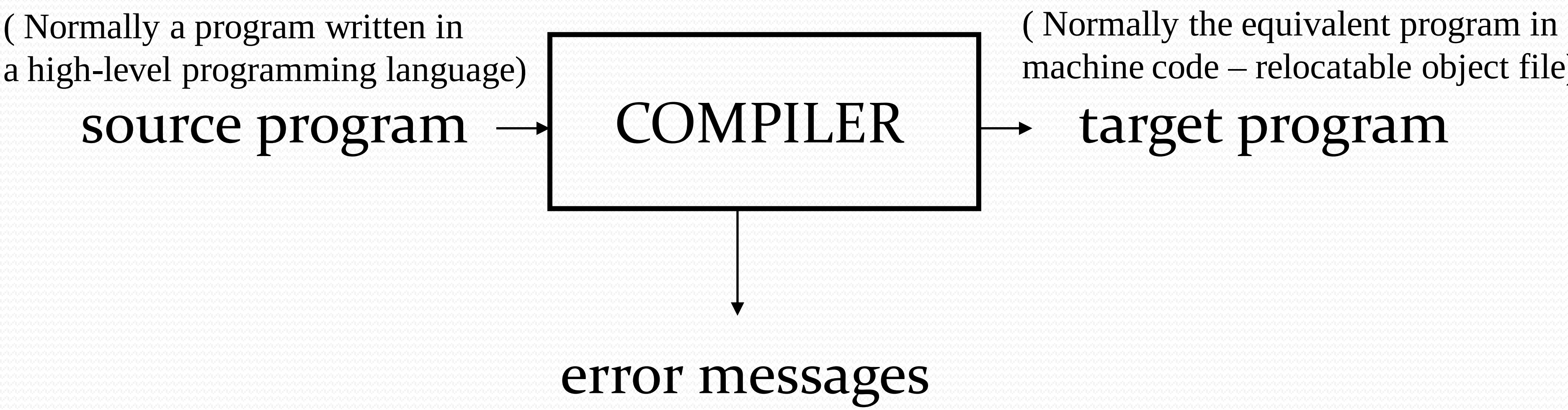
Compiler Design

Aysha Akther
Assistant Professor
CSE Discipline, Khulna University

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.



Compilers

- Compilers translate from a source language (typically a high level language) to a functionally equivalent target language (typically the machine code of a particular machine or a machine-independent virtual machine).
- Compilers for high level programming languages are among the larger and more complex pieces of software
  - Original languages included Fortran and Cobol
    - Often multi-pass compilers (to facilitate memory reuse)
  - Compiler development helped in better programming language design
    - Early development focused on syntactic analysis and optimization
  - Commercially, compilers are developed by very large software groups
    - Current focus is on optimization and smart use of resources for modern RISC (reduced instruction set computer) architectures.

Why Study Compilers?

- General background information for good software engineer
  - Increases understanding of language semantics
  - Seeing the machine code generated for language constructs helps understand performance issues for languages
  - Teaches good language design
  - New devices may need device-specific languages
  - New business fields may need domain-specific languages

Applications of Compiler Technology & Tools

- Processing XML/other to generate documents, code, etc.
- Processing domain-specific and device-specific languages.
- Implementing a server that uses a protocol such as http or imap
- Natural language processing, for example, spam filter, search, document comprehension, summary generation
- Translating from a hardware description language to the schematic of a circuit
- Automatic graph layout (graphviz, for example)
- Extending an existing programming language
- Program analysis and improvement tools

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design 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 design.
    - 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 design can be used in Natural Language Processing (NLP) systems.

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

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

- Each phase transforms the source program from one representation into another representation.

- They communicate with error handlers.

- They communicate with the symbol table.

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

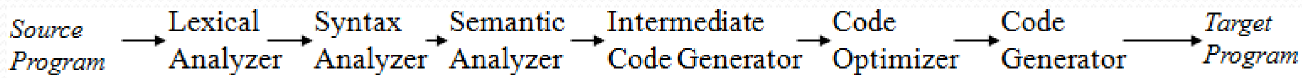
Lexical Analyzer

- **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:

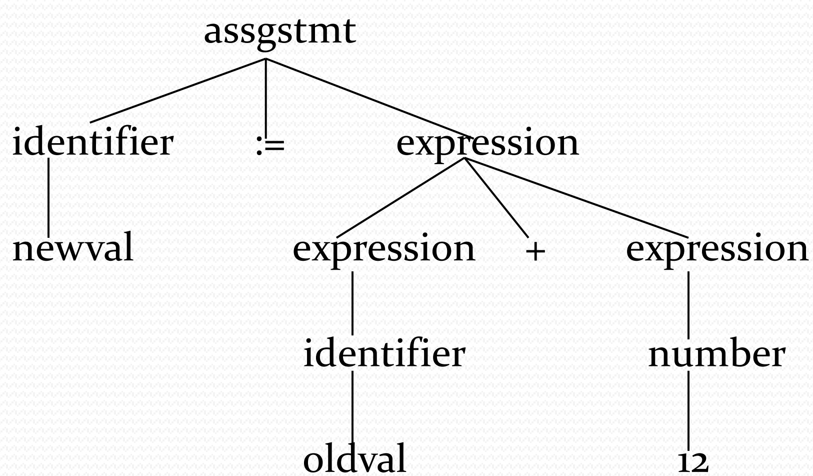
newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- 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.

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

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
```

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

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.

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

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

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

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)*

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

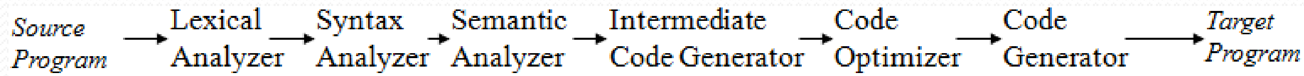
Intermediate Code Generation

- 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 *Intermediates Codes (Quadraples)*
ADD temp1,#1,temp2
MOV temp2,,id1



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
```

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → Target Program

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally 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
```

Compiler / Translator Design Decisions

- Choose a source language
  - Large enough to have many interesting language features
  - Small enough to implement in a reasonable amount of time
  - *Examples for us:* MicroJava, Decaf, MiniJava
- Choose a target language
  - Either a real assembly language for a machine with an assembler
  - Or a virtual machine language with an interpreter
  - *Examples for us:* MicroJava VM (μJVM), MIPS (a popular RISC architecture, for which there is a "SPIM" simulator)
- Choose an approach for implementation:
  - Either use an existing scanner and parser / compiler generator
    - lex/flex, yacc/bison/byacc, Antlr/JavaCC/SableCC/byaccj/Coco/R.
  - Or implement these yourself (limits the language somewhat)

lex Programming Utility

General Information:

- Input is stored in a file with *.l extension
- File consists of three main sections
- lex generates C function stored in lex.yy.c

Using lex:

1) Specify words to be used as tokens (Extension of regular expressions)
2) Run the lex utility on the source file to generate **yylex( )**, a C function
3) Declares global variables char* yytext and int yyleng

yacc Parser Generator

<u>General Information:</u>

- Input is specification of a language
- Output is a compiler for that language
- yacc generates C function stored in y.tab.c
- Public domain version available **bison**

<u>Using yacc:</u>

1) Generates a C function called **yyparse()**
2) **yyparse()** may include calls to yylex()
3) Compile this function to obtain the compiler

References

- Compilers: Principles, Techniques and Tools
  - →Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
- http://www.cs.bilkent.edu.tr/~ilyas/Courses/CS416