CS Exit Ethiopia

Complete Study Notes

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 6

How to use these study notes

START01 A practical study cycle

Start with a small set of questions before reading.

Mark the exact thing that stopped you: an unknown term, a rule, a calculation, a trace or a choice between alternatives.

Use the question lookup to open the relevant section.

Read one section, close it, and explain the rule in your own words.

Solve the question again and say why each other choice fails.

Then change one assumption or input and solve the variation.

This checks understanding beyond recognition.

A useful 60-minute session: 10 minutes of cold questions, 20 minutes of focused reading, 20 minutes of closed-book solving and 10 minutes updating an error log.

Adjust the timing to your needs.

Revisit mistakes after a gap, for example the next day, several days later and the following week.

These intervals are a practical starting plan, not a fixed learning law.

Record the section ID, your mistaken rule, the corrected rule and one boundary case.

Keep the log short enough to review.

Mix related courses during revision: programming with data structures; databases with security; networking with administration.

Explain how the same word changes meaning across contexts.

START02 What to emphasize

Emphasize definitions, key differences, algorithm costs, SDLC models, databases, networking and security.

Use short oral questions to check understanding and correct common misconceptions.

Combine those habits with the tracing, calculation, design and diagnosis demanded by the Ministry learning outcomes.

Allocate revision by both exam share and your error log.

A course with fewer items can still deserve more time if your foundations are weak.

The blueprint is a planning baseline, not a prediction of exact questions.

Definitions: distinguish nearby concepts, not just their names.

Calculations: write units, formulas, assumptions and intermediate steps.

Code and SQL: check validity before tracing the text; state any changes before analyzing a corrected version.

Algorithms and protocols: know the input, decision rule, state change, termination and failure cases.

Design choices: state the criterion.

A choice can change when cost, reliability, consistency or latency matters more.

Course group	Items in the 100-item blueprint	Revision implication
Web Programming	9	Reserve extra practice for HTML, CSS, JavaScript, PHP, client/server behavior and safe data handling.
Data Structures and Algorithms	7	Trace operations and compare their assumptions and costs.
Each of the other fourteen courses	6 each	Study all sixteen; do not abandon a whole course based on a short note or an alleged repeated-question list.

START03 How to handle exam traps

Underline NOT, FALSE, BEST and EXCEPT before comparing choices.

Check whether the stem supplies enough facts to select one answer.

A long scenario is not automatically an analysis question, and a familiar answer letter is not evidence.

When code is invalid, a diagram is absent, or two choices express the same rule, identify the clarification needed before answering.

Learn the correct rule and the smallest clarification needed.

Do not memorize an arbitrary key as a universal fact.

Distinguish worst case from average case, and an upper bound from a tight bound.

Check language version, SQL dialect, addressing assumptions, graph weights and tie-breaking rules.

For subnetting and scheduling, calculate rather than guess from a familiar number.

Use timed mixed papers only after you can explain your errors.

Revisit the reading sections behind recurring mistakes.

Compiler Design

Complete Study Notes

COMP01 | Translation, interpretation and compilation time

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.

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.

Compiler Design module

COMP02 | Tools, host, target and incremental work

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.

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.

Compiler Design module

COMP03 | Compiler phases and shared services

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.

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.

Compiler Design module

COMP04 | Token, lexeme and pattern

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.

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.

Example

Scan a statement

- int x = 42; gives keyword(int), ID(x), assignment(=), integer(42), semicolon(;): five tokens.

Compiler Design module

COMP05 | Regular expressions for lexical structure

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.

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.

Compiler Design module

COMP06 | Construct and run a scanner automaton

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.

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.

Compiler Design module

COMP07 | Longest match and counting tokens

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.

For the ordinary C++ header:
for (int i = 0; i <= 10; i++)
The tokens, split into two display lines, are:
for, (, int, i, =, 0, ;
i, <=, 10, ;, i, ++,)
There are 14 tokens, irrespective of spacing.

The increment operator is one token.

A comment contributes no ordinary parser token after preprocessing.

For the loop header above, 4,5,9 and 10 are incorrect token counts; the count is 14.

Compiler Design module

Flex manual: Matching

COMP08 | Locate the error by evidence

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.

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.

Recovery order and exact diagnostics depend on the compiler.

Do not silently replace o by 0 or fro by for.

Compiler Design module

COMP09 | CFGs, derivations and trees

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.

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.

Compiler Design module

COMP10 | Ambiguity, precedence and associativity

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.

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.

Compiler Design module

COMP11 | Top-down parsing

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.

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.

Compiler Design module

COMP12 | Remove left recursion and factor prefixes

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.

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.

Compiler Design module

COMP13 | Compute FIRST and FOLLOW

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.

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.

Example

Small grammar

- S→AB, A→a∣ε, B→b gives FIRST(A)={a,ε}, FIRST(S)={a,b}, FOLLOW(A)={b}, FOLLOW(B)={$}.

Compiler Design module

COMP14 | Build and use an LL(1) table

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.

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.

Exam tip

Exam trap

- Ambiguous useful grammars are not LL(1). Check overlapping FIRST sets, FIRST/FOLLOW conflicts and left recursion as well.

Compiler Design module

Allan Gottlieb, NYU Compilers Lecture 6

COMP15 | Bottom-up actions and handles

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.

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.

Compiler Design module

COMP16 | Construct LR(0) item states

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.

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.

Example

One transition

- For S→a, call the initial closure I0: I0 = {S′→·S, S→·a} GOTO(I0,a) = {S→a·} GOTO(I0,S) = {S′→S·} Each complete set above is one state, not one state per item.

Allan Gottlieb, NYU Compilers Lecture 7

COMP17 | LR families and conflicts

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.

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.

Compiler Design module

GNU Bison manual

COMP18 | Parser limits and operator grammars

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.

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.

Compiler Design module

COMP19 | Symbol tables and scope

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.

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.

Compiler Design module

COMP20 | Names, types and advanced language features

Distinguish lexical recognition, grammatical structure and semantic constraints.

In obj.method(x), the scanner recognizes identifiers and punctuation; the parser recognizes a member call; semantic analysis resolves the receiver, accessible member and argument types.

Overload resolution chooses among signatures using static types; overriding can select an implementation at runtime.

Nested scopes require declaration identity rather than spelling alone: an inner int x can shadow an outer float x. Closures capture free variables from lexical environments; lifetime can outlive the creating stack frame.

Generic functions need parameter constraints or instantiation checks.

These features require symbol/type information beyond a CFG, even when their surface syntax is context-free.

Example

Specify and check a member call

- For object.resize(3), the token kinds are id, dot, id, left parenthesis, integer, right parenthesis. One small grammar is Call → Expr . id (Args); Expr → id; Args → Expr | integer | ε. Derive Call ⇒ Expr . id (Args) ⇒ id . id (integer), then bind the tokens object, resize and 3. The parser checks this shape. Name and type analysis then require object to be in scope, resize to be an accessible member and an overload accepting one integer-compatible argument. If only resize() exists, the text still parses but fails the argument/type check.

Compiler Design module

COMP21 | Semantic checking: static and dynamic

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.

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.

Compiler Design module

COMP22 | Type rules, inference and compatibility

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.

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.

Compiler Design module

COMP23 | Conversions need a named language

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.

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.

C++ working draft, floating-integral conversions

Java Language Specification, conversions

COMP24 | Attributes and syntax-directed definitions

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.

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.

Compiler Design module

COMP25 | Translation actions, ASTs and postfix

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.

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.

Compiler Design module

COMP26 | Intermediate representations and three-address code

IR separates source analysis from target implementation, allowing optimizers and backends to be reused.

It should be easy to generate, analyze, transform and lower.

Portability is an architectural benefit, not proof that every IR omits all target-dependent details.

ASTs, graphs, bytecode and TAC can all be IRs.

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.

Compiler Design module

COMP27 | Control flow, arrays and SSA

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.

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.

Compiler Design module

LLVM Language Reference, phi instruction

COMP28 | Basic blocks, flow graphs and dominators

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.

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.

Compiler Design module

COMP29 | Data-flow questions and equations

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.

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.

Compiler Design module

COMP30 | Folding, propagation and dead code

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.

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.

Compare optimizations by their applicability and effect; several transformations may be appropriate.

Computing an unused volatile read or effectful call may still be required.

Compiler Design module

COMP31 | Reuse, strength reduction and peepholes

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.

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.

Compiler Design module

COMP32 | Loops: invariants, induction and fusion

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.

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.

Compiler Design module

COMP33 | Optimization legality and aliasing

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.

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.

Compiler Design module

COMP34 | Target code generation

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.

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.

Absolute machine code, relocatable object code, assembly and VM bytecode can be legitimate compiler targets; TAC is normally intermediate, but “cannot ever be target code” is an unjustified universal.

Compiler Design module

LLVM target-independent code generator

COMP35 | Descriptors and getReg

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.

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.

Allan Gottlieb, NYU Compilers Lecture 14

COMP36 | Interference graphs and spilling

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.

The general graph k-colorability decision problem is NP-complete for fixed $k \geq 3$; optimal register assignment/spill minimization is a hard optimization problem in general, so practical allocators use heuristics.

Restricted interference graphs or SSA formulations can have different properties.

Do not describe every possible register-allocation algorithm as NP-complete merely because a general model is difficult.

Cornell CS4120, Register Allocation notes

COMP37 | Runtime storage and calling conventions

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.

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.

Compiler Design module

COMP38 | Inlining and dynamic dispatch

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.

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.

Compiler Design module

COMP39 | Recovery and compiler-construction tools

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.

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.

Compiler Design module

COMP40 | Trace one expression through the compiler

For a=b+c*d;, the scanner recognizes names and operators, the parser forms an assignment with +(b,*(c,d)), and semantic analysis resolves declarations and operand/result types.

A declaration such as float a; int b,c,d; can require a conversion before storing the numeric result, according to language arithmetic rules.

Generate t1=c*d; t2=b+t1; a=convert_if_needed(t2).

Constant operands can fold; repeated unchanged expressions can share values.

Finally choose target instructions and registers.

The same spelling a can denote different declarations in nested scopes.

At each step ask what representation enters, what new information is added, and which errors can now be justified.

Compiler Design module

COMP41 | Distinguish tokens, lexemes and lexical analysis

The primary output of lexical analysis is a token stream, rather than a symbol table, AST or assembly; some are side structures. Distinguish a token category from the lexeme text that matches it.

Counting the loop header gives 14 tokens, even if no option matches.

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

Identify whether an analysis groups characters into tokens or tokens into syntactic constructs.

Symbol tables store identifiers and their attributes, not obligatorily every kind of token.

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

Compiler Design module

COMP42 | Compare parser methods and grammar conditions

Shift-reduce and LR are both bottom-up rather than top-down methods.

Recursive and nonrecursive predictive parsers can both be efficient.

Nested parentheses and parsing tables do not uniquely identify LR.

Ambiguity does not disappear merely by choosing LR, and leftmost derivation is not an ambiguity-resolution policy.

Overlapping FIRST sets, left recursion and ambiguity can each prevent a useful grammar from being LL(1); do not restrict the check to only the first two.

Check all applicable conditions and identify any conflicting table cell.

Precedence versus operator-precedence parsing needs a concrete conflict and policy, not a blanket claim that every shift-reduce conflict has one solution.

Compiler Design module

COMP43 | Distinguish semantic analysis and translation

Type checking and symbol tables are both important to semantic analysis: one is an activity and one a data structure.

Semantic rules generally compute attributes, including but not limited to types, values and code; no listed specialized purpose is automatically exhaustive.

Both names and types belong in symbol records.

Distinguish division by zero, bounds violations, indentation mistakes and syntax errors from type errors; a type-error example should specify the language and incompatible operands.

Code generation is conventionally last in a simplified pipeline even if later target optimization exists.

A compiler may emit an annotated AST or typed IR, so distinguish representation, annotation and phase before forcing one tree name.

Compiler Design module

COMP-S01 | Parser direction, expression parsing and coverage

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.

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.

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.

Compiler Design module · page 10

Compiler Design module · page 11

NIT Warangal Compiler Design Lab: operator-precedence parsing

NYU Compiler Construction: simple LR parsing and grammar coverage

Crespi Reghizzi and Pradella: Higher-order Operator Precedence Languages

COMP-S02 | Grammar notation and general CFG parsing algorithms

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.

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.

GNU Bison: Languages and Context-Free Grammars

Lark official documentation: parsing algorithms

Compiler Design module · page 8

COMP-S03 | Equivalent specifications for ordinary token languages

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.

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.

Automata and Complexity Theory module · page 7

Automata and Complexity Theory module · page 8

COMP-S04 | Target-aware instruction scheduling

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.

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.

LLVM - The Target-Independent Code Generator

LLVM - TargetSchedModel reference

Compiler Design module · page 18