

LANGUAGE PROCESSORS

(CST412)



Syllabus

UNIT-I Introduction to Compilers- Introduction to Compilers, Phases of compiler design, cross compiler, Bootstrapping, LEX, Tombstone Diagram, Relating Compilation Phases with Formal Systems

Lexical Analysis- Lexical analysis and tokens, input buffering, tokens, pattern and lexemes, Design of Lexical analyzer, Regular Expression, transition diagram, recognition of tokens, Lexical Errors, NFA and DFA for Lexical Analysis, Direct Method for Conversion of RE to DFA, DFA Minimisation.

UNIT-II Syntax Analysis- Specification of syntax of programming languages using CFG, Top-down parser, design of LL(1) parser, bottom up parsing technique, Handle and Viable Prefix, LR parsing, Design of SLR, CLR, LALR parsers, Parser Conflicts, Handling Ambiguous Grammars, Applications of the LR Parser, YACC.

UNIT-III Syntax directed translation- Study of syntax directed definitions & syntax directed translation schemes, Type and Type Checking, A Simple Type Checking System, Type Conversion and equivalence, implementation of SDTS, intermediate notations- postfix, syntax tree, TAC, translation of Assignment Statement, expressions, controls structures, declarations, procedure calls, Array reference.

Syllabus

UNIT-IV Storage allocation & Error Handling- Run time storage administration stack allocation, Activation of Procedures, Storage Allocation Strategies, Garbage Collection, symbol table management, Error detection and recovery- lexical, syntactic and semantic.

UNIT-V Code optimization- Machine-independent Optimisation- Algebraic Simplification and Strength Reduction, Dead Code Elimination, loop optimization, control flow analysis, data flow analysis, Loop invariant computation, Induction variable removal, Elimination of Common sub expression, Function Inlining and Cloning, other techniques and Machine-dependent Optimisation.

UNIT-VI Code generation – Problems in code generation, Target Machine, Instruction Cost, Simple code generator, Register allocation and assignment, Register allocation by Graph Colouring, Code generation from DAG, Code Generation by Dynamic Programming.

Textbooks




1. Aho, Sethi, and Ullman; Compilers Principles Techniques and Tools; Second Edition, Pearson education, 2008.
2. Alfred V. Aho and Jeffery D. Ullman; Principles of Compiler Design; Narosa Pub.House, 1977.
3. Vinu V. Das; Compiler Design using Flex and Yacc; PHI Publication, 2008.
4. Manoj B Chandak, Khushboo P Khurana; Compiler Design; Universities Press, 2018.

Course Outcomes



- CO1. Exhibit role of various phases of compilation, with understanding of types of grammars and design complexity of compiler.
- CO2. Design various types of parses and perform operations like string parsing and error handling.
- CO3. Demonstrate syntax directed translation schemes, their implementation for different programming language constructs.
- CO4. Implement different code optimization and code generation techniques using standard data structures.

Introduction



01000011010011110100110101010000010101
01010101000100010101010010.

Introduction



- **Problem:**

Very difficult to understand and find errors. A single digit if mistyped or left out while typing, changes the meaning of the program and may create errors.

- **Solution:** High Level Language

English like, not understood by computers

Again a problem?

Solution: **Compiler**

Compiler



- ❑ The output of compilation is the object code.
- ❑ Some compilers provide output in assembly language, which is then converted to machine language by an assembler
- ❑ C Compiler:
C program (".c") → C Compiler → object code (".o" / ".obj")
- ❑ Java Compiler produces byte code

Very first compiler

- When the very first compiler was developed, no other compilers were available- written in **assembly language**.
- For a completely new language - the compiler or interpreter must be **written in another language**.
- For example, Y Niklaus wrote the first Pascal compiler in Fortran.
- The compiler may be rewritten in its own language and become a self-hosting compiler.

C Compiler history

- 1960s: Digital Equipment Corporation (DEC) introduced the **PDP series of minicomputers**
- The first version of Unix was written in the low-level assembly language of PDP-7.
- TMG (TransMoGrifier) language was created for PDP-7 and used by Ken Thompson to develop a FORTRAN compiler.
- But he ended up creating compiler for a new high-level language called B.
- **B language** was influenced by an earlier language called Basic Common Programming Language (BCPL).
- BCPL - a huge amount of assembly code to accomplish a given task.
- B was designed to perform the same functionality in just a few lines of code.

C Compiler history

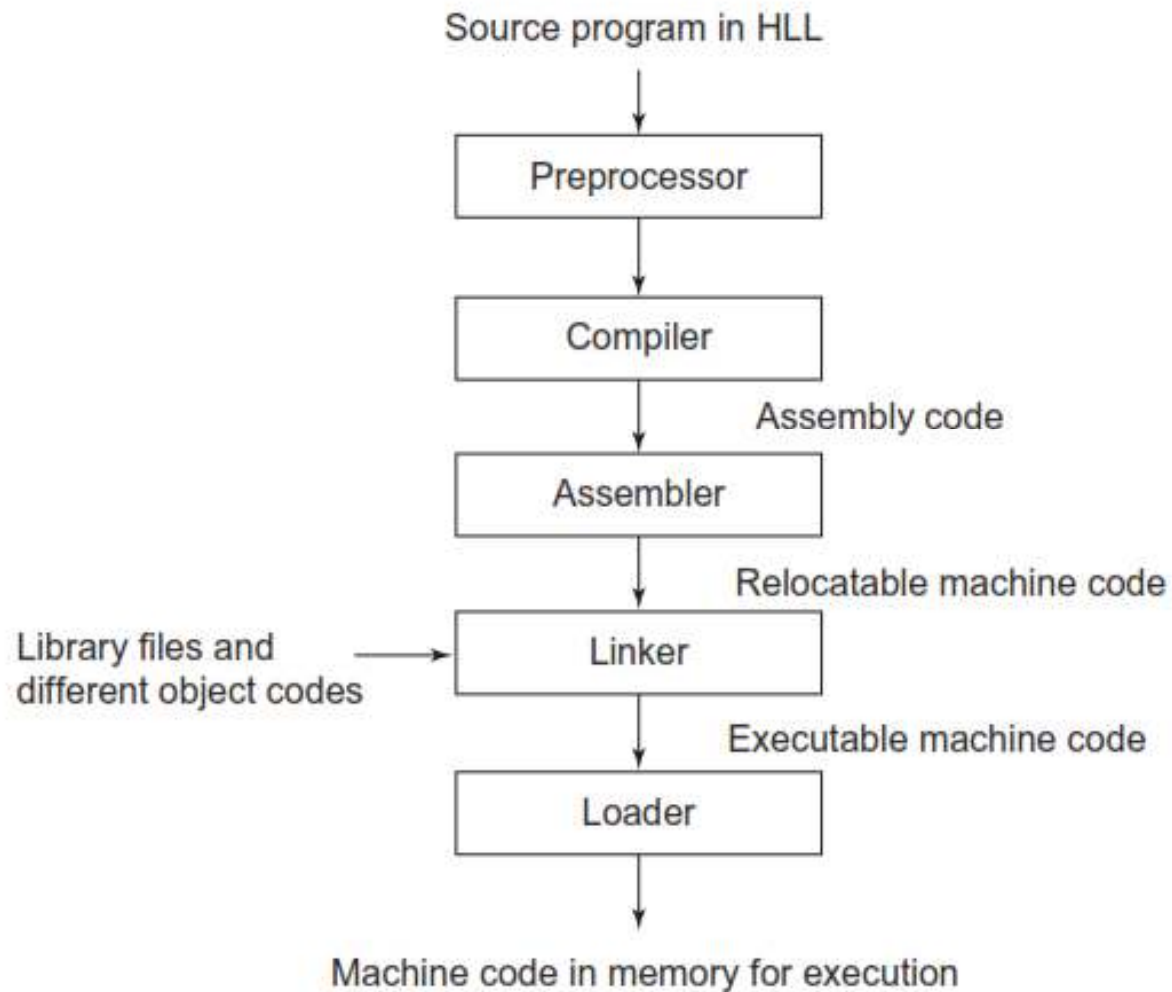
- C language is descendant of B language, written when the PDP-11 computer arrived at Bell Labs.
- Dennis Ritchie used B to create a new language-NB or NewB
- NB was later refined to C.
- The first C compiler created by Dennis Ritchie was written in the short-lived language NB.
- C itself was refined many times.
- The latest versions of GCC (GNU Compiler Collection) distribution containing C and C++ compilers that were written in C are moving towards C++.

Java Compiler history

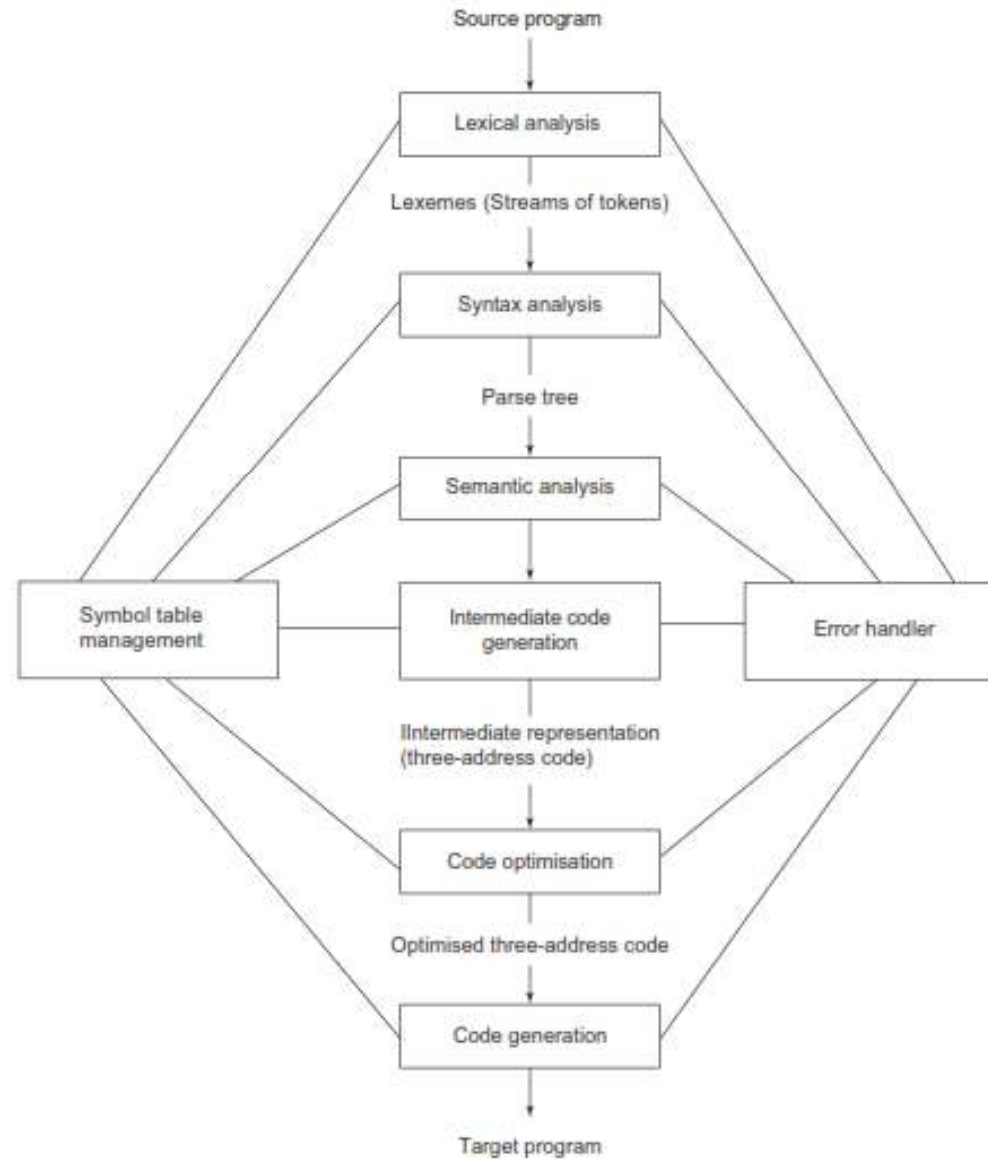


- ❑ The first Java compiler created by Sun Microsystems
- ❑ Written in C and used some libraries from C++
- ❑ Today, the Java compiler is written in Java while the Java Runtime Environment (JRE) is written in C.
- ❑ The Java compiler was written as a program in Java and compiled using a Java compiler written in C-bootstrap compiler.
- ❑ written in Java and which could compile Java programs.

Language Processing System



Phases of Compiler



Lexical Analysis

- first phase of the compilation process
- Source Program → Lexical analyser → Stream of tokens
- Tokens- logically cohesive sequence of characters
- Common examples of tokens are:
 - ▣ Keywords
 - ▣ Operators
 - ▣ Identifiers
 - ▣ Symbols
 - ▣ Constants
 - ▣ Strings

Lexical Analysis

C code which prints numbers from 1 to 10,

```
void main( )
{
int count;
for(count = 1; count <= 10; count ++ )
printf("%d", count);
printf("\n");
}
```

34 tokens will be identified as follows:

```
void      main      (      )
{
int       count     ;
for (    count     =      1  ; count     <=  10  ;      count     ++      )
printf (   "%d"    ,      count      )      ;
printf (   "\n"    )      ;
}
```


Lexical Analysis

- The symbol table stores
<token-type, attribute value> pairs for all the tokens identified.
- Consider the statement: `int count = 1;`

Token	Token type	Symbol table entries
int	keyword	<keyword, int>
count	identifier	<id, pointer to symbol table entry of count>
=	operator	<assign_op, >
1	constant	<constant, 1>
;	symbol	<symbol, ;>

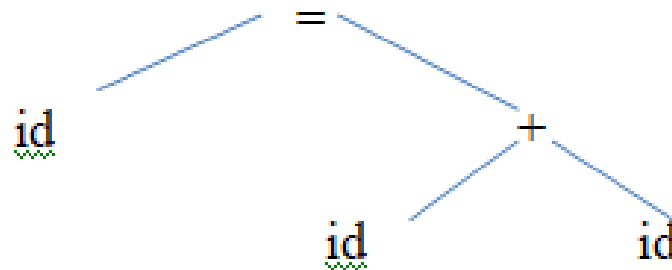
Syntax Analysis



- Parser
- Tokens → Syntax analyser → parse tree
- Validates syntax: checks if tokens appear in the patterns that are permitted by the specification for the source language

example

A=B+C → Lexical analyser → id = id + id → Syntax
Analyser → Parse tree



Parse tree for A=B+C

Semantic Analysis

- Checks the source program for semantic errors
- Performs type checking
- Checks if the token is declared before its use
- Checks if identifiers are used in the appropriate context

- Checks subroutine call arguments and labels
- Dynamic semantic checks are performed at run-time

Eg: array index within bounds, arithmetic errors such as division by zero, pointers not de-referenced unless pointing to valid objects and that there is no un-initialised variable.

Intermediate Code Generation

- Transforms the parse tree into an intermediate language representation
- Three address representation of continued example

$T1 = B + C$

$A = T1$

Code Optimisation

- apply transformations to the output of the intermediate code generator to generate faster or smaller object language programs.
- Simple optimisations can significantly improve the running time of the target program without slowing down the compilation time
- Some of these optimising techniques are as follows:
 - ▣ Local optimisation
 - ▣ Loop optimisation
 - ▣ Dead code elimination
 - ▣ Copy propagation
 - ▣ Common sub-expression elimination

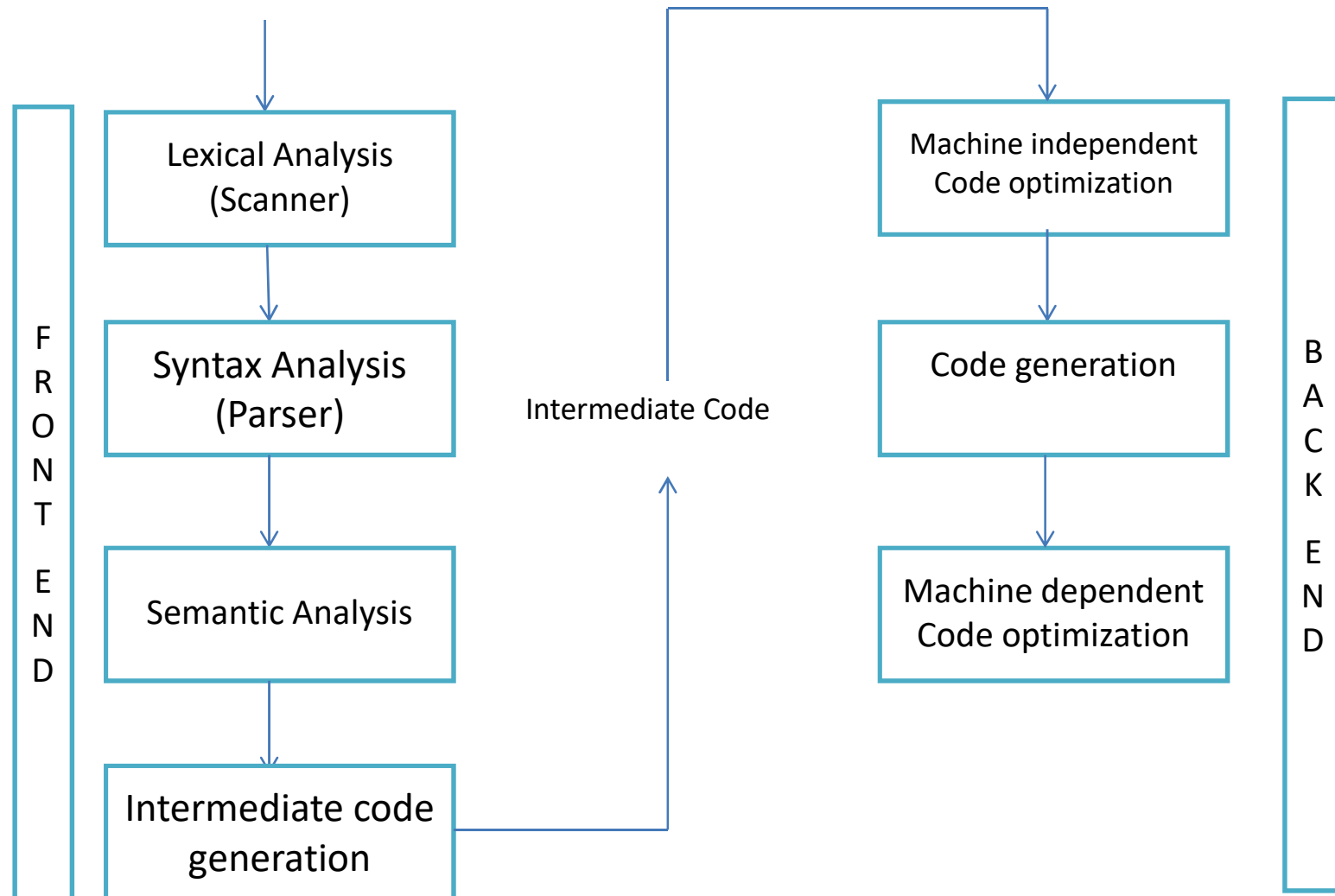
Code Generation

- ❑ This phase converts intermediate code into a sequence of machine instructions.
- ❑ Intermediate code in machine code as:
 - Load B
 - Add C
 - Store A
- ❑ Code optimisation can be applied after the code generation phase. This is called machine-dependent code optimisation,
- ❑ takes advantage of the special features of the target machine.
- ❑ It involves CPU registers and may have absolute memory references rather than relative references.
- ❑ Machine-dependent optimisers take maximum advantage of memory hierarchy.

Symbol Table and Error Handling

- Each phase uses a symbol table and error handler.
- The symbol table stores information about the occurrence of various entities such as
 - ▣ variable names
 - ▣ function names
 - ▣ Objects
 - ▣ classes
 - ▣ interfaces
- Phases may store information or use the information from the symbol table.
- The error handler is invoked by any phase if an error has occurred.

Overview of Compiler Design



Pass and Phase



Phase:

logically cohesive operation

takes input from the previous stage → processes the data → yields output (used as input for the next phase)

Pass:

traversal of a compiler through the entire program

Reads the source program or output of the previous pass → Applies transformations → writes the output in an intermediate file

Pass and Phase

Pass	Phase
It is a physical scan over a source program i.e, how many times the source code will be scanned.	It accepts the input in one representation of source program and produces output in another representation.
Scans over the source program, process it and stores it in an intermediate file.	It pass the processed information from one phase to next phase.
Intermediate file is needed between each pass.	Intermediate file is not needed between phases
Splitting into more number of passes reduces memory.	Splitting into more number of phases reduces complexity not memory
Single pass compiler is faster than multi-pass compiler	Reduction in number of phases gives complexity.

Application of Techniques Used in Compiler Design

CD techniques can be applied to other domains of computer science:

- Techniques used in the lexical analyser can be used in text editors, information retrieval systems, query languages, etc.
- Parser techniques can be used in query processing systems like SQL and in pattern recognition systems.
- Most of the techniques can be applied to natural language processing.

Relating Compilation Phases with Formal Systems



Lexical Analysis:

- Regular expression (RE)
- A finite automata (FA)

Syntax analysis:

- A context-free grammar (CFG) is used to specify the rules of the language.
- PDA (pushdown automata) to implement the CFG to recognise valid language constructs.

Relating Compilation Phases with Formal Systems

Intermediate code generation:

- Parse tree can be converted into a linear representation, for example, postfix notation.
- Intermediate code can be represented using quadruple, triple or indirect triple notation

code optimisation:

- Data flow analysis uses fixed-point algorithms.
- Dead code elimination uses graph algorithms.

Machine code generation:

- Computer architecture is used
- Greedy algorithms and graph-based algorithms used for register allocation.
- Dynamic programming techniques used for instruction selection.
- Complex data structures are utilised by symbol tables, parse trees and data-dependence graphs.

Compiler Construction Tools



- Scanner generators
- Parser generators
- Syntax-directed translation engines
- Automatic code generators
- Data flow engines