Compiler Construction

(CSC 631)

By
Bishnu Gautam,
M. Sc. Computer Science & Information Technology
CDCSIT, TU

Why study this Course?

……..…You may never write a commercial compiler,

But we study compiler construction.

- It gives an experience with large-scale applications development.
- Compiler writing is one of the shining triumphs of CS theory. It demonstrates the value of theory over the impulse to just pick up a solution.
- It is a basic element of programming language research.
- Many applications have similar properties to one or more phases of a compiler, and compiler expertise and tools can help an application programmer working on other projects besides compilers. (Next Slide)

Why study this Course?

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.

Prerequisites

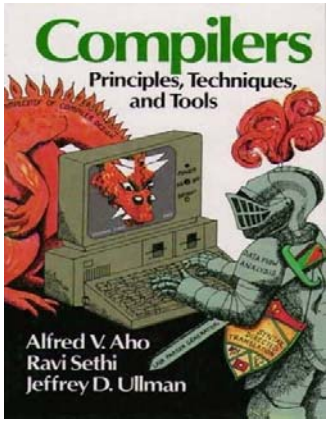
- Introduction to Automata and Formal Languages
- Introduction to Analysis of Algorithms and Data Structures
- Working knowledge of C/C++, gdb

Introduction to the Principles of Programming Languages, Operating System & Computer Architecture is plus

Resources

Text Book:

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986



The Red Dragon Book

References:

Research and technical papers.

Lab Tools & Resources

- Linux Machine with gcc/g++, flex and bison installed
- Manual pages for C/C++, flex and bison

Course Tentative Schedule

1. Introduction to Compiling: Phases of Compilers
2. Lexical Analysis: Role of Lexical analyzer, Token specification, Language for Lexical analysis
3. Syntax Analysis: Context Free Grammars, Top-Down Parsing, LL Parsing, Bottom-Up Parsing, LR Parsing

First Mid-Term Exam

4. Syntax-Directed Translation: Attribute Definitions, Evaluation of Attribute Definitions
5. Semantic Analysis, Type Checking
6. Run-Time Environments: Memory allocation, Parameters Passing

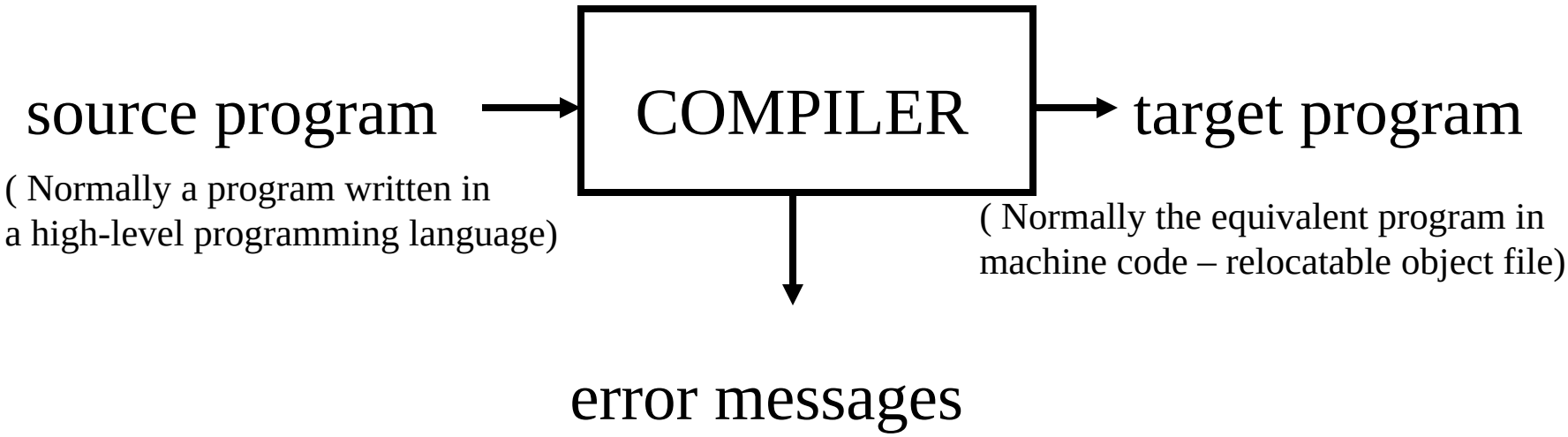
Second Mid-Term Exam

7. Code Generation & Optimization : Intermediate and final code generation concepts, compile time code optimization strategies

What is Compiler?

Reading: Chapter 1

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



Cousins of Compiler

There are several major kinds of compilers:

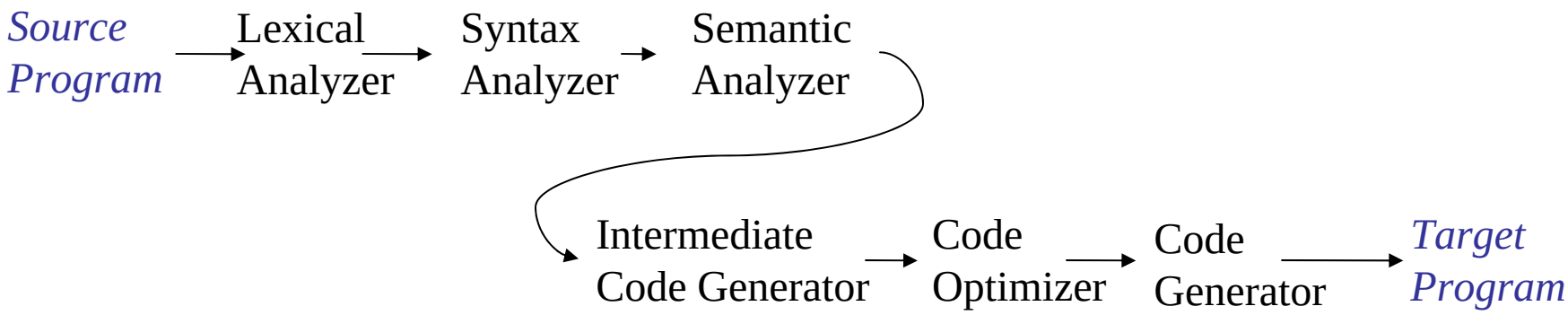
- Native Code Compiler
  - Translates source code into hardware (assembly or machine code) instructions. Example: gcc.
- Virtual Machine Compiler
  - Translates source code into an abstract machine code, for execution by a virtual machine interpreter. Example: javac.
- JIT Compiler
  - Translates virtual machine code to native code. Operates within a virtual machine. Example: Sun's HotSpot java machine.
- Preprocessor
  - Translates source code into simpler or slightly lower level source code, for compilation by another compiler. Examples: cpp, m4.
- Pure interpreter
  - Executes source code on the fly, without generating machine code. Example: Lisp.

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**.

Analysis of the Source Program

Lexical Analysis:

The stream of characters forming the source the program are scanned linearly to produce the stream of logical element called **tokens**

Syntax Analysis:

The streams of tokens are grouped into hierarchical collection called **syntax grouping**

Semantic Analysis:

Checks are performed to ensure that parts of the program fit together meaningfully.

Lexical Analyzer

Lexical Analyzer reads the source program 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, delimiters 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).

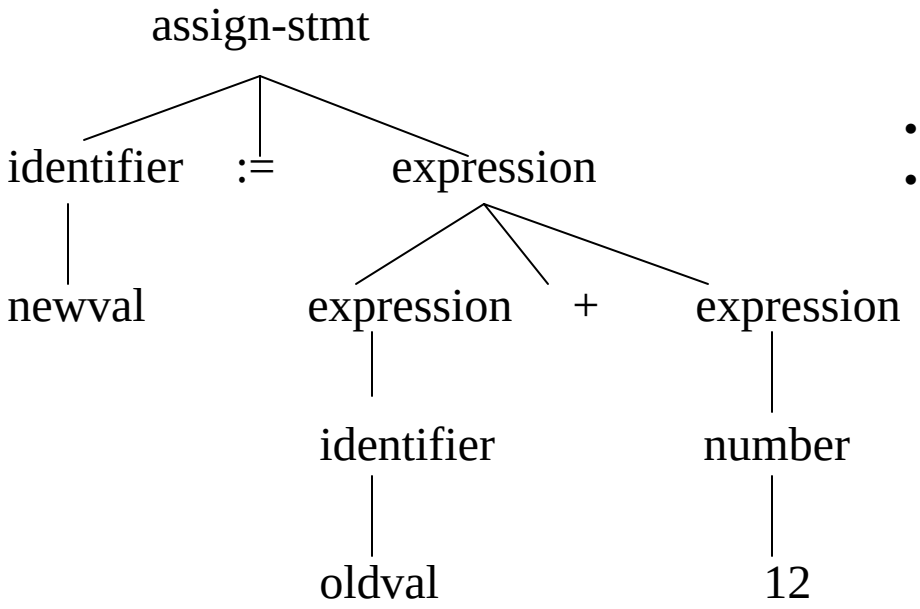
A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

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

assign-stmt -> identifier := expression
expression -> identifier
expression -> number
expression -> expression + expression

Syntax Analyzer vs. 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)*

Synthesis

Synthesis concerns the issues involving generating code in target language. It usually consists of the following phases:

- Intermediate code generation
- Code optimization
- Final code generation

Intermediate Code Generation

An intermediate language is often used by many compiler for analyzing and optimizing the source program. The intermediate language should have two important properties:

- It should be simple and easy to produce.
- It should be easy to translate to the target program

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
           ↓
temp1 = int2flot(1)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1     = temp3
```

Intermediates Codes, Three address code

Code Optimization

The compile time code optimization involves static analysis of the intermediate code to remove extraneous operation in of time and space. i.e

Detection of redundant function calls
Detection of loop invariants
Common subexpression compilation
Dead code detection and elimination

Ex:

```
temp1 = int2flot(1)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1   = temp3
```

```
temp1 = id2 * id3
id1 = temp1 + 1
```

Code Generation

This involves the translation of optimized intermediate code into the target language

The target code is normally is a relocatable object file containing the machine or assembly 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
```

The Symbol Table

A symbol table store information about keyword and tokens found during the lexical analysis. The symbol table is consulted in almost all phase of compiler.

Examples:

Insert(“dist”,id) // insert a symbol table entry associating the string “dist” with token type “id”

Lookup(“dist”) // an occurrence of string “dist” can be looked up in the symbol table. If found, the reference to the “id” is returned else lookup return 0.

Error Handling

- Error may be encountered in many phase of compiler
- Objective of error handling is to go as far as possible in compilation whenever an error is encountered

Examples:

- Handling missing symbols during the lexical analysis by inserting symbol
- Automatic type conversion during the semantic analysis

Home Works

1. Prepare a note on the history of compiler writing including the following scenarios
   - Research and development of programming language
   - Development of tools and technique of compiler design
   - Specify the parallel discovery of the same technique
   - Specify the impact of other fields of computer science with the development compiler design