How are Languages Implemented?

- Two major strategies:
  - Interpreters (older, less studied)
  - Compilers (newer, much more studied)

- Interpreters run programs "as is"
  - Little or no preprocessing

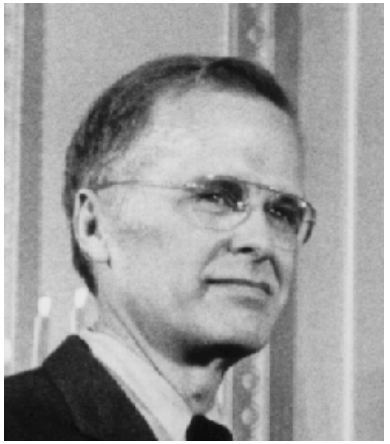
- Compilers do extensive preprocessing

Language Implementations

- Today, batch compilation systems dominate
  - gcc, clang, ...
- Some languages are primarily interpreted
  - Java bytecode compiler (javac)
  - Scripting languages (perl, python, javascript, ruby)
- Some languages (e.g. Lisp) provide both
  - Interpreter for development
  - Compiler for production

(Short) History of High-Level Languages

- 1953 IBM develops the 701
- Till then, all programming is done in assembly
- Problem: Software costs exceeded hardware costs!
- John Backus: "Speedcoding"
  - An interpreter
  - Ran 10-20 times slower than hand-written assembly



FORTRAN I

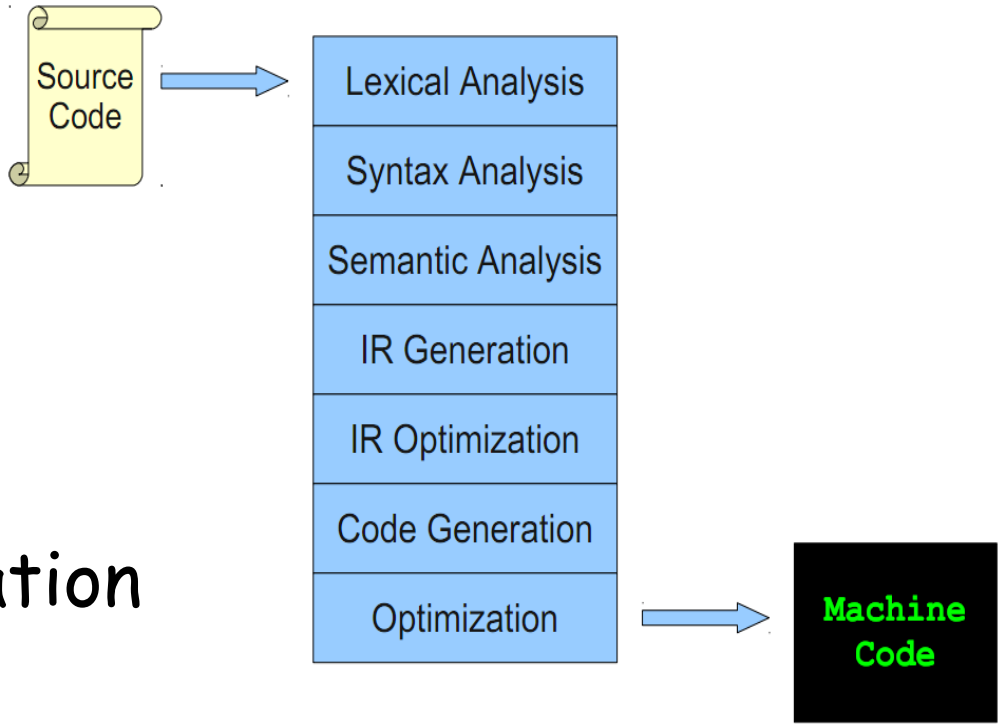
- 1954 IBM develops the 704
- John Backus
  - Idea: translate high-level code to assembly
  - Many thought this impossible
    - Had already failed in other projects
- 1954-7 FORTRAN I project
- By 1958, >50% of all software is in FORTRAN
- Cut development time dramatically
  - (2 weeks → 2 hours)

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

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 phases can be understood by analogy to how humans comprehend natural languages (e.g. English).

First Step: Lexical Analysis

- 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 trivial. 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:

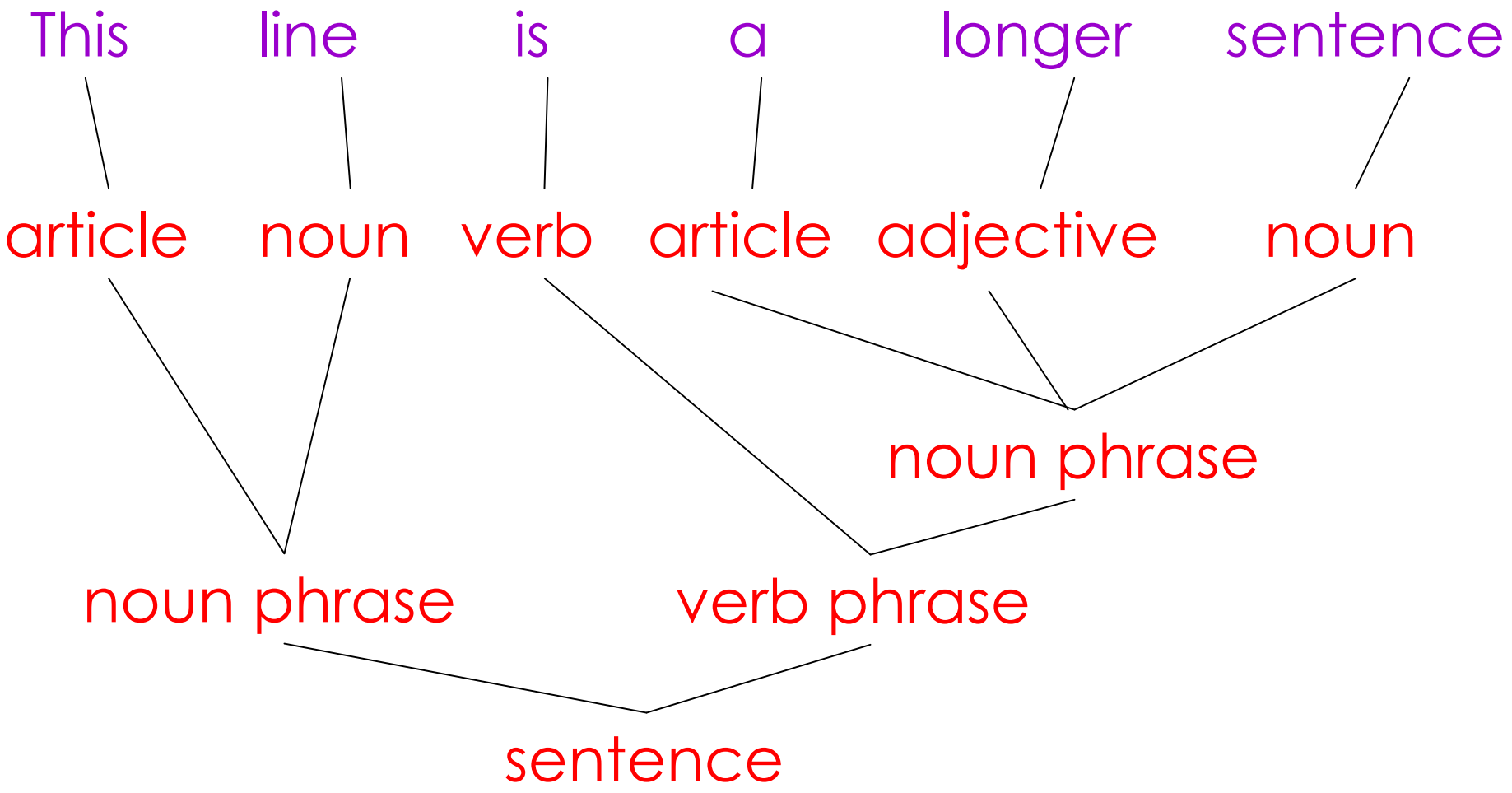
if, (, x, ==, y,), then, z, =, 1, ;, else, z, =, 2, ;

Second Step: Syntax Analysis (Parsing)

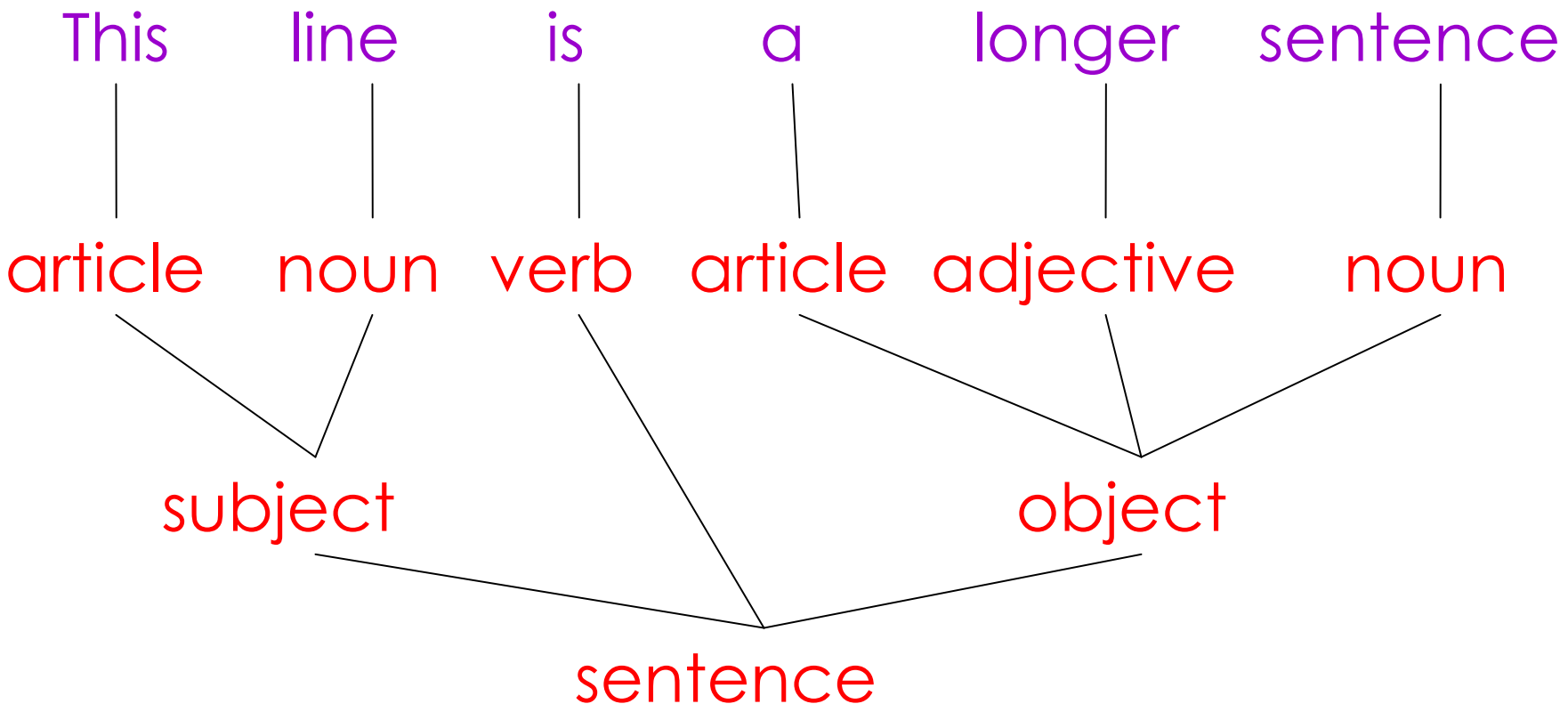
- Once words are identified, the next step is to understand the sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence (1)



Diagramming a Sentence (2)

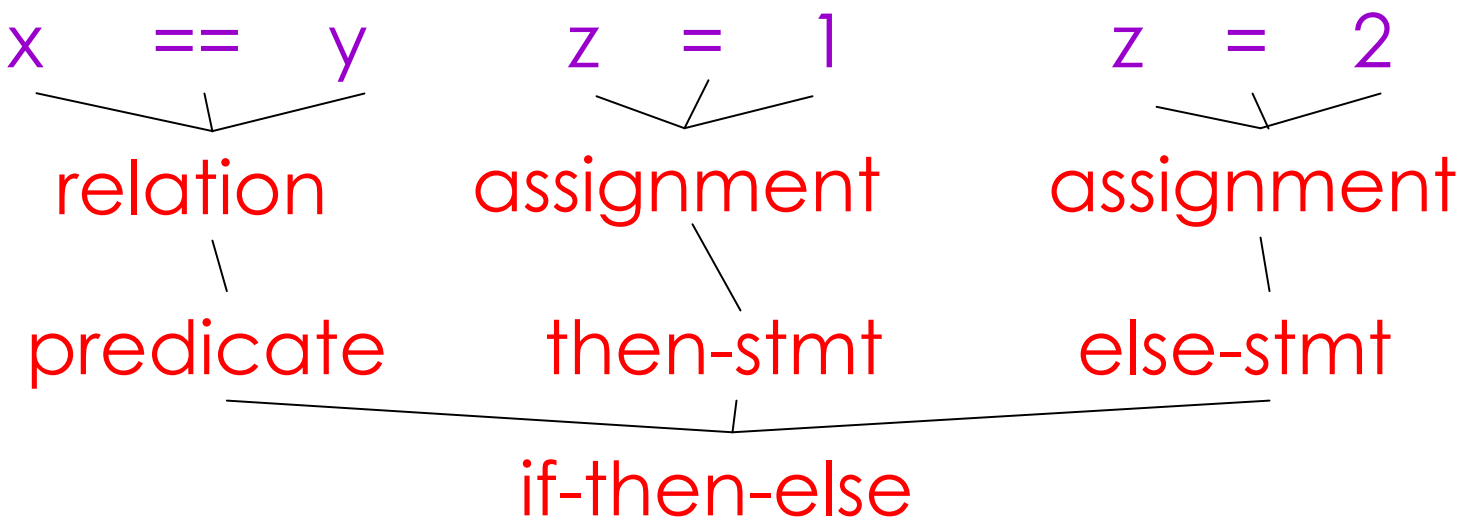


Parsing Programs

- Parsing program expressions is the same
- Consider:

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

- Diagrammed:



Third Step: Semantic Analysis

- Once the 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 ambiguities

- This C++ code prints 42; the inner definition is used

```
{
   int Jack = 17;
   {
      int Jack = 42;
      cout << Jack;
   }
}
```

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

- No strong counterpart in English, but akin to editing
- Automatically modify programs so that they
  - Run faster
    - avoid some source code redundancy
    - exploit the underlying hardware more effectively
  - Use less memory/cache/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
  - Analogous to human translation

Intermediate Languages

- 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

- Intermediate languages generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly

Intermediate Languages (Cont.)

- IL's are useful because lower levels expose features hidden by higher levels
  - registers
  - memory/frame layout
  - etc.

- But lower levels obscure high-level meaning

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
  - Course theme: many trade-offs in language design

Compilers Today

- The overall structure of almost every compiler adheres to our outline

- The proportions have changed since FORTRAN
  - Early:
    - lexical analysis, parsing most complex, expensive
  - Today:
    - lexical analysis and parsing are well-understood and cheap
    - semantic analysis and optimization dominate
    - focus on concurrency/parallelism and interactions with the memory model of the underlying platform
    - optimization for code size and energy consumption

Current Trends in Compilation

- Compilation for speed is less interesting. However, there are exceptions:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures, GPUs)

- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting data races
  - security properties
  - …

Why study Compilers?

- Increase your knowledge of common programming constructs and their properties
- Improve your understanding of program execution
- Increase your ability to learn new languages

- Learn how languages are implemented
- Learn new (programming) techniques
- See many basic CS concepts at work