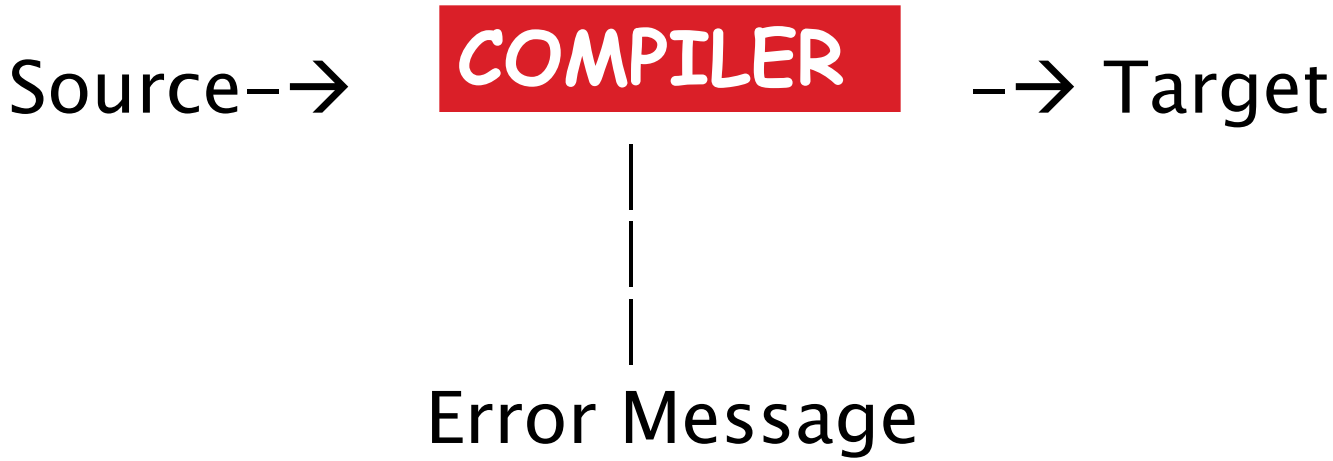
Harcourt butler technical University Kanpur

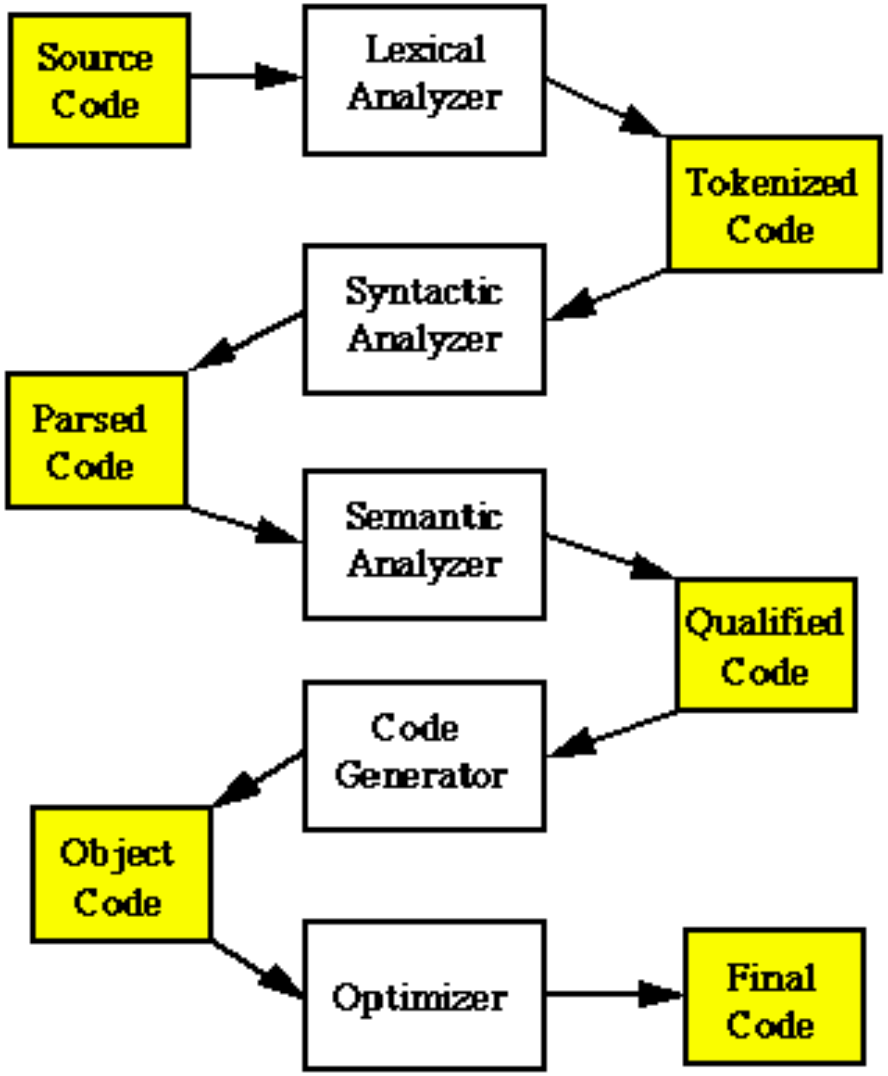
COMPILER Design

DEFINITION

- A compiler is a program that reads a program written in one language and translates into equivalent target language



Language Processing Technique



Structure of a compiler

- The **Front end** checks whether the program is correctly written in terms of the programming language syntax and semantics

- The **back end** is responsible for translating the source into assembly code.

Structure of a compiler

Front End :

- Lexical Analysis
- Preprocessing
- Syntax Analysis
- Semantic Analysis

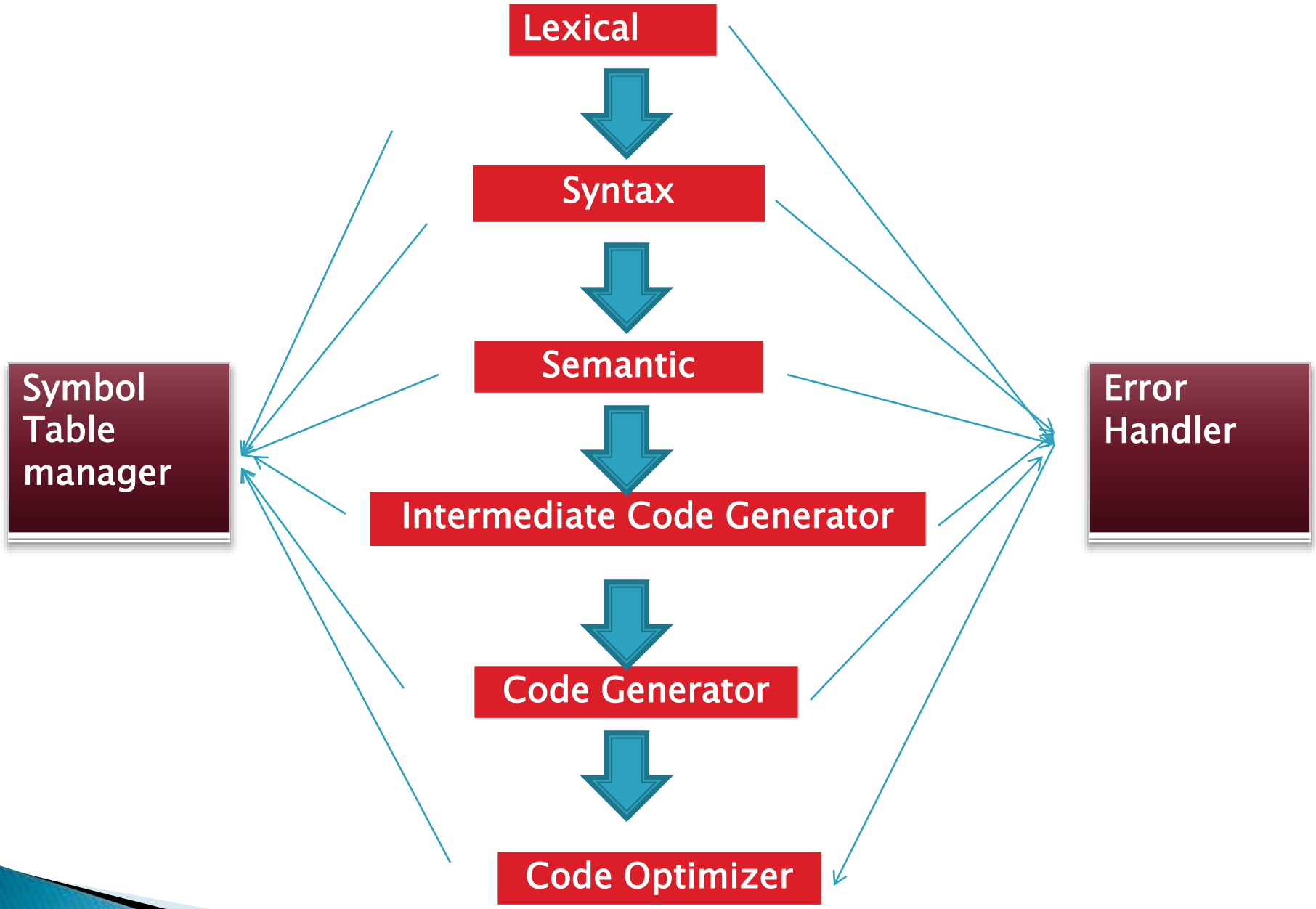
Structure of a compiler

Back End

- Analysis
- Optimization
- Code generation

Phases of a compiler

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate code generator
- Code optimizer
- Code generator


Lexical
Syntax
Semantic
Intermediate Code Generator
Code Generator
Code Optimizer
Symbol Table manager
Error Handler

Lexical analysis

- Also called Linear Analysis
- Characters read from left to right and grouped into tokens that are a sequence of characters with a collective meaning

→Scans Input

→Removes White spaces and comments

→Manufacture Tokens

→Generate Error if Any

Lexical analysis

- <u>Example</u>

- A=B+C

- Variable tokens -→ A ,B, C
- Symbolic token --→ = +

SKIP

Lexical Analyzer Generator

Lex(Flex in recent implementation)

SKIP

What is Lex?

- The main job of a *lexical analyzer (scanner)* is to break up an input stream into *tokens(***tokenize** input streams)*.
- *Ex :-*

  `a = b + c * d;`

  ID ASSIGN ID PLUS ID MULT ID SEMI

- Lex is an utility to help you rapidly generate your scanners

SKIP

Structure of Lex Program

- Lex source is separated into three sections by %% delimiters
- The general format of Lex source is

```
{definitions}
%%
{transition rules}
%%
{user Code}
```

(required)

(optional)

The absolute minimum Lex program is thus

```
%%
```

SKIP

Definitions

- Declarations of ordinary C variables ,constants and Libraries.

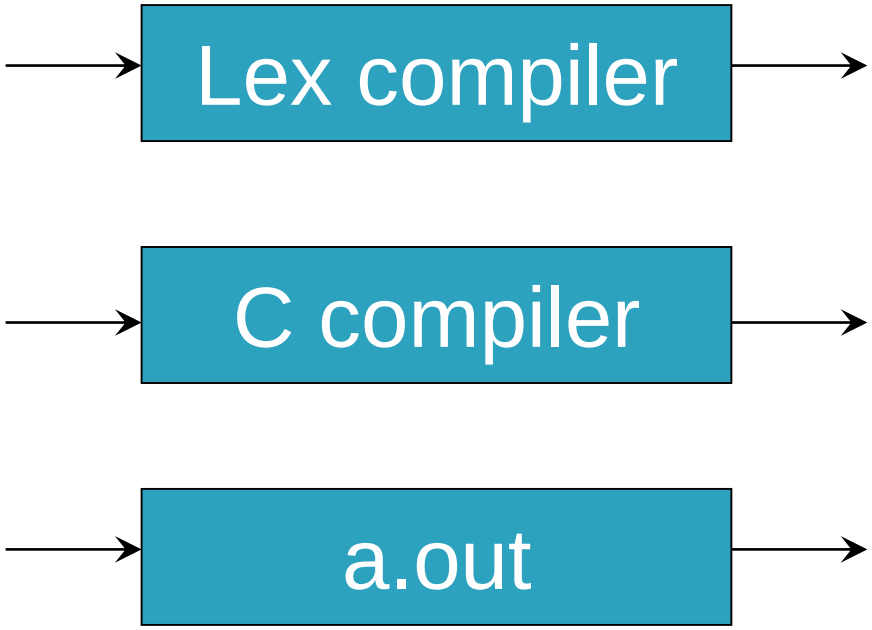
```
%{
#include <math.h>
 #include <stdio.h>
   #include <stdlib.h>
%}
```

- flex definitions :- name definition

  Digit [0-9] (Regular Definition)

An Overview of Lex

SKIP



SKIP

Lex Predefined Variables

- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - the default input of default main() is **stdin**
- yyout -- the output stream pointer
  - the default output of default main() is **stdout**.

SKIP

Lex Library Routines

- yylex()
  - The default main() contains a call of yylex()
- yymore()
  - return the next token
- yyless(n)
  - retain the first n characters in yytext
- yywarp()
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1

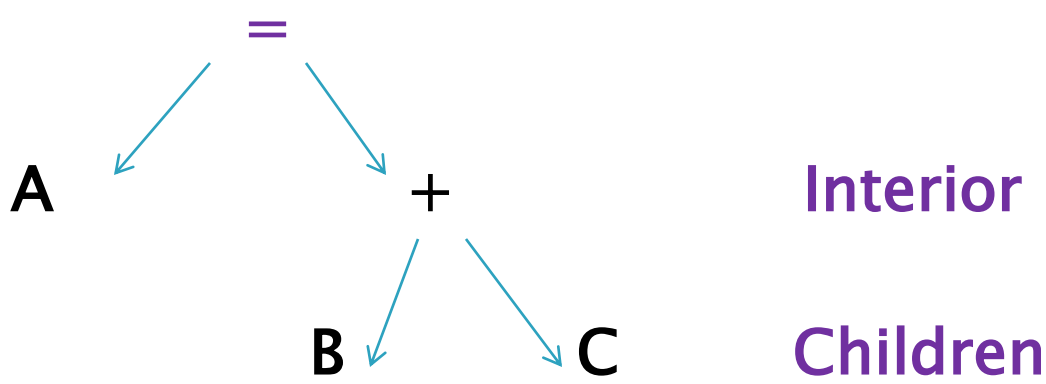
SKIP

Review of Lex Predefined Variables

Name	Function
`char *yytext`	pointer to matched string
`int yyleng`	length of matched string
`FILE *yyin`	input stream pointer
`FILE *yyout`	output stream pointer
`int yylex(void)`	call to invoke lexer, returns token
`char* yymore(void)`	return the next token
`int yyless(int n)`	retain the first n characters in yytext
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`ECHO`	write matched string
`REJECT`	go to the next alternative rule
`INITAL`	initial start condition
`BEGIN`	condition switch start condition

Syntax Analysis

- Also called as Hierarchial Analysis
- A syntax tree[also called as parse tree] is generated where
  - Operators → Interior nodes
  - Operands → Children of node for operators.



Semantic analysis

- Characters grouped as tokens in Lexical Analysis are recorded as Tables. Checks for semantic errors

- Collect TYPE information for the subsequent code generation phase

Intermediate Code Generator

- Sophisticated compilers typically perform multiple passes over various intermediate forms.
- Many algorithms for code optimization are easier to apply one at a time
- The input to one optimization relies on the processing performed by another optimization

Working of ICG

SKIP

Concrete Parse tree

Abstract syntax tree

Converted into a linear sequence of instructions

Results in 3AC [3 Address Code]

Code optimization

- This phase attempts to improve the intermediate code inorder to increase the running time
- Reduce the complexity of the code generated
- Leading to a faster execution of the program
- Increased Performance

Code optimization

- Platform Dependant/ Platform Independent
- Optimization can be automated by compilers or performed by programmers
- Usually, the most powerful optimization is to find a superior algorithm.
- Include activities like
  - Optimization of LOOPS
  - Optimization of Bottlenecks

Code generator

- Succeeding step of Intermediate code optimizer
- Consists of re-locatable machine code/assembly code
- Intermediate instructions are converted into a a sequence of machine instructions

SKIP

Work with Our Compiler

THANK

YOU