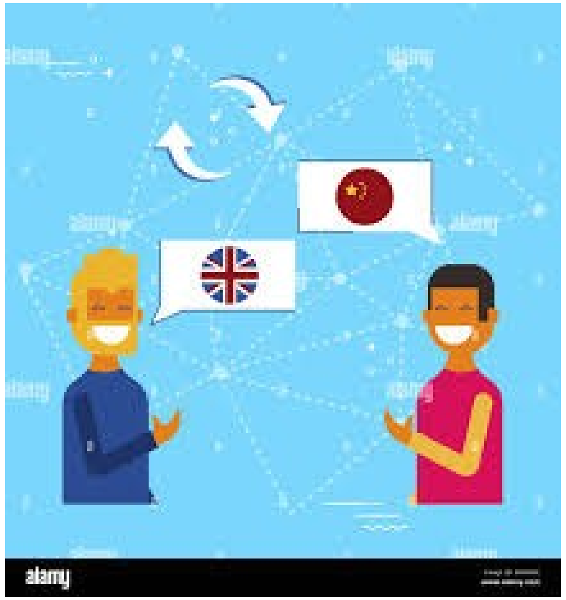
IT302 - Compiler Design

ThE iNtRoDuCtIoN

Motivation

- Language processing is an important component of programming.
  - Just like human languages need grammar for meaningful communication, programming requires understanding syntax, semantics, and structure to avoid ambiguity and ensure correctness.
  - Writing efficient programs demands knowledge of compiler phases—only then can you optimize logic, time, and space by anticipating how the compiler processes your code



Contd.,

- A large number of systems software and application programs require structured input.
  - Mean: many complex programs → **do not accept random input** — accepts **well-defined structure or format**, often following a **grammar**.
  - The reason is to ensure the input can be correctly **parsed, understood**, and **executed**.

1. Operating Systems (command line processing):

Structured Input: cp file1.txt /home/user/documents/

- **How it's processed:** OS uses a **parser** to:
  - Tokenize the command
  - Identify the structure
  - Map it to an internal function or system call

Ex: In Linux, running mkdir -p /new/folder must follow a specific format, or it results in an error.

Contd.,

2. **Databases (Query Language Processing):** SQL queries (Structured Input)

   - SELECT name, age FROM students WHERE age > 18;

- Why Structured**:** SQL follows a strict grammar with keywords (**SELECT, FROM, WHERE**), identifiers (**table/column names**), and expressions.
- How it's processed**:** A **query processor** parses and validates the syntax, then generates a query execution plan.
- Example**:** An incorrect query like SELECT FROM age students will raise a syntax error — because it violates structure.

Contd.,

3. Typesetting Systems like LaTeX: Structured Input: LaTeX documents are written using markup like:

- \documentclass{article}
- \begin{document}
- Hello, World!
- \end{document}

- **Why Structured**: LaTeX commands and environments have strict syntax rules (e.g., **\begin{}** must match **\end{}**).
- **How it's processed**: The LaTeX engine parses the input file line by line, builds a document object tree, and generates a PDF output.
- **Example:** A missing \end{document} or typo like \begn{document} will result in errors or failed compilation.

Motivation….. Summary

- Wherever input has a structure one can think of language processing
- Why study compilers?
  - Compilers use the whole spectrum of language processing technology

How are Languages Implemented?

- Two major strategies:
  - Interpreters (older, less studied)
  - Compilers (newer, much more studied, very well understood with mathematical foundations)

- Interpreters run programs “as is”
  - Little or no preprocessing

- Some environments provide both interpreter and compiler.
  - Interpreter for development
  - Compiler for deployment

- Java
  - Java compiler: Java to interpretable bytecode
  - Java (Just-In-Time) JIT: bytecode to executable image

- Compilers do extensive preprocessing

History of High-Level Languages

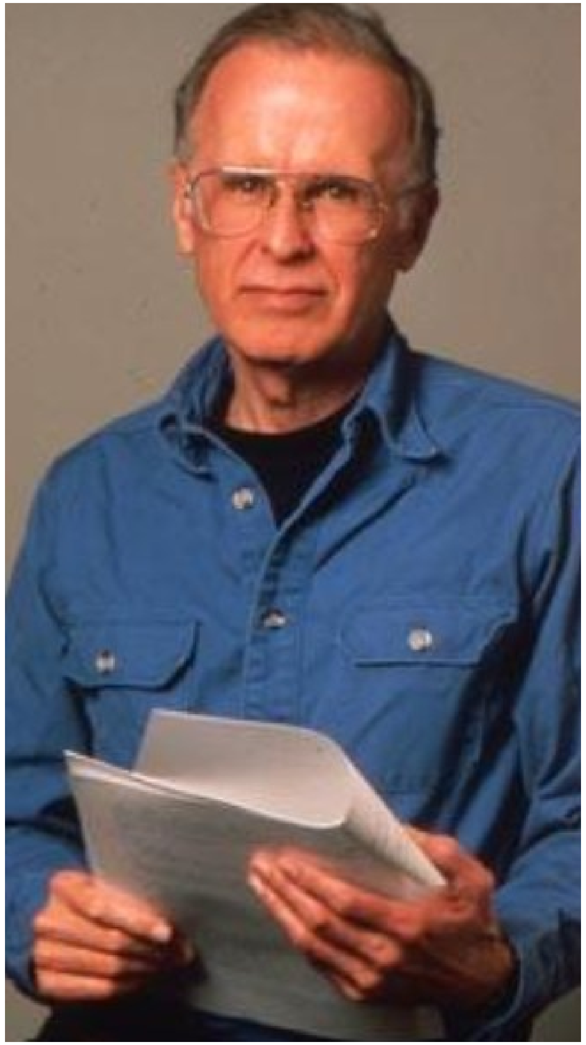
- 1953 IBM develops the 701 →→ Till then, all programming done in assembly
- Problem: Software costs exceeded hardware costs! Cost of software, Low productivity

- John Backus: “Speedcoding”
  - An interpreter
  - Ran 10-20 times slower than
  - hand-written assembly

- **John Backus (in 1954):** Proposed a program that translated high level expressions into native machine code.
- Skeptism all around. Most people thought it was impossible
- Fortran I project (1954-1957): The first compiler was released

FORTRAN I

- 1954 IBM develops the 704
- John Backus
  - Idea: translate high-level code to assembly
  - Many thought this impossible
    - Had already failed in other projects
- 1954 to 57: FORTRAN I project
- By 1958, >50% of all software is in FORTRAN
- Cut development time dramatically
  - (2 weeks → 2 hours)



FORTRAN I

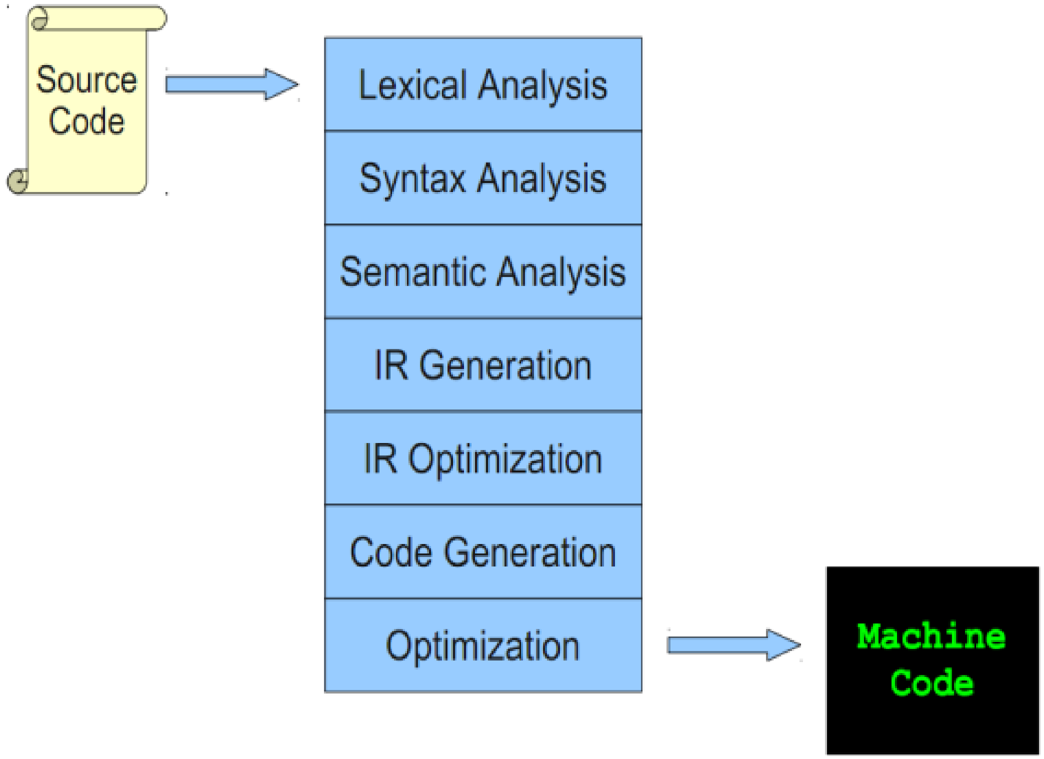
- The first compiler
  - Produced code almost as good as hand-written
  - Huge impact on computer science
- Led to an enormous body of theoretical work
- Modern compilers preserve the outlines of the FORTRAN I compiler

Organizing a Compiler

- Split work into different compiler phases
- Phases transform one program representation into another
- Every phase is relatively simple
- Phases can be between different types of program representations
- Phases can be on the same program representation

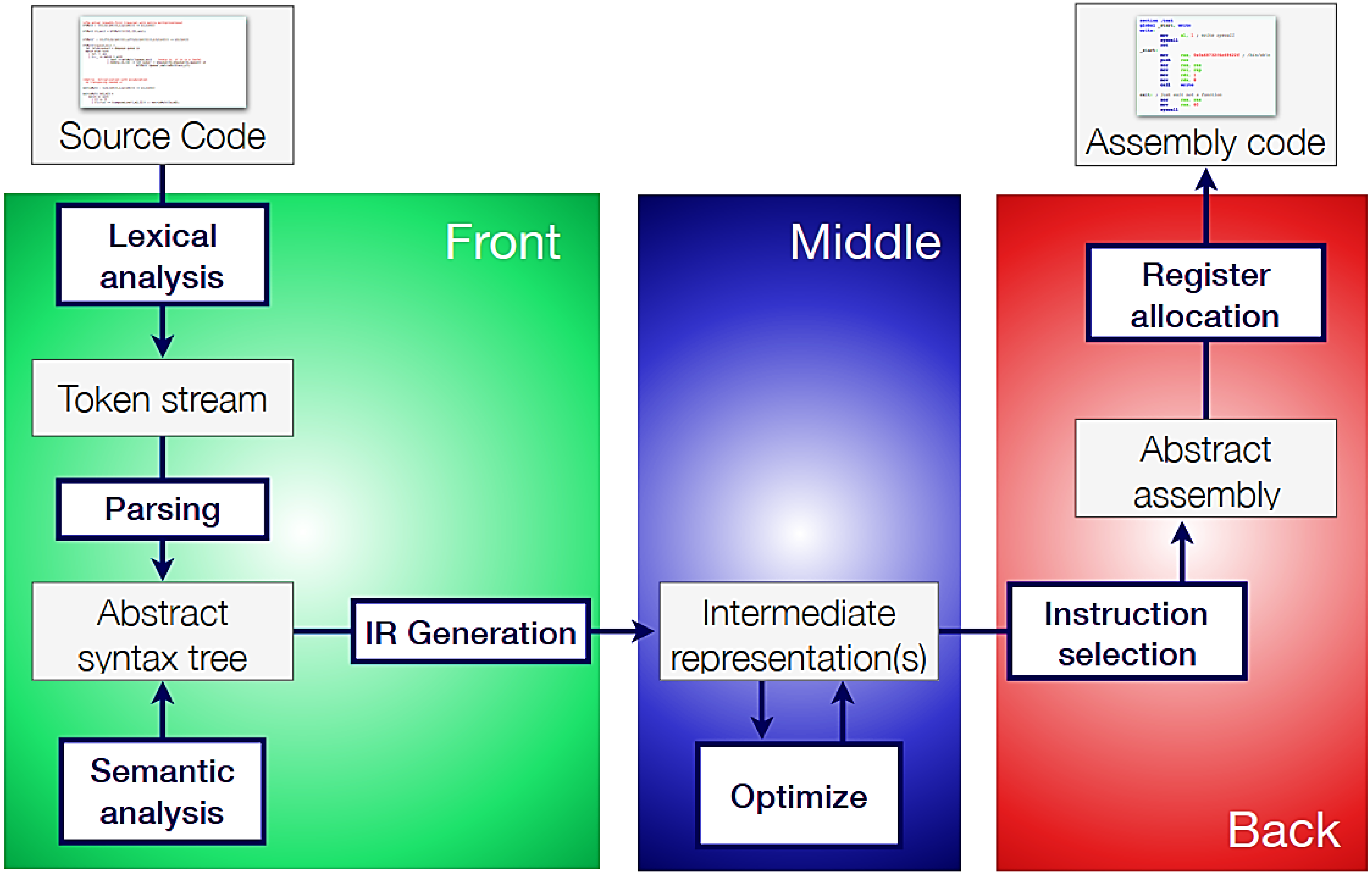
The Structure of a Compiler

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. IR Optimization
5. Code Generation
6. Low-level Optimization



The first 3, at least, can be understood by analogy to how humans comprehend English

Compiler Phases



keywords, identifiers, symbols

Builds a syntax tree from tokens, checking grammar rules.

Checks **types**, **scope**, and **logical correctness**.

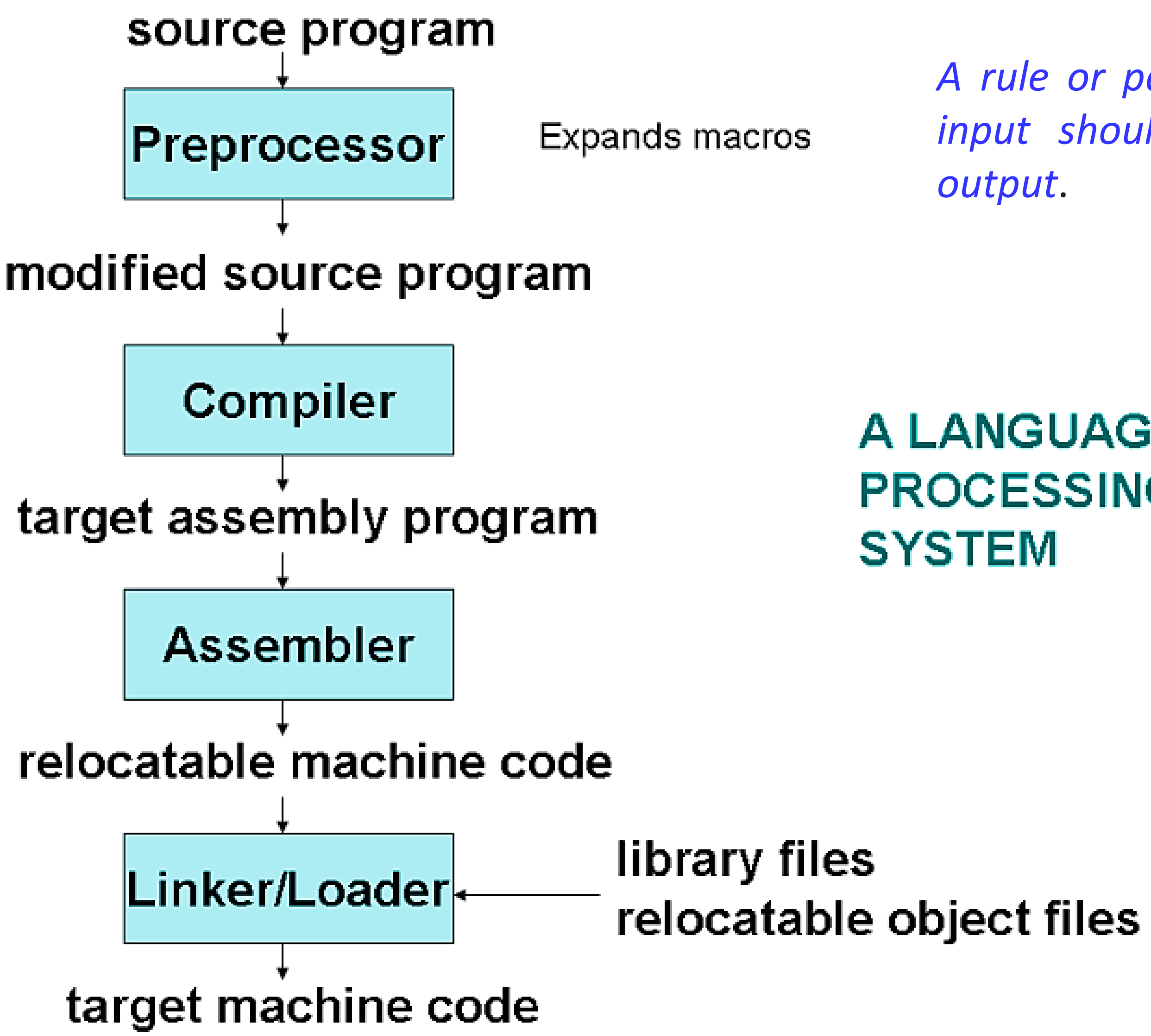
Performs **optimizations** on an **intermediate representation (IR)**

Generates **target code** from optimized IR

Assigns variables to physical CPU registers.

Maps high-level operations to machine-level

Language Processing System



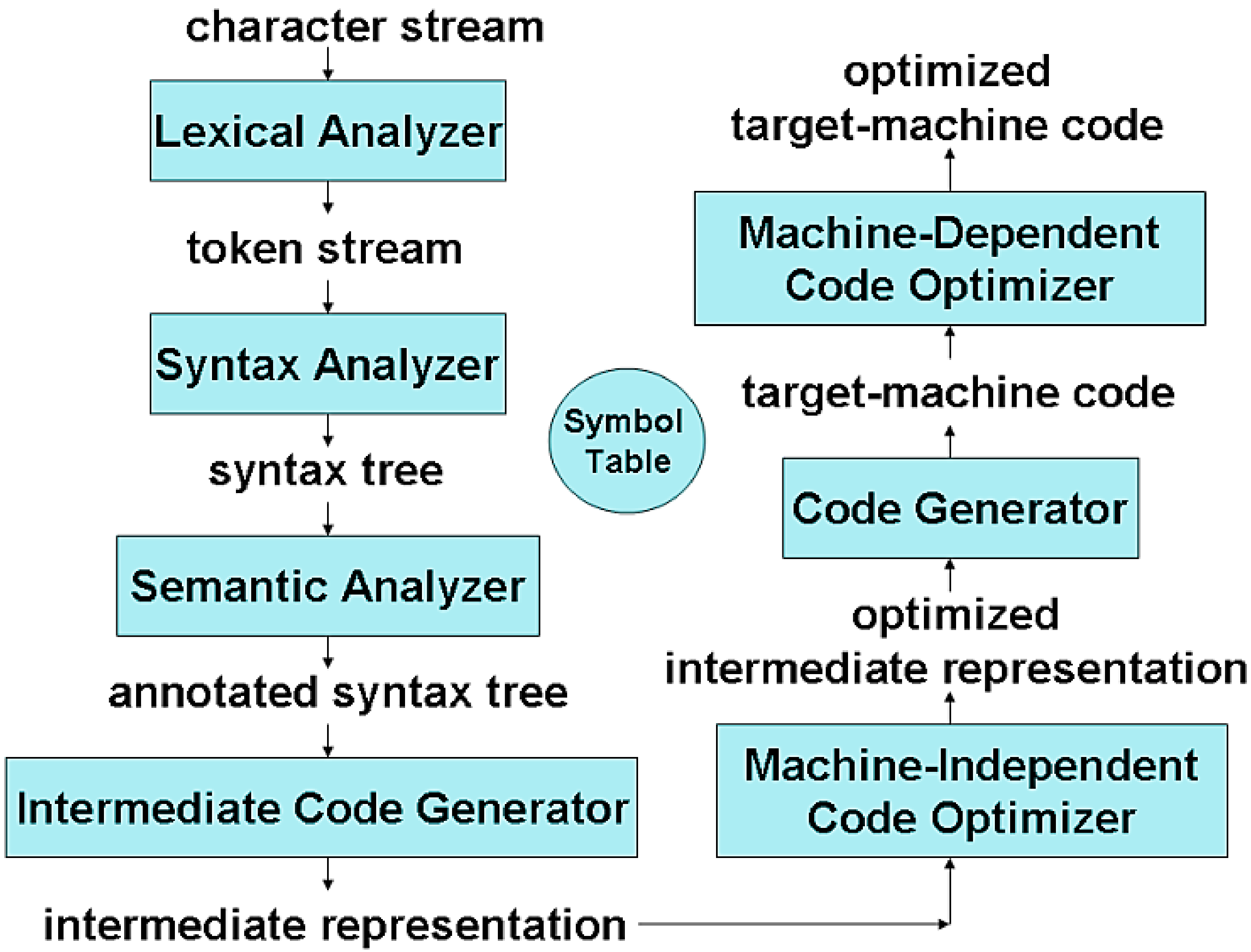
A rule or pattern that specifies how a certain input should be mapped to a replacement output.

A LANGUAGE PROCESSING SYSTEM

Mapping Previous Diagram to a Real-World Analogy

- Imagine **building a house**:
    1. **Source Program** = Blueprint
    2. **Preprocessor** = Designer expands repetitive elements
    3. **Compiler** = Converts design to construction instructions
    4. **Assembler** = Workers use instructions to build parts
    5. **Linker/Loader** = Assembles all parts into the final house and moves in furniture (libraries)

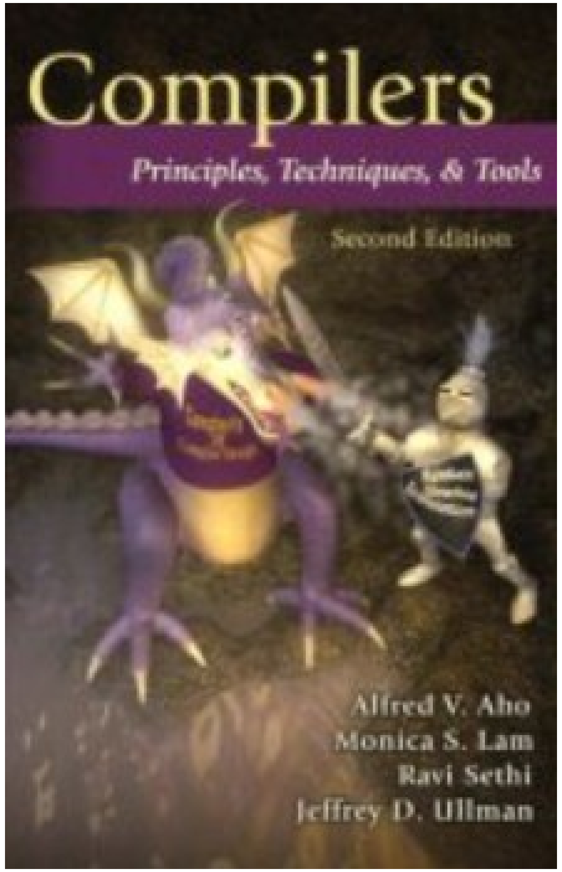
Compiler Overview

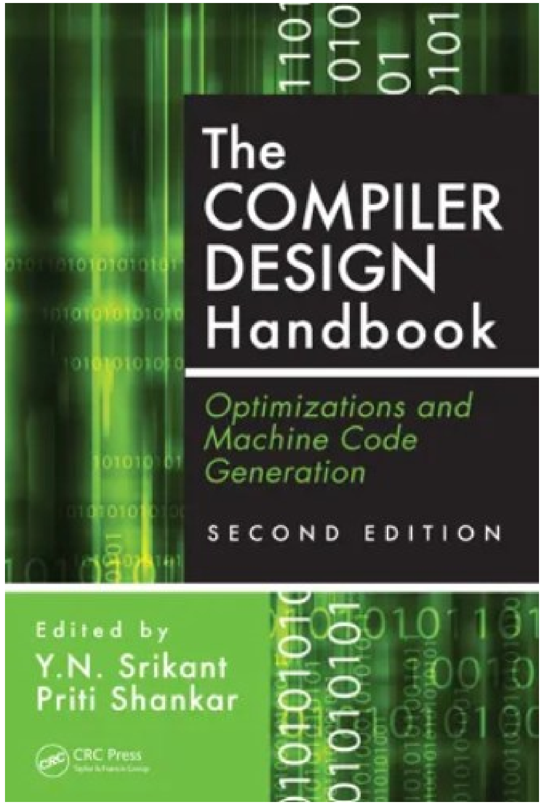


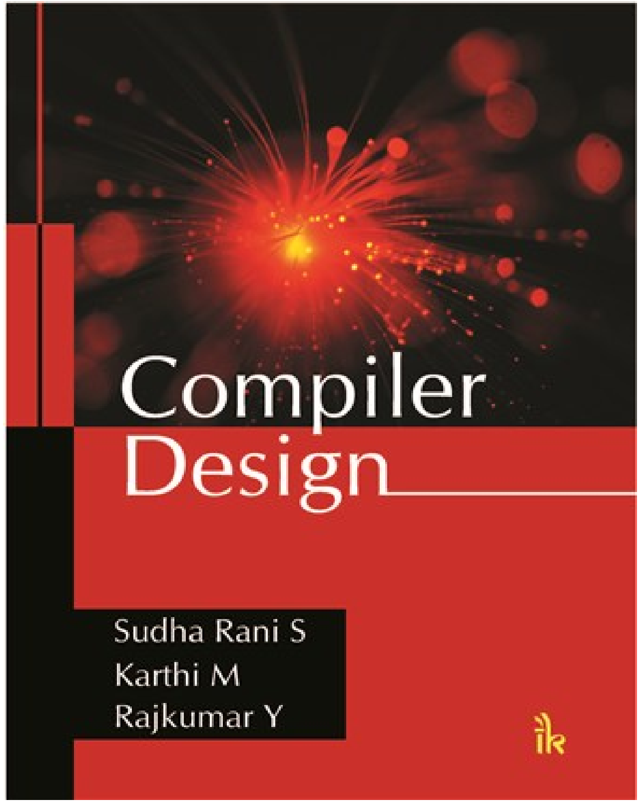
About the Complexity of Compiler Technology

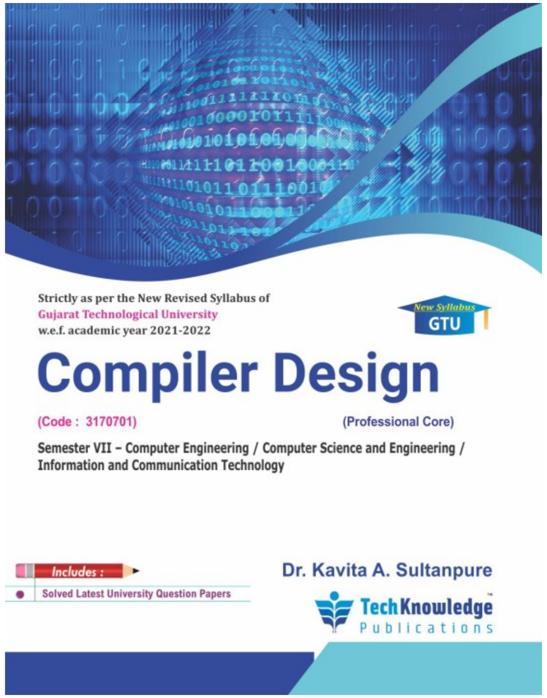
- A compiler is possibly the most complex system software and writing it is a significant exercise in software engineering
- The complexity arises from the fact that it is required to map a programmer’s requirements (in a HLL program) to architectural details
- It uses algorithms and techniques from a very large number of areas in computer science
- Translates difficult theory into practice - enables tool building.

Text/Reference Books









Finding a partner for the labs

I strongly suggest to work in teams of two.

Labs — Finding a Partner

Don’t panic.

There are two options **(new!)**

1. You fill out a questionnaire and we *suggest* a partner (staff selection)
    - Suggestion is not binding but it’s expected that you team up
2. You team up with somebody yourself (self selection)
    - Like in previous iterations of the course

Labs — Picking a Programming Language

- You can freely choose a programming language to use
- I strongly suggest to use a typed functional language
- Writing a compiler is a killer app for functional programming
- When picking a language also consider the availability of parser generators and libraries

Syllabus

- Syllabus Introduction to programming language translation. Lexical analysis: Specification and recognition of tokens.
- Syntax analysis: Top-down parsing-Recursive descent and Predictive Parsers. Bottom-up ParsingLR (0), SLR, and LR (1) Parsers.
- Semantic analysis: Type expression, type systems, symbol tables and type checking. Intermediate code generation: Intermediate languages. Intermediate representation-Three address code and quadruples. Syntax-directed translation of declarations, assignments statements, conditional constructs and looping constructs.
- Runtime Environments: Storage organization, activation records. Introduction to machine code generation and code optimizations.

Course Objectives

- The main outcome of the course ’Compiler Design’ is to make the students capable of applying the principles, algorithm, and data structures involved in the design of and compilers.
- Students should be able to design a lexical analyser in lex according to the specification. They should be able to design a parser in yacc when the specification is mentioned. They should be able to construct a compiler according to the rules and constrains given.
- ## Course Outcomes
  - To introduce the major concept areas of language translation and compiler design.
  - To enrich the knowledge in various phases of compiler and the design issues involved in compilation, code optimization techniques, machine code generation, and use of symbol table.

Course evaluation

- The course evaluation will be done in the following way:

Evaluation parameter	Weightage of Marks %
Mid Semester I	20
Assignments	20
Continuous Evaluation	20
End Semester	40

Administration of Exams: Exams will be in offline mode. The duration of the exam, will be 90(Mid)/120(end sem.) minutes respectively.

Continuous Evaluation

- The continuous evaluation scheme will consist of the following

<table>
<tr><th>Evaluation parameter</th><th>Weightage of Marks</th></tr>
<tr><td>Quiz</td><td rowspan="2">20</td></tr>
<tr><td>Lab Assignment</td></tr>
</table>

Each course should have:

- Minimum of One Quiz
- Minimum of Two Assignments

Why Should We Study Compiler Design?

- *Compilers are everywhere!*

- Many applications for compiler technology
    - Parsers for HTML in web browser
    - Interpreters for javascript/flash
    - Machine code generation for high level languages
    - Software testing
    - Program optimization
    - Malicious code detection
    - Design of new computer architectures
        - Compiler-in-the-loop hardware development
    - Hardware synthesis: VHDL to RTL translation
    - Compiled simulation
        - Used to simulate designs written in VHDL
        - No interpretation of design, hence faster

Why Should We Study Compiler Design? Contd.,

1. Parsers for HTML in Web Browsers

- **What:** Browsers parse HTML files to display content (text, images, links).
- **How it uses compiler tech:** When you open a webpage in Chrome, the browser **parses the HTML**, similar to how a compiler parses source code.

2. Software Testing

- **What:** Compilers assist in **static analysis** and **testing** tools.
- **How it uses compiler tech:** Abstract syntax trees (ASTs) are used to analyze code paths, detect unreachable code or potential bugs.

3. Program Optimization

- **What:** Making code run faster or use less memory.
- **How it uses compiler tech:** Compilers apply **loop unrolling**, **inlining**, **dead-code elimination**, etc.

Contd.,

6. Malicious Code Detection

- **What:** Analyzing code to detect trojans, backdoors, or malware patterns.
- **How it uses compiler tech:** Static code analyzers parse source or binary code to find suspicious patterns.

7. Design of New Computer Architectures

- **What:** Testing if new CPU instructions or pipelines can be supported by compilers.
- **How it uses compiler tech:** Simulate code generation and optimization strategies on imaginary hardware.

11. Used to Simulate Designs Written in VHDL (No Interpretation, Hence Faster)

- **What:** Simulations are more efficient when the VHDL is compiled, not interpreted.
- **How it uses compiler tech:** Compilation avoids repeated parsing during simulation.

About the Nature of Compiler Algorithms

- Draws results from mathematical logic, lattice theory, linear algebra, probability, etc.
    - type checking, static analysis, dependence analysis and loop parallelization, cache analysis, etc.

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

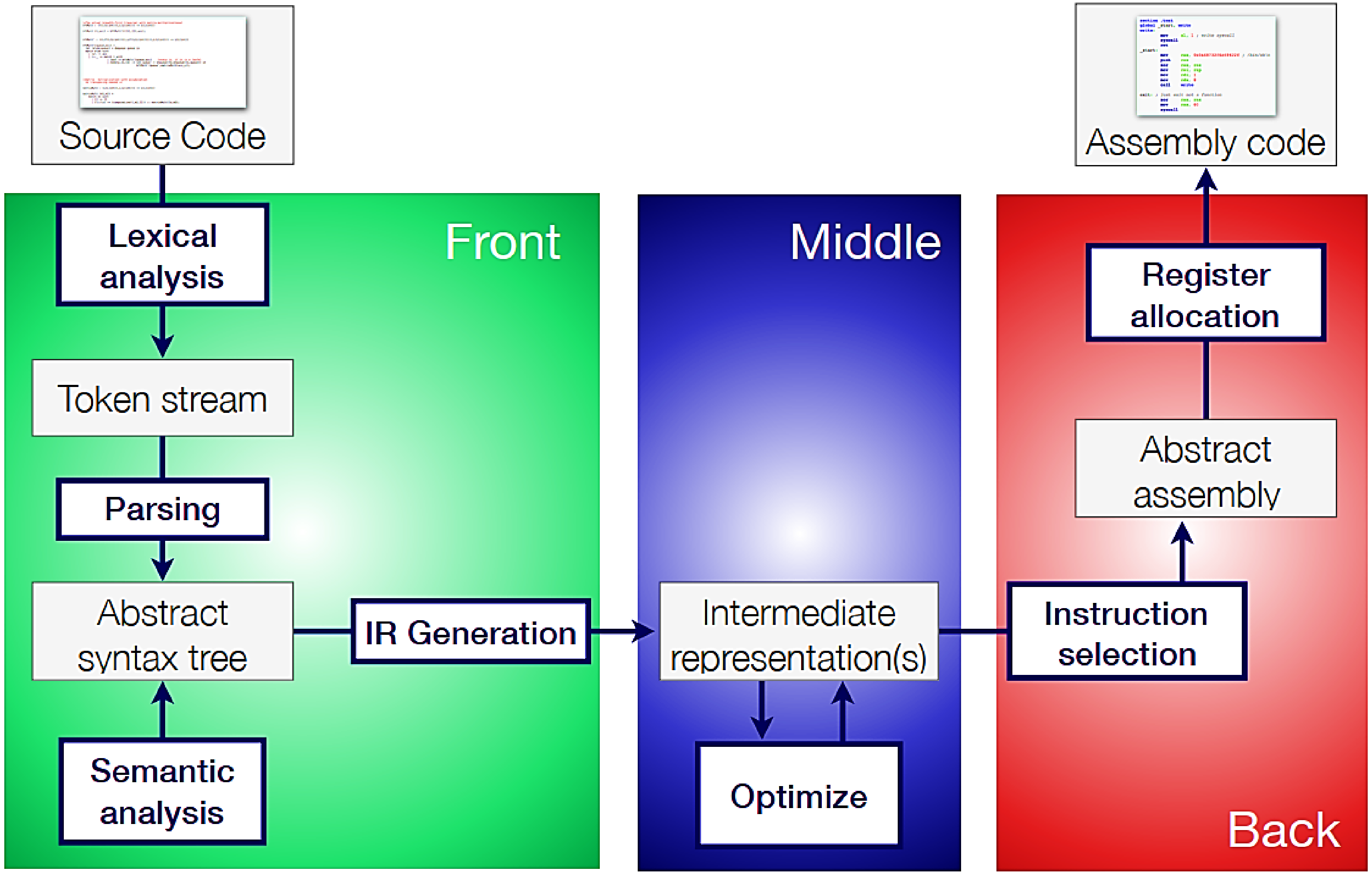
Application Area	Description	Compiler Techniques Involved	Example Tool
Assembler Implementation	Converts assembly to machine code	Lexical analysis, code generation	NASM
Online Text Search (GREP, AWK)	Pattern-based text filtering	Regular expressions, pattern matching	GREP, AWK
Website Filtering	Blocks/filters content using rules	Lexical/pattern matching	Pi-hole
Command Language Interpreters	Executes CLI commands	Tokenization, interpretation	Bash, Zsh
Scripting Language Interpretation	Parses and runs scripts dynamically	Parsing, interpretation/JIT	Python, Perl
XML Parsing	Builds tree structures from XML	Tree construction from structured input	DOM Parser
Database Query Interpretation	Analyzes and runs SQL queries	SQL parsing, semantic analysis, planning	PostgreSQL

Compiler Phases

keywords, identifiers, symbols

Builds a syntax tree from tokens, checking grammar rules.

Checks **types**, **scope**, and **logical correctness**.



Assigns variables to physical CPU registers.

Maps high-level operations to machine-level

Performs **optimizations** on an **intermediate representation (IR)**

Generates **target code** from optimized IR

Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

Note the

- Capital “T” (start of sentence symbol)
- Blank “ ” (word separator)
- Period “.” (end of sentence symbol)

More Lexical Analysis

- Lexical analysis is not small.
- Consider:

ist his ase nte nce

- Plus, programming languages are typically more cryptic than English:

```
*p->f ++ = -.12345e-5
```

And More Lexical Analysis

- Lexical analyzer divides program text into "words" or "tokens"

  if (x == y) then z = 1; else z = 2;

- Units:

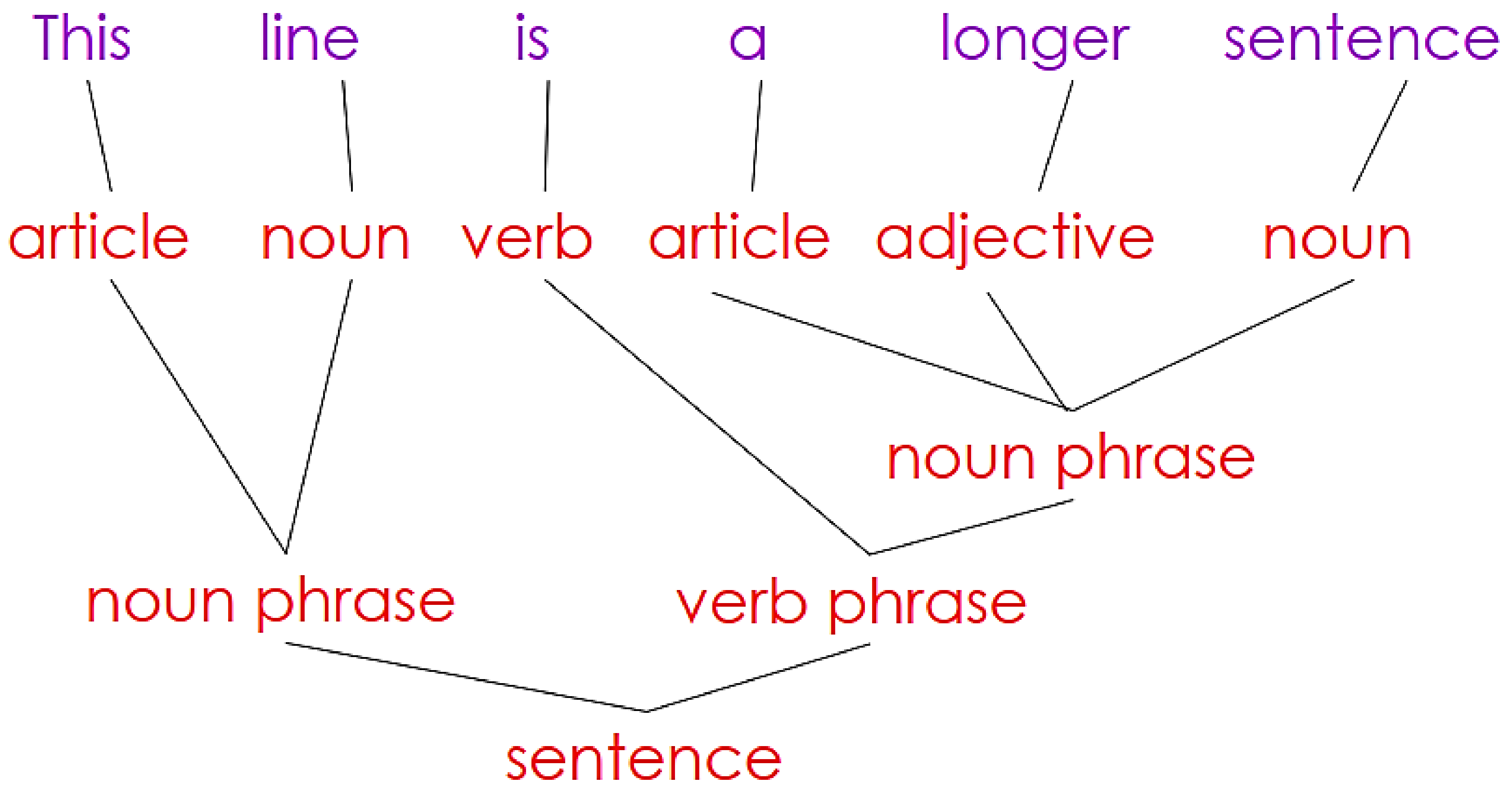
  if, (, x, ==, y,), then, z, =, 1, ;, else, z, =, 2, ;

Parsing

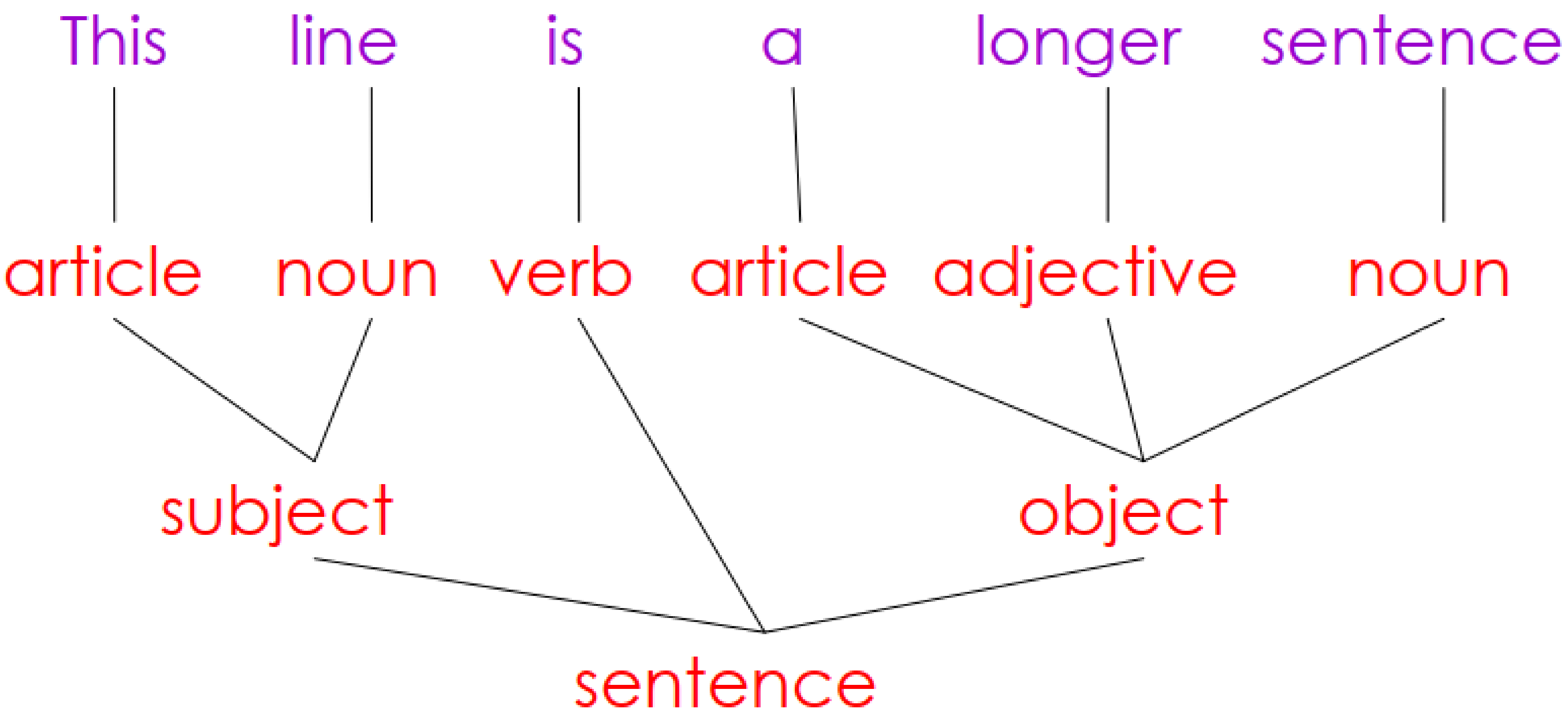
- Once words are understood, the next step is to understand the sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence (1)



Diagramming a Sentence (2)



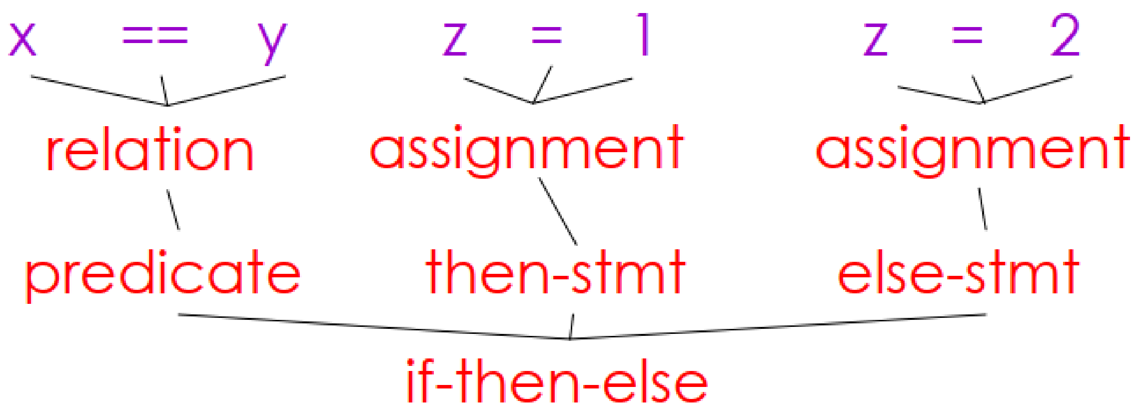
Parsing Programs

- Parsing program expressions is the same

- Consider:

  If (x == y) then z = 1; else z = 2;

- Diagrammed



Semantic Analysis

- Once sentence structure is understood, we can try to understand its "**meaning**"
  - But meaning is too hard for compilers
- Most compilers perform limited analysis to catch inconsistencies
- Some optimizing compilers do more analysis to improve the performance of the program

Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does “his” refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis in Programming Languages

- Programming languages define strict rules to avoid such doubts

```
{
    int Jack = 3;
    {
        int Jack = 4;
        cout << Jack;
    }
}
```

This C++ code prints “4”; the inner definition is used

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings
- Example:

  Arnold left her homework at home.

- A “type mismatch” between her and Arnold;
- we know they are different people
  - Presumably Arnold is male

Optimization

- Automatically modify programs so that they
  - Run faster
  - Use less memory/power
  - In general, conserve some resource more economically

Optimization Example

- X = Y * 0 is the same as X = 0

NO!

Valid for integers, but not for floating point numbers

Code Generation

- Produces assembly code (usually)

- A translation into another language
  - Similar to human translation

Intermediate Languages (IL’s)

- Many compilers perform translations between successive intermediate forms
  - All but first and last are intermediate languages internal to the compiler
  - Typically, there is **one IL**
- IL’s generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly
- IL’s are useful because lower levels expose features hidden by higher levels
  - registers
  - memory/frame layout, etc.

Issues

- Compiling is almost this simple, but there are many pitfalls
  - Example: How are erroneous programs handled?

- Language design has big impact on compiler
  - – Determines what is easy and hard to compile
  - – Note: many trade-offs in language design

Compilers Today

- The overall structure of almost every compiler stick to to our outline
- The proportions have changed since FORTRAN
  - Early:
    - lexical analysis, parsing most complex, expensive
  - Today:
    - semantic analysis and optimization dominate all other phases; lexing and parsing are well-understood and cheap

Current Trends

- **Language design**
  - Many new special-purpose languages
  - Popular languages to stay

- **Compilers**
  - More needed and more complex
  - Driven by increasing gap between
    - new languages
    - new architectures
  - Esteemed and healthy area

Why so many Programming Languages?

- Application domains have individual (and sometimes conflicting) needs
- Examples:
  - **Scientific computing**: High performance
  - **Business**: report generation
  - **Artificial intelligence**: symbolic computation
  - **Systems programming**: efficient low-level access
  - Other special purpose languages...

Language Design

- No universally accepted metrics for design
- "A good language is one people use"
- NO !
  - Is COBOL the best language?
- Good language design is hard

Language Evaluation Criteria

Characteristic	Criteria		
	Readability	Writeability	Reliability
Simplicity	YES	YES	YES
Data types	YES	YES	YES
Syntax design	YES	YES	YES
Abstraction		YES	YES
Expressivity		YES	YES
Type checking			YES
Exceptions			YES