Compiler Design – CS4L001

Introduction

Srinivas Pinisetty

Outline

- About the course
- What is a compiler
- Why study compiler design
- Course content and learning outcomes

Course Lecturers

- **Srinivas Pinisetty**– **`spinisetty@iitbbs.ac.in`**
- **Office**: A-205, SES

- Listen, understand, ask questions, ...

- Attend classes, be on-time

- Give feedback, clarify doubts

Course Text Book

- ***Compilers, Principles, Techniques and Tools*** (by Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman)

Evaluation

- **Internal** (20-30%): Quizzes, assignments, class participation,
- **Mid-semester** Exam: 25-30%
- **End-semester** Exam: 45-50%

About the course

- A detailed look at the **internals of a compiler**
- A compiler is an excellent example of theory translated into practice
- Solving **theoretical problems** and doing **programming assignments** are both essential

What is a compiler?

Move to Higher-Level Programming Languages

- Machine Languages (1st generation)
- Assembly Languages (2nd generation) – early 1950s
- High-Level Languages (3rd generation) – later 1950s
- 4th generation higher level languages (SQL, Postscript)
- 5th generation languages (logic based, eg, Prolog)

- Other classifications:
  - Imperative (how); declarative (what)
  - Object-oriented languages
  - Scripting languages

Goals

- Any program written in a programming language must be ***translated*** before it can be executed.

- This translation is typically accomplished by a software system called **compiler**.

- This course aims to introduce you to the ***principles and techniques used to perform this translation*** and the issues that arise in the construction of a compiler.

Why should we study compiler design?

Compilers are everywhere !!

- Many applications for compiler technology
  - Machine code generation for high level languages
  - Parsers for HTML in web browser
  - Interpreters for javascript/flash
  - Software testing
  - Program optimization
  - Malicious code detection
- Hardware synthesis: VHDL to RTL translation
- Compiled simulation
  - Used to simulate designs written in VHDL

Complexity of compiler technology

- A compiler is possibly the most complex system software and writing it is a substantial exercise in software engineering

- The complexity arises from the fact that it is required to map a programmer's requirements (in a HLL program) to **architectural** details

- It uses algorithms and techniques from a very large number of areas in computer science

- Translates intricate theory into practice - enables tool building

Compiler Algorithms

- **Makes practical application of**
  - Finite automata - lexical analysis
  - Pushdown automata - parsing
  - Greedy algorithms - register allocation
  - Heuristic search - list scheduling
  - Graph algorithms - dead code elimination, register allocation
  - Dynamic programming - instruction selection
  - Optimization techniques - instruction scheduling
  - Fixed point algorithms - data-flow analysis
  - Complex data structures - symbol tables, parse trees, data dependence graphs
  - Computer architecture - machine code generation

Other uses of parts of compiler technology

- **Scanning and parsing techniques**
  - Assembler implementation
  - Online text searching (GREP, AWK) and word processing
  - Command language interpreters
  - Scripting language interpretation (Unix shell, Perl, Python)
  - XML parsing and document tree construction
  - Database query interpreters

- **Program analysis techniques**
  - Parallelizing loops
  - Software testing (data-flow analysis approach)
  - WCET estimation
  - …………

Expectations?

- What will we learn in the course?

Learning Outcomes

- explain the principles governing **all phases** of the compilation process.
- explain the role of each of the basic components of a standard compiler.
- show awareness of the problems of and methods and techniques applied to each phase of the compilation process.
- apply standard techniques to solve basic problems that arise in compiler construction.

By the end of the course …

- Knowledge to

  design, develop, understand, modify/enhance, and maintain compilers for

  (even complex!) programming languages

Course contents (1)

- Introduction (Overview, and phases of compilation)
- Lexical Analysis (scanning)
    - Finite automata (NFA and DFA),
    - Regular expressions, regular languages
    - Designing a lexical analyser as a DFA
    - Lexical analyser generator
- Syntax Analysis (parsing)
    - Role of a parser,
    - context free grammars and context free languages,
    - parse trees and derivations
    - Top-down parsing
    - Bottom-up parsing
    - Error reporting and recovery
    - Parser generator

Course contents- (2)

- Semantic Analysis
- Syntax Directed Translation
- Symbol Table
- Intermediate Representations
- Storage management/ runtime environment
- Code Generation
- Code Optimisation

Bit of History

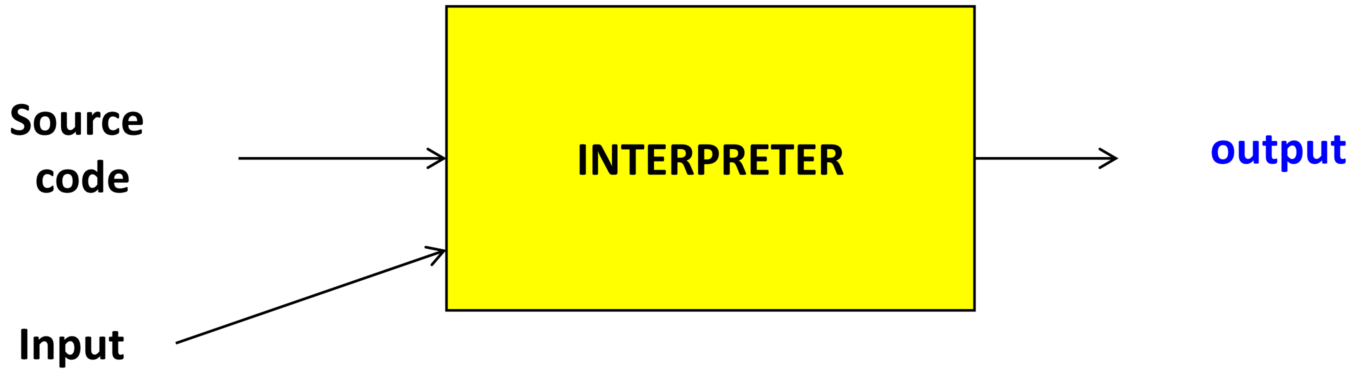
- How are programming languages implemented?

- Two major strategies:
  - Interpreters (old and much less studied)
  - Compilers (very well understood with mathematical foundations)

Interpreter

- A program that reads a source program and produces the results of executing this source.

- Directly execute the operations specified in the source program on inputs supplied by the user.

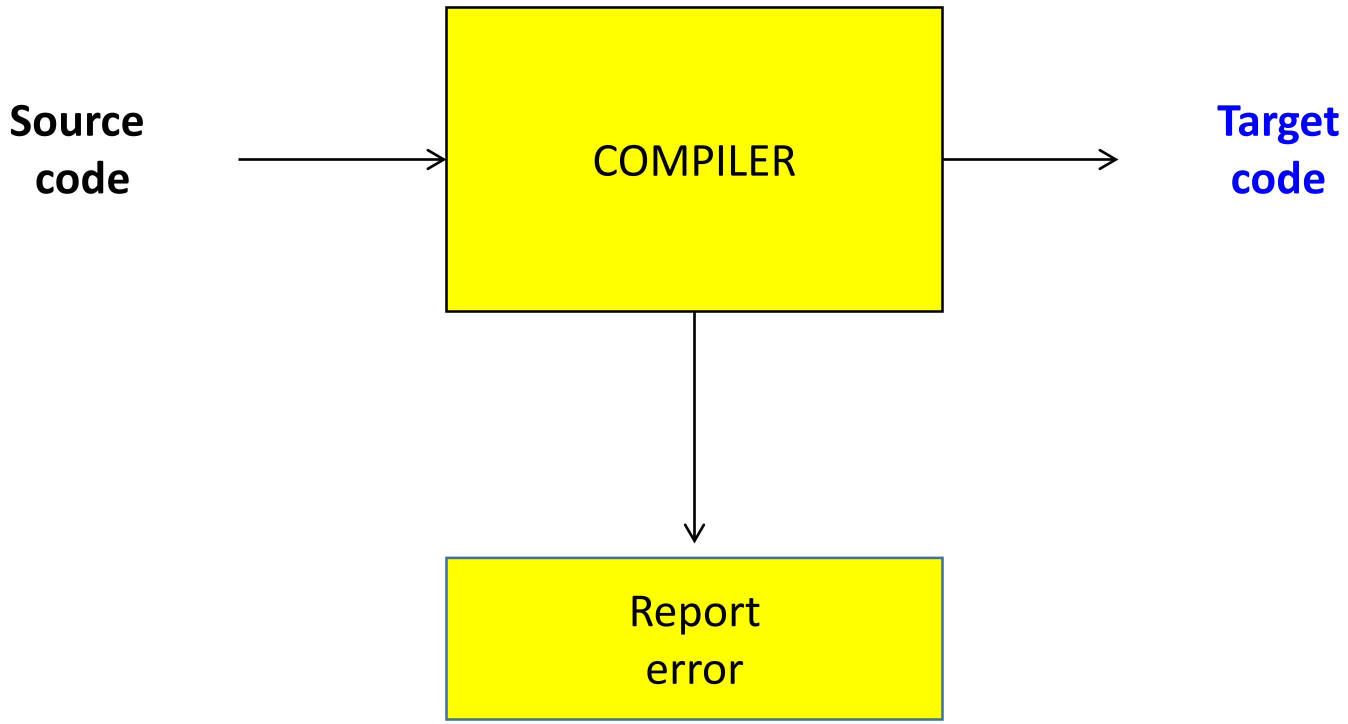


Compiler

- **Compiler**: Program that can read a program in one language (**Source**) and translate it into an ***equivalent*** program in another language (**Target**)

- An important role of the compiler is to report any errors in the source program that it detects during the translation process

Compiler



Target code : Target code is mostly an executable *machine-language program*.

It can be called by the user to process inputs and produce outputs.

Bit of History

- IBM developed 704 in 1954. All programming was done in assembly language. Cost of software development far exceeded cost of hardware. Low productivity.

- **Speedcoding interpreter**: programs ran about 10 times slower than hand written assembly code

- **John Backus** (in 1954): Proposed a program that translated high level expressions into native machine code. Most people thought it was impossible

- Fortran I project (1954-1957): The first compiler was released

Fortran I

- The first compiler had a huge impact on the programming languages and computer science. **The whole new field of compiler design was started**

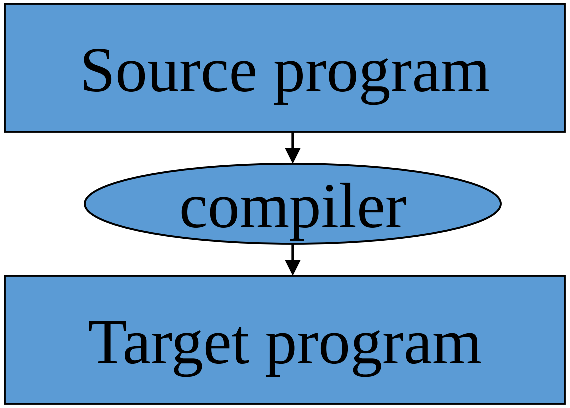
- More than half the programmers were using Fortran by 1958

- **The development time was cut down to half**

- **Led to enormous amount of theoretical work** (lexical analysis, parsing, optimization, structured programming, code generation, error recovery etc.)

- Modern compilers preserve the basic structure of the Fortran I compiler !!!

Examples



- C is typically compiled
- Python is typically interpreted
- Java is compiled to bytecodes, which are then interpreted

Qualities of a good compiler?

Principles of Compilation

The compiler must:

- *preserve the meaning of the program being compiled*.
- *"improve" the source code in some way*.

Other issues (depending on the setting):

- Speed (of compiled code)
- Space (size of compiled code)
- Feedback (information provided to the user)
- Debugging
- Compilation time efficiency (fast or slow compiler?)

Summary

- A compiler is a program that converts some input text in a source language to output in a target language.

- Compiler construction poses some of the most challenging problems in computer science.

- *Next lecture*: structure of a typical compiler.