CS Exit Ethiopia

Revision Outlines

Computer Science Exit Examination

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

Compiler Design

Revision Outlines

Translation, Interpretation and Compilation Timing

- Compiler translation contract
  - Source → equivalent target representation
  - Defined observable behavior preserved
  - Machine code; assembly; bytecode; source targets
  - Transpiler: source-to-source compiler
  - Language semantics versus unstated programmer intention
- Interpretation, AOT and JIT
  - Interpreter: executes program representation
  - Line-at-a-time misconception
  - Bytecode compilation then interpretation/JIT
  - AOT: before execution
  - JIT: during execution, runtime profiles
  - Speed: startup, optimization, workload
  - No universally faster method

Tools, host, target and incremental work

- Translation toolchain
  - Preprocessor: inclusion and macros
  - Assembler: assembly → object code
  - Linker: objects and libraries
  - External-symbol resolution and relocation
  - Loader: executable mapping and start
  - Compilation alone ≠ ready executable
- Host, target and compilation organization
  - Cross-compiler: host differs from target
  - Multiple targets: appropriate backends
  - Retargetable: changeable target descriptions
  - Incremental: affected fragments and dependent units

- Incremental ≠ changed-line replacement only
- One/multipass: traversal counts, not targets

Compiler phases and shared services | Exam focus

- Conventional compiler pipeline
  - Lexical → syntax → semantic analysis
  - IR generation → optimization → target generation
  - Scanning: characters → tokens
  - Parsing: structure
  - Semantics: declarations and types
  - Synthesis: implementation code
  - Phase: logical activity
  - Pass: traversal, possibly combined activities
- Compiler organization and services
  - Front end: source-language rules
  - Middle end: IR analysis/transformation
  - Backend: target details
  - Shared symbol tables and error reporting
  - No mandatory seventh final phase
  - Optimization at multiple levels
  - Post-instruction-selection optimization possible
  - Standard six-phase final: code generation

Token, lexeme and pattern

- Token, lexeme and pattern
  - Token: category plus optional attribute
  - (ID,symbol-entry); (INT_LITERAL,42)
  - Lexeme: actual characters total or 42
  - Pattern: token-class membership specification
  - Scanner output: ordered token stream
  - Unordered-set trap
- Token classes and scanner boundaries
  - Keywords; identifiers; literals; operators; punctuation

- Usually skipped whitespace/comments
- Retained line/column positions
- Indentation/newlines may carry tokens
- Parse trees: parser responsibility
- Scanning may intern identifier spellings
- Scope/types: later analysis

• Worked example Five-token declaration
- int x = 42;
- keyword; ID; assignment; integer; semicolon

Regular expressions for lexical structure | Exam focus

• Identifier regular expressions
- Concatenation; alternation r|s
- Kleene r*: zero-or-more
- r+: one-or-more
- Letter(Letter|Digit)*
- Single letter allowed; leading digit forbidden
- C-like ASCII underscore convention
- [A-Za-z_][A-Za-z0-9_]*
- Language-specific identifier rules

• Equivalent patterns and nesting limits
- Letter(Letter+|Digit*)* = Letter(Letter|Digit)*
- Repeated letter/digit blocks
- Equivalent-option trap
- Regular patterns: ordinary tokens
- Arbitrary balanced nesting: grammar/stack

Construct and run a scanner automaton | Exam focus

• Scanner-generator automata pipeline
- RE → Thompson NFA → subset DFA
- Optional minimization
- Concatenation: final-to-start ε edge
- Alternation: branching ε paths

- Star: skip and repetition ε paths

• Scanner subset construction
  - ε-closure: zero-or-more ε reachability
  - DFA state: NFA-state set
  - Transition: ε-closure(move(set,a))
  - Start: NFA start closure
  - Repeated unseen-subset discovery
  - Accepting subset contains NFA final
  - Token identity and rule-priority annotations
  - Empty subset: dead state

Longest match and counting tokens

• Maximal munch and rule priority
  - Longest matching lexeme
  - Equal-length flex matches: earliest rule
  - ifx: identifier, not if+x
  - <= and ++: single operators
  - Identifier recognition then keyword-table lookup

• Loop-header token count
  - for (int i = 0; i<=10; i++)
  - for,(,int,i,=,0,;
  - i,<=,10,;,i,++,)
  - Fourteen tokens irrespective of spacing
  - Increment: one token
  - Comments: no ordinary post-preprocessing parser token
  - Choices 4,5,9,10: correct result omitted

Locate the error by evidence

• Compiler-error classification
  - Lexical: no legal token formation
  - Invalid character; unterminated literal
  - Syntax: illegal token arrangement
  - Missing delimiter

- Semantic: name, scope, type constraints
- Wrong algorithms may compile
- All-logical-errors detection trap

• Malformed fro-line diagnosis

- fro and lowercase o: legal identifiers
- Misspelled keyword ≠ lexical error
- Mismatched syntax: definite
- Undeclared o: possible semantic diagnostic
- Compiler-dependent recovery order and diagnostics
- No silent fro→for or o→0 repair

CFGs, derivations and trees | Exam focus

• CFG vocabulary and derivations

- G=(N,T,P,S)
- Nonterminals; terminals; productions; start
- A→α: one nonterminal rewrite
- Sentential form: any S-derived string
- Sentence: terminals only
- Leftmost/rightmost: corresponding remaining nonterminal

• Parse tree versus AST

- Parse tree: production structure
- Root S; internal nonterminals
- Terminal leaves; optional ε leaves
- Left-to-right yield: input
- AST: removed scaffolding and punctuation
- Operands: leaves; operators: internal nodes
- Parser may construct either

Ambiguity, precedence and associativity | Exam focus

• Grammar ambiguity

- One terminal string, two parse trees
- Equivalent: two leftmost derivations
- Same-tree derivation order insufficient

- E→E+E|E*E|id
- id+id*id: alternative grouping

• Precedence, associativity and dangling else
- E→E+T|T; T→T*F|F; F→(E)|id
- Multiplication precedence; left association
- a−b−c: (a−b)−c
- Dangling else: alternative nested-if attachments
- Nearest-unmatched-if policy or grammar rewrite
- Leftmost derivation alone ≠ disambiguation

Top-down parsing | Exam focus

• Top-down parsing
- Start S → input expansions
- Usually leftmost derivation
- Recursive descent: procedure per nonterminal
- General descent may backtrack
- Predictive: lookahead choice without backtracking
- LL(1): Left-to-right, Leftmost, one lookahead

• Nonrecursive predictive parsing
- Explicit grammar-symbol stack
- Parsing-table consultation
- Same predictive language class
- Recursive/table-driven: linear for suitable fixed grammar
- Names alone do not rank efficiency

Left Recursion and Prefix Factoring | Exam focus

• Left-recursion elimination
- Immediate A→Aα
- A→Aα|β; β not A-leading
- A→βA′; A′→αA′|ε
- Generated language preserved
- Changed tree shape
- Semantic actions preserve intended association

- Indirect cycles through other nonterminals
- Ordinary predictive descent needs removal

• Left factoring and parser suitability
- $A \to \alpha\beta \mid \alpha\gamma \Rightarrow A \to \alpha A'$
- $A' \to \beta \mid \gamma$
- Common-prefix choice postponed
- Factoring ≠ guaranteed unambiguity/LL(1)
- LR handles suitable left recursion
- Unmodified descent: infinite recursion before input

Compute FIRST and FOLLOW | Exam focus

• FIRST of a sequence
- Beginning terminals
- ε when sequence vanishes
- Add each FIRST excluding ε
- Continue across nullable symbols
- ε only if all nullable

• FOLLOW fixed-point computation
- Next terminals after A
- End marker $, never ε
- $\$ \in FOLLOW(S)$
- $B \to \alpha A \beta$: $FIRST(\beta) - \{\varepsilon\}$ into FOLLOW(A)
- Nullable β: FOLLOW(B) into FOLLOW(A)
- Repeat until no growth

• Worked example FIRST/FOLLOW worked grammar
- $S \to AB$; $A \to a \mid \varepsilon$; $B \to b$
- $FIRST(A) = \{a, \varepsilon\}$
- $FIRST(S) = \{a, b\}$
- $FOLLOW(A) = \{b\}$
- $FOLLOW(B) = \{\$\}$

Build and use an LL(1) table | Exam focus

• LL(1) table construction

- A→α entries under FIRST(α) terminals
- Nullable α: entries under FOLLOW(A)
- Multiple productions per cell: conflict
- SELECT=FIRST(α)−{ε}
- Nullable: add FOLLOW(A)
- Pairwise disjoint SELECT sets
- Two nullable alternatives also conflict

• Predictive stack trace
- Initialize $S
- Terminal match and pop
- Nonterminal row plus lookahead
- Push selected RHS reversed
- ε production: no input consumed
- Accept: stack/input both $
- A→a|ε: a-lookahead selects a
- b∈FOLLOW(A): b-lookahead selects ε

• Exam trap LL(1) conflict traps
- Useful ambiguous grammars not LL(1)
- Overlapping FIRST
- FIRST/FOLLOW conflicts
- Left recursion
- A-and-B-only key: insufficient exclusion

Bottom-up actions and handles | Exam focus

• Bottom-up parsing and handles
- Input reduced toward S
- Reverse rightmost derivation
- Handle: next correct RHS occurrence
- Context determines valid reduction
- Arbitrary matching RHS ≠ handle

• Shift-reduce actions and GOTO
- Shift: next token and state
- Reduce: RHS → LHS, then GOTO

- Accept: complete; error: invalid situation
- A→XYZ: pop three symbols/states
- Exposed q → push A,GOTO[q,A]
- Go ≠ separate standard ACTION
- GOTO: essential table machinery

Construct LR(0) item states | Exam focus

- LR(0) augmentation and closure
  - S′→S
  - Item A→α·β: production progress
  - Dot before B triggers B→·γ
  - Repeat CLOSURE additions
- LR(0) canonical collection
  - GOTO(I,X): advance dots, take closure
  - Initial CLOSURE({S′→·S})
  - Nonempty GOTOs for all grammar symbols
  - Iterate until no new states
  - Canonical collection: item-set collection
  - GOTO alone ≠ canonical collection
- Worked example LR(0) state example
  - S→a
  - I0={S′→·S,S→·a}
  - GOTO(I0,a)={S→a·}
  - GOTO(I0,S)={S′→S·}
  - Complete item set: one state
  - One-state-per-item trap

LR families and conflicts | Exam focus

- LR parser families
  - SLR(1): LR(0) states, FOLLOW reductions
  - Canonical LR(1): context-specific item lookaheads
  - LALR(1): merge equal LR(0) cores
  - Combined lookaheads

- Merging may introduce reduce/reduce conflicts
- LR(0)⊆SLR(1)⊆LALR(1)⊆LR(1)
- Increasing capability, often larger tables

• Parser conflicts and limits
- Shift/reduce: consume or reduce
- Reduce/reduce: two reductions
- SLR/LALR conflict ≠ ambiguity proof
- LR(1) handles LR(1) grammars
- Not every unambiguous CFG
- Suitable expression conflicts: precedence/associativity
- Parser renaming ≠ universal ambiguity repair

Parser limits and operator grammars | Exam focus

• Nesting and deterministic parser limits
- LL; LR; recursive descent: suitable nesting
- Explicit/implicit stack supports unbounded nesting
- LL and LR: tables possible
- Deterministic LL/LR: unambiguous grammars
- Ambiguous grammar: rewrite or disambiguation
- Generalized method: GLR

• Operator grammar conditions
- No adjacent RHS nonterminals
- Usually no ε productions
- Operator precedence: conflict-free relations additionally
- Right-linear grammars also lack adjacent nonterminals
- Property alone: nonunique category
- Regular patterns: chiefly lexical
- CFGs: chiefly nested syntax

Symbol tables and scope | Exam focus

• Symbol-table contents and scope operations
- Names/identities → attributes
- Type; scope; width; storage offset

- Parameter signature; constants; linkage
- Hashing: expected O(1) insertion/lookup
- Stack of lexical scope tables
- Lookup nearest scope first
- Insert current-scope declarations
- Scope exit removes visibility

• Symbol-table timing and implementation
- Scanner may intern spelling
- Semantics binds declaration/type
- First interaction: conventional, implementation-dependent
- Punctuation need not have entries
- Literals may use separate table
- Hash table: implementation, not semantic role

Names, Types and Advanced Features | Exam focus

• Member-call phase responsibilities
- obj.method(x)
- Scanner: identifiers and punctuation
- Parser: member-call structure
- Semantics: receiver, accessibility, argument types
- Overloading: signatures using static types
- Overriding: possible runtime implementation selection

• Scope, closures and generics
- Declaration identity beyond spelling
- Inner int x shadows outer float x
- Closures: captured lexical free variables
- Lifetime beyond creating frame
- Generics: parameter constraints/instantiation checks
- Symbol/type information beyond CFG
- Surface syntax may remain context-free

• Worked example object.resize(3) worked call
- Tokens: id,dot,id,(,integer,)
- Call→Expr . id (Args)

- Expr→id; Args→Expr|integer|ε
- Call⇒Expr.id(Args)⇒id.id(integer)
- Bind object, resize, 3
- Parser: shape only
- Semantics: scope, accessible member, compatible overload
- Only resize(): parses, argument/type failure

Semantic checking: static and dynamic | Exam focus

- Static semantic checks
  - Before execution
  - Compatible types; valid names
  - Restricted-scope duplicate declarations
  - Argument counts; return compatibility
  - Control-flow placement
  - C/C++ break: loop or switch
  - Annotated AST or typed IR
  - No mandatory unique output-tree name
- Dynamic checks and static proofs
  - Runtime-dependent bounds/null checks
  - Some accesses statically provable safe/faulty
  - Universal static-memory-check impossibility trap
  - Unreachable-code rules language-dependent
  - Reject specified cases, warn, or eliminate

Type rules, inference and compatibility | Exam focus

- Type-checking operation rules
  - Strict integer mod: two integers
  - Indexing: array plus integer index
  - if: Boolean condition
  - Assignment: equal/convertible types
  - Calls: parameter count and compatibility
  - Returns: result-type match
- Type inference and error distinctions

- Constraints derive types
- Integer x, numeric +, promotion rules
- Static inference without annotations
- Operator overloading: multiple meanings
- Overloading ≠ replacement for checking
- Ordinary C++/Java int x="hello": mismatch
- Zero division/bounds ≠ automatic type mismatch

Conversions need a named language

• Conversions and precision
- Coercion: permitted implicit conversion
- Cast: explicit conversion request
- Widening: often increased range
- Java int→float: widening, possible rounding
- Java int→long: exact
- Narrowing: range/fraction loss

• Language-specific conversion cases
- C++ int x=3.14: 3 if representable
- C++ int x{3.14}: narrowing rejection
- Floating→integer: fraction discarded
- C++ unrepresentable truncated value: undefined behavior
- Java double→int assignment: explicit permitted cast
- C++/Java char/int: ordinary promotion
- Universal mixed-type ban trap

Attributes and Syntax-Directed Definitions | Exam focus

• Syntax-directed definitions and attributes
- CFG with grammar-symbol attributes
- Semantic rules on productions
- Annotated parse tree: computed attribute values
- Tree ≠ inherited/synthesized attribute
- Synthesized: children → parent
- Inherited: parent/sibling context

- Attribute evaluation examples and classes
  - E→E1+T; E.val=E1.val+T.val
  - Synthesized expression value
  - Inherited declaration type environment
  - S-attributed: synthesized only
  - Natural LR reduction-time evaluation
  - Suitable LL evaluation also possible
  - L-attributed: restricted left-to-right dependencies
  - Dependency cycles may prevent evaluation

Translation actions, ASTs and postfix | Exam focus

- Syntax-directed translation schemes
  - Executable actions embedded in productions
  - Actions during source translation
  - Not during compiler generation
  - Attribute results: types; values; AST pointers; code
  - Input-to-output translation
  - Beyond input-language recognition
- AST construction and postfix evaluation
  - Identifier: mkleaf(id,entry)
  - Number: mkleaf(num,value)
  - Operator: mknode(op,left,right)
  - a−4+c AST: +(−(a,4),c)
  - Postorder postfix: a 4 − c +
  - Stack: push operands
  - Pop right, then left
  - Stack-machine usefulness
  - Infix-to-postfix not universally mandatory

IR and Three-Address Code

- Intermediate-representation purpose
  - Source analysis separated from target implementation
  - Optimizer/backend reuse

- Easy generation, analysis, transformation, lowering
- Portability architectural benefit
- Possible target-dependent details
- AST; graphs; bytecode; TAC

• TAC and instruction records

- Ordinary arithmetic: one operation maximum
- Two sources; one destination
- t1=c*d; t2=b+t1; a=t2
- Quadruple: (op,arg1,arg2,result)
- Triple: position references, no result field
- Indirect triples: separate ordering list
- Easier rearrangement
- Trees: expressions; DAGs: shared subexpressions
- Quadruples/triples: instruction records

Control flow, arrays and SSA | Exam focus

• Control-flow and array translation

- if a<b goto Ltrue; goto Lfalse
- Labels join control paths
- Short-circuit AND: true-left permits right
- Short-circuit OR: false-left permits right
- Eager evaluation may change side effects
- Address: base+(i−L)*w
- Element width w; lower bound L
- Fixed-width contiguous-storage assumption

• Static single assignment

- Distinct name per definition
- Join: x3=φ(x1 from B1,x2 from B2)
- Incoming value follows actual predecessor
- φ ≠ arithmetic average
- SSA: IR naming discipline
- Coexisting control-flow graph

Basic Blocks, Flow Graphs and Dominators

- Basic blocks and leaders
  - Maximal straight-line instruction sequence
  - Entry at beginning; exit at end
  - Leaders: first instruction
  - Every jump target
  - Instruction after each branch
  - Block continues until next leader
- Flow graphs and dominators
  - CFG vertices: basic blocks
  - Edges: possible next transfers, fall-through
  - D dominates N: every entry-to-N path
  - Backedge N→H; H dominates N
  - H: natural-loop header
  - Loop-analysis structure
  - Expression DAG: value dependencies
  - DAG ≠ arbitrary control flow

Data-flow questions and equations

- Forward data-flow analyses
  - Reaching definitions: may-analysis
  - Assignments reaching without replacement
  - OUT=GEN∪(IN−KILL)
  - IN: predecessor-OUT union
  - Available expressions: must-analysis
  - Computed on every relevant path
  - Unchanged operands
  - Available-expression join: intersection
- Liveness and definite assignment
  - Liveness: backward may-analysis
  - Possible read before future redefinition
  - OUT[B]=∪successors IN
  - IN[B]=USE[B]∪(OUT[B]−DEF[B])

- Fixed-point iteration
- Register allocation and dead-value removal
- Definite assignment: initialized on every path
- Reaching definitions alone insufficient

Folding, Propagation and Dead Code

• Folding, propagation and elimination
- Constant folding: 3*4→12
- Constant propagation: x=3; y=x+2→y=3+2
- Then fold
- Copy propagation: source remains unchanged
- Dead code: unreachable or unused computation
- Necessary-side-effect exclusion

• Combined optimization worked case
- const x=8,y=16; if(x>y){z=x+y;}
- Propagate constants
- Fold 8>16 to false
- Remove unreachable branch
- Compile-time evaluation plus dead-code elimination
- Overlapping optimization choices
- Unused volatile read may remain necessary
- Effectful calls may remain necessary

Reuse, Strength Reduction and Peepholes

• Reuse, identities and strength reduction
- CSE: reuse earlier expression value
- Unchanged operands and relevant memory
- DAG sharing
- Valid integer identity: x+0→x
- Strength reduction: equivalent cheaper operation
- Unsigned x*8→x<<3
- Valid width and shift count
- Actual saving depends on target costs

- Peephole optimization
  - Short instruction windows
  - Redundant load/store removal
  - Machine-idiom replacement
  - Jump-chain shortening
  - jump L1; L1→L2 ⇒ jump L2
  - Local optimization: broader category
  - Peephole: concrete method
  - Overlapping choices require requested level

Loop Invariants, Induction and Fusion

- Loop invariants and induction variables
  - Unchanged expression inputs
  - Preheader hoisting: newly executed paths safe
  - Induction: loop-invariant increment
  - i=i+1
  - Repeated t=4*i
  - Equivalent running t increased by 4
  - Reduced repeated multiplication
- Loop transformation contrasts
  - Fusion/jamming: compatible bounds, safe dependencies
  - Unrolling: replicated body
  - Reduced branch overhead, exposed parallelism
  - Remainder iterations and code growth
  - LICM; fusion; unrolling; strength reduction
  - Distinct transformations
  - Machine-dependent/independent taxonomy
  - Local/global/loop taxonomy
  - Classification requires criterion

Optimization legality and aliasing | Exam focus

- Optimization legality and aliasing
  - Defined observable behavior preserved

- Required side effects and exceptions
- Aliases: same memory possibility
- *p=5 may invalidate q-load
- Reuse requires proven nonaliasing/equivalent proof
- Calls may invalidate memory facts

• Unsafe transformation boundary cases
- Potentially trapping division hoist
- Zero-iteration loop trap
- Floating reassociation: rounding/NaN changes
- Signed C/C++ overflow and shifts
- Unsigned modular arithmetic distinction
- Mathematical identity ≠ legal rewrite
- Dependence analysis: required memory-access order
- Fusion/scheduling legality

Target code generation

• Target code-generation tasks
- Checked IR → target form
- Instruction selection: target operations
- Register allocation: value locations
- Scheduling: dependency-respecting order
- Storage layout: offsets and ABI
- Register assignment: actual physical names

• Target instruction example and variants
- t=a+b
- LOAD R1,a; LOAD R2,b
- ADD R3,R1,R2; STORE t,R3
- Combined load/arithmetic or register-held t
- Machine idioms; register availability; instruction order
- Execution-time effects
- Absolute code; relocatable object; assembly; bytecode
- TAC normally intermediate
- Cannot-ever-be-target universal trap

Descriptors and getReg | Exam focus

- Register and address descriptors
  - Register descriptor: current held values
  - Address descriptor: all current value locations
  - Memory and multiple registers
  - Register-only assignment: stale memory copy
  - Code-generator bookkeeping
  - Descriptors ≠ source types
- getReg code-generation sequence
  - Select operand/result registers
  - Necessary loads → operation → descriptor update
  - Prefer suitable existing register
  - Reuse dead operand
  - Otherwise spill victim
  - Store live value lacking current alternative
  - Block exit: preserve successor-needed values

Interference graphs and spilling

- Liveness, interference and spilling
  - Live: possible use before redefinition
  - Edge: values need distinct coexisting registers
  - Registers as colors
  - Nonoverlapping live ranges may share
  - Insufficient register fit: spill to memory
  - Insert loads/stores
- Register-allocation complexity limits
  - General fixed k≥3 coloring: NP-complete
  - Optimal assignment/spill minimization: generally hard
  - Practical heuristic allocators
  - Restricted graphs/SSA: different properties possible
  - Not every allocation algorithm NP-complete
  - Decision-model versus optimization distinction

Runtime Storage and Calling Conventions | Exam focus

- Runtime storage and frames
  - Static: entire execution lifetime
  - Stack: nested calls and recursion
  - Heap: beyond-call lifetimes
  - Frame parameters and return address
  - Saved registers; locals; temporaries
  - Target-dependent sizes, alignment, offsets
  - Two-byte int: module example
  - Not C++ guarantee
- Calling convention and escape analysis
  - Argument/result locations
  - Caller/callee-saved registers
  - Stack alignment and frame responsibilities
  - Escape: outlive/access outside allocation region
  - Proven nonescape: stack allocation possible
  - Scalar replacement: allocation elimination
  - Preserved identity and lifetime semantics

Inlining and dynamic dispatch | Exam focus

- Inlining benefits and limits
  - Suitably renamed function-body copy
  - Preserved argument evaluation, returns, side effects
  - Reduced call overhead
  - Exposed optimization opportunities
  - Increased code size/cache pressure
  - Recursive expansion cannot be unlimited
- Dynamic dispatch and devirtualization
  - Overridden method: receiver runtime class
  - Common per-class virtual-method table
  - Object pointer to table
  - Load selected entry
  - Implementation strategy, not universal requirement

- Name mangling: linker-symbol distinctions/overloads
- Mangling ≠ runtime method selection
- Proven receiver type: devirtualization

Recovery and compiler-construction tools

- Error recovery methods
  - Panic mode: skip to synchronizing token
  - Semicolon or closing brace
  - Phrase-level: local insertion/deletion
  - Location and cause reporting
  - Avoid diagnostic cascades
  - Lexical, syntax, semantic error detection
  - Error handling: shared service
- Compiler-construction generators
  - Lex/flex: patterns/actions → scanner
  - Yacc: Yet Another Compiler Compiler
  - Yacc/Bison: grammar/actions → parser
  - Definitions; %% rules; %% user code
  - Classic Yacc: LALR
  - Bison: additional LR/generalized modes
  - Scanner generator ≠ CFG parser replacement

Expression Translation Through the Compiler | Exam focus

- Expression pipeline trace
  - a=b+c*d;
  - Scanner: names and operators
  - Parser: assignment with +(b,*(c,d))
  - Semantics: declarations and operand/result types
  - float a; int b,c,d;
  - Possible result conversion before store
  - Language arithmetic rules
- Expression IR to target trace
  - t1=c*d; t2=b+t1

- a=convert_if_needed(t2)
- Constant operands: folding
- Repeated unchanged expressions: sharing
- Target instruction and register selection
- Same a spelling, distinct scoped declarations
- Entering representation and added information
- Evidence for newly diagnosable errors

Missing and Conflated Lexical Answers

- Lexical-answer repairs
  - Primary output: token stream
  - Symbol table/AST/assembly: invalid substitutes
  - Side structures versus primary output
  - Smallest meaningful sequence: token/lexeme conflation
  - Required abstraction unspecified
  - Loop header: 14 despite missing option
- Grouping units and storage roles
  - Characters → token classes: lexical
  - Tokens → expressions/statements: syntax
  - Vague syntactic-class stem needs units
  - Symbol tables: identifiers and attributes
  - Not necessarily every token kind
  - Trees: hierarchical constructs
  - Symbol tables: declaration metadata

Overlapping Parser Choices

- Overlapping parser choices
  - Shift-reduce and LR: both bottom-up
  - Negative top-down question: multiple answers
  - Recursive/nonrecursive predictive: both potentially efficient
  - Nesting/tables: not uniquely LR
  - LR choice ≠ ambiguity removal
  - Leftmost derivation ≠ disambiguation policy

- LL(1) key and precedence repairs
  - Overlapping FIRST; left recursion; ambiguity
  - All rule out useful LL(1) grammar
  - A-and-B-only key invalid
  - Ask all conditions or conflicting cell
  - Precedence/operator-precedence: concrete conflict and policy
  - Universal shift-reduce solution trap

Semantic and Translation Claim Repairs

- Semantic activity versus data structure
  - Type checking: activity
  - Symbol table: data structure
  - Both support semantic analysis
  - Semantic rules compute attributes
  - Types; values; code: nonexhaustive examples
  - Symbol records: names and types
- Semantic and pipeline claim repairs
  - Zero division/bounds ≠ clean type-error list
  - Indentation/syntax mistakes ≠ clean type errors
  - Incompatible operand plus named language
  - Conventional final phase: code generation
  - Later target optimization possible
  - Annotated AST or typed IR
  - Representation versus annotation versus phase
  - Forced unique tree-name trap