Incorrect: Predictive recursive descent is top-down.

B - Correct: Shift-reduce is bottom-up.

C - Incorrect: Nonrecursive prediction is top-down.

D - Correct: LR is bottom-up.

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 10 | lines 11-17

Parsing
• Process of constructing parse tree for a sentence generated by a given grammar.
• 2 types of parsers
● Top down parsing (predictive parsers)
• LL(1)
● Bottom up parsing (SR parsers)
• LR(1)

Compiler Design module | PDF page 11 | lines 2-17

LL(1) Grammar
• L - left to right
• L - left most derivation
• 1 - number of look ahead
• First( ) and Follow( )
● the first terminal in a string and the terminal that follows a variable respectively.
LR Parsing
• LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.
Types of LR Parsers
• LR (0) , SLR (1) , LALR (1) , CLR (1)
LL LR
Leftmost derivation Rightmost derivation in reverse
Starts with root non-terminal on stack Ends with root non-terminal on the stack
Builds the parse tree top-down Builds the parse tree bottom-up
Expands the non-terminals Reduces the non-terminals

Question sources

Computer Science Exit Exam practice collection · Question 121

Question 140 | CS0267

Which one of the following is a type of compiler that compiles only the changed lines from source code and updates the object code accordingly?

A. Retargetable compiler

B. One pass compiler

C. Incremental compiler

D. Parallelizing compiler

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer C: An incremental compiler.

COMP02 defines incremental compilation as recompiling affected fragments or dependent units after edits. This is the compiler category intended by updating code after source changes. The question's only changed lines wording is simplified: real dependency tracking can require recompiling more than literal edited lines. Retargeting and one-pass organization describe different properties.

Why each choice is right or wrong

A - Incorrect: Retargetable compilers support changing target descriptions rather than selecting edited source fragments.

B - Incorrect: One pass counts traversals of the program, not whether edits trigger selective recompilation.

C - Correct: Incremental compilation updates affected fragments after edits, accounting for dependencies beyond literal changed lines.

D - Incorrect: Parallelizing compilation is not the category defined by selective recompilation after source changes.

Primary note proof

Study-note section COMP02

> A preprocessor handles directives such as file inclusion and macros. An assembler translates assembly to object code. A linker combines object files and libraries, resolves external symbols and performs relocation. A loader maps an executable into memory and starts it. Compilation alone does not necessarily produce a ready executable.

Study-note section COMP02

> A cross-compiler runs on a host platform and generates code for a different target. Multiple targets require appropriate backends, not a special definition of cross-compilation. A retargetable compiler supports changing target descriptions. Incremental compilation recompiles affected fragments or dependent units after edits; it is more than literally replacing changed lines. One-pass versus multipass counts traversals, not supported targets.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 2 | lines 13-15

> What is Compiler
> • a program that reads a program written in one language and translates it into an equivalent program in another language.

Compiler Design module | PDF page 3 | lines 20-24

> • Cross-Compiler
> • compiler that runs on platform (A) and generates executable code for another platform (B).

- Source-to-source Compiler
- compiler that translates source code of one programming language to another

Question sources

Computer Science Exit Exam practice collection · Question 267

Question 141 | CS0268

Which technique of code optimization shifts computations from run time to compile time by evaluating constant expressions at compile time and replacing them with their values?

A. Copy propagation

B. Compile time evaluation

C. Code motion

D. Dead code elimination

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B: Compile time evaluation

PDF p18 lists constant folding and illustrates R2 = 3 * 2. Evaluating this constant expression early gives 6, eliminating that runtime computation. B describes this operation as compile time evaluation, although constant folding would be the module's exact name.

Why each choice is right or wrong

A - Incorrect: P18 copy propagation example copies r1 into r2, not evaluates a constant arithmetic expression.

B - Correct: Evaluating 3 * 2 to 6 is the constant-folding example on p18, matching the stem.

C - Incorrect: P18 code movement relocates a block outside a loop, not necessarily evaluating it at compilation.

D - Incorrect: P18 dead-code elimination removes unreachable statements, rather than computing constant values.

Primary note proof

Study-note section COMP30

Constant folding evaluates constant expressions during compilation: 3*4→12. Constant propagation substitutes known values: x=3; y=x+2 becomes y=3+2 before folding. Copy propagation substitutes a copied name while its source remains unchanged. Dead-code elimination removes unreachable code or computations whose values are unused and whose evaluation has no necessary side effects.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Compiler Design module | PDF page 18 | lines 32-41

- Constant folding
  - Eg.
  - R2 = 3 * 2
- Constant Propagation
  - Eg.
  - r1 = 3
  - r2 = r1 * 2
- Copy Propagation
  - Eg.
  - r2 = r1

Compiler Design module | PDF page 18 | lines 2-10

Techniques
- Common sub-expression elimination
  - Repeated appearance computed previously
- Strength reduction
  - Replacement of expensive expressions with simple ones
- Code movement
  - Moving a block of code outside a loop
- Dead code elimination
  - Eliminated code statements that are either never executed or unreachable

Question sources

Computer Science Exit Exam practice collection · Question 268

Question 142 | NS0222

For const int x=8; const int y=16; if(x>y){int z=x+y;}, which techniques directly evaluate the known condition and remove its unreachable body? Select all that apply. Arithmetic simplification means algebraic identity rewriting here.

A. Code motion

B. Dead code elimination

C. Compile time evaluation

D. Arithmetic simplification

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B, C: Dead code elimination; Compile time evaluation

Both x and y are known constants, so evaluating 8>16 at compile time yields false. The body can never execute and dead-code elimination can remove it. These are distinct but compatible transformations, corresponding to B and C in this option order. Code motion relocates computations, while algebraic identity simplification is not the decisive operation on this constant-false condition.

Why each choice is right or wrong

A - Incorrect: No relocation is needed to evaluate or delete the branch.

B - Correct: The unreachable branch body can be removed.

C - Correct: The condition is evaluated from known constants at compile time.

D - Incorrect: No algebraic identity rewrite is the decisive step.

Primary note proof

Study-note section COMP30

> For const int x=8,y=16; if(x>y){int z=x+y;}, propagate the constants, fold 8>16 to false, then remove the unreachable branch. Both compile-time evaluation and dead-code elimination apply.

Study-note section COMP31

> Algebraic simplification removes identities when valid: integer x+0→x.

Question sources

Computer Science Exit Exam practice collection · Question 263

Question 143 | NS0223

What is the difference between a compiler and an interpreter?

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer: A compiler translates; an interpreter executes a representation.

COMP01 distinguishes translating source into an equivalent target from executing a program representation. Bytecode systems may combine compilation, interpretation and JIT, so the distinction is not simply all-at-once versus line-by-line execution.

Primary note proof

Study-note section COMP01

> A compiler translates a source program into an equivalent target representation while preserving defined observable behavior. The target may be machine code, assembly, bytecode or another source language. A transpiler is a source-to-source compiler, not a fundamentally separate noncompiler technology. A compiler preserves language semantics, not the programmer's unstated intention.

Study-note section COMP01

> An interpreter executes a program representation; it need not work one source line at a time. Systems can compile to bytecode and then interpret or JIT-compile it. AOT compilation happens before execution; JIT compilation happens during execution and may use runtime profiles. Neither is universally faster; startup cost, optimization and workload matter.

Question sources

Compiler Design review questions · Question 1

Question 144 | NS0224

What is the difference between lexical analysis and syntax analysis?

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer: Lexical analysis groups characters into tokens; syntax analysis groups tokens into structures.

COMP41 separates character-to-token classification from token-to-expression or statement structure. The scanner recognizes lexemes such as identifiers and operators and emits an ordered token stream. The parser checks how those tokens form grammatical constructs and can build a parse tree or AST; legal tokens can still appear in an illegal syntactic arrangement.

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Question sources

Compiler Design review questions · Question 2

Question 145 | NS0225

What is the purpose of a symbol table?

Hint

Ask whether the required information is a declaration’s identity and attributes, program structure or the ordered source tokens.

Answer: A symbol table associates names with declaration attributes.

COMP19 maps names or identities to type, scope, width, offsets, signatures and linkage information. These records support declaration lookup and semantic checks; a hash table is one implementation rather than the purpose itself.

Primary note proof

Study-note section COMP19

> A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage. A hash table gives expected O(1) insertion/lookup with suitable hashing; a stack of scope tables supports lexical nesting. Search the nearest scope first, insert declarations in the current scope, and remove visibility when leaving it.

Study-note section COMP19

> Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal. Not every punctuation token needs a symbol-table entry; literals may use a separate literal table. A hash table is an implementation, not the information’s semantic role.

Question sources

Compiler Design review questions · Question 3

Question 146 | NS0226

What is the difference between top-down and bottom-up parsing?

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer: Top-down expands from the start symbol; bottom-up reduces input toward it.

COMP15 describes bottom-up recognition as input reductions toward S, reversing a rightmost derivation. COMP11 describes top-down expansion from S toward the input, normally a leftmost derivation.

Primary note proof

Study-note section COMP11

> Top-down parsing starts with S and expands productions toward the input, usually constructing a leftmost derivation. Recursive descent typically has one procedure per nonterminal. A general recursive-descent implementation can backtrack; a predictive parser chooses a production using lookahead without backtracking. LL(1) scans Left to right, produces a Leftmost derivation and uses one lookahead token.

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse. A handle is a production RHS occurrence whose reduction is the next step of that reverse derivation. Not every substring matching a RHS is a handle in the current context.

Question sources

Compiler Design review questions · Question 4

Question 147 | NS0227

What is intermediate code and why is it used?

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer: IR separates source analysis from target lowering and enables reuse.

COMP26 defines IR's role in separating source meaning from target implementation so analysis, optimization and backends can be reused. ASTs, graphs, TAC and bytecode are possible forms. A useful IR is easy to generate, analyze, transform and lower; portability does not require every IR to exclude all target-specific details.

Primary note proof

Study-note section COMP26

> IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

> Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Question sources

Compiler Design review questions · Question 5

Question 148 | NS0228

Which of the following is NOT a valid token type produced during lexical analysis?

A. Keyword |

B. Parse tree node

C. Identifier |

D. Numeric literal

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Parse-tree node is not a lexical token type.

COMP04 lists keywords, identifiers and literals among token classes. A parse-tree node belongs to the grammatical structure built after scanning, not the lexical class of a source lexeme. Consequently only parse-tree node is the invalid token-type choice, while the other three are ordinary scanner output categories.

Why each choice is right or wrong

A - Incorrect: Keywords are explicitly listed as a valid lexical token class.

B - Correct: Parse-tree nodes represent grammatical structure, not a lexical token category emitted by a scanner.

C - Incorrect: Identifiers are an ordinary lexical token class.

D - Incorrect: Numeric literals are literals and therefore valid token categories.

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Question sources

Compiler Design review questions · Question 1

Question 149 | NS0229

What algorithm is commonly used to convert a Non-deterministic Finite Automaton (NFA) to a Deterministic Finite Automaton (DFA) in lexer construction?

A. Thompson's construction |

B. Subset construction (Powerset construction)

C. Cocke-Younger-Kasami (CYK) |

D. Earley's algorithm

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer B: Subset (powerset) construction.

COMP06 distinguishes Thompson construction, which builds an NFA from a regular expression, from subset construction, which determinizes the NFA. Each DFA state represents an NFA-state subset, with epsilon closure when needed. CYK and Earley are grammar-parsing algorithms rather than this automaton determinization, so the powerset option is the relevant stage.

Why each choice is right or wrong

A - Incorrect: Thompson construction goes from regular expression to NFA rather than NFA to DFA.

B - Correct: Subset construction creates DFA states from sets of NFA states.

C - Incorrect: CYK performs CFG membership parsing and does not determinize a lexical NFA.

D - Incorrect: Earley is a grammar-parsing method rather than the stated NFA-to-DFA construction.

Primary note proof

Study-note section COMP06

> A common generator pipeline is regular expressions → NFA using Thompson construction → DFA using subset construction; minimization is optional.

Study-note section COMP06

> The ε-closure of states includes those reachable by zero or more ε edges. A DFA state is a set of NFA states; its a-transition is ε-closure(move(set,a)). Start with the NFA start closure, repeatedly add unseen subsets, and mark subsets containing an NFA final state as accepting. Token identity and rule priority annotate accepting states; the empty subset acts as a dead state.

Study-note section COMP-S02

> CYK and Earley are context-free grammar parsing algorithms. They determine grammatical structure or membership of an input sequence. They are different tasks from Thompson construction, which builds a token-recognizing NFA from a regular expression, and subset construction, which turns an NFA into a DFA.

Question sources

Compiler Design review questions · Question 2

Question 150 | NS0230

Given the grammar: E → E + T | T, T → T * F | F, F → (E) | id - this grammar enforces:

A. Right-associativity for both + and * |

B. Left-associativity for both + and *, with * having higher precedence than +

C. Right-associativity for + and left-associativity for * |

D. No precedence or associativity rules

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Both operators associate left; multiplication has higher precedence.

COMP10 states exactly these productions and their effect. Multiplication occurs within T below the addition level E, so it binds more tightly. The recursive operand on the left of E→E+T and T→T*F enforces left association for each operator. Parentheses in F can override the default grouping.

Why each choice is right or wrong

A - Incorrect: The recursive nonterminal is on the left, producing left rather than right association.

B - Correct: Separate T multiplication and E addition levels give higher multiplication precedence and left association.

C - Incorrect: Addition is left-recursive too, so it does not have the claimed right association.

D - Incorrect: The grammar explicitly encodes both precedence levels and associativity.

Primary note proof

Study-note section COMP10

> Use E→E+T|T, T→T*F|F, F→(E)|id to give * higher precedence and both operators left associativity. For subtraction, a−b−c then groups (a−b)−c.

Question sources

Compiler Design review questions · Question 3

Question 151 | NS0231

For a reduced CFG (all symbols reachable and productive), which conditions prevent it from being LL(1)? Select all that apply. In A, overlap means a shared terminal in competing RHS FIRST sets.

A. Two productions for the same non-terminal have overlapping FIRST sets |

B. The grammar contains a left-recursive rule

C. The grammar is ambiguous |

D. A and B only

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A, B, C: Two productions for the same non-terminal have overlapping FIRST sets |; The grammar contains a left-recursive rule; The grammar is ambiguous |

A shared FIRST terminal for alternatives of one nonterminal creates a predictive-table collision. Reachable left recursion prevents ordinary LL(1) prediction, and an ambiguous useful grammar cannot have a unique LL(1) parse. Thus A, B and C are all obstructions. The combined D alternative wrongly says only A and B, excluding ambiguity, so it is not a complete correct selection.

Why each choice is right or wrong

A - Correct: Shared FIRST terminals create competing table entries.

B - Correct: Reachable left recursion rules out ordinary LL(1) parsing.

C - Correct: A useful ambiguous grammar is not LL(1).

D - Incorrect: The word only wrongly excludes the valid ambiguity obstruction.

Primary note proof

Study-note section COMP14

> For every A→α, put it in M[A,a] for each terminal a in FIRST(α). If α is nullable, also enter it under every symbol in FOLLOW(A). More than one production in a cell is a conflict. Pairwise SELECT sets must be disjoint; SELECT(α)=FIRST(α)−{ε}, plus FOLLOW(A) when α is nullable. This also detects two nullable alternatives.

Study-note section COMP12

> Indirect left recursion follows a cycle through other nonterminals and also needs removal for ordinary predictive descent.

Study-note section COMP12

> LR parsers can handle suitable left-recursive grammars; ordinary unmodified recursive descent can recurse forever before consuming input.

Study-note section COMP18

> Deterministic LL/LR grammars must be unambiguous; a general ambiguous grammar requires rewriting, disambiguation policy or a generalized method such as GLR.

Question sources

Compiler Design review questions · Question 4

Question 152 | NS0232

Which of the following is an example of a synthesized attribute in attribute grammars?

A. The type of an expression computed from its sub-expressions |

B. The scope environment passed down to a block

C. The inherited type context for a function body |

D. The enclosing loop level passed to a break statement

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer A: Expression type computed from subexpressions is synthesized.

COMP24 defines synthesized attributes by flow from children to a parent. Computing an expression's type from its subexpressions fits this direction. Scope, type context and enclosing-loop information passed down from surrounding constructs instead supply inherited context. The direction of dependency, rather than the fact that types are involved, distinguishes these choices.

Why each choice is right or wrong

A - Correct: The parent expression derives its type from child subexpressions, which is synthesized upward flow.

B - Incorrect: A scope environment passed down to a block is inherited contextual information.

C - Incorrect: A type context supplied to a function body flows from its surroundings, so it is inherited.

D - Incorrect: Enclosing-loop context is passed inward to a break statement rather than synthesized from its children.

Primary note proof

Study-note section COMP24

> A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an inherited or synthesized attribute. Synthesized attributes flow from children to a parent; inherited attributes carry context from parent or siblings.

Study-note section COMP24

For E→E1+T, E.val=E1.val+T.val synthesizes an expression value. A declaration's type environment passed into its body is inherited. S-attributed definitions use only synthesized attributes and fit LR reduction-time evaluation naturally; suitable LL parsers can also compute them. L-attributed definitions restrict dependencies so a left-to-right traversal can evaluate inherited context. Attribute dependency cycles can prevent evaluation.

Question sources

Compiler Design review questions · Question 5

Question 153 | NS0233

What is the primary purpose of Three-Address Code (TAC) as an intermediate representation?

A. To directly execute on the target CPU |

B. To provide a machine-independent representation suitable for optimization and code generation

C. To replace the symbol table |

D. To encode only arithmetic expressions

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer B: An optimization-friendly intermediate representation for later target generation.

COMP26 gives reuse and separation as the purpose of IR and describes TAC as simple instruction records. TAC is normally not directly executed by a native CPU, does not replace declaration metadata and is not limited to arithmetic: branches and assignments are also representable. B states its architectural optimization/code-generation role most directly.

Why each choice is right or wrong

A - Incorrect: TAC is normally intermediate and requires lowering rather than direct execution on the target CPU.

B - Correct: TAC provides analyzable intermediate operations supporting reuse, optimization and later target-code generation.

C - Incorrect: A symbol table retains declaration metadata and is not replaced by the instruction representation.

D - Incorrect: TAC can encode control transfers and other operations, not only arithmetic expressions.

Primary note proof

Study-note section COMP26

IR separates source analysis from target implementation, allowing optimizers and backends to be reused. It should be easy to generate, analyze, transform and lower.

Study-note section COMP26

Ordinary TAC arithmetic has at most one operation, two sources and one destination: t1=c*d; t2=b+t1; a=t2. Quadruples store (op,arg1,arg2,result); triples omit an explicit result field and refer to instruction positions. Indirect triples reference a separate ordering list, easing rearrangement. Trees describe expressions; DAGs share common subexpressions, while quadruples/triples are instruction-record structures.

Question sources

Compiler Design review questions · Question 6

Question 154 | NS0234

The 'maximal munch' rule in lexical analysis means:

A. The lexer always produces the shortest possible token |

B. The lexer always produces the longest possible token matching the current position

C. Keyword tokens are prioritized over identifiers |

D. Comments are included in the token stream

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer B: Choose the longest matching lexeme at the current position.

COMP07 defines longest matching lexeme selection at each input position. Equal-length matches use rule order, which is a separate tie-breaking rule; keywords do not override a longer identifier such as ifx. Comments are normally skipped rather than retained merely because maximal munch is used. This establishes B as the definition.

Why each choice is right or wrong

A - Incorrect: Shortest-match selection is the opposite of maximal munch.

B - Correct: The lexer consumes the longest available match beginning at the current input position.

C - Incorrect: Keyword priority can break equal-length rule ties but does not define longest match.

D - Incorrect: Comment retention is independent of longest match; ordinary comments are skipped.

Primary note proof

Study-note section COMP07

> At each input position, choose the longest matching lexeme; if equal lengths match multiple flex rules, choose the earliest rule. Thus ifx is an identifier rather than if followed by x; <= and ++ are each one operator token. An implementation may recognize an identifier first and then consult a keyword table.

Question sources

Compiler Design review questions · Question 7

Question 155 | NS0235

In bottom-up parsing, a 'shift-reduce conflict' occurs when:

A. There are two possible reductions for the same stack top |

B. The parser cannot decide whether to shift the next input token or reduce the current stack contents

C. The grammar is left-recursive |

D. The FOLLOW set is empty

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: A competing shift and reduction on the same situation.

COMP17 defines a shift/reduce conflict as a choice between consuming a token and reducing. Two competing reductions describe reduce/reduce instead. Left recursion may be entirely acceptable to LR, and an empty FOLLOW set is not the definition of this table conflict. The conflict concerns competing actions for the same state and lookahead.

Why each choice is right or wrong

A - Incorrect: Two possible reductions define a reduce/reduce conflict, a different conflict type.

B - Correct: The same parser situation permits both shifting input and reducing the recognized right-hand side.

C - Incorrect: LR can handle suitable left recursion; that property does not itself define a shift/reduce conflict.

D - Incorrect: FOLLOW-set emptiness does not express competing shift and reduction actions.

Primary note proof

Study-note section COMP17

> SLR(1) uses LR(0) states and FOLLOW(A) for reduction by a completed A production. Canonical LR(1) attaches context-specific lookahead to items. LALR(1) merges canonical states with the same LR(0) core, combining lookaheads; merging can introduce reduce/reduce conflicts. The standard grammar-class hierarchy is LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1), with increasing capability and often table size.

Study-note section COMP17

> A shift/reduce conflict offers both consuming a token and reducing; reduce/reduce offers two reductions. An SLR or LALR conflict does not by itself prove the grammar ambiguous. Canonical LR(1) handles LR(1) grammars, not every unambiguous CFG. Precedence and associativity can resolve suitable expression conflicts; changing parser names alone does not repair all ambiguity.

Question sources

Compiler Design review questions · Question 8

Question 156 | NS0236

Which optimization technique moves loop-invariant computations outside the loop body?

A. Strength reduction |

B. Loop unrolling

C. Loop-invariant code motion (LICM) |

D. Dead code elimination

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer C: Loop-invariant code motion.

COMP32 explains that an expression whose inputs remain unchanged can be moved to a preheader when safe on newly executed paths. That is loop-invariant code motion. Strength reduction changes operation cost, unrolling replicates the body, and dead-code elimination removes unused or unreachable work. Hoisting also requires safety, not merely unchanged operands.

Why each choice is right or wrong

A - Incorrect: Strength reduction substitutes a cheaper equivalent operation rather than necessarily moving it outside a loop.

B - Incorrect: Unrolling replicates loop bodies instead of hoisting invariant computations.

C - Correct: LICM moves invariant computations to an appropriate outside location when executing them there is safe.

D - Incorrect: Dead-code elimination removes unnecessary computations rather than relocating a needed invariant value.

Primary note proof

Study-note section COMP32

> A loop-invariant expression has unchanged inputs during a loop. Hoist it to a preheader only when evaluation is safe on the paths where it will newly run. Induction variables change by a loop-invariant increment, such as i=i+1. For repeated t=4*i, maintain a running t increased by 4 when equivalent, reducing repeated multiplication.

Study-note section COMP32

> Loop fusion or jamming combines loops with compatible bounds and safe dependence order. Unrolling replicates a body to reduce branch overhead and expose parallelism; handle remaining iterations and increased code size. Loop-invariant code motion, fusion, unrolling and strength reduction are different transformations. A finite taxonomy needs a criterion: machine dependent/independent versus local/global/loop are different classifications.

Question sources

Compiler Design review questions · Question 9

Question 157 | NS0237

What does 'register spilling' mean in the context of register allocation?

A. Allocating the same register to two variables |

B. Storing a variable in memory instead of a register because no register is available

C. Using floating-point registers for integer values |

D. Eliminating unnecessary register copies

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer B: Store selected values in memory and insert loads/stores when registers do not suffice.

COMP36 says spilling places selected values in memory when the register assignment cannot fit available registers. Loads and stores then connect memory-resident values to instructions needing registers. Register sharing can be legal for nonoverlapping live ranges, while copy elimination and choosing a different register class are separate operations rather than this storage fallback.

Why each choice is right or wrong

A - Incorrect: Register sharing may be legal for disjoint live ranges and does not itself describe spilling.

B - Correct: Spilling moves selected values to memory because a feasible register-only assignment is unavailable.

C - Incorrect: Using another register class is not the definition of writing spilled values to memory.

D - Incorrect: Removing redundant register copies is a different optimization from memory-backed spilling.

Primary note proof

Study-note section COMP36

> A value is live where it can be used before redefinition. In the interference model, values that must coexist in distinct registers are connected by an edge; each register is a color. Nonoverlapping live ranges may share a register. If a valid assignment cannot fit the available registers, spilling places selected values in memory and inserts loads/stores.

Study-note section COMP36

The general graph k-colorability decision problem is NP-complete for fixed k≥3; optimal register assignment/spill minimization is a hard optimization problem in general, so practical allocators use heuristics.

Question sources

Compiler Design review questions · Question 10

Question 158 | NS0238

For Java numeric conversions such as int→long and int→float, which statement correctly describes widening?

A. Converting a wider type to a narrower type, possibly losing information |

B. Converting to a numeric type with a wider representable range; some widening conversions can still lose precision.

C. Declaring a variable with an explicit type annotation |

D. Assigning a value to a pointer of a different type

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer B: Converting to a numeric type with a wider representable range; some widening conversions can still lose precision.

Java int→long is widening and exact, while int→float is also widening even though some large integer values round. Widening therefore does not universally guarantee preservation of every bit of precision. Narrowing goes toward a more restricted representation; type annotations and arbitrary pointer changes are separate concepts. Option B captures the relevant larger-range conversion without making the false no-loss promise.

Why each choice is right or wrong

A - Incorrect: This describes narrowing rather than widening.

B - Correct: Widening can increase range while still losing precision, as int→float illustrates.

C - Incorrect: An explicit annotation declares a type rather than converting a value.

D - Incorrect: An arbitrary pointer-type change does not define Java numeric widening.

Primary note proof

Study-note section COMP23

Coercion is an implicit conversion permitted by the language; a cast requests an explicit conversion. Widening often increases representable range, but can still lose precision: Java int→float is widening although large integers round. Java int→long is exact. Narrowing can lose range or fractional information.

Study-note section COMP23

> In C++ ordinary int x=3.14 converts to 3 when representable; it is not inherently a type error. int x{3.14} is rejected as narrowing. C++ floating-to-integer conversion discards the fraction and has undefined behavior if the truncated value is unrepresentable. Java ordinary double→int assignment needs a permitted explicit cast. Mixing char and int is ordinarily valid after promotion in C++/Java; do not import a fictitious universal ban.

Question sources

Compiler Design review questions · Question 11

Question 159 | NS0239

Which of the following data structures is most commonly used to implement a symbol table for fast insertion and lookup?

A. Sorted linked list |

B. Hash table

C. Binary search tree |

D. Stack

Hint

Ask whether the required information is a declaration’s identity and attributes, program structure or the ordered source tokens.

Answer B: A hash table.

COMP19 directly identifies hashing as giving expected constant-time insertion and lookup under suitable hashing. A stack supports nested scope order but is not the primary keyed mapping. Lists and trees are possible alternatives, yet neither is the specific common expected-constant-time structure emphasized in the note and requested by the stem.

Why each choice is right or wrong

A - Incorrect: A sorted linked list generally requires linear search and does not supply the highlighted expected constant lookup.

B - Correct: Hash tables commonly implement expected O(1) symbol insertion and lookup.

C - Incorrect: A search tree is possible, but the note specifically identifies hashing for the usual fast keyed implementation.

D - Incorrect: A stack organizes scope nesting rather than providing fast arbitrary name lookup within a scope.

Primary note proof

Study-note section COMP19

> A symbol table maps names or symbol identities to attributes: type, scope, width, storage offset, parameter signature, constant information or linkage. A hash table gives expected O(1)

insertion/lookup with suitable hashing; a stack of scope tables supports lexical nesting. Search the nearest scope first, insert declarations in the current scope, and remove visibility when leaving it.

Study-note section COMP19

Scanning may intern a spelling first; semantic analysis determines the declaration and type. Implementations differ, so first interaction is conventional rather than universal. Not every punctuation token needs a symbol-table entry; literals may use a separate literal table. A hash table is an implementation, not the information’s semantic role.

Question sources

Compiler Design review questions · Question 12

Question 160 | NS0240

A grammar is said to be 'left-factored' when:

A. All productions are left-recursive |

B. Productions for the same non-terminal that share a common prefix are rewritten to factor out the prefix

C. The grammar has no ε-productions |

D. The grammar can be parsed by a bottom-up parser only

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Common prefixes are factored out of alternatives.

COMP12 rewrites $A \to \alpha\beta \mid \alpha\gamma$ as $A \to \alpha A'$ and $A' \to \beta \mid \gamma$. The transformation postpones the alternative choice until after consuming the common prefix. It does not mean making productions left-recursive, removing all epsilon rules or restricting parsing to bottom-up methods; left factoring alone also does not guarantee LL(1).

Why each choice is right or wrong

A - Incorrect: Left recursion and left factoring are distinct properties and transformations.

B - Correct: Factoring extracts shared prefixes and delays selection between the remaining suffix alternatives.

C - Incorrect: Factoring does not require removing every epsilon production.

D - Incorrect: Factored grammars can support top-down prediction; bottom-up exclusivity is not a defining condition.

Primary note proof

Study-note section COMP12

> Left factoring rewrites $A \to \alpha\beta \mid \alpha\gamma$ using:
> $A \to \alpha A'$
> $A' \to \beta \mid \gamma$
> It postpones a choice until after the common prefix. It does not by itself remove ambiguity or guarantee LL(1). LR parsers can handle suitable left-recursive grammars; ordinary unmodified recursive descent can recurse forever before consuming input.

Question sources

Compiler Design review questions · Question 13

Question 161 | NS0241

What is 'constant propagation' in compiler optimization?

A. Evaluating constant expressions at compile time |

B. Replacing uses of a variable with its known constant value at each program point

C. Removing code that has no effect on program output |

D. Moving constants from memory to registers

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B: Replace uses with their known constant values.

COMP30 gives x=3; y=x+2 becoming y=3+2 before folding. Replacing x by its known value is propagation; evaluating 3+2 is folding. Removing unnecessary code or choosing register locations addresses different transformations. The program-point qualification matters because the value must still be known on all relevant incoming paths.

Why each choice is right or wrong

A - Incorrect: Evaluating an already constant expression is constant folding, which can follow propagation.

B - Correct: Propagation substitutes a variable’s established constant value at uses where that fact holds.

C - Incorrect: Removing effect-free unused or unreachable work is dead-code elimination.

D - Incorrect: Memory-to-register placement is not the substitution of a known compile-time value.

Primary note proof

Study-note section COMP30

> Constant folding evaluates constant expressions during compilation: 3*4→12. Constant propagation substitutes known values: x=3; y=x+2 becomes y=3+2 before folding. Copy propagation substitutes a copied name while its source remains unchanged. Dead-code elimination removes unreachable code or computations whose values are unused and whose evaluation has no necessary side effects.

Question sources

Compiler Design review questions · Question 14

Question 162 | NS0242

In an LR parsing table, what does a 'reduce/reduce conflict' indicate?

A. Two different productions can both be used to reduce the same stack contents with the same lookahead |

B. The parser must both shift and reduce on the same input symbol

C. The grammar has a dangling else problem |

D. The grammar contains ε-productions

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A: Two competing reductions for the same parser state/lookahead.

COMP17 defines reduce/reduce as two reductions being available. In a parsing table that means the same state and lookahead permit different completed productions. A shift competing with reduction is another conflict type. Dangling else is a grammar ambiguity example, and epsilon productions alone do not guarantee any reduce/reduce collision.

Why each choice is right or wrong

A - Correct: The table offers two different production reductions for the same state and lookahead.

B - Incorrect: Simultaneous shift and reduce actions define shift/reduce rather than reduce/reduce.

C - Incorrect: Dangling else commonly creates an attachment ambiguity, not the definition of two competing reductions.

D - Incorrect: Epsilon rules can appear in conflict-free grammars and do not alone imply this collision.

Primary note proof

Study-note section COMP17

> SLR(1) uses LR(0) states and FOLLOW(A) for reduction by a completed A production. Canonical LR(1) attaches context-specific lookahead to items. LALR(1) merges canonical states with the same LR(0) core, combining lookaheads; merging can introduce reduce/reduce conflicts. The standard grammar-class hierarchy is LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1), with increasing capability and often table size.

Study-note section COMP17

> A shift/reduce conflict offers both consuming a token and reducing; reduce/reduce offers two reductions. An SLR or LALR conflict does not by itself prove the grammar ambiguous. Canonical LR(1) handles LR(1) grammars, not every unambiguous CFG. Precedence and associativity can resolve suitable expression conflicts; changing parser names alone does not repair all ambiguity.

Question sources

Compiler Design review questions · Question 15

Question 163 | NS0243

Which compiler phase is primarily responsible for detecting the incompatible assignment in the Java declaration int x = "hello";?

A. Lexical analysis |

B. Syntax analysis

C. Semantic analysis |

D. Code generation

Hint

Check operand compatibility and permitted conversions under the stated language rules; distinguish grammatical structure from semantic meaning.

Answer C: Semantic analysis |

The declaration has recognizable tokens and a valid declaration-shaped syntax, but assigns a string to an integer variable. Under Java's type rules this violates assignment compatibility, so semantic analysis and type checking identify the error. Scanning recognizes the identifier, type keyword and literal; parsing organizes them. Code generation uses the checked representation later. The named language removes any assumption that every statically typed language has identical implicit conversions.

Why each choice is right or wrong

A - Incorrect: The individual tokens are recognizable; the error is their type compatibility.

B - Incorrect: The declaration's grammar structure is not the primary problem.

C - Correct: Semantic type checking rejects the string-to-integer assignment.

D - Incorrect: Code generation follows the frontend type check rather than defining this compatibility rule.

Primary note proof

Study-note section COMP08

> A lexical error prevents forming a legal token, such as an invalid character or unterminated literal. A syntax error is an illegal token arrangement, such as a missing delimiter. A semantic error violates name, scope or type rules after grammatical recognition. A wrong algorithm can compile successfully; compilers do not detect all logical mistakes.

Study-note section COMP22

> int x="hello" is a type mismatch in ordinary C++/Java; division by zero and bounds violations are not automatically operand-type mismatches.

Question sources

Compiler Design review questions · Question 16

Question 164 | NS0244

In Thompson's construction for building an NFA from a regular expression, the NFA for the concatenation r1r2 is built by:

A. Merging the start states of both NFAs |

B. Adding an ε-transition from the accepting state(s) of the NFA for r1 to the start state of the NFA for r2

C. Creating a new start state with ε-transitions to both NFAs |

D. Taking the cross product of the two NFAs

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer B: Connect the first fragment’s accepting end to the second start with ε.

COMP06 directly gives the concatenation rule: connect r’s final state to s’s start by epsilon. This forces any complete path through the first fragment to continue through the second in order. Branching epsilon paths instead implement alternation; merged starts and product constructions do not implement the specified Thompson concatenation step.

Why each choice is right or wrong

A - Incorrect: Merging starts does not force a complete r1 match followed by an r2 match.

B - Correct: The epsilon connection joins the first accepting end to the second start without consuming input.

C - Incorrect: A new branching start with epsilon alternatives implements union rather than concatenation.

D - Incorrect: A state cross product is not the Thompson fragment connection for concatenation.

Primary note proof

Study-note section COMP06

> A common generator pipeline is regular expressions → NFA using Thompson construction → DFA using subset construction; minimization is optional. For concatenation rs, connect r’s final state to s’s start by ε. Alternation introduces branching ε paths; star adds a way to skip and repeat the fragment.

Study-note section COMP06

> The ε-closure of states includes those reachable by zero or more ε edges. A DFA state is a set of NFA states; its a-transition is ε-closure(move(set,a)). Start with the NFA start closure, repeatedly add unseen subsets, and mark subsets containing an NFA final state as accepting. Token identity and rule priority annotate accepting states; the empty subset acts as a dead state.

Question sources

Compiler Design review questions · Question 17

Question 165 | NS0245

What is the 'dangling else' problem?

A. An else branch that can never be reached due to preceding conditions |

B. Ambiguity in grammars that allows an else to be matched with either of two if statements

C. An else clause that lacks a body |

D. An else used outside an if statement

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Ambiguous attachment of else to one of two nested ifs.

COMP10 describes an else that can attach to either nested if and resolves it with nearest-unmatched-if policy or a grammar rewrite. That grammatical ambiguity is the dangling-else problem. It does not mean unreachable code, a missing body or an else outside any if, which are different semantic or syntactic situations.

Why each choice is right or wrong

A - Incorrect: Unreachability concerns control-flow feasibility, not the competing grammatical attachment of else.

B - Correct: The ambiguity is whether else belongs to the inner or outer unmatched if.

C - Incorrect: A missing else body is a syntax defect rather than the classic attachment ambiguity.

D - Incorrect: An else outside any if is not the two-attachment situation defining dangling else.

Primary note proof

Study-note section COMP10

> The dangling-else grammar permits else to attach to either nested if; a nearest-unmatched-if rule or grammar rewrite resolves it. Taking a leftmost derivation by itself does not resolve ambiguous interpretation.

Question sources

Compiler Design review questions · Question 18

Question 166 | NS0246

Which representation explicitly uses basic blocks as vertices and possible transfers as edges, making it the direct structural basis for dominator and natural-loop analysis?

A. Abstract Syntax Tree (AST) |

B. Three-Address Code (TAC)

C. Static Single Assignment (SSA) form |

D. Control Flow Graph (CFG)

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer D: Control Flow Graph (CFG)

A control-flow graph explicitly records basic blocks and their possible successor transfers. Dominator relationships describe which blocks occur on every path to another block; back edges to dominating headers identify natural loops. TAC can populate the blocks, and SSA can coexist with this graph, but neither alone names the requested vertex/edge structure. An AST emphasizes source syntax rather than this control-flow representation.

Why each choice is right or wrong

A - Incorrect: An AST represents source hierarchy rather than the specified block-transfer graph.

B - Incorrect: TAC gives instructions, not by itself the requested graph.

C - Incorrect: SSA is a definition-naming discipline that can coexist with a CFG.

D - Correct: A CFG explicitly provides the specified blocks, edges and path structure.

Primary note proof

Study-note section COMP28

> A basic block is a maximal straight-line instruction sequence entered at its beginning and exited at its end. Leaders are the first instruction, every jump target, and each instruction after a branch. Start a block at each leader and continue until the next leader.

Study-note section COMP28

> A control-flow graph has blocks as vertices and possible next-block transfers as edges, including fall-through. A node D dominates N when every entry-to-N path passes through D. For an edge N→H where H dominates N, H is a loop header of a natural loop. This structure supports loop analysis. An expression DAG records value dependencies, not arbitrary control flow.

Question sources

Compiler Design review questions · Question 19

Question 167 | NS0247

In Static Single Assignment (SSA) form, what is the purpose of a 'φ (phi) function'?

A. To represent a conditional branch in the control flow graph |

B. To select the correct version of a variable at join points where multiple control flow paths merge

C. To mark the entry point of a function |

D. To represent a loop-back edge in the CFG

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer B: Select the incoming variable version corresponding to the predecessor taken.

COMP27 defines phi at a join as choosing an incoming value according to the actual predecessor path. It is not a conditional branch instruction, a function-entry marker or a back edge, and does not average its operands. The function reconciles distinct definition names when different paths reach the same use.

Why each choice is right or wrong

A - Incorrect: A phi function selects an incoming value; it does not transfer control between blocks.

B - Correct: It merges definition versions by selecting the value associated with the predecessor actually taken.

C - Incorrect: Function entry is a control-flow location, not the value-selection purpose of phi.

D - Incorrect: A back edge represents loop control transfer rather than a merge of SSA values.

Primary note proof

Study-note section COMP27

> A conditional expression can generate if a<b goto Ltrue; goto Lfalse; labels join control paths. Short-circuit AND evaluates the right operand only after a true left operand; OR only after a false one. Translating both eagerly can change side effects. For a zero-based array with element width w and lower bound L, element i uses base+(i−L)*w, assuming fixed-width contiguous storage.

Study-note section COMP27

> SSA gives each definition a distinct name. At a join, x3=φ(x1 from B1,x2 from B2) chooses the incoming value according to the predecessor actually taken; it is not an arithmetic average. SSA is a naming discipline over IR and commonly coexists with a control-flow graph.

Question sources

Compiler Design review questions · Question 20

Question 168 | NS0248

The FOLLOW set of the start symbol S always contains:

A. ε (epsilon) |

B. The end-of-input marker $

C. The first terminal of every production |

D. Nothing; FOLLOW(S) is always empty

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: The end-of-input marker $.

COMP13 explicitly places $ in FOLLOW(S), representing that a completed start-symbol derivation can be followed by end of input. FOLLOW never contains epsilon. Possible initial terminals belong to FIRST, not automatically FOLLOW. Therefore the marker is guaranteed even when other following terminals depend on the grammar.

Why each choice is right or wrong

A - Incorrect: FOLLOW contains terminals and possibly the end marker, never epsilon.

B - Correct: The end-of-input marker is placed in FOLLOW of the start symbol by initialization.

C - Incorrect: Initial terminals are FIRST information and are not automatically following terminals.

D - Incorrect: FOLLOW(S) is not empty because it contains the end marker.

Primary note proof

Study-note section COMP13

> FIRST(α) contains terminals that can begin strings derived from α, plus ε when α can vanish. For a sequence, add FIRST of each symbol except ε, continue while symbols are nullable, and add ε only if every symbol is nullable.

Study-note section COMP13

> FOLLOW(A) contains possible next terminals after A, plus end marker $ where appropriate; never ε. Put $ in FOLLOW(S). For B→αAβ, add FIRST(β)−{ε} to FOLLOW(A); if β is nullable, add FOLLOW(B). Repeat all rules until no sets grow.

Question sources

Compiler Design review questions · Question 21

Question 169 | NS0249

For an unsigned fixed-width integer x with width greater than three and modular arithmetic, which optimization replaces x = x * 8 with x = x << 3?

A. Constant folding |

B. Dead code elimination

C. Strength reduction |

D. Loop unrolling

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer C: Strength reduction |

The transformation replaces multiplication by a power of two with an equivalent shift under the stated unsigned model. This is strength reduction when the target makes that replacement cheaper. It does not evaluate a constant-only expression because x remains variable, does not remove an unused statement, and does not replicate loop iterations. The explicit arithmetic model matters because signed-language overflow or shift rules can make a superficially similar rewrite invalid.

Why each choice is right or wrong

A - Incorrect: The operand x remains variable, so this is not evaluation of an all-constant expression.

B - Incorrect: The assignment remains and still computes a needed result.

C - Correct: Replacing an operation by an equivalent cheaper form is strength reduction.

D - Incorrect: No loop body is replicated.

Primary note proof

Study-note section COMP31

> Algebraic simplification removes identities when valid: integer x+0→x. Strength reduction replaces an operation with an equivalent cheaper form, such as unsigned x*8→x<<3 for a valid width and shift count. Target costs determine whether it is actually cheaper.

Study-note section COMP33

> Signed C/C++ overflow and shift rules differ from unsigned modular arithmetic; a mathematical identity is not automatically a legal transformation.

Study-note section COMP32

> Unrolling replicates a body to reduce branch overhead and expose parallelism; handle remaining iterations and increased code size.

Question sources

Compiler Design review questions · Question 22

Question 170 | NS0250

In a compiler's code generator, 'calling convention' refers to:

A. The order in which source files are compiled |

B. The agreed-upon protocol for how functions pass arguments, return values, and manage the stack

C. The naming convention for generated labels |

D. The format of debug information in the object file

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer B: The protocol for arguments, results, registers and stack management.

COMP37 defines the locations of arguments/results, saved-register responsibilities, stack alignment and frame duties as the calling convention. These permit caller and callee to agree on the runtime call interface. Source-file build order, label spelling and debug format concern other compiler/toolchain conventions rather than this execution protocol.

Why each choice is right or wrong

A - Incorrect: Compilation order concerns build organization rather than the runtime interface between callers and callees.

B - Correct: The calling convention specifies argument/result passing and register/stack responsibilities.

C - Incorrect: Label naming is separate from the runtime storage and control-transfer call protocol.

D - Incorrect: Debug information format is not the protocol used to invoke and return from functions.

Primary note proof

Study-note section COMP37

> Static storage lasts throughout execution; stack frames support nested calls and recursion; heap storage supports lifetimes beyond a call. A frame may contain parameters, return address, saved registers, locals and temporaries. Sizes, alignment and offsets are target-dependent; a two-byte int is an example, not a C++ guarantee.

Study-note section COMP37

> A calling convention specifies argument and result locations, caller/callee-saved registers, stack alignment and frame responsibilities. Escape analysis asks whether an object/reference can

outlive or be accessed outside a region such as its allocating call. Proven nonescaping allocations may use stack storage or be eliminated by scalar replacement, provided identity and lifetime semantics remain correct.

Question sources

Compiler Design review questions · Question 23

Question 171 | NS0251

What does 'peephole optimization' involve?

A. Optimizing entire functions at once |

B. Examining and replacing small sequences of generated instructions with more efficient equivalents

C. Inlining small functions at call sites |

D. Eliminating unreachable basic blocks

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B: Examine short instruction sequences and replace them with better equivalents.

COMP31 defines peephole optimization by short instruction windows and lists redundant-load removal, machine idioms and jump-chain shortening. B states that method directly. Whole-function optimization has a broader scope; inlining changes calls to bodies, and unreachable-block removal is a particular control-flow simplification rather than the definition of examining small windows.

Why each choice is right or wrong

A - Incorrect: Whole-function scope is broader than the short instruction windows defining peephole optimization.

B - Correct: The optimizer inspects short instruction sequences and substitutes equivalent more efficient forms.

C - Incorrect: Inlining expands calls into bodies rather than defining the local instruction-window method.

D - Incorrect: Unreachable-block removal is one control-flow optimization, not the defining peephole inspection mechanism.

Primary note proof

Study-note section COMP31

> Peephole optimization examines short instruction windows. It can remove redundant loads/stores, replace machine idioms and shorten jump chains: jump L1 where L1 immediately jumps L2 can become jump L2.

Question sources

Compiler Design review questions · Question 24

Question 172 | NS0252

Which statement correctly compares LALR(1) with SLR(1) and canonical LR(1), allowing the case where no equal-core canonical states need merging?

A. LALR(1) is weaker than SLR(1) |

B. LALR(1) has no more states than canonical LR(1) and can resolve some conflicts SLR(1) cannot.

C. LALR(1) and SLR(1) produce identical parsing tables for all grammars |

D. LALR(1) requires more lookahead than SLR(1)

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: LALR(1) has no more states than canonical LR(1) and can resolve some conflicts SLR(1) cannot.

LALR combines lookaheads of canonical LR(1) states with equal LR(0) cores. This gives no more states than the canonical machine, with a reduction when distinct equal-core states merge. LALR can handle some grammars beyond SLR because its lookaheads are more precise. Both use one lookahead token, and their tables are not identical for every grammar.

Why each choice is right or wrong

A - Incorrect: The hierarchy places SLR within LALR rather than the reverse.

B - Correct: This states both the correct state-count bound and the greater parsing capability.

C - Incorrect: The methods can use different reduction lookaheads and tables.

D - Incorrect: Both use one lookahead token.

Primary note proof

Study-note section COMP17

> SLR(1) uses LR(0) states and FOLLOW(A) for reduction by a completed A production. Canonical LR(1) attaches context-specific lookahead to items. LALR(1) merges canonical states with the same LR(0) core, combining lookaheads; merging can introduce reduce/reduce conflicts. The standard grammar-class hierarchy is LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1), with increasing capability and often table size.

Study-note section COMP17

> A shift/reduce conflict offers both consuming a token and reducing; reduce/reduce offers two reductions. An SLR or LALR conflict does not by itself prove the grammar ambiguous. Canonical LR(1) handles LR(1) grammars, not every unambiguous CFG. Precedence and associativity can resolve suitable expression conflicts; changing parser names alone does not repair all ambiguity.

Question sources

Compiler Design review questions · Question 25

Question 173 | NS0253

In data flow analysis, 'live variable analysis' is a:

A. Forward may-analysis |

B. Backward may-analysis

C. Forward must-analysis |

D. Backward must-analysis

Hint

Track which facts can hold along one path and which must hold along every path. Apply the specified direction, transfer rule and join operation.

Answer B: Backward may-analysis.

COMP29 explicitly calls liveness backward may-analysis. A value is live if some future path reads it before redefinition, so information flows backward from successor uses and combines possible uses rather than requiring every path. This distinguishes both the direction and may-versus-must dimensions requested by the options.

Why each choice is right or wrong

A - Incorrect: May is appropriate, but liveness propagates from future uses backward rather than forward.

B - Correct: Liveness asks whether some future path uses a value before redefinition, a backward may property.

C - Incorrect: Both direction and must quantification differ from the liveness definition.

D - Incorrect: Backward direction fits, but liveness requires a possible use, not use on every path.

Primary note proof

Study-note section COMP29

> Liveness is backward may-analysis: a value might be read before redefinition on some future path. OUT[B]=union of IN over successors; IN[B]=USE[B]∪(OUT[B]−DEF[B]). Iterate to a fixed point. Liveness supports register allocation and dead-value removal. Definite-assignment analysis asks whether initialization occurs on every path before use; reaching definitions alone does not prove this.

Question sources

Compiler Design review questions · Question 26

Question 174 | NS0254

The 'closure' operation in NFA-to-DFA subset construction computes:

A. The set of all NFA states reachable from a given set of states using only ε-transitions |

B. The set of all NFA states reachable using any number of any transitions

C. The minimal DFA equivalent to the NFA |

D. The FIRST set of a non-terminal

Hint

Follow the representation at each construction step and apply the pattern or state-set rules consistently, including empty cases.

Answer A: States reachable by zero or more ε transitions.

COMP06 defines epsilon closure as all states reachable without consuming input through zero or more epsilon edges. Zero moves includes the starting set itself. Arbitrary labeled transitions would consume symbols and belong to a different reachability relation; minimization and FIRST-set computation are separate algorithms. A therefore matches the closure used in subset construction.

Why each choice is right or wrong

A - Correct: Epsilon closure includes the starting set and every state reachable using only epsilon edges.

B - Incorrect: Arbitrary symbol transitions consume input and are not part of epsilon closure.

C - Incorrect: Closure supplies subsets during determinization; it does not itself minimize the resulting DFA.

D - Incorrect: FIRST is a grammar lookahead set, not epsilon reachability among NFA states.

Primary note proof

Study-note section COMP06

> A common generator pipeline is regular expressions → NFA using Thompson construction → DFA using subset construction; minimization is optional. For concatenation rs, connect r's final state to s's start by ε. Alternation introduces branching ε paths; star adds a way to skip and repeat the fragment.

Study-note section COMP06

> The ε-closure of states includes those reachable by zero or more ε edges. A DFA state is a set of NFA states; its a-transition is ε-closure(move(set,a)). Start with the NFA start closure, repeatedly add unseen subsets, and mark subsets containing an NFA final state as accepting. Token identity and rule priority annotate accepting states; the empty subset acts as a dead state.

Question sources

Compiler Design review questions · Question 27

Question 175 | NS0255

For a real-machine backend, which listed optimization chiefly depends on the target's instruction latencies and execution resources?

A. Constant folding |

B. Dead code elimination

C. Instruction scheduling |

D. Loop-invariant code motion

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer C: Instruction scheduling |

Instruction scheduling chooses a legal instruction order while considering target resources and latencies. It is therefore the target-dependent backend choice in this comparison. Constant folding computes known expressions, dead-code elimination removes unnecessary computation while preserving effects, and loop-invariant motion relocates safe unchanged expressions. Their validity depends on program semantics, but these three need not be organized around a particular processor's scheduling resources.

Why each choice is right or wrong

A - Incorrect: Folding evaluates known expressions from program semantics rather than scheduling machine resources.

B - Incorrect: Dead-code elimination uses reachability, use and side-effect information.

C - Correct: Instruction scheduling considers target execution resources and instruction latencies.

D - Incorrect: Invariant motion uses loop values, paths and safety conditions rather than target issue scheduling.

Primary note proof

Study-note section COMP-S04

> Instruction scheduling orders selected target instructions while preserving dependencies. A target-aware scheduler also considers instruction latencies and the processor resources needed by each operation. This is a backend optimization whose performance choices depend on the target machine, even when a reusable scheduling framework serves many targets.

Study-note section COMP-S04

> By contrast, constant folding, dead-code elimination and safe loop-invariant motion can be expressed on a target-neutral IR using values, control flow and side-effect rules. Target costs can still influence whether a legal transformation is worthwhile. Machine independence therefore describes the abstraction of a pass, not freedom from all language semantics or implementation costs.

Question sources

Compiler Design review questions · Question 28

Question 176 | NS0256

In a context-free grammar, a 'sentential form' is:

A. A string consisting only of terminal symbols |

B. Any string of terminals and non-terminals derivable from the start symbol

C. A string that cannot be further reduced |

D. The right-hand side of a production rule

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Any terminal/nonterminal string derivable from the start symbol.

COMP09 defines a sentential form as any string of terminals and nonterminals derivable from S. Terminal-only sentences are a special case, not the full definition. A production right side need not itself be reachable from S, and irreducibility does not define derivability. B captures the required reachability and permits both kinds of symbols.

Why each choice is right or wrong

A - Incorrect: Terminal-only strings are sentences, a subset of sentential forms rather than their full definition.

B - Correct: Derivability from the start symbol with terminals and possibly nonterminals is the defining property.

C - Incorrect: A sentential form can still contain expandable nonterminals; inability to reduce is not required.

D - Incorrect: A right-hand side is not necessarily reachable from the grammar's start symbol.

Primary note proof

Study-note section COMP09

> A context-free grammar G=(N,T,P,S) has nonterminals N, terminals T, productions P and start S. A production A→α rewrites one nonterminal. A sentential form is any string of terminals/nonterminals derivable from S; a sentence contains only terminals. A leftmost derivation rewrites the leftmost remaining nonterminal; a rightmost one rewrites the rightmost.

Study-note section COMP09

> A parse tree shows grammar productions: root S, internal nonterminals, and terminal leaves, with ε sometimes shown for empty productions. Its left-to-right yield is the input. An AST omits grammar scaffolding and punctuation; expression operands are leaves and operators internal nodes. The parser can build either representation, depending on implementation.

Question sources

Compiler Design review questions · Question 29

Question 177 | NS0257

What is 'function inlining' and what is its primary risk?

A. Replacing a function call with the function body; risk is incorrect semantics |

B. Replacing a function call with the function body; risk is increased code size (code bloat)

C. Moving a function to a different module; risk is linker errors |

D. Specializing a function for specific argument values; risk is type errors

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer B: Replace a call with its body; excessive code size is a risk.

COMP38 defines inlining as substituting a suitably renamed body while preserving argument evaluation, returns and side effects. It can remove call overhead but increase code size and instruction-cache pressure. Correct inlining preserves semantics; corruption is an implementation bug rather than its defining trade-off. Moving modules or specializing arguments describes different transformations.

Why each choice is right or wrong

A - Incorrect: Inlining must preserve semantics; incorrect behavior is a bug rather than the normal primary trade-off stated here.

B - Correct: Body substitution can reduce call overhead while increasing generated code size.

C - Incorrect: Moving a function between modules is not call-site body substitution.

D - Incorrect: Argument specialization is a different optimization and does not define general function inlining.

Primary note proof

Study-note section COMP38

> Inlining replaces a call with a suitably renamed copy of its body, preserving argument evaluation, returns and side effects. It can remove call overhead and expose optimization opportunities, but may increase code size and instruction-cache pressure. A recursive function cannot simply be expanded without limit.

Study-note section COMP38

> Virtual dispatch chooses an overridden method based on a receiver’s runtime class. A common implementation uses a per-class virtual-method table and an object pointer to the table; the call loads the selected entry. This is an implementation strategy, not a universal language requirement. Name mangling distinguishes linker symbols, often for overloads; it is not itself runtime method selection. Proven receiver types may permit devirtualization.

Question sources

Compiler Design review questions · Question 30

Question 178 | NS0258

Which of the following best describes the 'handle' in bottom-up parsing?

A. The leftmost terminal in the input string |

B. A substring of the current sentential form that matches the right-hand side of a production and whose reduction leads to a valid parse

C. The top element of the parsing stack |

D. The lookahead token used for parsing decisions

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: The RHS occurrence whose next reverse-rightmost reduction leads toward the valid parse.

COMP15 defines a handle as a production RHS occurrence whose reduction is the next reverse-rightmost derivation step. B is the closest complete description: its valid-parse condition must mean that next legitimate step, not merely any matching substring. Neither a single stack item nor the lookahead alone specifies that contextual production occurrence.

Why each choice is right or wrong

A - Incorrect: The leftmost terminal need not be the handle and does not identify a production RHS occurrence.

B - Correct: Read valid reduction as the next reverse-rightmost step; contextual validity distinguishes a handle from arbitrary matches.

C - Incorrect: A handle can span multiple symbols and is not simply the top stack element.

D - Incorrect: Lookahead guides action selection but is not the substring being reduced.

Primary note proof

Study-note section COMP15

> Bottom-up parsing starts from input and reduces toward S, constructing a rightmost derivation in reverse. A handle is a production RHS occurrence whose reduction is the next step of that reverse derivation. Not every substring matching a RHS is a handle in the current context.

Study-note section COMP15

> Shift pushes the next input token and a state; reduce replaces a recognized RHS by its LHS and consults GOTO; accept recognizes completion; error reports an invalid situation. For A→XYZ, pop three grammar symbols and their states, expose state q, then push A and GOTO[q,A]. “Go” is not a separate standard shift-reduce ACTION, although GOTO is essential table machinery.

Question sources

Compiler Design review questions · Question 31

Question 179 | NS0259

In the context of code generation, what is a 'basic block'?

A. A block of memory allocated on the heap |

B. A maximal straight-line sequence of instructions with one entry point and one exit point

C. A function or procedure in the source code |

D. A group of related variable declarations

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer B: A maximal straight-line sequence with entry at the beginning and exit at the end.

COMP28 directly defines this sequence as a basic block. Maximality means it is not arbitrarily shortened while retaining the same entry/exit structure. Heap allocation, whole functions and related declarations describe different entities; a function may contain many basic blocks separated by branch targets and control transfers.

Why each choice is right or wrong

A - Incorrect: A heap memory region is storage, not a straight-line control-flow unit.

B - Correct: This matches the maximal straight-line single-entry, end-exit definition in COMP28.

C - Incorrect: A function may contain branches and multiple basic blocks rather than one straight-line block.

D - Incorrect: Related declarations are not the instruction sequence defined by control-flow boundaries.

Primary note proof

Study-note section COMP28

> A basic block is a maximal straight-line instruction sequence entered at its beginning and exited at its end. Leaders are the first instruction, every jump target, and each instruction after a branch. Start a block at each leader and continue until the next leader.

Study-note section COMP28

> A control-flow graph has blocks as vertices and possible next-block transfers as edges, including fall-through. A node D dominates N when every entry-to-N path passes through D. For an edge N→H where H dominates N, H is a loop header of a natural loop. This structure supports loop analysis. An expression DAG records value dependencies, not arbitrary control flow.

Question sources

Compiler Design review questions · Question 32

Question 180 | NS0260

What is 'alias analysis' and why is it important for optimization?

A. Determining which variables have the same name in different scopes; important for scope resolution |

B. Determining whether two pointer expressions can refer to the same memory location; important because aliasing restricts reordering and CSE

C. Detecting variables with the same type; important for type coercion |

D. Finding duplicate function names; important for linking

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer B: Determine whether references may denote the same memory, constraining transformations.

COMP33 asks whether references can designate the same memory. A write through p can invalidate a value loaded through q unless nonaliasing or another proof permits reuse. Therefore alias facts constrain CSE and reordering. Identical spellings, types or function names concern scope, typing or linkage rather than memory-location overlap.

Why each choice is right or wrong

A - Incorrect: Equal spelling in different scopes concerns declaration identity rather than whether references share memory.

B - Correct: Potential shared locations determine whether writes can invalidate reused loads or reordered memory operations.

C - Incorrect: Equal types do not prove shared addresses and are not the alias-analysis question.

D - Incorrect: Duplicate linker names concern symbol resolution rather than potential overlap of referenced memory.

Primary note proof

Study-note section COMP33

> Optimization preserves defined observable behavior, including required side effects and exceptions. Alias analysis asks whether references can denote the same memory. After *p=5, a previous load through q cannot be reused unless p and q are known not to alias or an equivalent proof applies. Function calls can also invalidate memory facts.

Study-note section COMP33

> Do not move a potentially trapping division outside a loop that might execute zero times. Floating-point reassociation can change rounding and NaN behavior. Signed C/C++ overflow and shift rules differ from unsigned modular arithmetic; a mathematical identity is not automatically a legal transformation. Dependence analysis must show that fusion or scheduling preserves the required order of memory accesses.

Question sources

Compiler Design review questions · Question 33

Question 181 | NS0261

Which statement about recursive-descent parsers is TRUE?

A. They cannot handle left-recursive grammars without modification |

B. They inherently support all context-free grammars

C. They require a parsing table to function |

D. They build the parse tree bottom-up

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer A: Ordinary recursive descent needs modification to handle left recursion.

COMP12 explains that ordinary unmodified recursive descent can recurse forever on left recursion before consuming a token. Removing left recursion or changing the algorithm is therefore necessary for that direct implementation. Recursive descent is top-down and normally procedural, so it does not inherently require an explicit table or support every CFG unchanged.

Why each choice is right or wrong

A - Correct: Without transformation or algorithmic modification, left recursion can cause infinite recursive calls before input consumption.

B - Incorrect: Ordinary recursive descent is not an unrestricted parser for every context-free grammar.

C - Incorrect: Procedures can encode production choices directly; an explicit parsing table is not required.

D - Incorrect: Recursive descent expands productions top-down rather than constructing a reverse-rightmost bottom-up parse.

Primary note proof

Study-note section COMP12

> Immediate left recursion has $A \to A\alpha$. For $A \to A\alpha \mid \beta$, where β does not start with A, introduce a new nonterminal A' and use:
> $A \to \beta A'$
> $A' \to \alpha A' \mid \varepsilon$
> This preserves the generated language; tree shape changes, so semantic actions must preserve intended associativity. Indirect left recursion follows a cycle through other nonterminals and also needs removal for ordinary predictive descent.

Study-note section COMP12

Left factoring rewrites A→αβ|αγ using:
A→αA′
A′→β|γ
It postpones a choice until after the common prefix. It does not by itself remove ambiguity or guarantee LL(1). LR parsers can handle suitable left-recursive grammars; ordinary unmodified recursive descent can recurse forever before consuming input.

Question sources

Compiler Design review questions · Question 34

Question 182 | NS0262

In attribute grammar terminology, which of the following is an 'inherited' attribute?

A. The value of an expression node computed from its children |

B. The type environment passed from a function declaration down to its body

C. The size of an array determined from its element type and length |

D. The result type of an arithmetic operation

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer B: A type environment passed from a declaration into its body is inherited.

COMP24 explicitly gives the declaration’s type environment passed into its body as an inherited attribute. This carries context inward rather than computing a parent from child values. Expression values, array sizes computed from components and arithmetic result types are ordinarily synthesized examples when obtained from the node’s own child information.

Why each choice is right or wrong

A - Incorrect: Computing an expression from its children is synthesized upward flow.

B - Correct: The surrounding declaration supplies type context to the nested body, matching the explicit inherited example.

C - Incorrect: Computing size from element information and length is a synthesized result, not supplied surrounding context.

D - Incorrect: An operation’s result type derived from operands is normally synthesized from its child information.

Primary note proof

Study-note section COMP24

A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions. An annotated parse tree displays computed attribute values; it is not itself an

inherited or synthesized attribute. Synthesized attributes flow from children to a parent; inherited attributes carry context from parent or siblings.

Study-note section COMP24

For E→E1+T, E.val=E1.val+T.val synthesizes an expression value. A declaration’s type environment passed into its body is inherited. S-attributed definitions use only synthesized attributes and fit LR reduction-time evaluation naturally; suitable LL parsers can also compute them. L-attributed definitions restrict dependencies so a left-to-right traversal can evaluate inherited context. Attribute dependency cycles can prevent evaluation.

Question sources

Compiler Design review questions · Question 35

Question 183 | NS0263

A compiler emits the error 'variable used before initialization'. This is detected during:

A. Lexical analysis |

B. Syntax analysis

C. Semantic analysis / data flow analysis |

D. Code generation

Hint

Track which facts can hold along one path and which must hold along every path. Apply the specified direction, transfer rule and join operation.

Answer C: Semantic/data-flow analysis.

COMP29 distinguishes definite assignment, which asks whether initialization occurs on every incoming path, from reaching definitions alone. Such control/data-flow facts go beyond recognizing legal tokens or grammar. The error therefore belongs to semantic/data-flow analysis before target emission, even though exact acceptance and diagnostic rules depend on the language.

Why each choice is right or wrong

A - Incorrect: Scanning recognizes identifier tokens but does not prove initialization along every control-flow path.

B - Incorrect: A grammatically valid variable use can still occur before initialization.

C - Correct: Semantic data-flow analysis checks that required initialization precedes each use on every relevant path.

D - Incorrect: Target generation ordinarily consumes checked input rather than supplying the primary definite-assignment analysis.

Primary note proof

Study-note section COMP29

> Liveness is backward may-analysis: a value might be read before redefinition on some future path. OUT[B]=union of IN over successors; IN[B]=USE[B]∪(OUT[B]−DEF[B]). Iterate to a fixed point. Liveness supports register allocation and dead-value removal. Definite-assignment analysis asks whether initialization occurs on every path before use; reaching definitions alone does not prove this.

Question sources

Compiler Design review questions · Question 36

Question 184 | NS0264

In graph-coloring register allocation, what is an 'interference graph'?

A. A graph where nodes represent machine instructions and edges represent data dependencies |

B. A graph where nodes represent variables and edges connect pairs of variables that are simultaneously live

C. A graph where nodes represent registers and edges represent compatibility |

D. A graph representing the control flow of the program

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer B: A graph connecting values that must coexist in distinct registers.

COMP36 models concurrently needed values as vertices joined by interference edges, with physical registers represented by colors. Thus the variable/live-range graph in B is the intended model. Instruction dependencies and control-flow edges have different meanings; a graph of register compatibility reverses the allocation model's vertices and does not express live-value interference.

Why each choice is right or wrong

A - Incorrect: Instruction-dependency edges constrain scheduling rather than describing interference among live values.

B - Correct: Variables or live ranges form vertices; an edge requires distinct registers for simultaneously needed values.

C - Incorrect: Registers are colors in this model, not the vertices whose live ranges interfere.

D - Incorrect: A CFG connects basic blocks by possible transfers, not values by register interference.

Primary note proof

Study-note section COMP36

> A value is live where it can be used before redefinition. In the interference model, values that must coexist in distinct registers are connected by an edge; each register is a color. Nonoverlapping live ranges may share a register. If a valid assignment cannot fit the available registers, spilling places selected values in memory and inserts loads/stores.

Study-note section COMP36

> The general graph k-colorability decision problem is NP-complete for fixed k≥3; optimal register assignment/spill minimization is a hard optimization problem in general, so practical allocators use heuristics.

Question sources

Compiler Design review questions · Question 37

Question 185 | NS0265

What is 'dead code' in the context of compiler optimization?

A. Code that causes a runtime error |

B. Code that is syntactically incorrect

C. Code that is unreachable or whose computed results are never used |

D. Code that depends on uninitialized variables

Hint

Trace whether computation is evaluated, moved, shared or removed. Preserve observable effects and use the stated arithmetic and target assumptions.

Answer C: Unreachable code or unused computations without required effects.

COMP30 permits removing unreachable code and unused computations only when evaluation has no necessary side effects. C states the intended two dead-code categories, with this semantic caveat. Runtime faults, syntax defects and uninitialized uses are different error categories, not definitions of code being safely removable. An unused volatile read or effectful call may still be required.

Why each choice is right or wrong

A - Incorrect: A runtime fault does not make arbitrary reachable code safely removable as dead code.

B - Incorrect: Syntactically invalid code is a compilation error, not the definition of dead computation.

C - Correct: Unreachability or unused results identify candidates, provided necessary observable side effects are preserved.

D - Incorrect: An uninitialized dependency is an analysis or correctness issue, not automatically dead code.

Primary note proof

Study-note section COMP30

> Constant folding evaluates constant expressions during compilation: 3*4→12. Constant propagation substitutes known values: x=3; y=x+2 becomes y=3+2 before folding. Copy propagation substitutes a copied name while its source remains unchanged. Dead-code elimination removes unreachable code or computations whose values are unused and whose evaluation has no necessary side effects.

Question sources

Compiler Design review questions · Question 38

Question 186 | NS0266

What is the precise relationship between the general term compiler and the term transpiler?

A. A transpiler is a source-to-source compiler; compilers can also target assembly, machine code or bytecode.

B. A compiler is faster than a transpiler

C. A transpiler performs more optimizations than a compiler |

D. There is no meaningful distinction

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: A transpiler is a source-to-source compiler; compilers can also target assembly, machine code or bytecode.

A compiler translates a source program to an equivalent target representation, which may be machine code, assembly, bytecode or another source language. A transpiler is the source-to-source subtype of compiler. This gives a meaningful target-based distinction while retaining the shared category. Speed and optimization count depend on implementations and workloads; neither defines the relationship between the two terms.

Why each choice is right or wrong

A - Correct: A transpiler is a compiler whose target is another source language.

B - Incorrect: There is no universal speed ordering.

C - Incorrect: Optimization count does not define the subtype.

D - Incorrect: The source-to-source target is a meaningful distinction.

Primary note proof

Study-note section COMP01

> A compiler translates a source program into an equivalent target representation while preserving defined observable behavior. The target may be machine code, assembly, bytecode or another source language. A transpiler is a source-to-source compiler, not a fundamentally separate noncompiler technology. A compiler preserves language semantics, not the programmer's unstated intention.

Study-note section COMP01

> An interpreter executes a program representation; it need not work one source line at a time. Systems can compile to bytecode and then interpret or JIT-compile it. AOT compilation happens before execution; JIT compilation happens during execution and may use runtime profiles. Neither is universally faster; startup cost, optimization and workload matter.

Question sources

Compiler Design review questions · Question 39

Question 187 | NS0267

In the context of lexical analysis, 'lexeme' vs 'token' means:

A. Lexeme is the abstract category; token is the actual text |

B. Lexeme is the actual sequence of characters in the source; token is the abstract category with optional attribute value

C. They are synonymous terms |

D. Lexeme refers to keywords only; token refers to all other units

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: Lexeme is the source text; token is its category with optional attributes.

COMP04 directly distinguishes the actual character sequence from its token category and optional attribute. For example, total is a lexeme that may produce an ID token associated with a symbol entry. Neither term is restricted to keywords; the same distinction applies to identifiers, numeric literals, operators and other lexical units.

Why each choice is right or wrong

A - Incorrect: This reverses the roles: the lexeme is concrete text and the token identifies its class.

B - Correct: The concrete source sequence is the lexeme; category and optional information form the token.

C - Incorrect: The note explicitly separates the two abstractions rather than treating them as synonyms.

D - Incorrect: Lexemes include identifiers, literals and operators, not just keywords.

Primary note proof

Study-note section COMP04

> A token is a category with optional attribute, such as (ID, symbol-entry) or (INT_LITERAL,42). A lexeme is the actual character sequence, such as total or 42. A pattern specifies which lexemes belong to a token class. The scanner emits an ordered token stream, not an unordered mathematical set.

Study-note section COMP04

> Keywords, identifiers, literals, operators and punctuation are token classes. Whitespace and comments are usually skipped while line/column positions are retained. In indentation-sensitive or newline-sensitive languages, layout can carry tokens. Parse-tree construction is a parser task; scanning can intern identifier spellings, but declaration scope and types need later analysis.

Question sources

Compiler Design review questions · Question 40

Question 188 | NS0268

Under the general interference-graph model with fixed k≥3, which task is an NP-complete decision problem?

A. Converting an NFA to a DFA |

B. Constructing a LL(1) parsing table

C. Deciding whether an arbitrary interference graph can use k registers without spilling, for fixed k≥3

D. Computing FIRST sets for a CFG

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer C: Deciding whether an arbitrary interference graph can use k registers without spilling, for fixed k≥3

In the general interference model, simultaneously needed values are adjacent and registers are colors. Deciding whether the graph can be colored with k registers without spilling is general graph k-colorability, NP-complete for fixed k≥3. Optimal allocation is instead an optimization task. The other alternatives are construction or fixed-point tasks and do not state this membership-plus-hardness decision problem.

Why each choice is right or wrong

A - Incorrect: NFA determinization is a construction task, even when its output is exponential.

B - Incorrect: LL(1) table construction is not the specified graph-coloring decision task.

C - Correct: No-spill allocation is k-colorability under the stipulated general model.

D - Incorrect: FIRST-set computation is a fixed-point task rather than this NP-complete decision form.

Primary note proof

Study-note section COMP36

> A value is live where it can be used before redefinition. In the interference model, values that must coexist in distinct registers are connected by an edge; each register is a color. Nonoverlapping live ranges may share a register. If a valid assignment cannot fit the available registers, spilling places selected values in memory and inserts loads/stores.

Study-note section COMP36

> The general graph k-colorability decision problem is NP-complete for fixed k≥3; optimal register assignment/spill minimization is a hard optimization problem in general, so practical allocators use heuristics.

Question sources

Compiler Design review questions · Question 41

Question 189 | NS0269

What is the 'reaching definitions' data flow problem used for?

A. Determining which variables are live at each program point |

B. Determining which variable assignments might reach a given program point without being overwritten

C. Determining which expressions are available at each program point |

D. Determining which functions can be called from a given call site

Hint

Track which facts can hold along one path and which must hold along every path. Apply the specified direction, transfer rule and join operation.

Answer B: Assignments that may reach a point without being overwritten.

COMP29 directly defines reaching definitions as assignments that can arrive at a point without a replacement definition on the path. Its join is union because one possible path suffices. Liveness asks about future uses, available expressions asks about previous computations on every path, and possible call targets is a separate analysis question.

Why each choice is right or wrong

A - Incorrect: Future reads before redefinition are the liveness question, not reaching definitions.

B - Correct: This is the exact assignment-and-no-overwrite property defined for reaching definitions.

C - Incorrect: Available expressions concerns prior computations with unchanged operands, not assignment definitions.

D - Incorrect: Possible function targets concern call-graph or points-to analysis rather than reaching assignment definitions.

Primary note proof

Study-note section COMP29

> Reaching definitions is forward may-analysis: which assignments can reach this point without a replacement definition? OUT=GEN∪(IN−KILL), with IN the union of predecessor OUT sets. Available expressions is usually forward must-analysis: an expression has been computed on every relevant path and its operands have not changed. Join with intersection.

Question sources

Compiler Design review questions · Question 42

Question 190 | NS0270

In predictive (LL(1)) parsing, when should the parser use an ε-production A → ε?

A. When the lookahead token is in FIRST(A) |

B. When the lookahead token is in FOLLOW(A) and no other production for A begins with that token

C. When the stack is empty |

D. Whenever the next token is not a keyword

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Use ε on an appropriate FOLLOW(A) lookahead with no competing alternative.

COMP14 enters a nullable alternative under the left-hand nonterminal's FOLLOW symbols and requires no conflicting production in that cell. An epsilon reduction consumes no input. FIRST(A) alone can refer to a nonempty alternative, while stack emptiness and token being a keyword do not define nullable-alternative selection. Thus B supplies the correct lookahead rule.

Why each choice is right or wrong

A - Incorrect: A FIRST terminal may require a nonempty alternative; FIRST(A) alone does not select epsilon.

B - Correct: FOLLOW lookahead selects the nullable alternative when its predictive table cell has no competing production.

C - Incorrect: An empty stack is not the rule for selecting a production for a pending nonterminal.

D - Incorrect: Keyword status has no general relation to a nonterminal's nullable-production table entries.

Primary note proof

Study-note section COMP14

> For every A→α, put it in M[A,a] for each terminal a in FIRST(α). If α is nullable, also enter it under every symbol in FOLLOW(A). More than one production in a cell is a conflict. Pairwise SELECT sets must be disjoint; SELECT(α)=FIRST(α)−{ε}, plus FOLLOW(A) when α is nullable. This also detects two nullable alternatives.

Study-note section COMP14

> Initialize stack with $S. Match and pop terminals; for a nonterminal consult its row and lookahead, then push the chosen RHS in reverse. An ε production consumes no input. Accept when stack and input reach $. For A→a|ε with b∈FOLLOW(A), choose a on lookahead a and ε on b.

Question sources

Compiler Design review questions · Question 43

Question 191 | NS0271

Which of the following correctly describes a 'dominator' in a control flow graph?

A. A node that is executed before all other nodes unconditionally |

B. Node D dominates node N if every path from the entry node to N must pass through D

C. The node with the most outgoing edges |

D. The unique successor of every conditional branch

Hint

Identify what the representation stores, what it omits and which later transformations it must support; separate a representation property from an algorithmic task.

Answer B: Every entry-to-N path passes through D.

COMP28 defines domination relative to a particular node N: every path from entry to N must pass D. This is not a claim that D precedes every node in the program, nor is domination measured by out-degree or being a unique successor. The definition concerns unavoidable traversal on paths reaching the specified node.

Why each choice is right or wrong

A - Incorrect: A dominator is defined relative to a node, not necessarily as preceding all program nodes.

B - Correct: This exactly states the all-entry-to-N-paths definition in COMP28.

C - Incorrect: The number of outgoing edges does not establish unavoidable traversal on paths.

D - Incorrect: Branch successor uniqueness is unrelated to the general dominator definition.

Primary note proof

Study-note section COMP28

> A basic block is a maximal straight-line instruction sequence entered at its beginning and exited at its end. Leaders are the first instruction, every jump target, and each instruction after a branch. Start a block at each leader and continue until the next leader.

Study-note section COMP28

> A control-flow graph has blocks as vertices and possible next-block transfers as edges, including fall-through. A node D dominates N when every entry-to-N path passes through D. For an edge N→H where H dominates N, H is a loop header of a natural loop. This structure supports loop analysis. An expression DAG records value dependencies, not arbitrary control flow.

Question sources

Compiler Design review questions · Question 44

Question 192 | NS0272

In SLR(1) parsing, the reduce action for production A → α is triggered when the lookahead is in:

A. FIRST(A) |

B. FOLLOW(A)

C. FIRST(α) |

D. FOLLOW(α)

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: FOLLOW(A), once the item for A→α is complete.

COMP17 says SLR uses LR(0) states and FOLLOW(A) for reduction by a completed A production. The completed item establishes that the right side has been recognized; FOLLOW restricts which lookaheads permit reducing it. FIRST information predicts beginnings instead, and FOLLOW of an arbitrary right-hand string is not the table rule used here.

Why each choice is right or wrong

A - Incorrect: FIRST(A) concerns beginnings of derived strings rather than lookaheads after a completed A.

B - Correct: SLR places a completed A-production reduction under the lookaheads in FOLLOW(A).

C - Incorrect: FIRST of the right side predicts starting tokens, not permitted post-recognition reduction lookaheads.

D - Incorrect: SLR uses FOLLOW of the left-hand nonterminal A, not this right-side expression.

Primary note proof

Study-note section COMP17

> SLR(1) uses LR(0) states and FOLLOW(A) for reduction by a completed A production. Canonical LR(1) attaches context-specific lookahead to items. LALR(1) merges canonical states with the same LR(0) core, combining lookaheads; merging can introduce reduce/reduce conflicts. The standard grammar-class hierarchy is LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1), with increasing capability and often table size.

Study-note section COMP17

> A shift/reduce conflict offers both consuming a token and reducing; reduce/reduce offers two reductions. An SLR or LALR conflict does not by itself prove the grammar ambiguous. Canonical LR(1) handles LR(1) grammars, not every unambiguous CFG. Precedence and associativity can resolve suitable expression conflicts; changing parser names alone does not repair all ambiguity.

Question sources

Compiler Design review questions · Question 45

Question 193 | NS0273

What is 'escape analysis' and how does it help optimization?

A. Analysis to detect out-of-bounds array accesses; helps prevent security vulnerabilities |

B. Analysis to determine whether an object's reference can 'escape' the function where it is created; enables stack allocation of non-escaping objects

C. Analysis to find unreachable exception handlers; enables dead code elimination |

D. Analysis to track pointer arithmetic; enables bounds checking elimination

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer B: Determine whether references outlive or escape a region, enabling safe allocation changes.

COMP37 asks whether an object/reference can outlive or be accessed outside its allocating region. A proven nonescaping object may use stack storage or be eliminated when lifetime and identity are preserved. Bounds checking, exception reachability and pointer arithmetic are separate analyses and do not define the reference-escape property described.

Why each choice is right or wrong

A - Incorrect: Array bounds concern index validity rather than whether references leave their allocation region.

B - Correct: Reference escape information can justify stack allocation when lifetime and identity remain correct.

C - Incorrect: Unreachable exception handlers concern control-flow reachability, not reference lifetime or external access.

D - Incorrect: Pointer arithmetic tracking does not by itself define escape beyond a function or allocation region.

Primary note proof

Study-note section COMP37

> Static storage lasts throughout execution; stack frames support nested calls and recursion; heap storage supports lifetimes beyond a call. A frame may contain parameters, return address, saved registers, locals and temporaries. Sizes, alignment and offsets are target-dependent; a two-byte int is an example, not a C++ guarantee.

Study-note section COMP37

> A calling convention specifies argument and result locations, caller/callee-saved registers, stack alignment and frame responsibilities. Escape analysis asks whether an object/reference can outlive or be accessed outside a region such as its allocating call. Proven nonescaping allocations may use stack storage or be eliminated by scalar replacement, provided identity and lifetime semantics remain correct.

Question sources

Compiler Design review questions · Question 46

Question 194 | NS0274

Which of the following is a valid reason why LALR(1) parsers may have reduce/reduce conflicts that canonical LR(1) parsers do not?

A. LALR(1) uses a larger parsing table than canonical LR(1) |

B. LALR(1) merges LR(1) states with the same core but different lookaheads, potentially merging incompatible lookahead sets

C. LALR(1) does not support left-recursive grammars |

D. LALR(1) computes FOLLOW sets incorrectly

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Merging equal-core LR(1) states can combine incompatible reduction lookaheads.

COMP17 directly explains that LALR merges canonical states with identical LR(0) cores while unioning their lookahead sets. Distinctions that prevented competing reductions in separate states can disappear, creating reduce/reduce conflicts. This is not due to incorrect FOLLOW computation or rejection of left recursion; the loss comes from deliberate state merging.

Why each choice is right or wrong

A - Incorrect: Merging aims to reduce state count, and a larger table is not the reason for new conflicts.

B - Correct: Combining lookaheads of equal-core states can make previously separated reductions compete.

C - Incorrect: LALR can handle suitable left-recursive grammars, as other LR methods can.

D - Incorrect: The conflict can arise from correct merging, without any erroneous FOLLOW-set computation.

Primary note proof

Study-note section COMP17

> SLR(1) uses LR(0) states and FOLLOW(A) for reduction by a completed A production. Canonical LR(1) attaches context-specific lookahead to items. LALR(1) merges canonical states with the same LR(0) core, combining lookaheads; merging can introduce reduce/reduce conflicts. The standard grammar-class hierarchy is LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1), with increasing capability and often table size.

Study-note section COMP17

A shift/reduce conflict offers both consuming a token and reducing; reduce/reduce offers two reductions. An SLR or LALR conflict does not by itself prove the grammar ambiguous. Canonical LR(1) handles LR(1) grammars, not every unambiguous CFG. Precedence and associativity can resolve suitable expression conflicts; changing parser names alone does not repair all ambiguity.

Question sources

Compiler Design review questions · Question 47

Question 195 | NS0275

Which is a common implementation mechanism for runtime virtual dispatch in an object-oriented language, when the receiver's class cannot be resolved statically?

A. Static binding of function calls at compile time |

B. A virtual method table (vtable) per class, with function pointers resolved at runtime

C. Inline expansion of all polymorphic calls |

D. Name mangling to distinguish overloaded functions

Hint

Match stored information to its lifetime, access path and calling-convention role; distinguish compile-time organization from run-time storage.

Answer B: A virtual method table (vtable) per class, with function pointers resolved at runtime

A common implementation stores virtual-method entries in a per-class vtable and lets an object refer to the appropriate table. A virtual call selects the entry using the receiver's runtime class. Static binding alone does not provide that selection for an unknown class. Mandatory inlining is not generally possible, and name mangling distinguishes linker symbols rather than selecting overrides at runtime.

Why each choice is right or wrong

A - Incorrect: Static binding alone does not choose by an unknown runtime class.

B - Correct: A per-class vtable is the common mechanism described in the note.

C - Incorrect: Unknown runtime receivers prevent mandatory inlining of all polymorphic calls.

D - Incorrect: Name mangling identifies symbols rather than selecting overrides.

Primary note proof

Study-note section COMP38

Inlining replaces a call with a suitably renamed copy of its body, preserving argument evaluation, returns and side effects. It can remove call overhead and expose optimization opportunities, but may increase code size and instruction-cache pressure. A recursive function cannot simply be expanded without limit.

Study-note section COMP38

Virtual dispatch chooses an overridden method based on a receiver's runtime class. A common implementation uses a per-class virtual-method table and an object pointer to the table; the call loads the selected entry. This is an implementation strategy, not a universal language requirement. Name mangling distinguishes linker symbols, often for overloads; it is not itself runtime method selection. Proven receiver types may permit devirtualization.

Question sources

Compiler Design review questions · Question 48

Question 196 | NS0276

Which of the following is true about a 'well-formed' grammar for LL(1) parsing?

A. It may contain left recursion |

B. For every non-terminal A with productions A → α1 | α2 | ... | αn, no two alternatives share a common token in their FIRST sets, and if any αi derives ε, then FIRST(αj) and FOLLOW(A) are disjoint for all j ≠ i

C. It must be ambiguous |

D. It must have at least one ε-production

Hint

Use construction direction, lookahead and grammar restrictions. Distinguish algorithm families, implementations and coverage before comparing them.

Answer B: Disjoint predictive selections, including nullable FIRST/FOLLOW constraints.

COMP14 requires that alternatives for a nonterminal select disjoint lookahead sets. B gives the standard FIRST separation and nullable-alternative FOLLOW restriction; shared epsilon alternatives would also violate pairwise FIRST disjointness. Left recursion and useful ambiguity are not acceptable LL(1) properties, and epsilon productions are optional rather than required.

Why each choice is right or wrong

A - Incorrect: Ordinary LL(1) grammars cannot retain left recursion that recurs without input consumption.

B - Correct: Disjoint alternative selections prevent multiple productions from occupying the same predictive table cell.

C - Incorrect: A useful ambiguous grammar cannot give a unique deterministic LL(1) parse.

D - Incorrect: A grammar may be LL(1) without any epsilon production.

Primary note proof

Study-note section COMP14

For every A→α, put it in M[A,a] for each terminal a in FIRST(α). If α is nullable, also enter it under every symbol in FOLLOW(A). More than one production in a cell is a conflict. Pairwise SELECT

sets must be disjoint; SELECT(α)=FIRST(α)−{ε}, plus FOLLOW(A) when α is nullable. This also detects two nullable alternatives.

Study-note section COMP14

Initialize stack with $S. Match and pop terminals; for a nonterminal consult its row and lookahead, then push the chosen RHS in reverse. An ε production consumes no input. Accept when stack and input reach $. For A→a|ε with b∈FOLLOW(A), choose a on lookahead a and ε on b.

Question sources

Compiler Design review questions · Question 49

Question 197 | NS0277

The primary difference between a 'compiler' and a 'just-in-time (JIT) compiler' is:

A. A JIT compiler produces faster code than an ahead-of-time compiler always |

B. A JIT compiler compiles code at runtime (during program execution), allowing it to use runtime profiling information for optimization, while an AOT compiler compiles before execution

C. A JIT compiler does not perform any optimization |

D. A JIT compiler only works for interpreted languages

Hint

Identify each activity’s input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer B: JIT compiles during execution; AOT compiles before execution.

COMP01 defines AOT before execution and JIT during execution, possibly using runtime profiles. Both are compilation, so the heading compiler versus JIT is better understood as AOT versus JIT. The timing contrast in B is correct, while no universal speed guarantee, absence of optimization or restriction to interpreted languages follows.

Why each choice is right or wrong

A - Incorrect: Neither AOT nor JIT is universally faster; startup and workload affect the comparison.

B - Correct: Runtime compilation can use execution profiles, whereas AOT performs compilation before the run.

C - Incorrect: JIT may optimize using runtime information rather than performing no optimization.

D - Incorrect: JIT is an execution-time compilation strategy, not a technique restricted to a separately defined interpreted-language category.

Primary note proof

Study-note section COMP01

A compiler translates a source program into an equivalent target representation while preserving defined observable behavior. The target may be machine code, assembly, bytecode or another source language. A transpiler is a source-to-source compiler, not a fundamentally separate noncompiler technology. A compiler preserves language semantics, not the programmer's unstated intention.

Study-note section COMP01

An interpreter executes a program representation; it need not work one source line at a time. Systems can compile to bytecode and then interpret or JIT-compile it. AOT compilation happens before execution; JIT compilation happens during execution and may use runtime profiles. Neither is universally faster; startup cost, optimization and workload matter.

Question sources

Compiler Design review questions · Question 50

Question 198 | NS0299

What is the main purpose of syntax-directed translation in a compiler?

A. To interleave semantic analysis with syntax analysis using attributes attached to grammar symbols

B. To generate optimized machine code directly

C. To tokenize the source code

D. To replace the parser with a finite automaton

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer A: Associate semantic computations with grammatical structure and attributes.

COMP24 attaches attributes and semantic rules to grammar symbols and productions; COMP25 executes translation actions during processing. This permits semantic work to follow parsing structure, as A states. It does not inherently bypass IR to optimized machine code, reduce to tokenization, or replace the parser with a finite automaton.

Why each choice is right or wrong

A - Correct: Attributes and rules connect grammatical structure to semantic computations during translation.

B - Incorrect: Direct optimized machine-code production is not the required or exclusive purpose of syntax-directed translation.

C - Incorrect: Tokenization is lexical analysis rather than grammar-directed semantic computation.

D - Incorrect: Finite automata recognize ordinary token patterns and do not replace general nested-syntax parsing.

Primary note proof

Study-note section COMP24

> A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions.

Study-note section COMP25

> A syntax-directed translation scheme embeds executable actions in productions; the actions run while translating a source program, not while generating the compiler. Attributes may compute types, values, AST pointers or generated code. The scheme defines an input-to-output translation, not merely an input language.

Question sources

Computer Science Exit Exam · 2018 · Question 16

Question 199 | NS0303

Straight directed translation uses:

A. Purely lexical rules

B. Backtracking algorithms

C. Grammar without attributes

D. Grammar with sematic rule

Hint

Track where each computed value comes from and when each production action runs. Distinguish the representation from its attached information.

Answer D: Grammar with semantic rules, reading the misspelled term as syntax-directed translation.

The stem's Straight directed translation appears to be a damaged rendering of syntax-directed translation. Under that explicit reading, COMP24 defines a CFG with attributes and semantic rules, and COMP25 embeds translation actions in productions. D expresses that structure despite its spelling error. The other choices omit semantics or prescribe unrelated implementation details.

Why each choice is right or wrong

A - Incorrect: Pure lexical rules do not supply grammar-directed semantic attributes and computations.

B - Incorrect: Backtracking is a parsing implementation choice, not the defining translation structure.

C - Incorrect: A grammar without semantic attributes or rules does not describe the syntax-directed definition requested.

D - Correct: Reading sematic as semantic, this is the grammar-plus-semantic-rules structure of syntax-directed translation.

Primary note proof

Study-note section COMP24

> A syntax-directed definition is a CFG with attributes on grammar symbols and semantic rules on productions.

Study-note section COMP25

> A syntax-directed translation scheme embeds executable actions in productions; the actions run while translating a source program, not while generating the compiler. Attributes may compute types, values, AST pointers or generated code. The scheme defines an input-to-output translation, not merely an input language.

Question sources

Computer Science Exit Exam · 2018 · Question 21

Question 200 | NS0331

A compiler has produced checked three-address code for a function. Which description incorrectly places the ensuing target-lowering task within the standard compilation pipeline?

A. Code generation can be considered as the start phase of compilation

B. The target program is the output of the code generator

C. Checked intermediate code is supplied as input to the code generator.

D. It used to produce the target code for three-address statements.

Hint

Identify each activity's input and output. Distinguish character recognition, grammatical structure, semantic checks and target implementation.

Answer A: Code generation can be considered as the start phase of compilation

Code generation uses checked intermediate code to produce target instructions after the frontend has analyzed the source. It can lower each three-address operation into a suitable target sequence. Lexical analysis, not code generation, begins the standard pipeline. The input condition here means checked IR suitable for lowering; it is not a claim that all possible program bugs have been proved absent.

Why each choice is right or wrong

A - Correct: Target lowering follows scanning, parsing, semantic checks and IR construction; it does not start compilation.

B - Incorrect: The generator's output is target code.

C - Incorrect: Checked IR is the stated input to target lowering.

D - Incorrect: Three-address statements can be translated into target instructions.

Primary note proof

Study-note section COMP03

> The conventional sequence is lexical analysis → syntax analysis → semantic analysis → intermediate-code generation → optimization → target-code generation. Scanning turns characters into tokens; parsing builds structure; semantics checks declarations and types; synthesis produces implementation code. A phase is a logical activity; a pass is a traversal that can combine activities.

Study-note section COMP34

> Code generation lowers checked IR to a target form.

Question sources

Computer Science Exit Exam · 2018 · Question 58