Compiler Techniques

CS 308

Course Roadmap

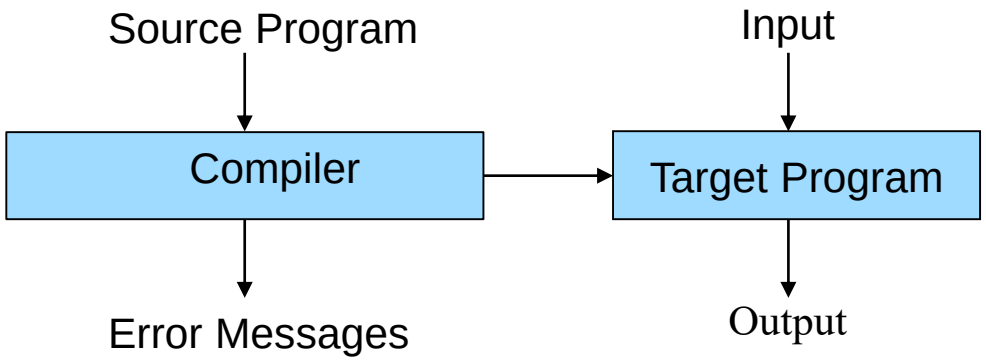
Chapter 1	Introduction to Compiler
Chapter 2	Lexical Analyzer and Grammars
Chapter 3	Syntax Analysis
Chapter 4	Semantic Analysis
Chapter 5	Intermediate Code Generation
Chapter 6	Code Optimization

Course Roadmap

Chapter 1	Introduction to Compiler
Chapter 2	Lexical Analyzer and Grammars
Chapter 3	Syntax Analysis
Chapter 4	Semantic Analysis
Chapter 5	Intermediate Code Generation
Chapter 6	Code Optimization

What is Compiler

- Most programming is done using a high-level programming language
  - But this high-level language can be very different from the machine language that a computer can execute
- A compiler translates a program written in a high-level programming language into the low-level machine language
  - High-level programming language is suitable for human programmers and low-level machine language is required by computers



- Why we don't write program in machine language?

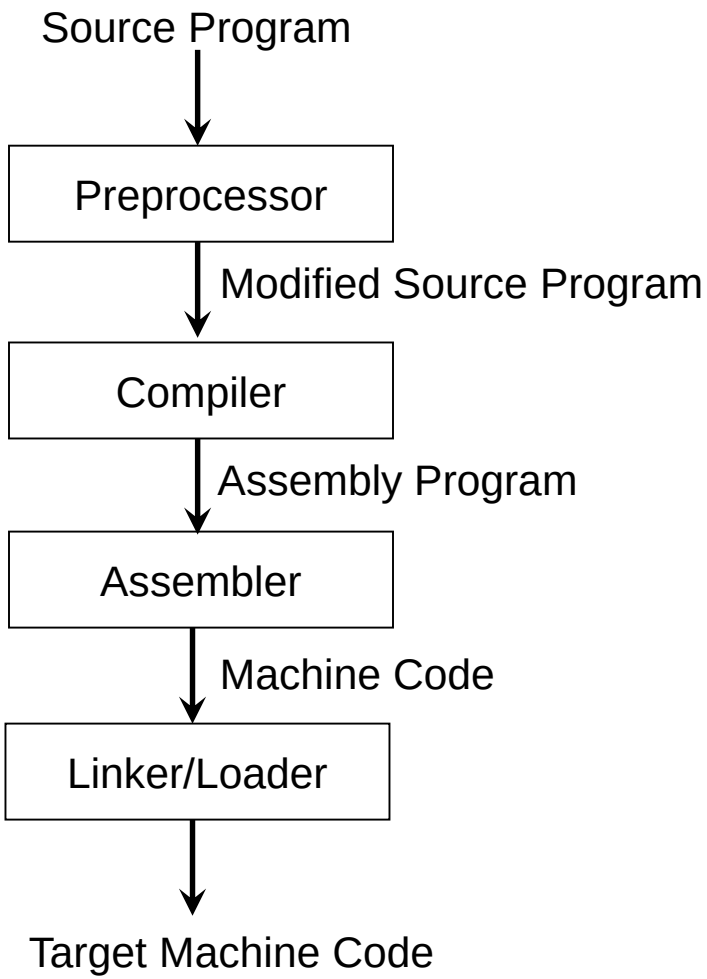
Why Compiler

- **Using a high-level language for programming has a large impact on how fast programs can be developed**
  - Compared to machine language, the notation used by programming languages is closer to the way humans think about problems
    - Programmars are comfortable to write a program in a language which is similar to familiar language such as English
  - The compiler can spot some obvious programming mistakes
  - Programs written in a high-level language tend to be shorter than equivalent programs written in machine language
  - The same program can be compiled to many different machine languages and, hence, can be brought to run on many different machines

Language Processing System

- Computer system is made of hardware and software
  - Hardware understands a language, which is difficult for humans to understand
- We write programs in high-level language, which is easier for us to understand and remember
- These Programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine
  - This is known as Language Processing System

Language Processing System



Language Processing System

- User writes a program in C language (high-level language)
- Pre-processor deals with macro-processing, file inclusion, etc.
  - Coverts high-level language into pure high-level language
- Compiler compiles the program and translates it to assembly program (low-level language)
- An assembler then translates the assembly program into machine code (object)
  - Assembler is machine dependent
  - We can see assembler as a manual using that we can operate in one machine but not in other machine
  - In earlier days, people use to write program in assembly language

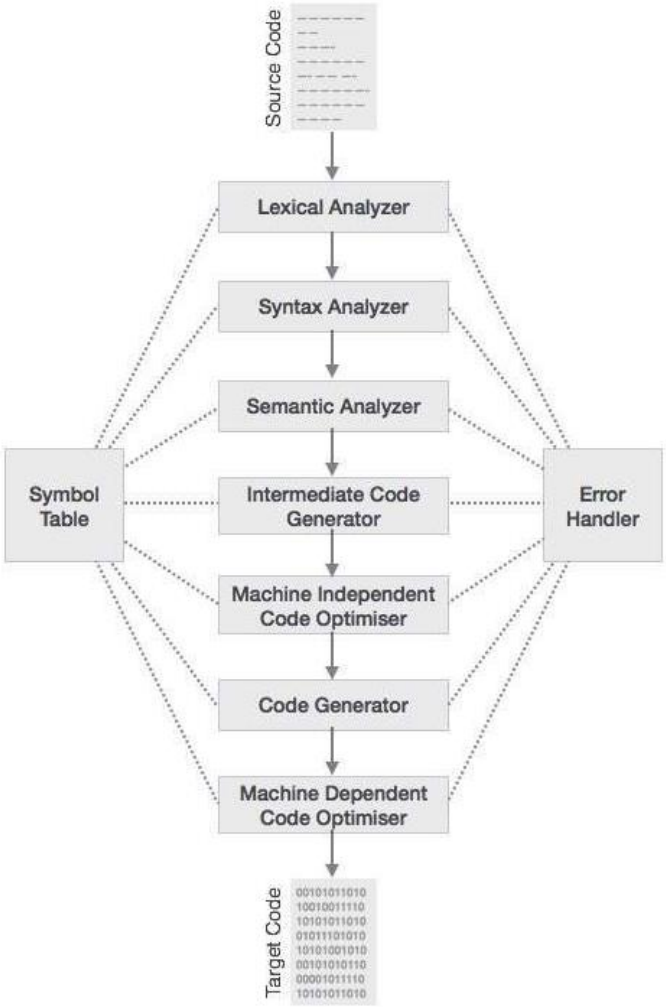
Language Processing System

- A linker tool is used to link all the parts of the program together for execution (executable machine code)
- A loader loads all of them into memory and then the program is executed
- In modern compiler such as GCC, all the tools are kept together as one tool
- We will focus on compiler

Compiler Architecture

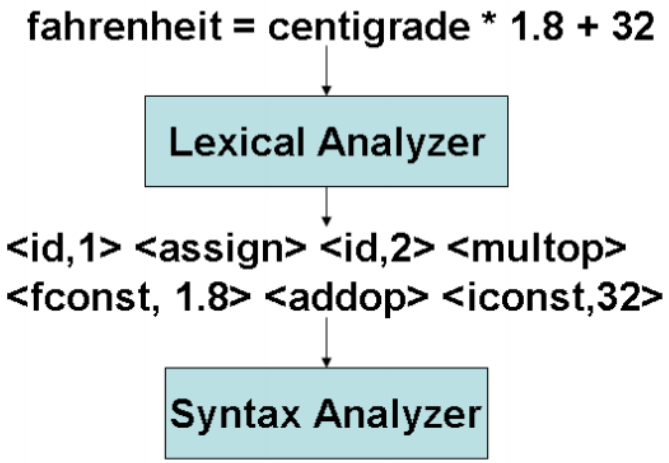
- A compiler can broadly be divided into two phases
  - Analysis Phase
    - Known as the front-end of the compiler
      - Reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors
      - Generates an intermediate representation of the source program, which is fed to the Synthesis phase as input
  - Synthesis Phase
    - Known as the back-end of the compiler
      - Generates the target program with the help of intermediate source code

Phases of Compiler



Lexical Analysis

- The first phase of scanner works as a text scanner
- This phase scans the source code as a stream of characters and converts it into meaningful lexemes
  - Lexical analyzer represents these lexemes in the form of tokens
  - <token-name, attribute-value>



Lexical Analysis

- Lexical Analyzer (LA) can be generated automatically from regular expression specifications
- LA is a deterministic finite state automaton
- LA based on finite automata are more efficient to implement

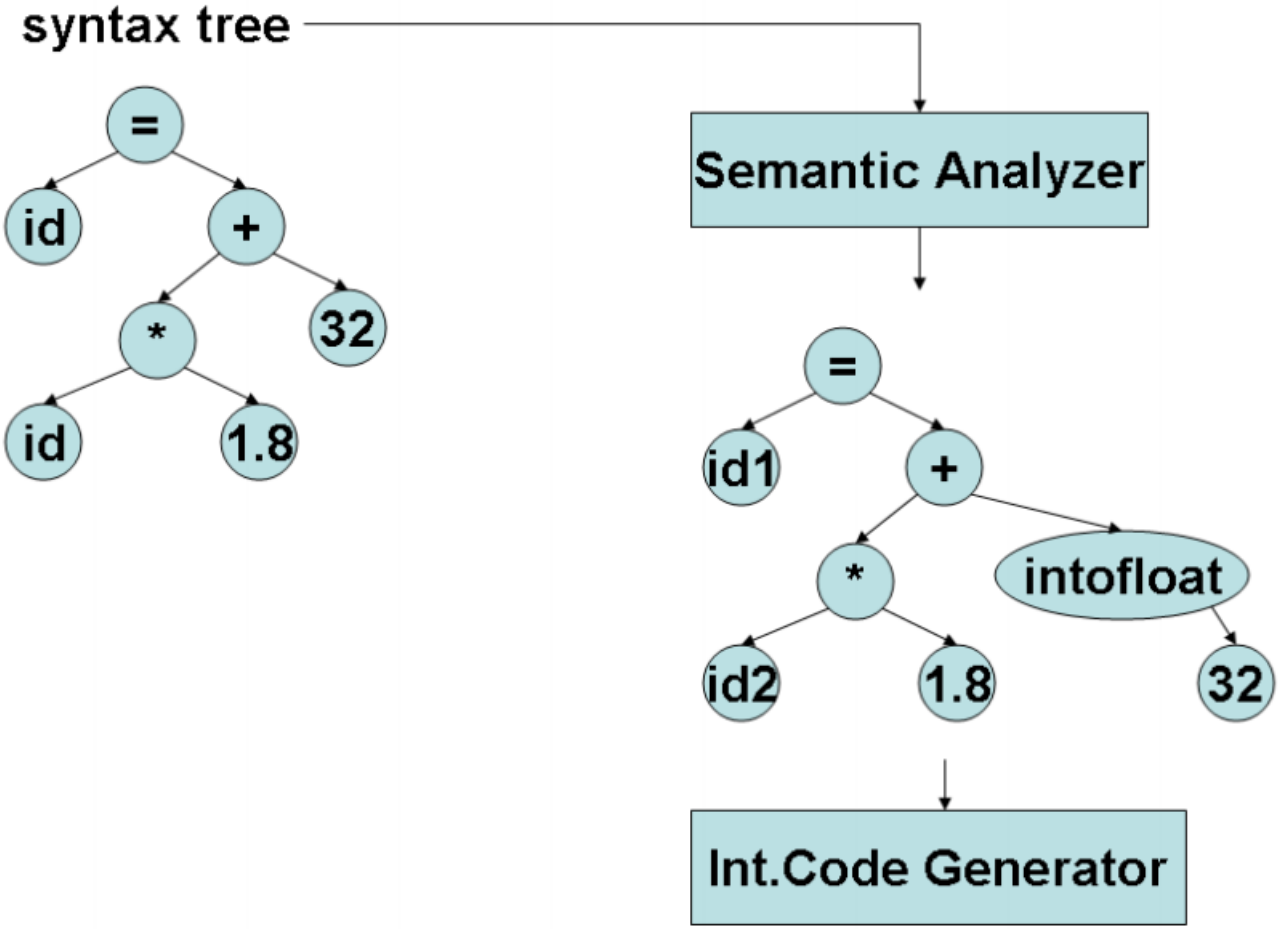
Syntax Analysis

- Syntax Analysis takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree)
- Token arrangements are checked against the source code grammar
  - For example: parser checks if the expression made by the tokens is syntactically correct
- Syntax Analyzers (or Parsers) are deterministic push-down automata
  - LL(1), and LALR(1) are the most popular parsers

Semantic Analysis

- Semantic Analyzer checks if parse tree is meaningful or not
- Semantic analysis checks whether the parse tree constructed follows the rules of language
  - For example, assignment of values is between compatible data types, adding string to an integer, etc.
- Type checking of various programming language constructs is one of the most important tasks

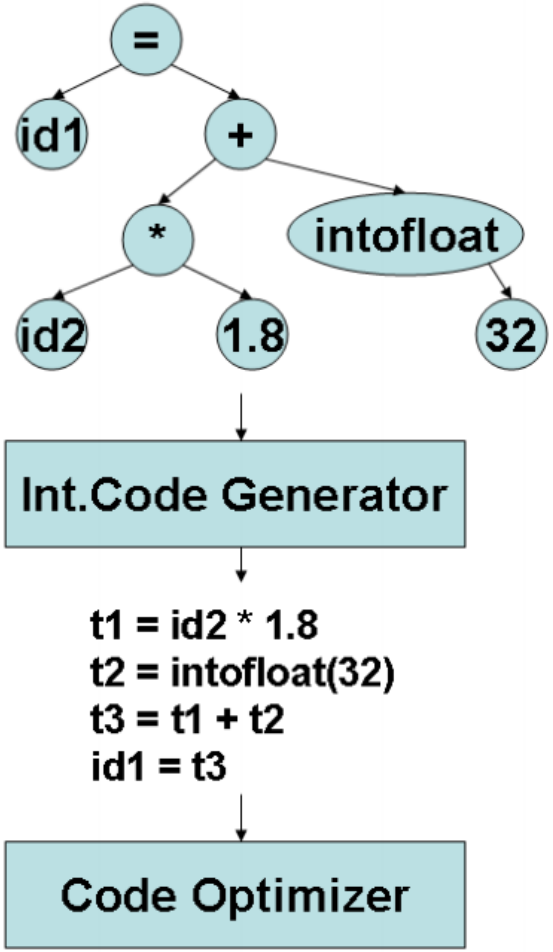
Semantic Analysis



Intermediate Code Generation

- Compiler generates an intermediate code of the source code for the target machine
  - For Example: Three Address Code
- It is in between the high-level language and the machine language
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)

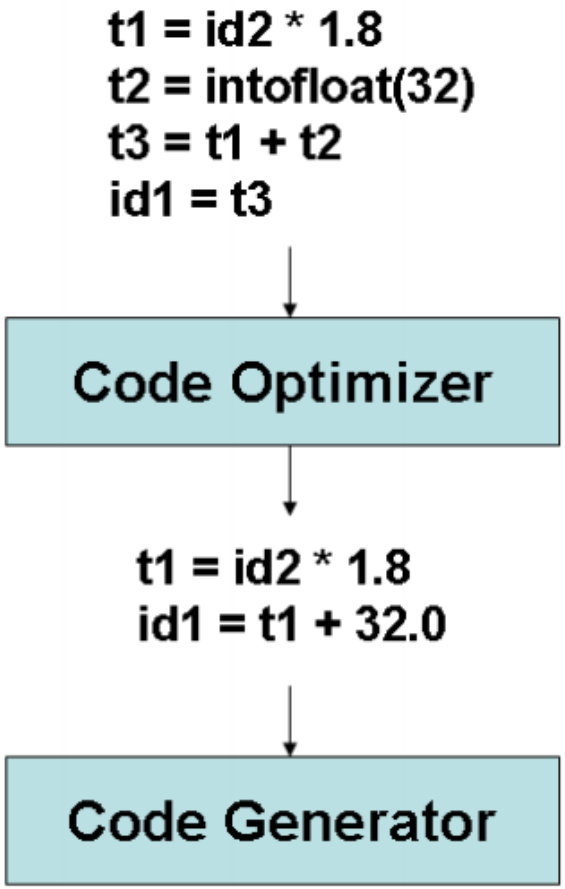
Intermediate Code Generation



Code Optimization

- Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements
- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
    - Code optimization removes such inefficiencies and improves code
    - For Example:
      - Dead code elimination
      - Loop invariant code motion
    - Improvement may be time, space, or power consuming

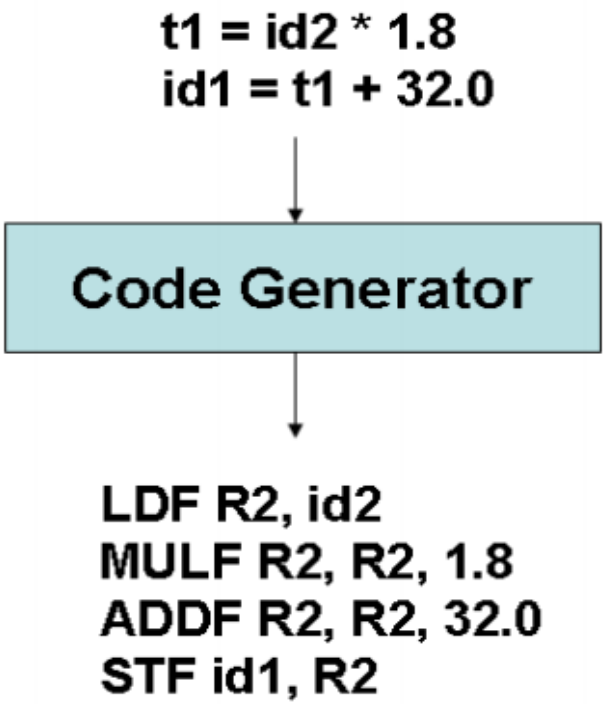
Code Optimization



Code Generation

- Converts intermediate code to machine code
- Each intermediate code instruction may result in many machine instructions or vice-versa
- Must handle all aspects of machine architecture
  - Registers, pipelining, etc.
- Generating efficient code is an NP-complete problem
- Storage allocation decisions are made here
  - Register allocation and assignment are the most important problems

Code Generation



Machine Dependent Optimization

- Peephole optimizations
  - Redundant instruction elimination
  - Analyze sequence of instructions in a small window (*peephole*) and using preset patterns, replace them with a more efficient sequence
    - Example: Strength reduction (left shift, right shift), replace slower instruction with faster instruction (inc, dec)

- If you want to design a C compiler for a new machine, you can take same phases till intermediate code generation from existing C compiler and change the last few phases

History of Compiler

- The “compiler” word was first used in the early1950s by Grace Murray Hopper
- The first compiler was build by John Backum and his group between 1954 and 1957 at IBM
  - It was FORTRAN, which is usually considered the first high-level language
- COBOL was the first programming language which was compiled on multiple platforms in 1960
  - It was developed for business applications
- C compiler was developed in the 1970s at Bell Laboratories
- C++ was developed at Bell Laboratories over a period starting in 1979
  - Earlier it was called C with Classes, renamed as C++ in 1983

References

- Compilers – Principles, Techniques and Tools, Second Edition by Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman
- Advanced Compiler Design and Implementation by Steven S. Muchnick

End of Chapter 1