CC Lecture 1

Prepared for: 7th Sem, CE, DDU

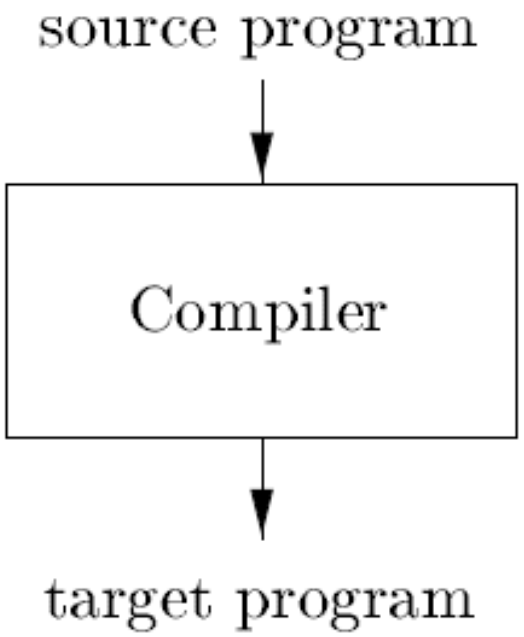
Prepared by: Niyati J. Buch

Introduction

- **Programming languages** are notations for describing computations to people and to machines.
- But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.
- The software systems that do this translation are called **compilers**.

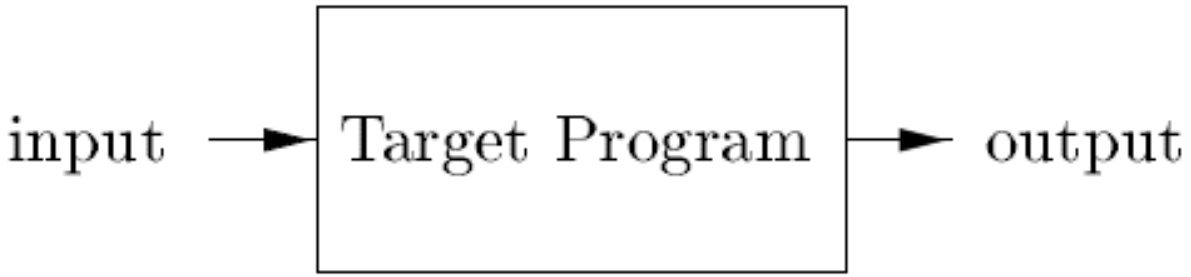
Compiler

- A **compiler** is a program that can read a program in one language the **source language** and translate it into an equivalent program in another language the **target language**.



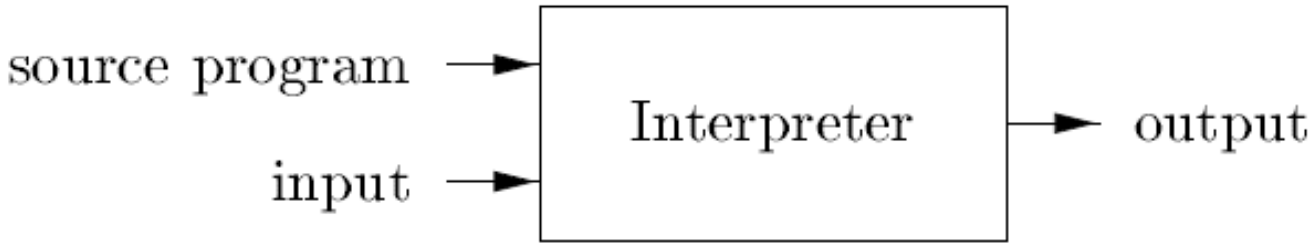
Running the target program

- If the target program is an **executable machine-language program**,
  - it can then be called by the user to process inputs
  - and produce outputs.



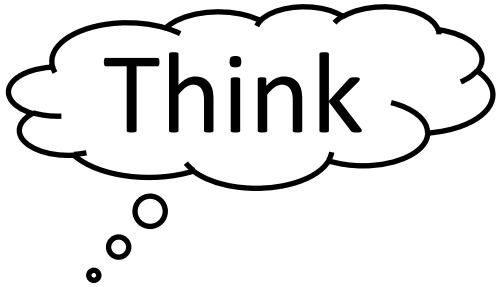
Interpreter

- Instead of producing a target program as a translation, an **interpreter** appears to directly execute the operations specified in the source program on inputs supplied by the user.



Compiler vs. Interpreter

- The machine-language target program produced by a **compiler** is usually much **faster than an interpreter** at mapping inputs to outputs.
- An **interpreter**, however, can usually give **better error diagnostics than a compiler**, because it executes the source program statement by statement.

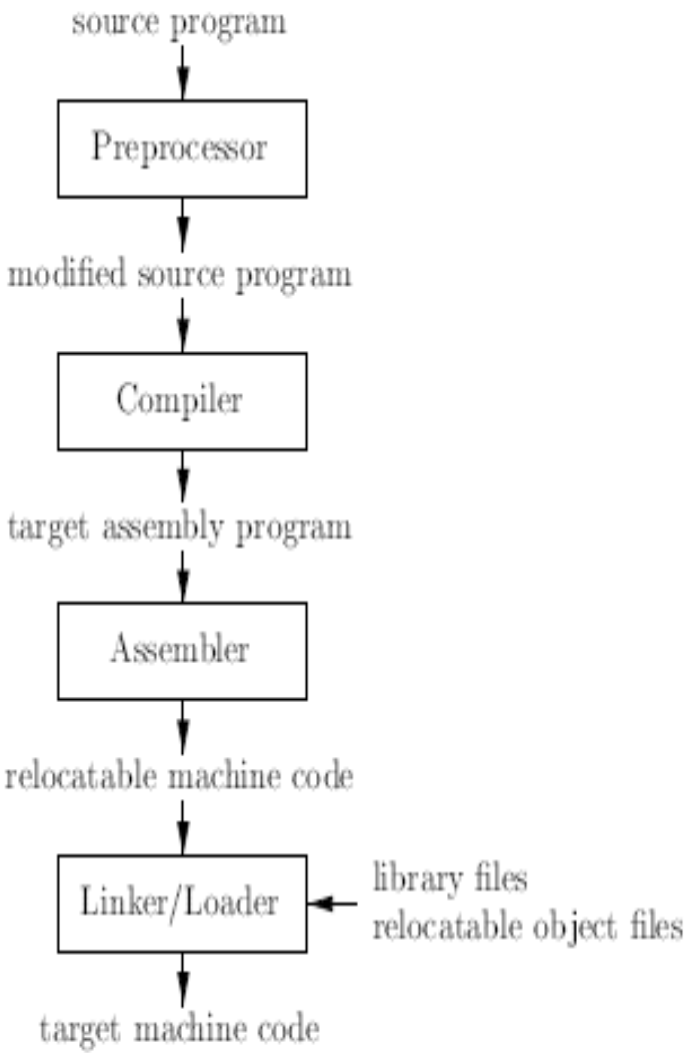


1. What is the difference between a compiler and an interpreter?
2. What are the advantages of

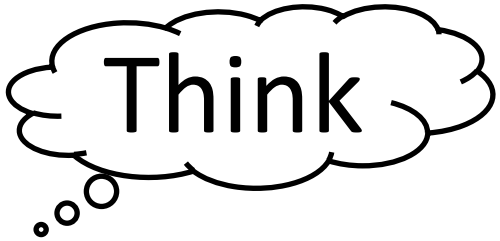
   (a) a compiler over an interpreter

   (b)an interpreter over a compiler

Language Processing System



- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The **linker** resolves external memory addresses, where the code in one file may refer to a location in another file.
- The **loader** then puts together all of the executable object files into memory for execution.

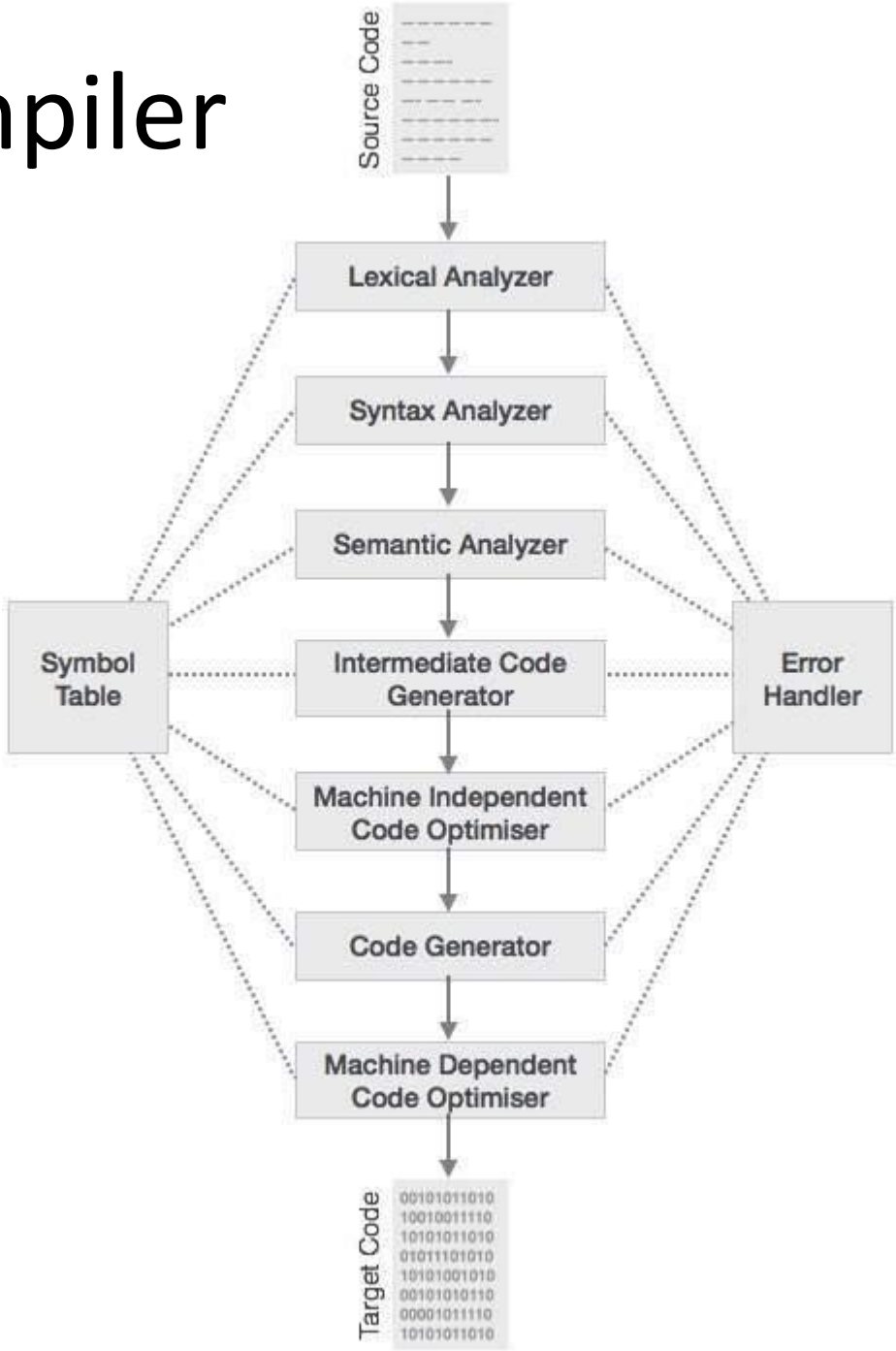


1. What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

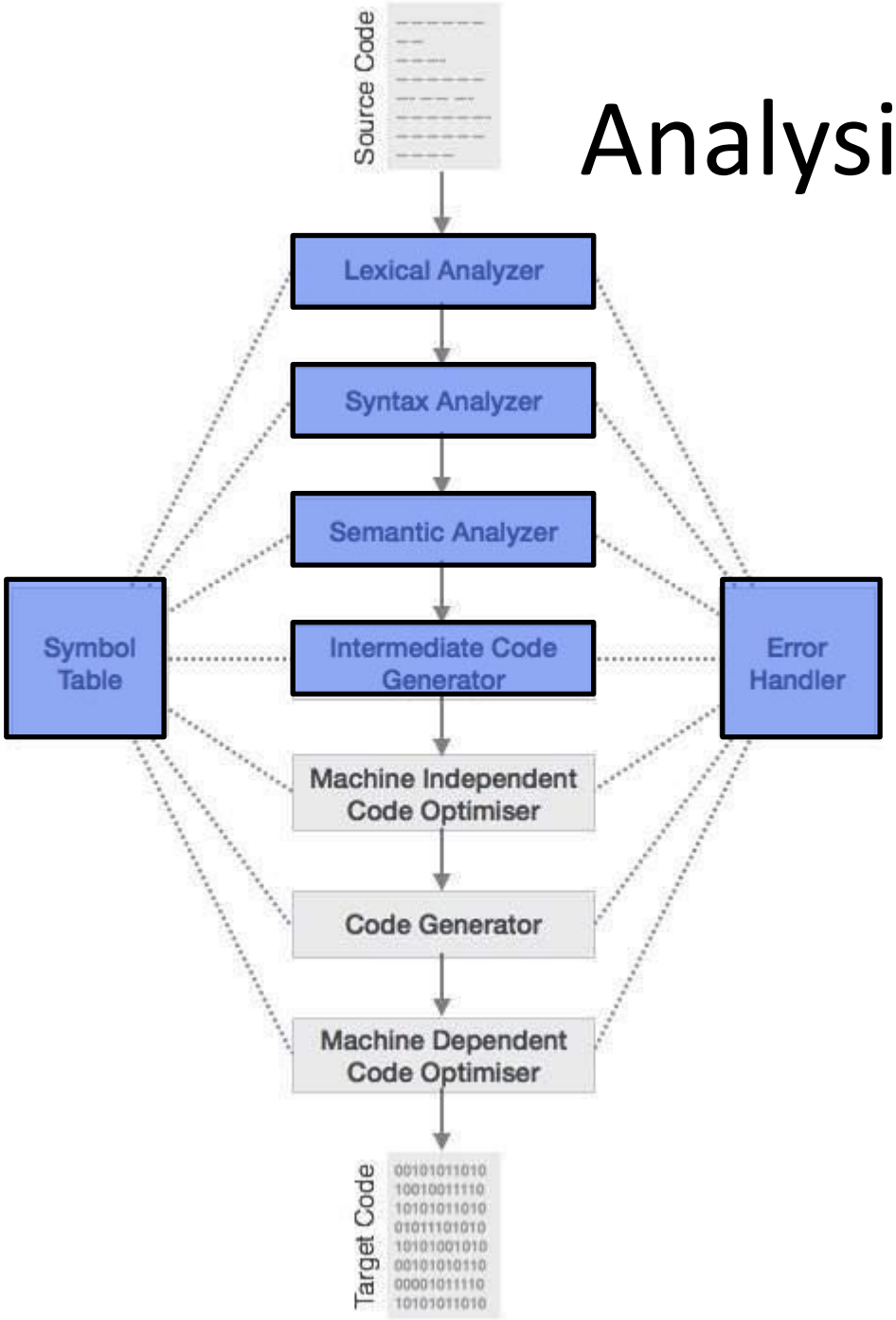
2. A compiler that translates a high-level language into another high-level language is called **a source-to-source translator**.

   What advantages are there to using C as a target language for a compiler?

Phases of Compiler

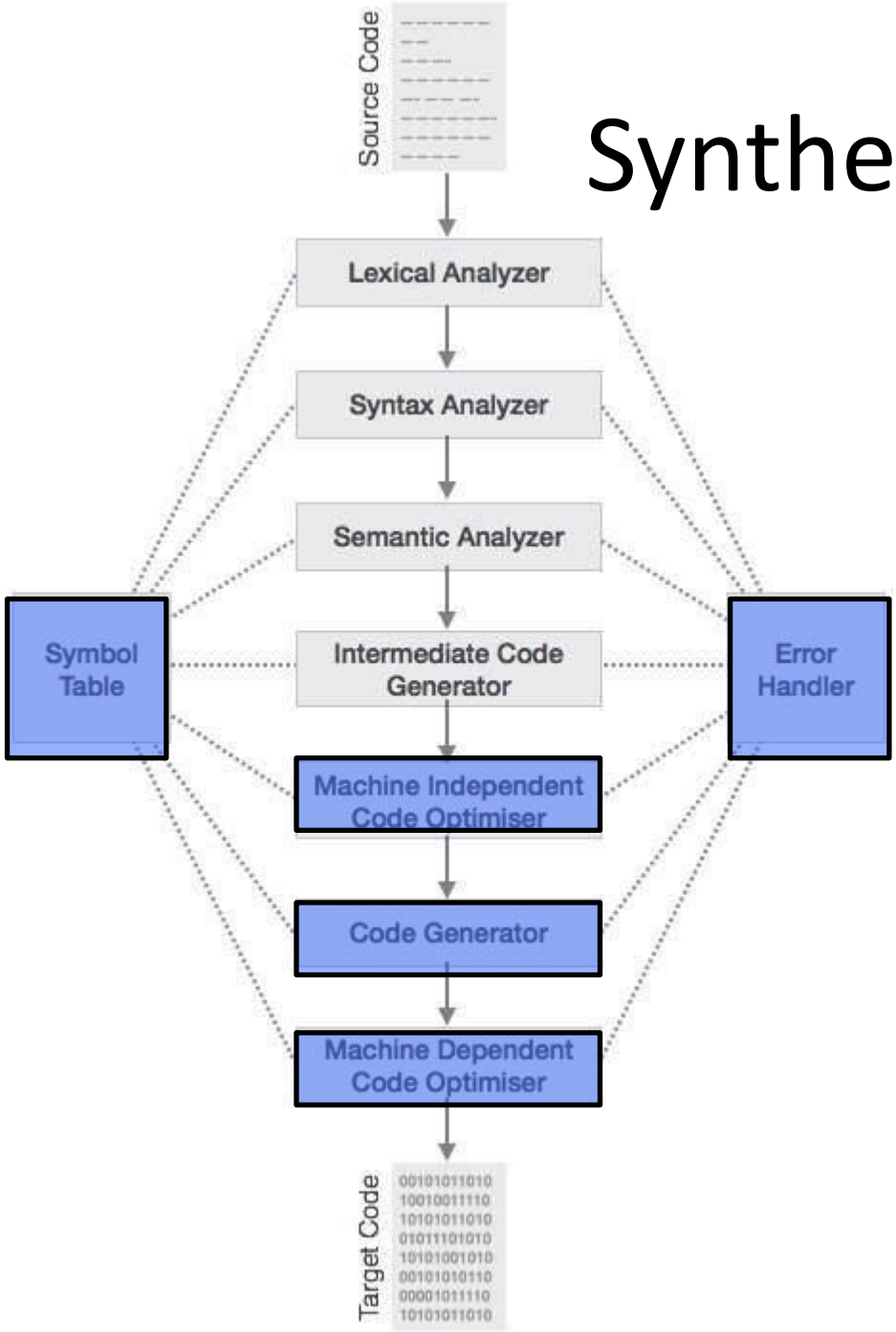


Analysis Part



- The **analysis part** is often called the front end of the compiler.
- The **analysis part** also collects information about the source program and stores it in a data structure called **a symbol table**, which is passed along with the intermediate representation to the synthesis part.

Synthesis Part



- The **synthesis part** constructs the desired target program from the intermediate representation and the information in the symbol table.

Analysis Part (Step 1 & 2)

- **Lexical Analysis or Scanning**
  - The **lexical analyzer** reads the stream of characters making up the source program and groups the characters into meaningful sequences called **lexemes**.
  - For each lexeme, the lexical analyzer produces as output a token of the form **<token-name; attribute-value>**

- **Syntax analysis or Parsing**
  - The **parser** uses the rest components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation (**syntax tree**) that depicts the grammatical structure of the token stream.

Analysis Part (Step 3 & 4)

- **Semantic Analysis**
  - The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
  - Type checking and type conversion (coercions) are part of semantic analysis.

- **Intermediate Code Generator**:
  - After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation (e.g. three-address code) which should be easy to produce and it should be easy to translate into the target machine.

Symbol Table Management

- The **symbol table** is a data structure containing a record for each variable name, with fields for the attributes of the name.

- The data structure should be designed to allow the compiler **to add** the record for each name **quickly** and **to store or retrieve data** from that record **quickly**.

Synthesis Part (Step 5)

- **Code Optimization**:
  - The machine-independent code-optimization phase attempts to improve the intermediate code so that **better** target code will result.
  - Usually better means **faster**, but other objectives may be desired, such as **shorter code**, or target code that consumes **less power**.

Synthesis Part (Step 6)

- **Code Generation**:
  - The code generator takes as input an intermediate representation of the source program and maps it into the **target language**.
  - If the target language is **machine code**, registers or memory locations are selected for each of the variables used by the program
  - Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Synthesis Part (Step 6)

  - A crucial aspect of code generation is the **judicious assignment of registers to hold variables**.
  - The organization of storage at run-time depends on the language being compiled.
- Storage-allocation decisions are made either during intermediate code generation or during code generation.

Synthesis Part (Step 7)

- **Machine Dependent Code Optimization**:
  - Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture.
  - It involves CPU registers and may have absolute memory references rather than relative references.

Code Optimization

- **Code Optimization** aims at improving the execution efficiency of a program.

- This is achieved in two ways:
  1. **Redundancies** in a program are **eliminated**.
  2. **Computations** in a program are **rearranged** or **rewritten** to make it **execute efficiently**.

- Code optimization must not change the meaning of a program.

Scope of optimization

1. Optimization seeks to improve a program rather than the algorithm used in a program.

   Thus, ***replacement of an algorithm by a more efficient algorithm is beyond the scope of optimization***.

2. Efficient code generation for a specific target machine (e.g. by fully exploiting its instruction set) is also beyond its scope; it belongs in the back end of the compiler.

   The optimization techniques are thus independent of both the PL(programming language) and the target machine.

Two pass schematic for language processing

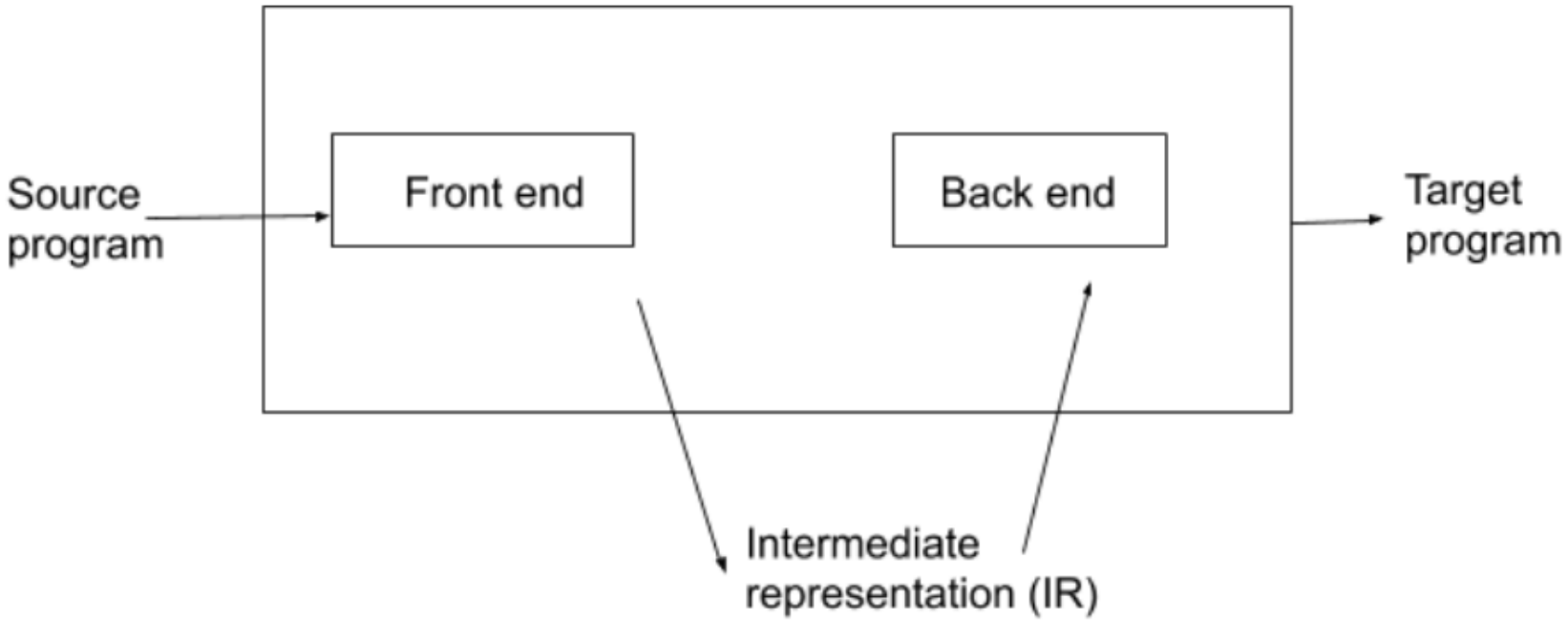


Figure 1 Two pass schematic for language processing

Schematic of an optimizing compiler

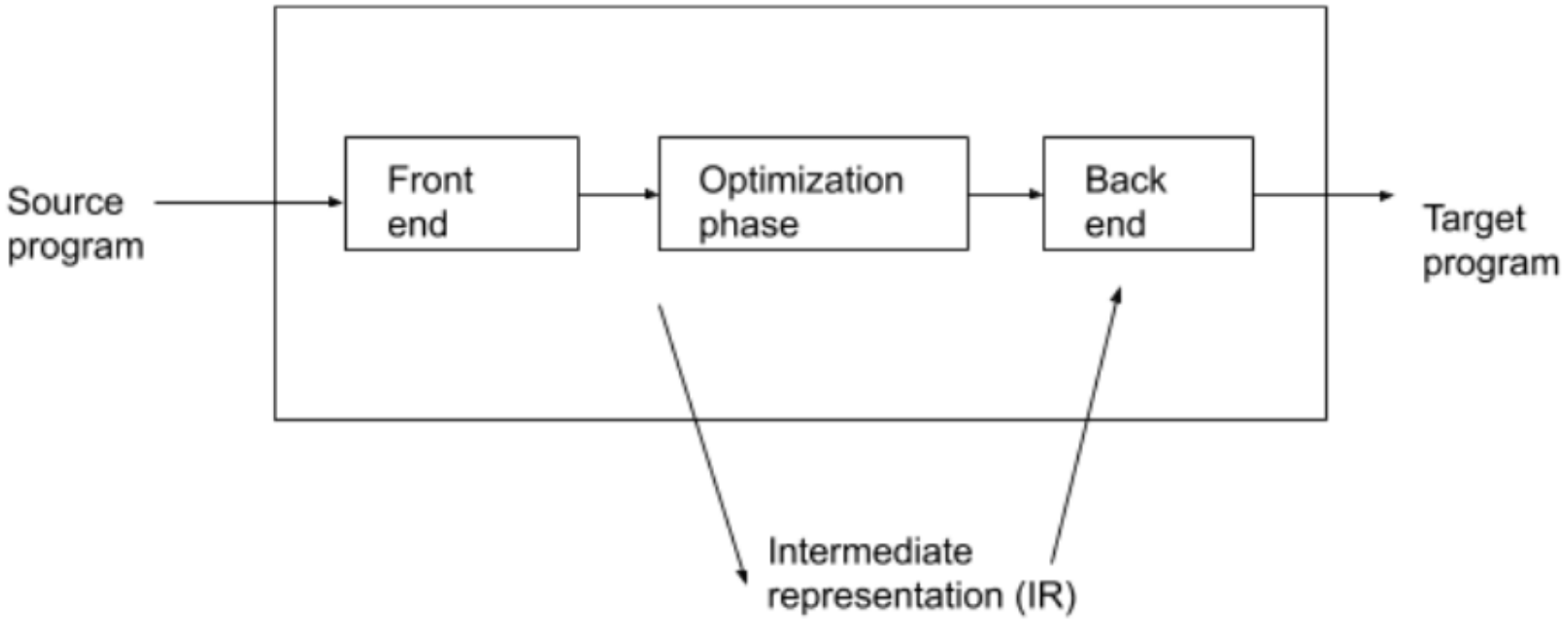


Figure 2 Schematic of an optimizing compiler

Optimizing Transformations

- An ***optimizing transformation*** is a rule for rewriting a segment of a program to improve its execution efficiency without affecting its meaning.

- Optimizing transformations are classified into ***local*** **and** ***global*** transformations depending on whether they are applied over small segments of a program consisting of a few source statements or over large segments consisting of loops or function bodies.

- The reason for this distinction is the difference in the costs and benefits of the optimizing transformations.

Few of the optimizing transformations

1. Compile time evaluation
2. Elimination of common subexpression
3. Dead code elimination
4. Frequency reduction
5. Strength reduction