

Introduction to Compilation

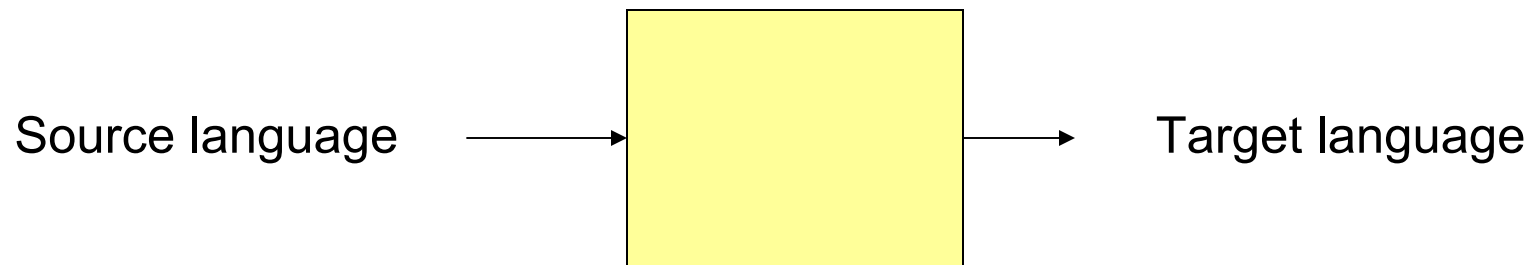
Lecture 01

What is a compiler?

- Programming problems are easier to solve in high-level languages
 - Languages closer to the level of the problem domain, e.g.,
 - SmallTalk: OO programming
 - JavaScript: Web pages
- Solutions are usually more efficient (faster, smaller) when written in machine language
 - Language that reflects to the cycle-by-cycle working of a processor
- Compilers are the bridges:
 - Tools to translate programs written in high-level languages to efficient executable code

What is a compiler?

A program that reads a program written in one language and translates it into another language.



Traditionally, compilers go from high-level languages to low-level languages.

Compilation task is full of variety??

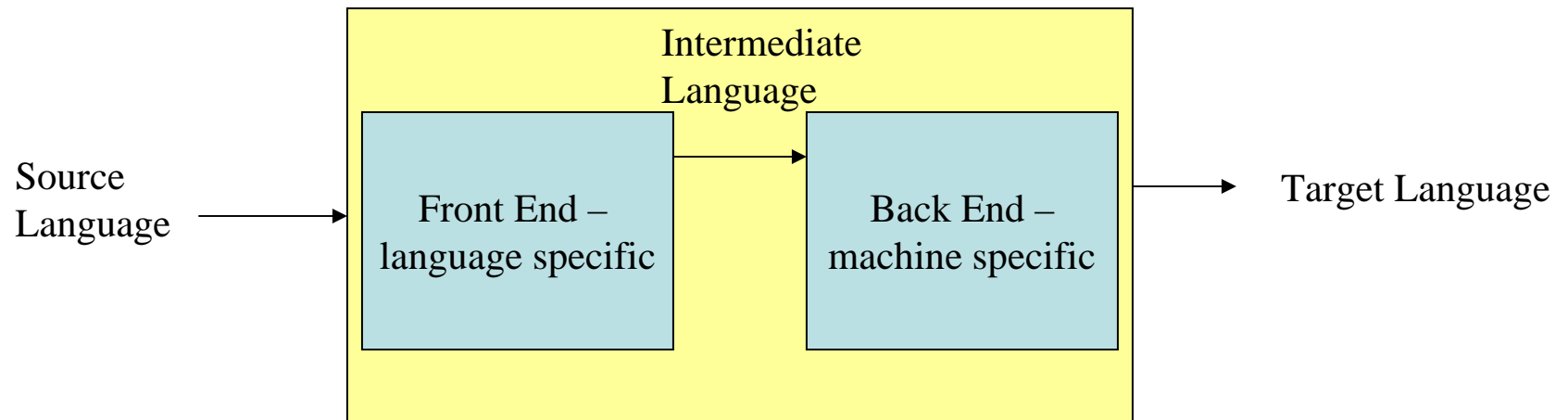
- Thousands of source languages
 - Fortran, Pascal, C, C++, Java,
- Thousands of target languages
 - Some other lower level language (assembly language), machine language
- Compilation process has similar variety
 - Single pass, multi-pass, load-and-go, optimizing....
- Variety is overwhelming.....
- Good news is:
 - Few basic techniques is sufficient to cover all variety
 - Many efficient tools are available

Requirement

- In order to translate statements in a language, one needs to understand both
 - the structure of the language: the way “sentences” are constructed in the language, and
 - the meaning of the language: what each “sentence” stands for.
- Terminology:
 - Structure \equiv Syntax
 - Meaning \equiv Semantics

Analysis-Synthesis model of compilation

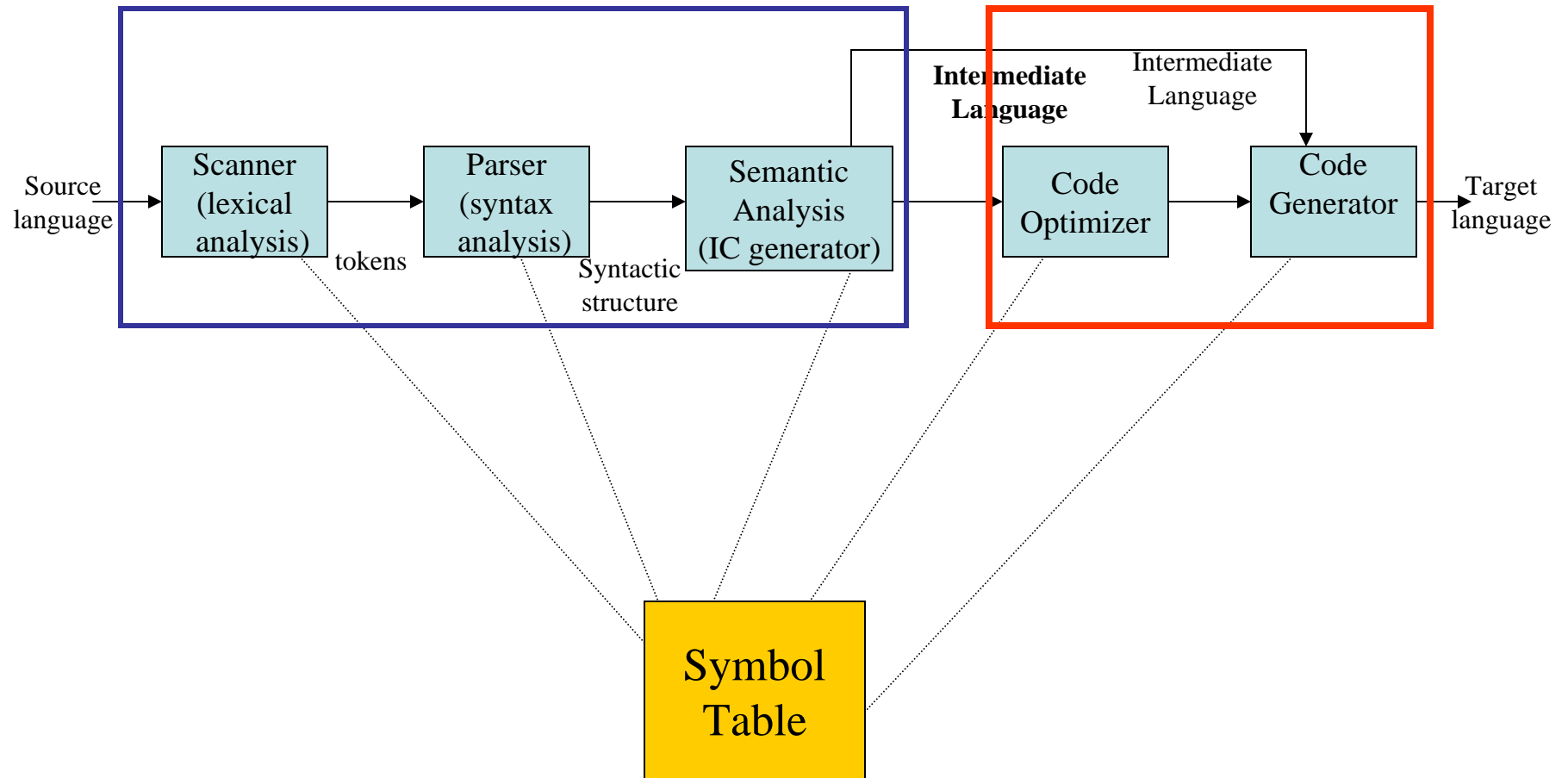
- Two parts
 - **Analysis**
 - Breaks up the source program into constituents
 - **Synthesis**
 - Constructs the target program



Software tools that performs analysis

- Structure Editors
 - Gives hierarchical structure
 - Suggests keywords/structures automatically
- Pretty Printer
 - Provides an organized and structured look to program
 - Different color, font, indentation are used
- Static Checkers
 - Tries to discover potential bugs without running
 - Identify logical errors, type-checking, dead codes identification etc
- Interpreters
 - Does not produce a target program
 - Executes the operations implied by the program

Compilation Steps/Phases



Compilation Steps/Phases

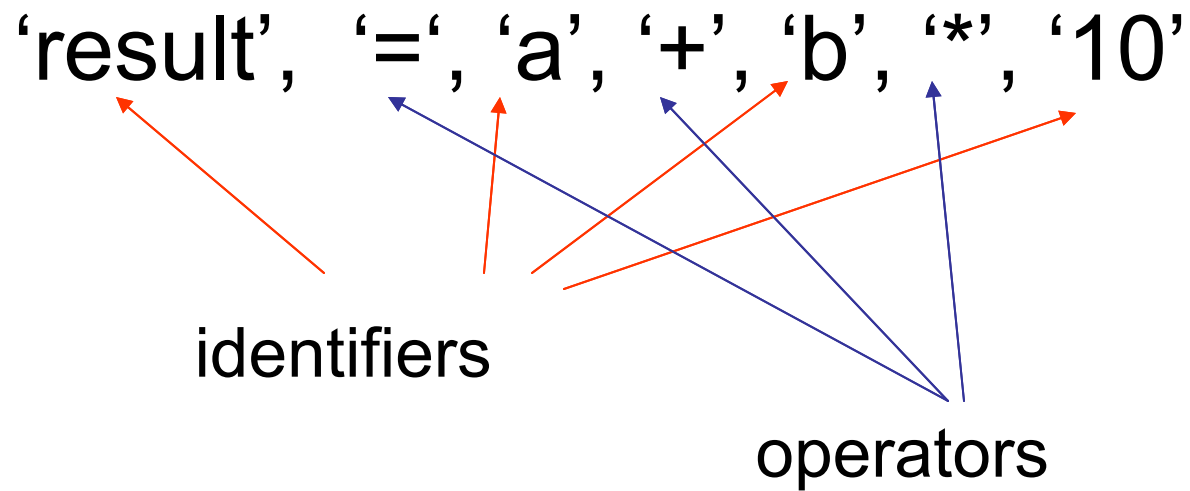
- **Lexical Analysis Phase:** Generates the “tokens” in the source program
- **Syntax Analysis Phase:** Recognizes “sentences” in the program using the syntax of the language
- **Semantic Analysis Phase:** Infers information about the program using the semantics of the language
- **Intermediate Code Generation Phase:** Generates “abstract” code based on the syntactic structure of the program and the semantic information from Phase 2
- **Optimization Phase:** Refines the generated code using a series of optimizing transformations
- **Final Code Generation Phase:** Translates the abstract intermediate code into specific machine instructions

Lexical Analysis

- Convert the stream of characters representing input program into a sequence of tokens
- Tokens are the “words” of the programming language
- Lexeme
 - The characters comprising a token
- For instance, the sequence of characters “static int” is recognized as two tokens, representing the two words “static” and “int”
- The sequence of characters “*x++” is recognized as three tokens, representing “*”, “x” and “++”
- Removes the white spaces
- Removes the comments

Lexical Analysis

- Input: result = a + b * 10
- Tokens:



Syntax Analysis (Parsing)

- Uncover the structure of a sentence in the program from a stream of tokens.
- For instance, the phrase “x = +y”, which is recognized as four tokens, representing “x”, “=”, “+” and “y”, has the structure **=(x,+ (y))**, i.e., an assignment expression, that operates on “x” and the expression “+(y)”.
- Build a tree called a parse tree that reflects the structure of the input sentence.

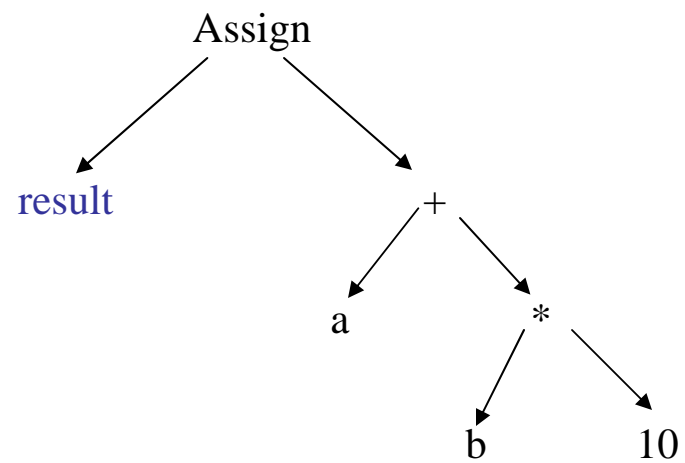
Syntax Analysis: Grammars

- Expression grammar

$$\begin{aligned}\text{Exp} &::= \text{Exp '+' Exp} \\ &| \text{Exp '*' Exp} \\ &| \text{ID} \\ &| \text{NUMBER}\end{aligned}$$
$$\text{Assign} ::= \text{ID '=' Exp}$$

Syntax Tree

Input: result = a + b * 10



Semantic Analysis

- Concerned with the semantic (meaning) of the program
- Performs type checking
 - Operator operand compatibility

Intermediate Code Generation

- Translate each hierarchical structure decorated as tree into intermediate code
- A program translated for an abstract machine
- Properties of intermediate codes
 - Should be easy to generate
 - Should be easy to translate
- Intermediate code hides many machine-level details, but has instruction-level mapping to many assembly languages
- Main motivation: portability
- One commonly used form is “Three-address Code”

Code Optimization

- Apply a series of transformations to improve the time and space efficiency of the generated code.
- Peephole optimizations: generate new instructions by combining/expanding on a small number of consecutive instructions.
- Global optimizations: reorder, remove or add instructions to change the structure of generated code
- Consumes a significant fraction of the compilation time
- Optimization capability varies widely
- Simple optimization techniques can be vary valuable

Code Generation

- Map instructions in the intermediate code to specific machine instructions.
- Memory management, register allocation, instruction selection, instruction scheduling, ...
- Generates sufficient information to enable symbolic debugging.

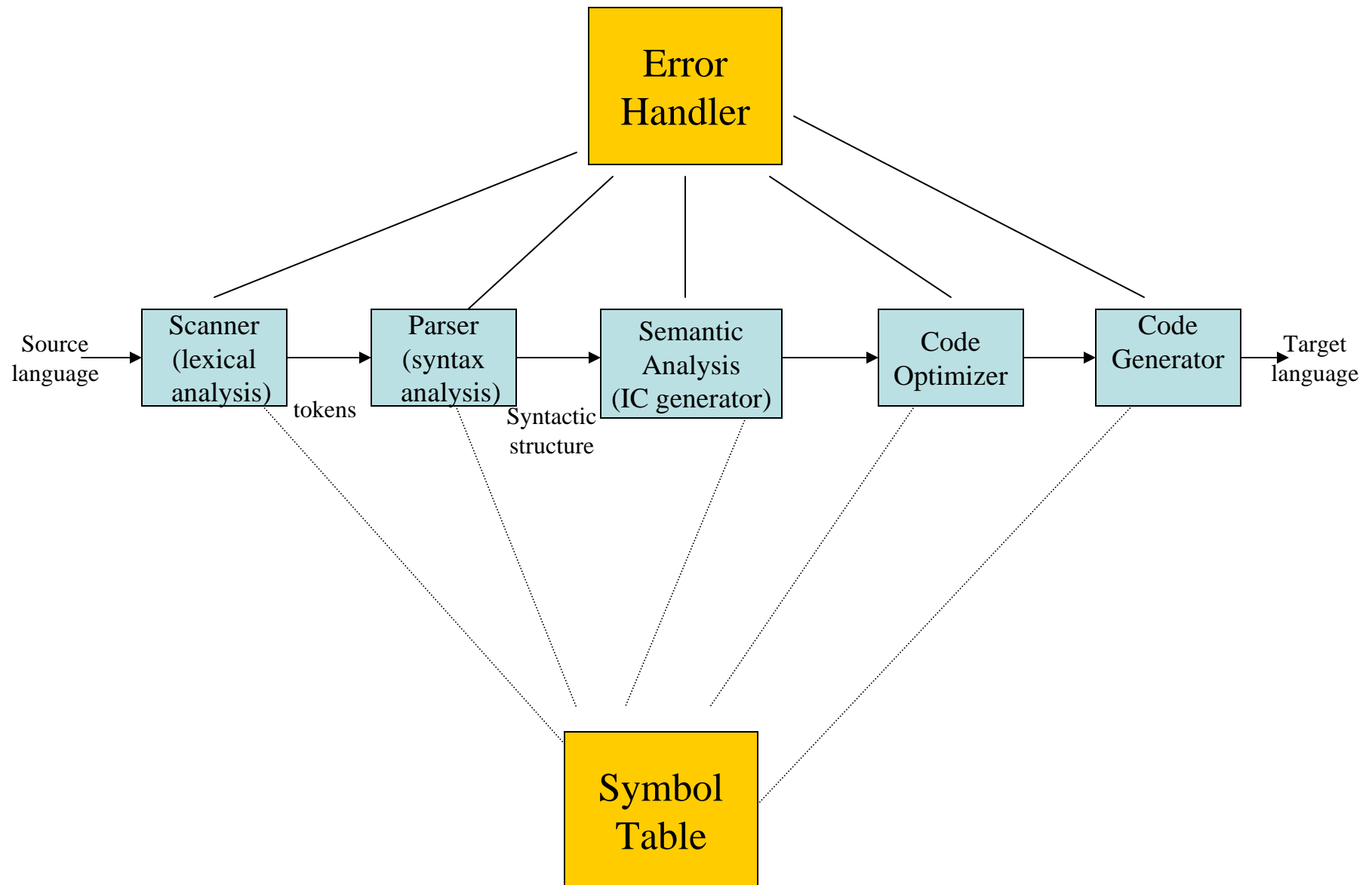
Symbol Table

- Records the identifiers used in the source program
 - Collects various associated information as attributes
 - Variables: type, scope, storage allocation
 - Procedure: number and types of arguments method of argument passing
- It's a data structure with collection of records
 - Different fields are collected and used at different phases of compilation

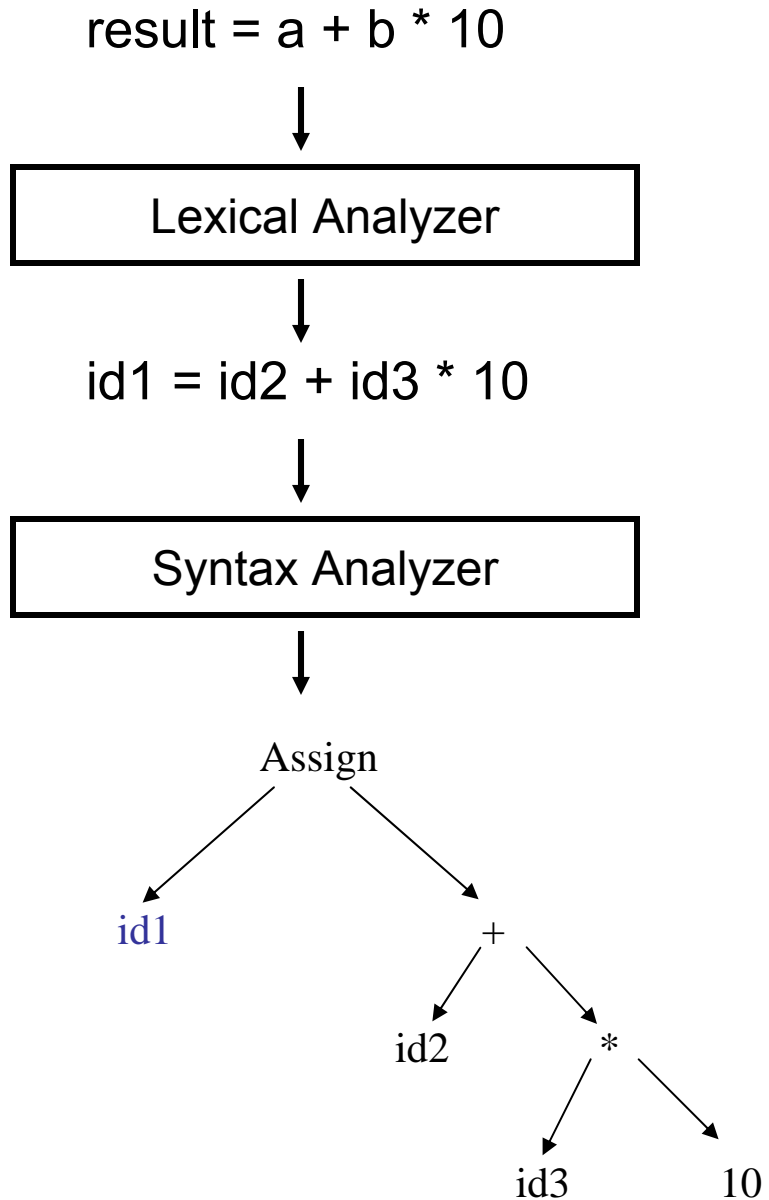
Error Detection, Recovery and Reporting

- Each phase can encounter error
- Specific types of error can be detected by specific phases
 - Lexical Error: `int abc, lnum;`
 - Syntax Error: `total = capital + rate year;`
 - Semantic Error: `value = myarray [realIndex];`
- Should be able to proceed and process the rest of the program after an error detected
- Should be able to link the error with the source program

Error Detection, Recovery and Reporting



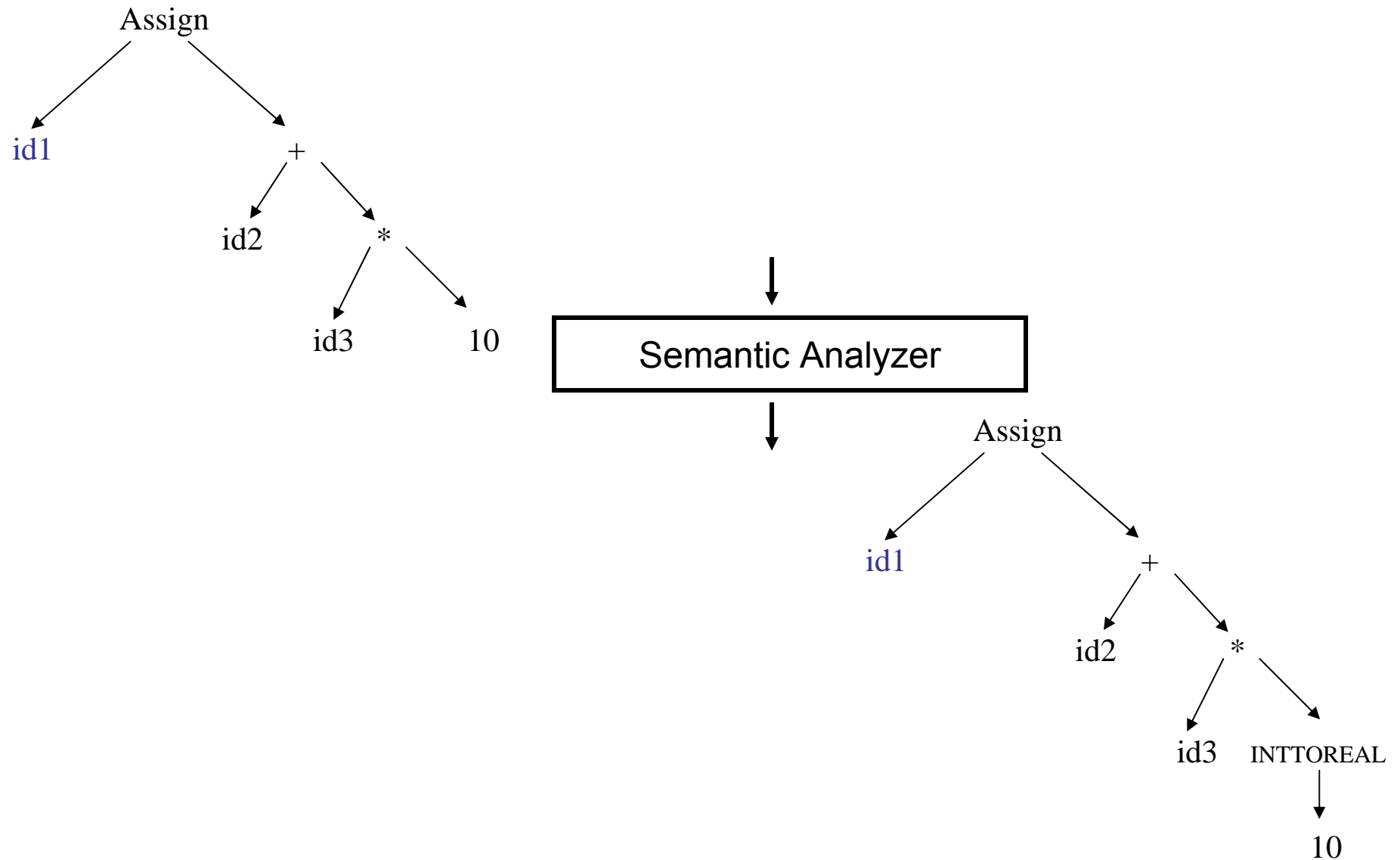
Translation of a statement



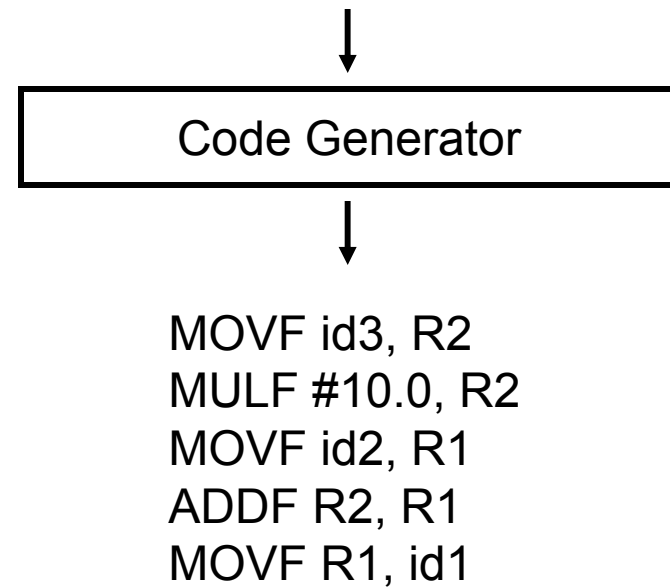
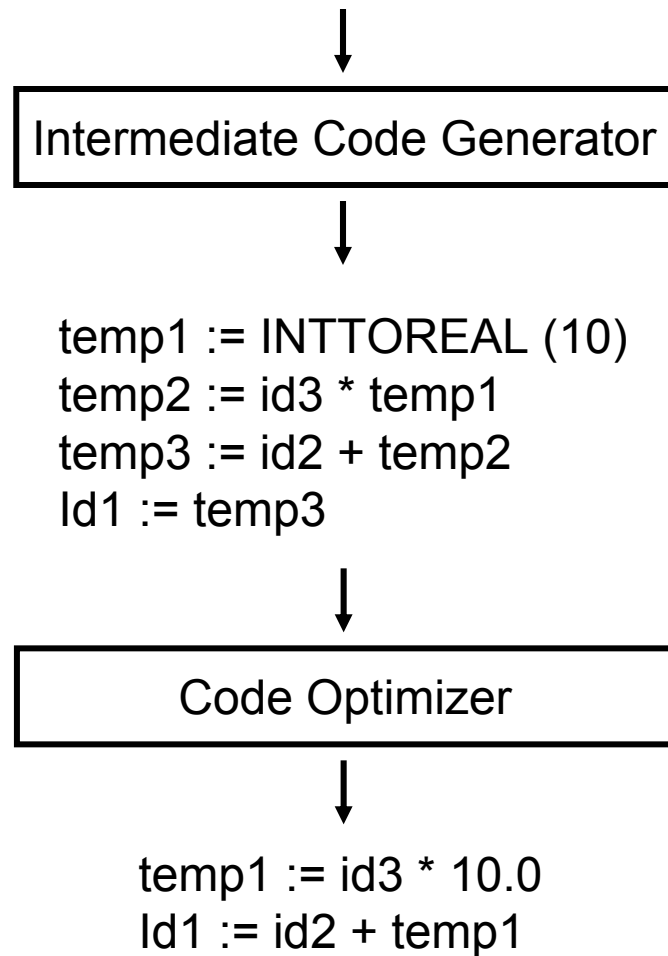
Symbol Table

result
a
b

Translation of a statement



Translation of a statement



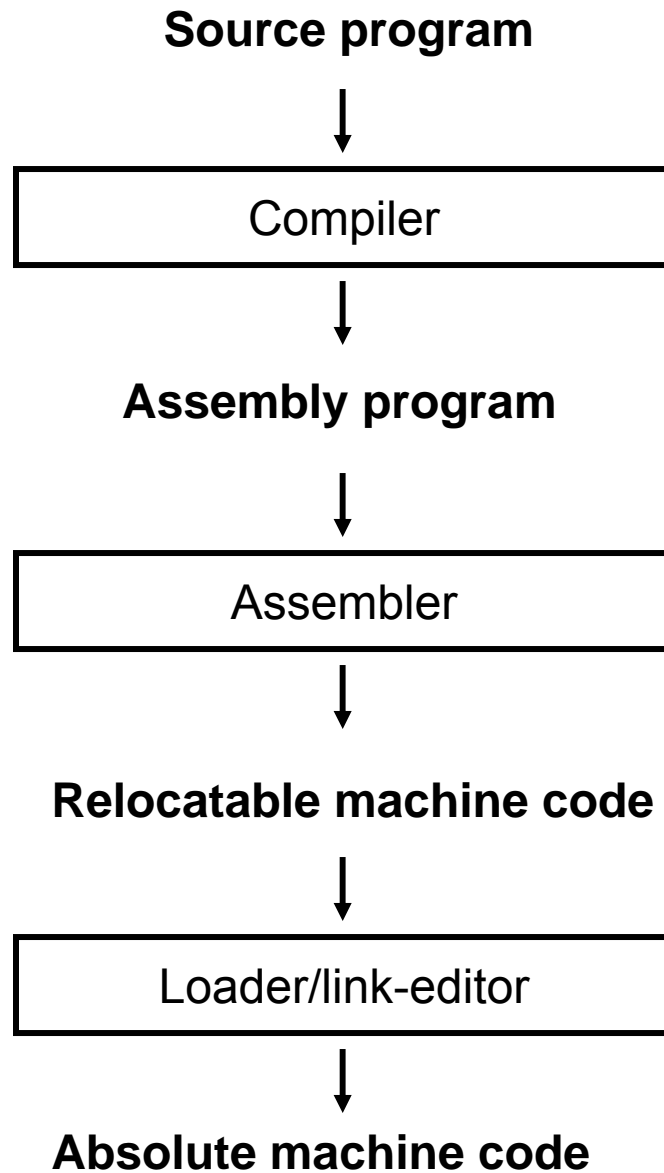
Syntax Analyzer versus Lexical Analyzer

- 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.

Cousins of the Compiler

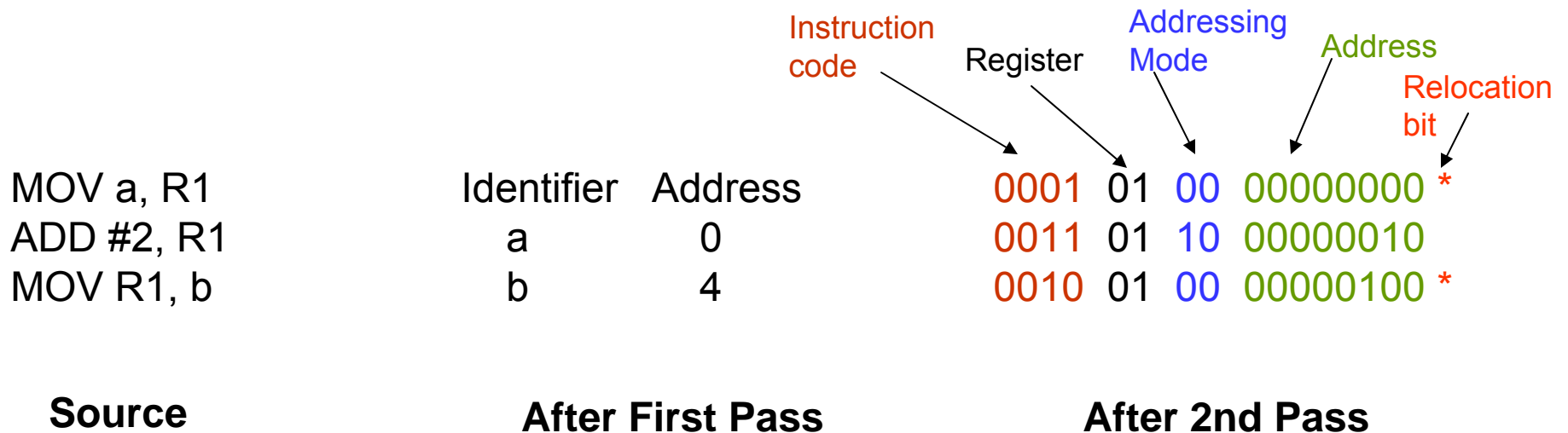
- Preprocessor
 - Macro preprocessing
 - Define and use shorthand for longer constructs
 - File inclusion
 - Include header files
 - “Rational” Preprocessors
 - Augment older languages with modern flow-of-control or data-structures
 - Language Extension
 - Add capabilities to a language
 - Equel: query language embedded in C

Assemblers



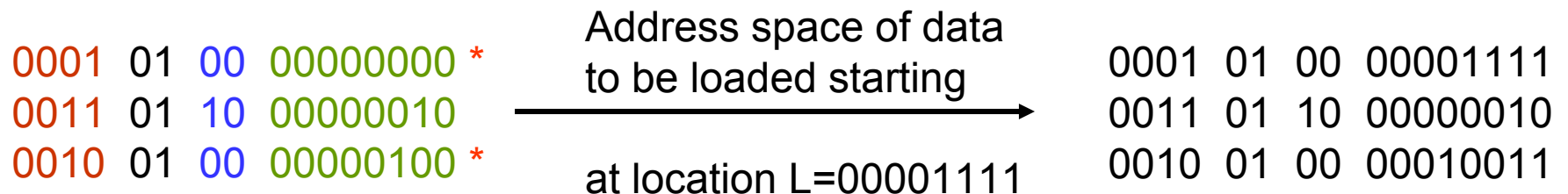
Two-Pass Assembly

- Simplest form of assembler
- First pass
 - All the identifiers are stored in a symbol table
 - Storage is allocated
- Second pass
 - Translates each operand code in the machine language



Loaders and Link-Editors

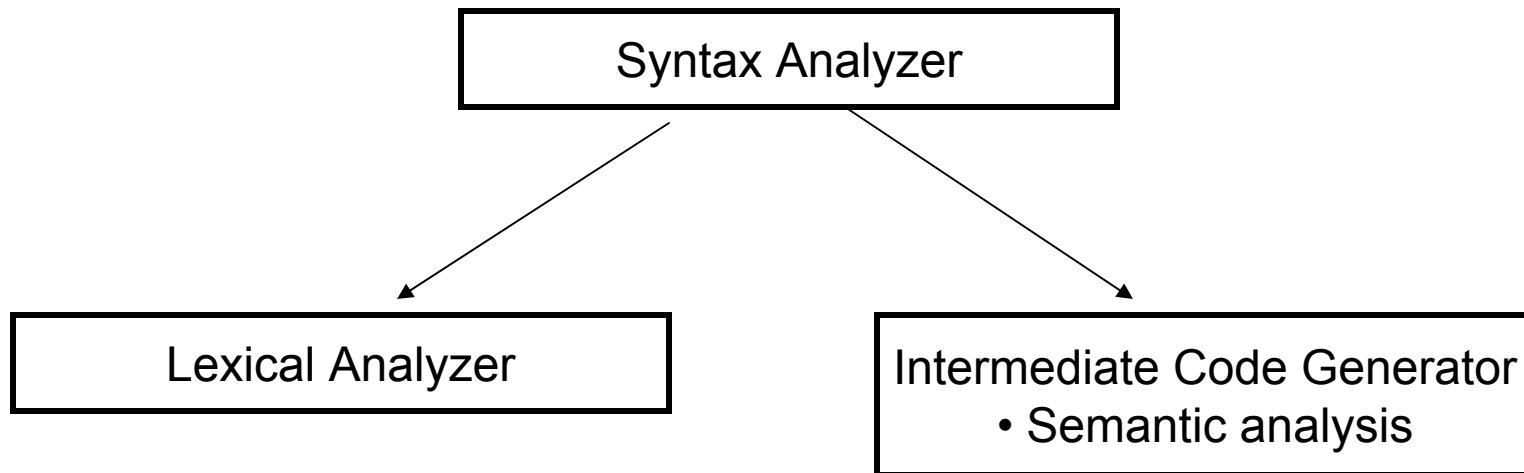
- Convert the relocatable machine code into absolute machine code
 - Map the relocatable address
- Place altered instructions and data in memory



- Make a single program from several files of relocatable machine code
 - Several files of relocatable codes
 - Library files

Multi Pass Compilers

- Passes
 - Several phases of compilers are grouped in to passes
 - Often passes generate an explicit output file
 - In each pass the whole input file/source is processed



How many passes?

- Relatively few passes is desirable
 - Reading and writing intermediate files take time
 - It may require to keep the entire file in memory
 - One phase generate information in different order than that is needed by the next phase
 - Memory space is not trivial in some cases
- Grouping into same pass incurs some problems
 - Intermediate code generation and code generation in the same pass is difficult
 - e.g. Target of 'goto' that jumps forward is now known
 - 'Backpatching' can be a remedy

Issues Driving Compiler Design

- Correctness
- Speed (runtime and compile time)
 - Degrees of optimization
 - Multiple passes
- Space
- Feedback to user
- Debugging

Other Applications

- 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.