Detailed Notes on Compiler Design based on Syllabus

These notes are based on the provided syllabus and the suggested textbooks, aiming to provide a comprehensive understanding of compiler design principles and techniques.

Module I: Introduction to Compilers and Lexical Analysis

1.1 Compiler Introduction:

Definition: A compiler is a program that translates source code written in a high-level language (e.g., C, Java, Python) into an equivalent target code, typically in a lower-level language like assembly language or machine code. The target code can then be executed by a computer.

Purpose: The main goals of a compiler are:

Translation: Convert the source code into a semantically equivalent target code.

Error Detection: Identify and report errors in the source code (syntax, semantic).

Optimization: Improve the target code's efficiency in terms of execution time, memory usage, or code size.

Relationship to Interpreters: Unlike compilers, interpreters execute source code directly, line by line, without generating an intermediate target code. Interpreters offer more flexibility in terms of debugging and platform independence (bytecode-based interpreters). Compilers generally produce faster executing code.

1.2 Analysis of the Source Program:

The compilation process can be divided into two main phases:

1. Analysis Phase (Front End): Breaks down the source program into its constituent parts and creates an intermediate representation. This phase is language-specific.
2. Synthesis Phase (Back End): Constructs the desired target program from the intermediate representation. This phase is target-machine-specific.

1.3 Phases of a Compiler:

A typical compiler consists of several distinct phases, each performing a specific task:

1. Lexical Analysis (Scanning):

Task: Reads the source program character stream and groups them into meaningful sequences called lexemes. Each lexeme is associated with a token, which represents a syntactic category (e.g., identifier, keyword, operator).

Example: `position = initial + rate 60;` becomes:

`id, position`

`=`

`id, initial`

`+`

`id, rate`

``

`number, 60`

`;`

Output: A stream of tokens.

Reference: Aho, Sethi & Ullman, Chapter 3.

1. Syntax Analysis (Parsing):

Task: Takes the token stream from the lexical analyzer and groups the tokens into hierarchical structures representing grammatical phrases. This structure is often a parse tree.

Example: Based on the token stream from the previous example, the parser would create a parse tree reflecting the arithmetic expression's structure.

Output: A parse tree or abstract syntax tree (AST).

Reference: Aho, Sethi & Ullman, Chapter 4.

1. Semantic Analysis:

Task: Checks the parse tree for semantic errors and gathers type information. This includes type checking, ensuring that operators are applied to compatible operands, and resolving variable references.

Example: Checks if `position`, `initial`, and `rate` are of compatible types for the operations performed on them.

Output: An annotated syntax tree or AST.

Reference: Aho, Sethi & Ullman, Chapter 6.

1. Intermediate Code Generation:

Task: Translates the annotated syntax tree into an intermediate representation (IR). IRs are designed to be machine-independent and easy to optimize. Common IRs include three-address code, quadruples, triples, and indirect triples.

Example: The expression `position = initial + rate 60;` could be translated into the following three-address code:

```
t1 = rate 60

t2 = initial + t1

position = t2
```

Output: Intermediate code.

Reference: Aho, Sethi & Ullman, Chapter 6.

1. Code Optimization:

Task: Improves the intermediate code to produce faster-running or more efficient target code. Optimization techniques include constant folding, dead code elimination, common subexpression elimination, and loop unrolling.

Example: The above three-address code could be optimized by constant folding:

```
t1 = rate 60 //This remains the same

position = initial + t1
```

Output: Optimized intermediate code.

Reference: Aho, Sethi & Ullman, Chapter 9.

1. Code Generation:

Task: Transforms the optimized intermediate code into the target language (assembly or machine code). This involves selecting registers, allocating memory, and generating machine instructions.

Example: The optimized intermediate code could be translated into assembly code for a specific architecture (e.g., x86).

Output: Target code.

Reference: Aho, Sethi & Ullman, Chapter 8.

1. Symbol Table Management:

Task: Maintains a symbol table that stores information about identifiers (variables, functions, etc.) used in the source program. This information includes the identifier's name, type, scope, and memory location.

Used by: All phases of the compiler.

Reference: Aho, Sethi & Ullman, Chapter 2.

1. Error Handling:

Task: Detects and reports errors that occur during any phase of compilation. Error messages should be informative and provide context to help the programmer understand and fix the errors.

Integrated with: All phases of the compiler.

1.4 Lexical Analysis:

Role of the Lexical Analyzer (Scanner):

Read the source code character by character.

Group characters into lexemes.

Produce tokens for each lexeme.

Remove whitespace and comments.

Interface with the symbol table.

Report lexical errors (e.g., invalid characters).

Input Buffering:

Purpose: Improves the efficiency of lexical analysis by reducing the number of I/O operations.

Techniques:

Two-Buffer Scheme: Uses two buffers of equal size to read the source code. The scanner reads characters from one buffer while the other buffer is being filled. Requires end-of-file markers.

Sentinel: Placing a special character (e.g., EOF) at the end of each buffer simplifies boundary checking.

Reference: Aho, Sethi & Ullman, Section 3.2.

1.5 Specification of Tokens:

Terminology:

Token: A pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit (e.g., ID, NUMBER, PLUS). The attribute value points to an entry in the symbol table (for identifiers) or holds the numeric value (for numbers).

Lexeme: A sequence of characters in the source program that matches the pattern for a token. (e.g., "abc" is a lexeme for the token ID)

Pattern: A description of the form that the lexemes of a token can take. Patterns are often expressed using regular expressions.

Regular Expressions: A powerful notation for specifying patterns.

Basic Symbols:

\`a\`: Matches the character 'a'.

\`ε\`: Matches the empty string.

Operators:

\`.\`: Concatenation. \`ab\` matches 'a' followed by 'b'.

\`\`: Kleene closure (zero or more occurrences). \`a\` matches ε, a, aa, aaa, ...

\`+\`: Positive closure (one or more occurrences). \`a+\` matches a, aa, aaa, ...

\`?\`: Optional (zero or one occurrence). \`a?\` matches ε or a.

Examples:

\`letter = [A-Za-z]\`

`digit = [0-9]`

Regular Definitions: Assign names to regular expressions. This makes regular expressions more concise and readable. Example: see `letter` and `digit` defined above.

1.6 Recognition of Tokens:

Finite Automata (FA): Mathematical model of a machine that recognizes a regular language. Used to implement lexical analyzers.

Deterministic Finite Automaton (DFA): For each state and input symbol, there is exactly one transition to another state. Easier to implement.

Non-deterministic Finite Automaton (NFA): Allows multiple transitions from a state for the same input symbol or ε-transitions (transitions without consuming any input). Easier to construct from regular expressions.

Conversion from Regular Expressions to DFAs:

1. Convert the regular expression to an NFA. (Thompson's Construction)
2. Convert the NFA to a DFA. (Subset Construction)
3. Minimize the DFA. (Hopcroft's Algorithm)

1.7 Lexical Analyzer Generators:

Purpose: Automate the process of creating lexical analyzers. They take a specification of the tokens (typically a set of regular expressions) as input and generate code for the lexical analyzer.

Examples:

Lex (or Flex): A popular lexical analyzer generator. Takes a `.lex` file containing regular expressions and associated actions (C code to be executed when a token is matched) and generates a C program (usually `lex.yy.c`) containing the lexical analyzer.

ANTLR: Another popular parser generator that also includes lexical analysis capabilities.

Structure of a Lex Specification File:

```

declarations

%%

rules
```

```
%%

user subroutines
```

Example Lex Rule:

```
[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[A-Za-z]+ { yylval = lookup(yytext); return ID; }
```

Module II: Syntax Analysis

2.1 Role of the Parser:

Input: Token stream from the lexical analyzer.

Task: Constructs a parse tree or an abstract syntax tree (AST) based on the grammar of the programming language.

Output: Parse tree or AST.

Error Detection: Reports syntax errors if the token stream does not conform to the grammar.

Interface with: Lexical analyzer and semantic analyzer.

2.2 Context-Free Grammars (CFGs):

Definition: A formal grammar that describes the syntactic structure of a programming language.

Components:

Terminals (T): Basic symbols of the language (e.g., tokens).

Nonterminals (N): Variables representing syntactic categories (e.g., `expression`, `statement`).

Start Symbol (S): A special nonterminal that represents the top-level syntactic category.

Productions (P): Rules that define how nonterminals can be expanded into sequences of terminals and nonterminals.

Example: A grammar for simple arithmetic expressions:

```

```

Derivations: The process of applying production rules to expand the start symbol until a string of terminals is obtained.

Leftmost Derivation: Always expands the leftmost nonterminal.

Rightmost Derivation: Always expands the rightmost nonterminal.

Parse Tree: A graphical representation of the derivation. The root is the start symbol, internal nodes are nonterminals, and leaf nodes are terminals.

Ambiguity: A grammar is ambiguous if there are two or more distinct parse trees (or leftmost/rightmost derivations) for the same input string. Ambiguous grammars can cause problems during parsing and are generally avoided.

2.3 Top-Down Parsing:

Approach: Starts with the start symbol and tries to derive the input string by expanding nonterminals. Builds the parse tree from the root down to the leaves.

Techniques:

Recursive-Descent Parsing: Implements each nonterminal as a function. The function tries to match the production rules for that nonterminal.

Requires: Grammar must be LL(1) (see below).

Predictive Parsing: A special form of recursive-descent parsing where the next production rule to apply is determined by looking at the next input token (lookahead). Avoids backtracking.

LL(1) Grammar: A grammar for which a predictive parser can be built. LL(1) stands for "Left-to-right scan, Leftmost derivation, 1 symbol lookahead." A grammar is LL(1) if, for each nonterminal, the production rules starting with that nonterminal can be distinguished based on the first terminal symbol that can be derived from each rule.

Problems:

Left Recursion: A production rule where the leftmost symbol on the right-hand side is the same as the nonterminal on the left-hand side (e.g., `E -> E + T`). Causes infinite recursion in recursive-descent parsers. Must be eliminated.

Backtracking: Trying different production rules and undoing the parse if a rule does not lead to a successful derivation. Inefficient. Avoided in predictive parsing.

Left Factoring: If two or more production rules for the same nonterminal have a common prefix, the grammar can be left-factored to remove the common prefix and make the grammar more suitable for predictive parsing.

2.4 Bottom-Up Parsing:

Approach: Starts with the input string and tries to reduce it to the start symbol by repeatedly applying production rules in reverse. Builds the parse tree from the leaves up to the root.

Techniques:

Shift-Reduce Parsing: Uses a stack to hold grammar symbols and an input buffer to hold the remaining input string.

Shift: Moves the next input token from the input buffer onto the stack.

Reduce: Replaces a sequence of grammar symbols on the top of the stack (a handle) with the nonterminal on the left-hand side of a corresponding production rule.

Accept: The input string is accepted when the stack contains only the start symbol and the input buffer is empty.

Error: A syntax error is detected when no action (shift or reduce) is possible.

Operator Precedence Parsing: A simple bottom-up parsing technique that is suitable for grammars where the precedence and associativity of operators are clearly defined. Uses precedence relations between operators to guide the parsing process.

LR Parsing: A more powerful bottom-up parsing technique that can handle a wider range of grammars than operator precedence parsing. LR parsers use a state machine to keep track of the current parsing context. LR stands for "Left-to-right scan, Rightmost derivation in reverse."

SLR Parsing: Simple LR parsing.

Canonical LR Parsing: Most powerful LR parsing technique, but can require a large state machine.

LALR Parsing: Look-Ahead LR parsing. A compromise between SLR and Canonical LR. Reduces the size of the state machine compared to Canonical LR, while retaining much of its power. LALR(1) is the most common parsing technique and used in most parser generators.

Conflicts: Shift-reduce and reduce-reduce conflicts can occur in bottom-up parsing. These conflicts indicate that the grammar is not suitable for the parsing technique being used and must be resolved by modifying the grammar or using a more powerful parsing technique.

2.5 Parser Generators:

Purpose: Automate the process of creating parsers. They take a grammar specification as input and generate code for the parser.

Examples:

Yacc (or Bison): A popular parser generator. Takes a `.y` file containing the grammar specification and associated actions (C code to be executed when a production rule is reduced) and generates a C program (usually `y.tab.c`) containing the parser.

ANTLR: A powerful parser generator that supports multiple target languages and grammars including LL() and LR grammars.

Structure of a Yacc Specification File:

```
declarations

%%

grammar rules

%%

user subroutines
```

Example Yacc Rule:

```
expression : expression '+' term { $$ = $1 + $3; }

;
```

Module III: Syntax-Directed Translation, Type Checking, and Run-Time Environment

3.1 Syntax-Directed Translation:

Concept: Augmenting a grammar with rules or actions that describe how to translate the syntax of the language into an intermediate representation or target code.

Syntax-Directed Definitions (SDDs): A generalization of context-free grammars in which each grammar symbol has a set of attributes, and each production has a set of semantic rules that define how the attributes are computed.

Attributes: Variables associated with grammar symbols. They can represent information about the symbol, such as its type, value, or location.

Semantic Rules: Define how to compute the values of attributes. Semantic rules can access the attributes of the grammar symbols in the production rule.

3.2 S-Attributed Definitions:

Definition: SDD where all attributes are synthesized attributes.

Synthesized Attribute: Its value at a node in the parse tree is determined only by the attribute values at its children.

Evaluation: Can be evaluated during a bottom-up parse. Easy to implement.

3.3 L-Attributed Definitions:

Definition: SDD where attributes can be synthesized or inherited, but the values of inherited attributes must be computed in a way that allows the parse tree to be evaluated in a single pass, either top-down or bottom-up.

Inherited Attribute: Its value at a node in the parse tree is determined by the attribute values at its parent and/or siblings.

Evaluation: Can be evaluated during a top-down parse or a bottom-up parse with some restrictions.

Restriction: For a production `A -> X1 X2 ... Xn`, the inherited attribute `Xi.a` can depend only on:

The attributes of `A`.

The attributes of `X1, X2, ..., Xi-1`.

Example: Type checking of declarations can be implemented using L-attributed definitions.

3.4 Top-Down and Bottom-Up Translation:

Top-Down Translation: Evaluation of semantic rules is performed during a top-down parse.

Bottom-Up Translation: Evaluation of semantic rules is performed during a bottom-up parse. Requires S-attributed definitions or L-attributed definitions that can be evaluated bottom-up.

3.5 Type Checking:

Purpose: Verify that the type of each expression is consistent with the operations being performed on it.

Tasks:

Type Inference: Inferring the type of an expression based on the types of its operands.

Type Conversion (Coercion): Converting a value from one type to another.

Type Checking Errors: Reporting type errors when the types of operands are incompatible or when a type conversion is not possible.

Static Type Checking: Type checking performed at compile time. Catches errors early. Languages like Java, C++, and C# use static type checking.

Dynamic Type Checking: Type checking performed at runtime. More flexible, but errors are not detected until runtime. Languages like Python, JavaScript, and Ruby use dynamic type checking.

3.6 Type Systems:

Definition: A set of rules for assigning types to expressions.

Components:

Basic Types: Integer, real, boolean, character.

Type Constructors: Arrays, pointers, functions, records, classes.

Type Equivalence: Rules for determining when two types are the same. (Name equivalence, Structural equivalence)

Type Compatibility: Rules for determining when a value of one type can be used in a context where a value of another type is expected.

3.7 Specification of a Type Checker:

Type checkers can be implemented using syntax-directed translation.

Attributes are used to store type information.

Semantic rules are used to check the types of expressions and report errors.

Example: Consider an assignment statement:

```
E -> id = E1
```

Semantic rule:

```
E.type = if (id.type == E1.type) then id.type else ERROR
```

3.8 Run-Time Environment:

Purpose: Provides the environment in which the target program executes.

Source Language Issues:

Static vs. Dynamic Scope: How variable names are resolved to memory locations.

Parameter Passing Mechanisms: How arguments are passed to functions (e.g., by value, by reference).

Memory Management: How memory is allocated and deallocated (e.g., static allocation, stack allocation, heap allocation).

3.9 Storage Organization:

Code Area: Stores the executable code of the program.

Static Data Area: Stores global variables and constants.

Stack Area: Stores activation records for function calls.

Heap Area: Stores dynamically allocated memory.

3.10 Storage Allocation Strategies:

Static Allocation: Memory is allocated at compile time. Suitable for global variables and constants.

Stack Allocation: Memory is allocated on the stack for function calls. Activation records are pushed onto the stack when a function is called and popped off the stack when the function returns.

Heap Allocation: Memory is allocated dynamically at runtime using functions like `malloc` and `free` (in C) or `new` and `delete` (in C++).

3.11 Access to Nonlocal Names:

Static Scope: The scope of a variable is determined by the static structure of the program (the nesting of functions). Lexical scope.

Dynamic Scope: The scope of a variable is determined by the calling sequence of functions.

Implementation Techniques:

Static Links: Each activation record contains a pointer to the activation record of its lexically enclosing function.

Display: An array that stores pointers to the activation records of the currently active functions along the static nesting chain.

3.12 Symbol Tables:

Purpose: Store information about identifiers (variables, functions, etc.).

Information Stored: Name, type, scope, memory location, parameters (for functions), etc.

Implementation Techniques:

Linear Lists: Simple but inefficient for large symbol tables.

Hash Tables: More efficient for large symbol tables. Provide fast insertion, deletion, and lookup operations.

Binary Search Trees: Provide logarithmic time complexity for insertion, deletion, and lookup operations.

Module IV: Intermediate Code Generation, Code Optimization, and Code Generation

4.1 Intermediate Code Generation:

Purpose: Translate the annotated syntax tree into an intermediate representation (IR).

Benefits of IR:

Machine independence.

Ease of optimization.

Portability: A compiler can be built for multiple target machines by changing only the back end.

Common Intermediate Languages:

Three-Address Code: A sequence of instructions of the form `x = y op z`, where `x`, `y`, and `z` are variables, constants, or temporaries, and `op` is an operator.

Quadruples: A record structure with four fields: `op`, `arg1`, `arg2`, `result`.

Triples: Similar to quadruples, but uses the position of the instruction in the triple list to refer to intermediate values, rather than explicit names.

Indirect Triples: Uses a list of pointers to triples, allowing for easier code optimization.

4.2 Declarations:

Generating intermediate code for declarations involves allocating space for variables and entering information about the variables into the symbol table.

4.3 Assignment Statements:

Generating intermediate code for assignment statements involves evaluating the expression on the right-hand side and storing the result in the memory location of the variable on the left-hand side.

4.4 Boolean Expressions:

Short-Circuit Evaluation: Evaluating boolean expressions only as far as necessary to determine the result.

Translation to Three-Address Code: Boolean expressions can be translated into three-address code using jump instructions.

Example:

```

// ...

}

```

Could be translated into:

```
if a < b goto L1

goto L2

L1:

if c > d goto L3

L2:

goto L4

L3:

// ...

L4:
```

4.5 Procedure Calls:

Generating intermediate code for procedure calls involves passing arguments to the procedure, saving the return address, and transferring control to the procedure.

4.6 Code Optimization:

Purpose: Improve the intermediate code to produce faster-running or more efficient target code.

Sources of Optimization:

Local Optimization: Optimizations performed within a single basic block (a sequence of instructions with no jumps in or out, except at the beginning and end).

Global Optimization: Optimizations performed across multiple basic blocks.

Loop Optimization: Optimizations performed within loops.

Optimization Techniques:

Constant Folding: Evaluating constant expressions at compile time.

Dead Code Elimination: Removing code that is never executed.

Common Subexpression Elimination: Replacing multiple occurrences of the same expression with a single calculation.

Copy Propagation: Replacing a variable with its value if the variable is assigned a constant value or another variable.

Code Motion: Moving code out of loops to reduce the number of times it is executed.

Strength Reduction: Replacing expensive operations with cheaper ones (e.g., replacing multiplication by a power of 2 with a left shift).

Loop Unrolling: Replicating the body of a loop multiple times to reduce loop overhead.

4.7 Introduction to Data Flow Analysis:

Purpose: Collect information about the flow of data in a program.

Used for: Global code optimization.

Techniques:

Reaching Definitions: Determining which definitions of a variable may reach a given point in the program.

Live Variable Analysis: Determining which variables may be used after a given point in the program.

4.8 Code Generator:

Purpose: Translate the optimized intermediate code into the target language (assembly or machine code).

Issues in the Design of a Code Generator:

Target Machine Selection: The choice of the target machine affects the design of the code generator.

Register Allocation: Assigning registers to variables to reduce memory access.

Instruction Selection: Choosing the best machine instructions to implement the intermediate code.

Instruction Scheduling: Ordering the instructions to improve performance.

4.9 The Target Machine:

Characteristics of Target Machines: Instruction set, addressing modes, register set, memory architecture.

Example: x86 architecture.

4.10 A Simple Code Generator:

Input: Optimized intermediate code.

Output: Assembly code.

Example: Generating x86 assembly code from three-address code.

Tasks:

Register allocation.

Instruction selection.

Generating labels for jumps and branches.

These notes provide a comprehensive overview of compiler design based on the provided syllabus and suggested textbooks. Remember to refer to the textbooks for more detailed explanations and examples. Good luck with your studies!

`	**`: Alternation (or). `a**	**b` matches either 'a' or 'b'.**
`id = letter (letter	digit)`	
`number = digit+ (. digit+)? (E (+	-)? digit+)?`	
E -> E + T	T	
T -> T F	F	
F -> (E)	id	
if (a < b	c > d) {	