CS 346: Compilers

Course Information

- Instructors

  Asif Ekbal

  - Office: STPI, Room no-111
  - Phone: 062122090 (O), 8521274830 (M)
  - Email: asif@iitp.ac.in, asif.ekbal@gmail.com
- Raju Halder (few lectures)
  - Office: STPI
  - Phone: 062122009 (O)
  - Email: halder@iitp.ac.in
- Meeting time - send me an email (preferably!)

Preliminaries Required

- Basic knowledge of programming languages
- Basic knowledge of FSA and CFG
- Basic level of some high level programming languages

Textbook:

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman,

"*Compilers: Principles, Techniques, and Tools*"

Addison-Wesley, 1986.

Grading

- Midterm: 30%
- End Semester : 50%
- Assignments, Quizzes and Attendance : 20%

Course Outline (Proposed)

- Introduction to Compilers

- Lexical Analysis
  - Role
  - Recognition of tokens
  - Fine automata
  - Design of lexical analyzer

- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing and their various approaches
  - Bottom-Up Parsing and their various approaches
  - Powerful parsers: Canonical LR, LALR
  - Parsing with ambiguous grammars

Course Outline (Proposed)

- Syntax-Directed Definition and Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
  - Syntax directed translation schemes

- Intermediate Code Generation
  - Different representations (Syntax tree, three-address codes etc.)
  - Three-address codes
  - Translation of expressions
  - Type checking
  - Backpatching

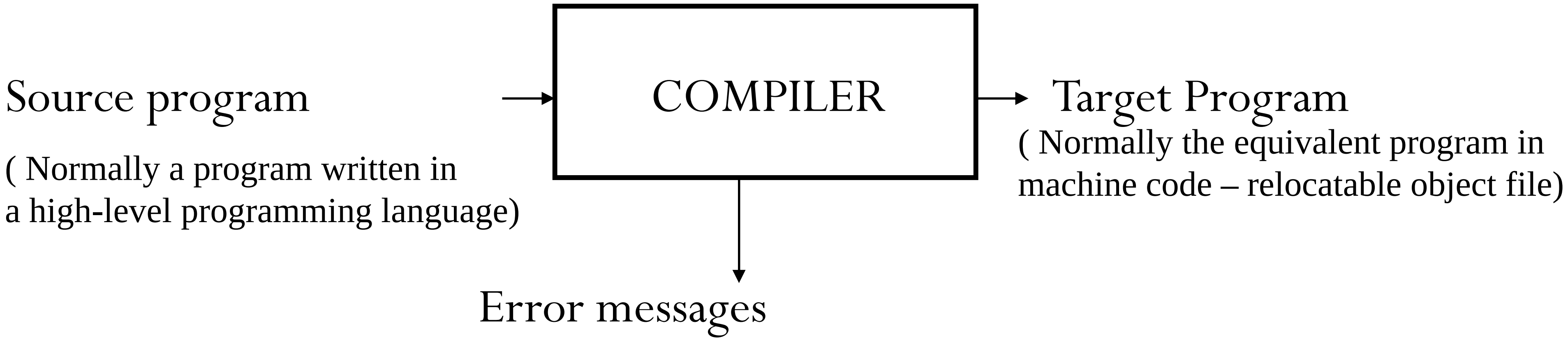
Course Outline (Proposed)

- Run-time Environments
  - Storage organization
  - Stack allocation of spaces

- Code generation
  - Various issues
  - Addressing in target codes
  - Basic blocks and flow graphs
  - Optimization of basic blocks
- Code optimization
  - Various techniques and issues

Compilers

Compiler: a component that takes a program written in a source language and translates it into an equivalent program in a target language



Source program

(Normally a program written in
a high-level programming language)

COMPILER

Target Program

(Normally the equivalent program in
machine code – relocatable object file)

Error messages

Machine Language

- The only language that is "understood" by a computer
- Varies from machine to machine
- The only choice in the 1940s

```
0001 01 00 00001111
0011 01 10 00000010
0010 01 00 00010011
```

b = a + 2

Assembly Languages

- Also known as symbolic languages
- First developed in the 1950s
- Easier to read and write
- Assembler converts to machine code
- Still different for each type of machine

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

b = a + 2

High-Level Languages

- Developed in 1960s and later
- Much easier to read and write
- Portable to many different computers
- Languages include C, Pascal, C++, Java, Perl, etc.
- Still must be converted to machine code!

Compilers (History)

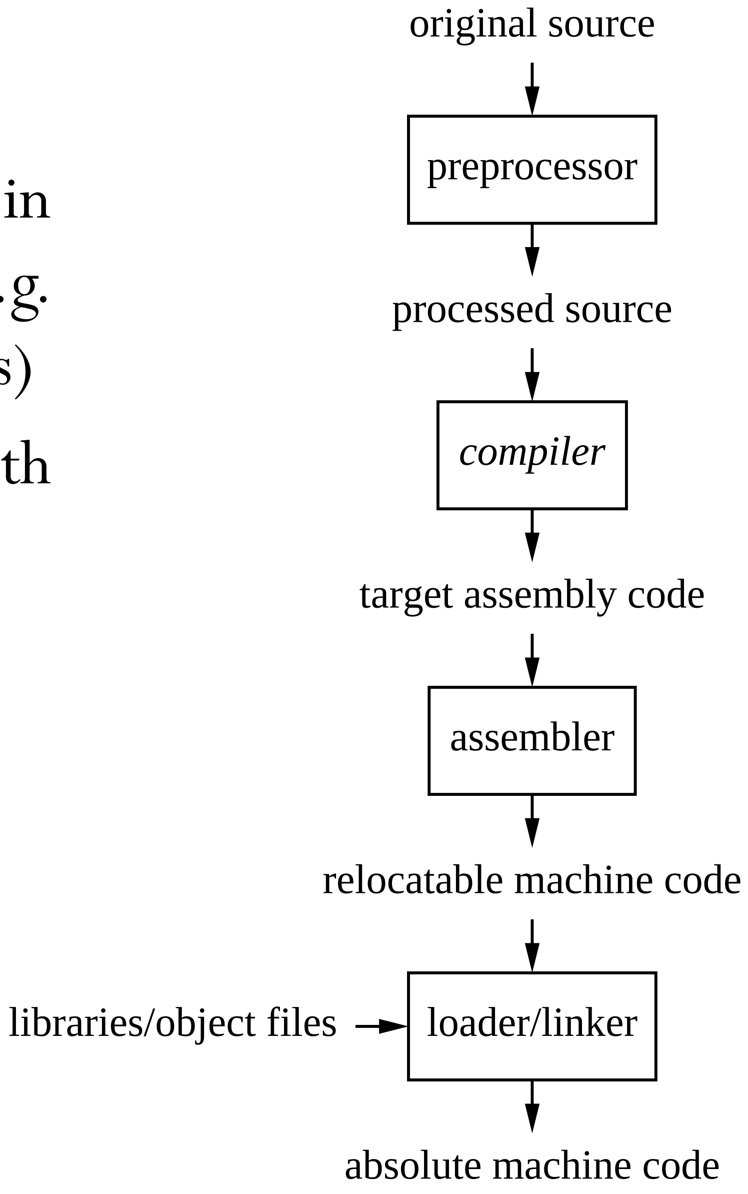
- Early compilers
  - 1950s: by Grace Hopper
  - Late 1950s: Fortran

- Broad applications
  - Typesetting: TeX, LaTeX
  - Portable document representation: PostScript
  - Symbolic and numeric problem solving: Mathematica
  - VLSI: Verilog, VHDL

Major Parts of Compilers

- TWO major parts
  - **Analysis**
  - **Synthesis**
- Analysis phase
  - Intermediate representation is created from the given source program
  - **Components:** Lexical Analyzer, Syntax Analyzer and Semantic Analyzer
- Synthesis phase
  - Equivalent target program is created from the intermediate representation obtained from the analysis phase
  - **Components**: Intermediate Code Generator, Code Generator, and Code Optimizer

- Some tools often work in conjunction with compilers (e.g. assemblers, linkers, preprocessors)
- Often, they are coupled with compiler



Compilers (Conjunct Tools)

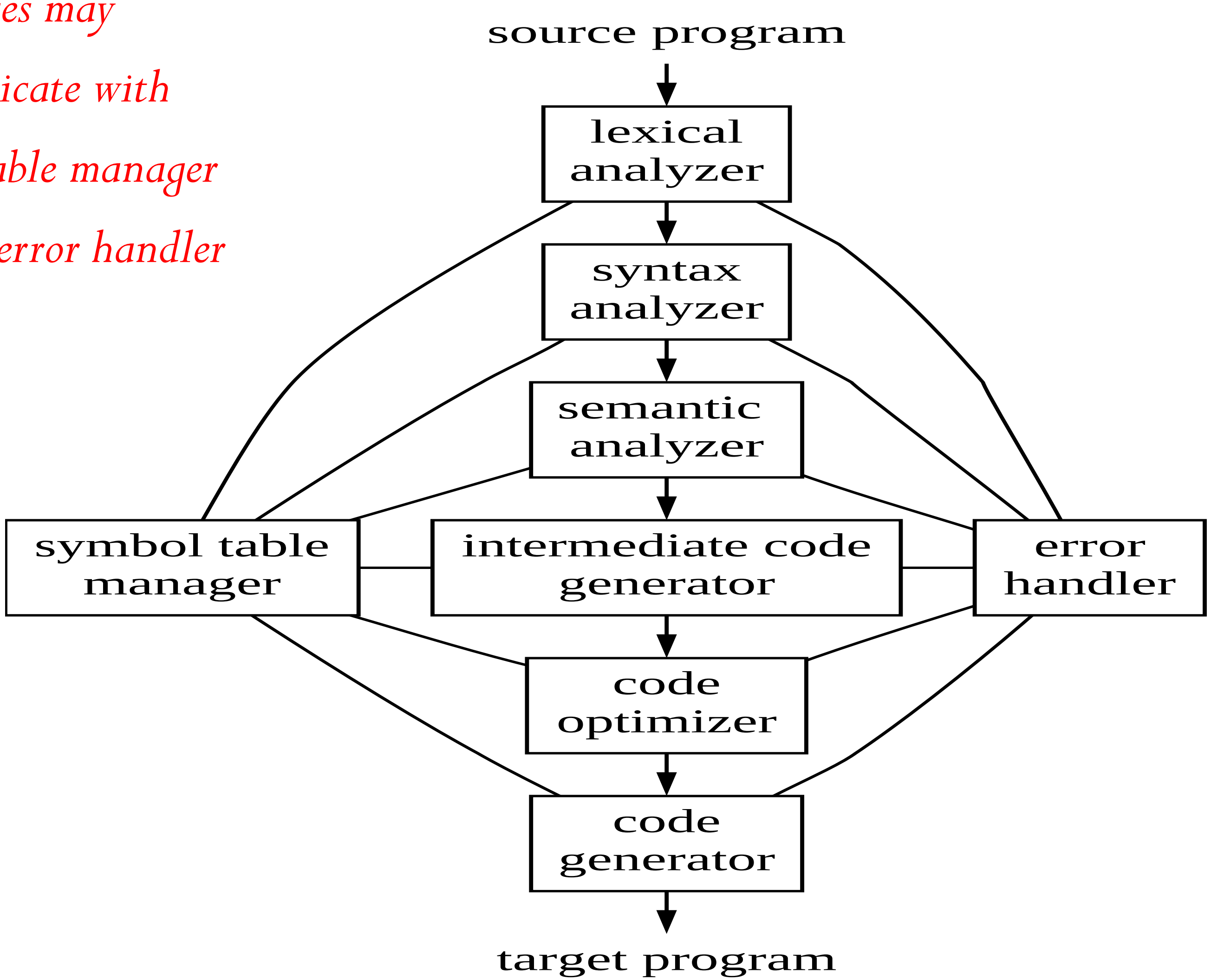
- Preprocessor
  - Source program may be divided into several pieces
  - Task of collecting the source programs known as preprocessor
- Assembler
  - Compiler may produce assembly language as output
  - Assembly is easier to produce as output and debug
  - Assembly language processed by assembler and its output is relocatable machine code
- Large programs compile in components
- Relocatable machine codes may need to be linked together with other relocatable object files

Compilers (Conjunct Tools)

- Linker
  - Resolves external memory addresses, where the code in one file may refer to the location in other files
- Loader
  - Puts together all of the executable object files into memory for execution

Phases of Compiler

All phases may communicate with symbol table manager and/or error handler



Lexical Analyzer

- **Lexical Analyzer**
  - reads the source program character by character
  - returns the *tokens* of the source program
- A *token*
  - a pattern of characters having same meaning in the source program

    E.g. *identifiers*, *operators*, *keywords*, *numbers*, *delimeters* etc.
- *Lexeme*: Character sequence forming token

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

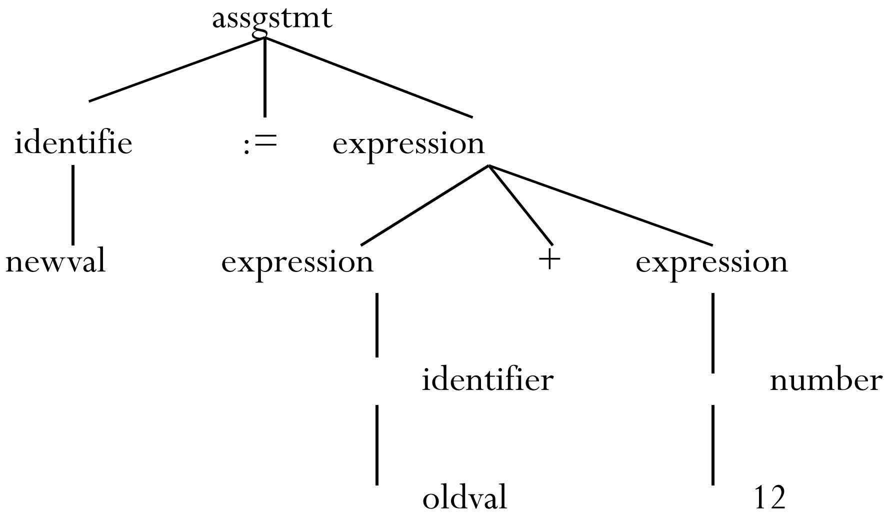
newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

Lexical Analyzer

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

Syntax Analyzer

- **Syntax Analyzer**
  - creates the syntactic structure (generally a *parse tree*) of the given program
- Syntax analyzer- **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)

- Syntax of a language is specified by a **context free grammar** (CFG)
- Rules in a CFG-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

- Both do similar things but at the *different levels*

- Lexical analyzer deals with simple non-recursive constructs of the language

- Syntax analyzer deals with recursive constructs of the language

- Lexical analyzer simplifies the job of the syntax analyzer
  - Lexical analyzer recognizes the smallest meaningful units (tokens) in a source program
  - Syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language

Parsing Techniques

- Different parsing techniques exist depending upon the way of how the parse tree is created
- Two main categories
  - *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
  - E.g. 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
  - 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

- Checks the source program for *semantic errors*
- Collects the *type information* for the code generation
- *Type-checking* is an important part of semantic analyzer
- In general, semantic information cannot be represented by a context-free language used in syntax analyzers
- CFG of syntax analysis 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)*

Symbol Tables

- A mechanism (or, data structure) that allows information to be associated with *identifiers* and shared among *compiler phases*
  - Identifier declaration
  - Identifier use
  - Type checking

- Attributes for variables
  - storage, type, scope, etc.
- Attributes for procedures
  - name, parameters, etc.

When a lexical analyzer sees an identifier for the first time, it adds it to the symbol table

Intermediate Code Generation

- Compiler may produce an explicit intermediate codes to represent source program
- Intermediate codes are generally machine (*architecture) independent*
- Level of intermediate codes is close to the level of machine codes
- Should have at least two qualities:
  - Easy to produce
  - Easy to translate into target program
- Three-address code is common
  - At most three operands per instruction
  - At most one operator (plus assignment)

```
temp1 := 2
temp2 := a + temp1
b = temp2
```

b = a + 2

Intermediate Code Generation

Ex:

```
id1  :=  id2 * id3 + 1

MULT  id2,id3,temp1 Intermediates Codes (Quadraples)
ADD   temp1,#1,temp2
MOV   temp2,,id1
```

Code Optimizer (for Intermediate Code Generator)

- Essential in terms of space and time
- Code optimization operates on intermediate code
  - General
  - Not really optimal

Ex:

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

Code Generator

- Produces the target language in a specific architecture
- Target program is normally a re-locatable object file
- Object file contains the machine codes

Ex: Assume that we have an architecture with instructions, at least one of its operands is a machine register

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

Front End vs. Back End

- Front end deals with source language
  - Includes analysis, creation of symbol table, generation of intermediate code, some optimization
  - Independent of target machine

- Back end deals with target code
  - Includes some optimization, code generation
  - Depends on target machine, not on source language

Passes

- Many compilers make multiple passes
  - Several phases often grouped into single pass (e.g. all of analysis and intermediate code generation)
  - Desirable to have relatively few passes

- Backpatching
  - Leaves blanks for unknown values to be filled in later
  - Allows merging of phases

Compiler Writing Tools

- Compiler generators (compiler compilers)
  - Scanner generator
  - Parser generator
  - Symbol table manager
  - Attribute grammar evaluator
  - Code-generation tools
- Much of the effort in crafting a compiler lies in writing and debugging the semantic phases
  - Usually hand-coded

Useful Tools

- Writing compilers used to be extremely complicated

- Now tools make the task much easier
  - Lexical analyzers (e.g. Lex)
  - Compiler-compilers (e.g.Yacc)

Relevance to Other Subjects

- Natural Language Processing
  - Summarization, Machine Translation, Question-Answering, Search
  - Also often separates analysis and generation
  - Also deals with syntax and semantics
- Theoretical (Regular Expressions, Finite Automata, Context-free Grammars, etc.)