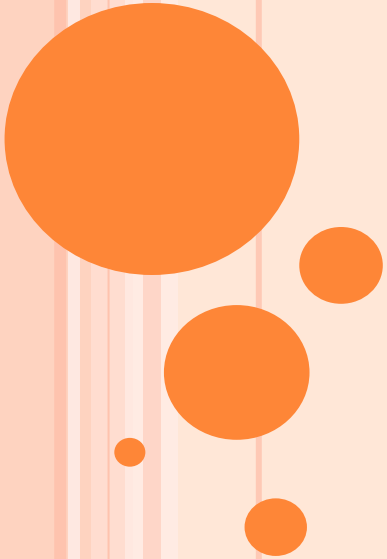
UCS 802 COMPILER CONSTRUCTION



PRELIMINARIES REQUIRED

- Basic knowledge of programming languages.
- Basic knowledge of FSA and CFG.
- Knowledge of a high programming language for the programming assignments.

Laboratory Work:

- Lexical Analysis and Syntax Analysis: Construction of DFA from RE and construction of SLR

Text Books:

- A. V. Aho, Ravi Sethi, J. D. Ullman, "Compilers Tools and Techniques", Addison-Wesley
- Allen I. Holoub, "Compiler Design in C", PHI

Reference Books:

- Barret W. A., J. D. Couch, "Compiler Construction Theory and Practice", Computer Science Series- Asian Student Edition
- D.M. Dhamdere, "Compiler Construction –Principles and Practice", Mcmillan , India.

Evaluation pattern followed Earlier

Total Marks (100)

- MST : 24
- Quizzes : 16
- Practical evaluation : 20
- EST : 40

WHAT IS A TRANSLATOR?

A **translator** is a system software which translates the language written by the user in a to the form understandable by machine.

Machine understands the language of 0's and 1's only so the role of translator is to convert human readable language to machine language.

TYPE OF TRANSLATORS

1) **Compiler:** Converts HLL to Machine language or Assemble Language

2) **Interpreter:** Converts HLL to Machine language

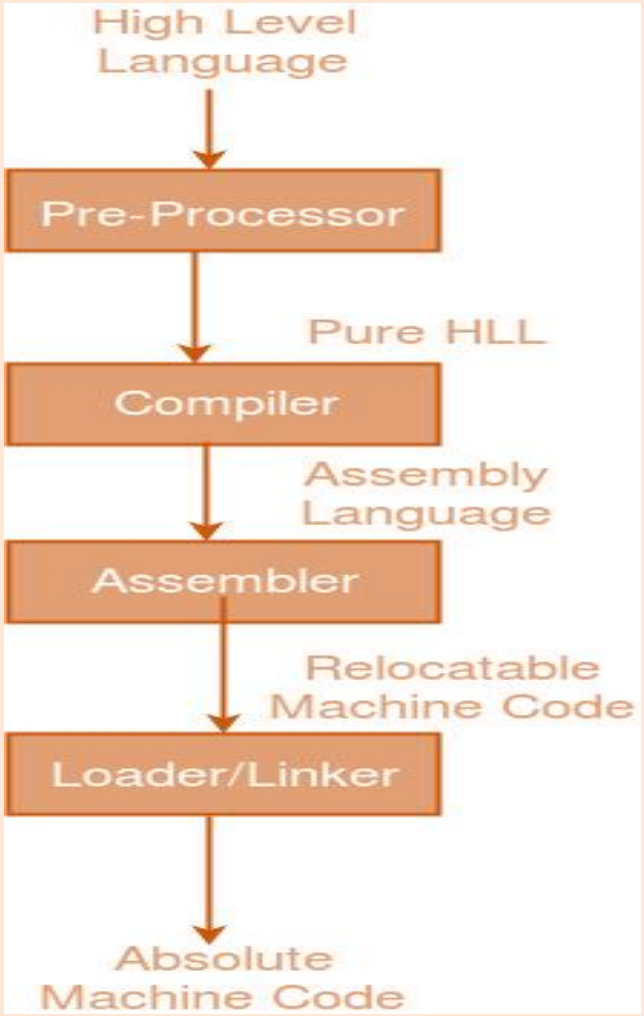
3) **Assembler**: Assemble language i.e. language of pneumonic to machine language. MUL,ADD,SUB

Difference between Compiler and Interpreter:

Compiler takes the entire program and converts it to Machine language in a single go.

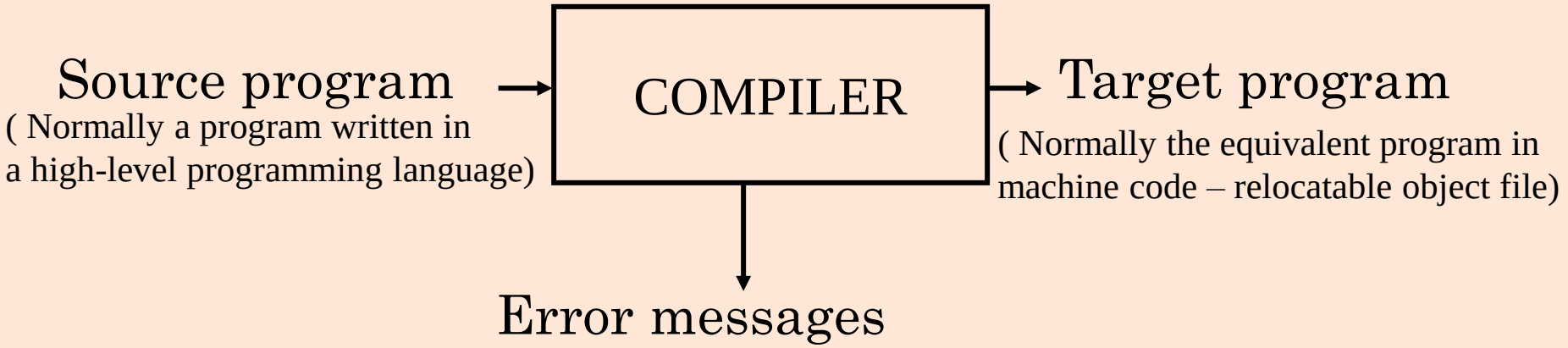
Interpreter does the conversion line by line

LANGUAGE PROCESSING SYSTEM



COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.

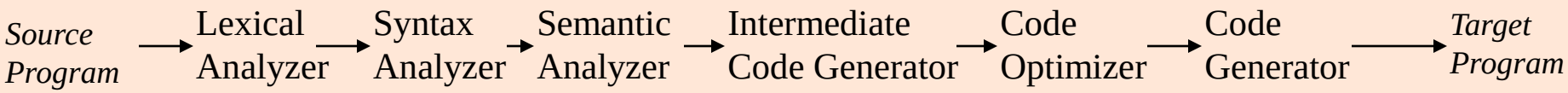


Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

PHASES OF A COMPILER



- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

LEXICAL ANALYZER

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

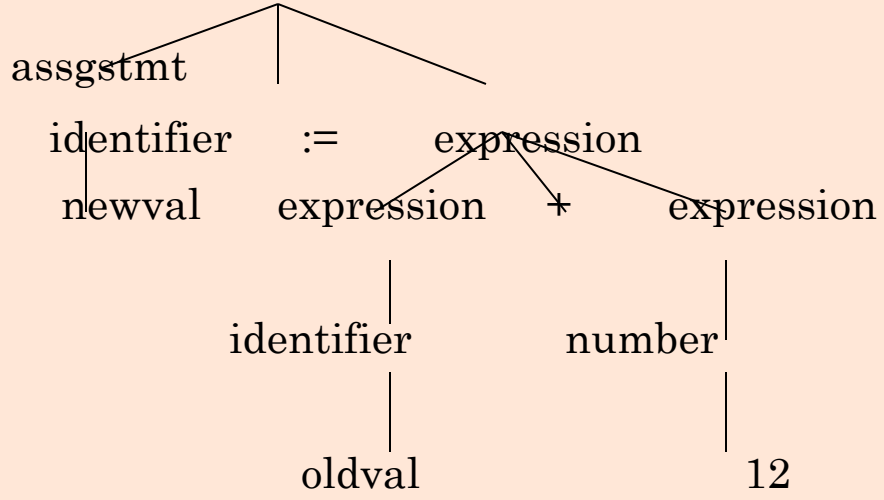
Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

SYNTAX ANALYZER

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

SYNTAX ANALYZER (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

```
assgstmt    ->  identifier  := expression
expression  ->  identifier
expression  ->  number
expression  ->  expression  +  expression
```

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

PARSING TECHNIQUES

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

SEMANTIC ANALYZER

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  ```
  newval := oldval + 12
  ```

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

INTERMEDIATE CODE GENERATION

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
- Ex:

newval := oldval * fact + 1

id1 := id2 * id3 + 1

MULT id2,id3,temp1

ADD temp1,#1,temp2

MOV temp2,,id1

Intermediates Codes (Quadraples)

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT        id2,id3,temp1
ADD    temp1,#1,id1
```

CODE GENERATOR

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

- Ex:

(assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

Other Applications

- In addition to the development of a compiler, the techniques used in COMPILER CONSTRUCTION can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in COMPILER CONSTRUCTION.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in COMPILER CONSTRUCTION can be used in Natural Language Processing (NLP) systems.

Course Outline

- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation