Concepts of Programming Languages

The Evolution of Major Programming Languages

Objectives

- Learn about the language of a computer
- Learn about the evolution of programming languages
- Examine high-level programming languages
- Discover what a compiler is and what it does

An Overview of the History of Computers

- 1950s: Very large devices available to a few
- 1960s: Large corporations owned computers
- 1970s: Computers get smaller and cheaper
- 1990s: Computers get cheaper and faster and are found in most homes

Elements of a Computer System

- A computer has 2 components
  - Hardware
  - Software

Hardware Components of a Computer

- Central Processing Unit (CPU)
- Main Memory

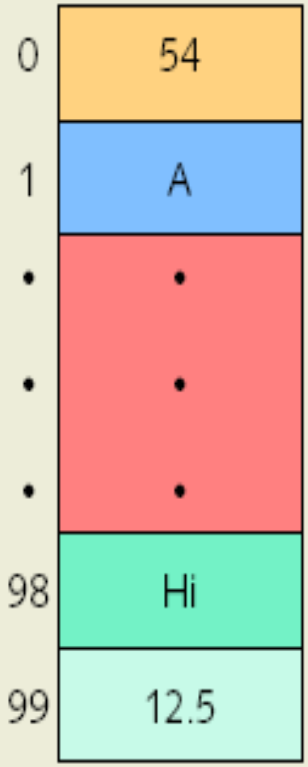
Central Processing Unit

- Control Unit (CU)
- Arithmetic Logic Unit (ALU)
- Program Counter (PC)
- Instruction Register (IR)
- Accumulator (ACC)

Main Memory

- Ordered sequence of cells (memory cells)
- Directly connected to CPU
- All programs must be brought into main memory before execution
- When power is turned off, everything in main memory is lost

Main Memory with 100 Storage Cells



Secondary Storage

- Provides permanent storage for information
- Examples of secondary storage:
  - Hard Disks
  - Floppy Disks
  - ZIP Disks
  - CD-ROMs
  - Tapes

Input Devices

- Definition: devices that feed data and computer programs into computers

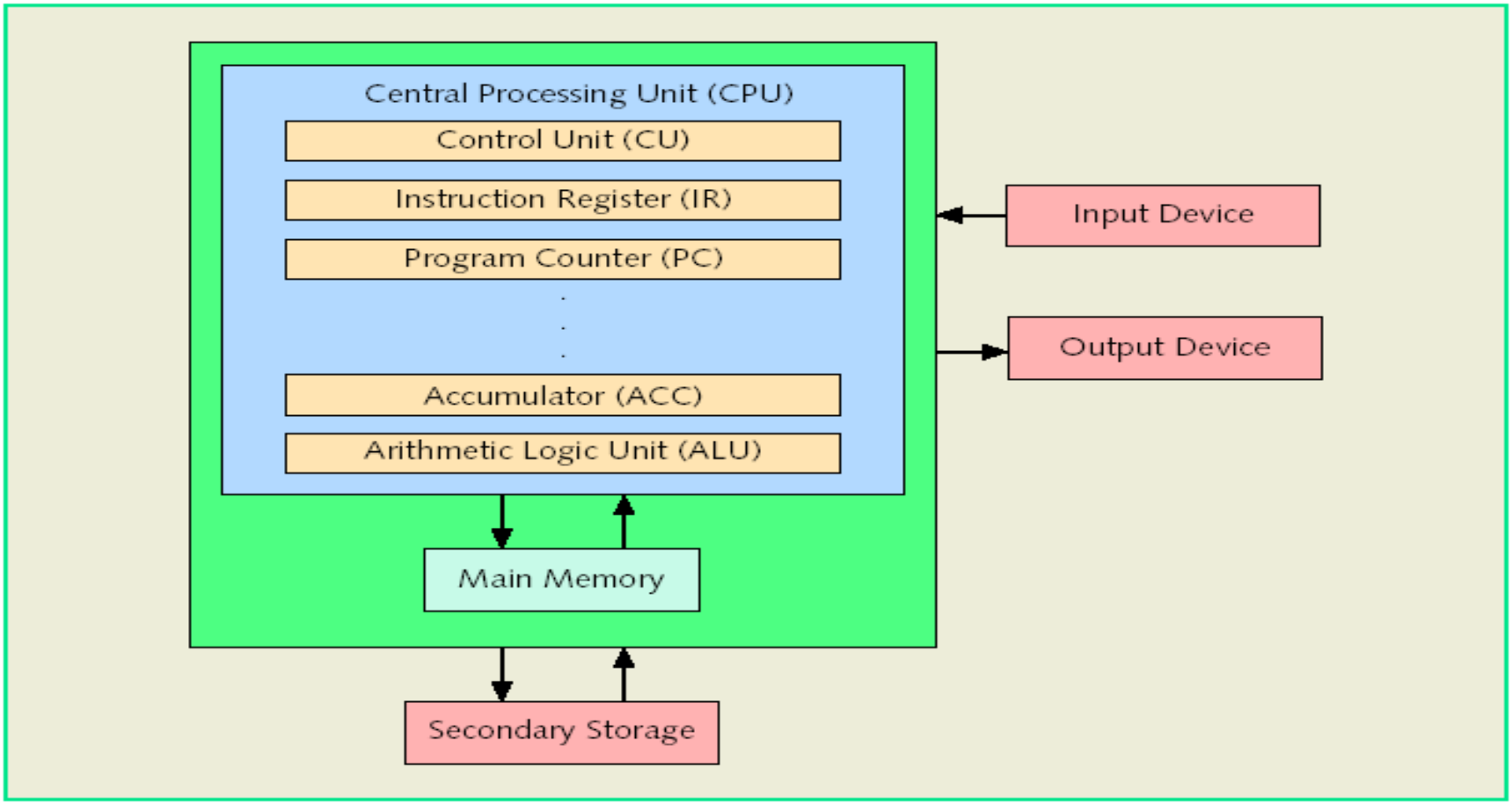
- Examples:
  - Keyboard
  - Mouse
  - Secondary Storage

Output Devices

- Definition: devices that the computer uses to display results

- Examples:
  - Printer
  - Monitor
  - Secondary Storage

Hardware Components of a Computer



Software

- Software consists of programs written to perform specific tasks

- Two types of programs
  - System Programs
  - Application Programs

System Programs

- System programs control the computer
- The operating system is first to load when you turn on a computer

Operating System (OS)

- OS monitors overall activity of the computer and provides services

- Example services:
  - memory management
  - input/output
  - activities
  - storage management

Application Programs

- Written using programming languages
- Perform a specific task
- Run by the OS

- Example programs:
  - Word Processors
  - Spreadsheets
  - Games

Language of a Computer

- Machine language: the most basic language of a computer
- A sequence of 0s and 1s
- Every computer directly understands its own machine language
- A bit is a binary digit, 0 or 1
- A byte is a sequence of eight bits

Evolution of Programming Languages

- Early computers programmed in machine language
- Assembly languages were developed to make programmer's job easier
- In assembly language, an instruction is an easy-to-remember form called a mnemonic
- **Assembler:** translates assembly language instructions into machine language

Instructions in Assembly and Machine Language

Assembly Language	Machine Language
LOAD	100100
STOR	100010
MULT	100110
ADD	100101
SUB	100011

Evolution of Programming Languages

- High-level languages make programming easier
- Closer to spoken languages
- Examples:
  - Basic
  - FORTRAN
  - COBOL
  - C/C++
  - Java

Evolution of Programming Languages

To run a Java program:

1. Java instructions need to be translated into an intermediate language called bytecode
2. Then the bytecode is interpreted into a particular machine language

Evolution of Programming Languages

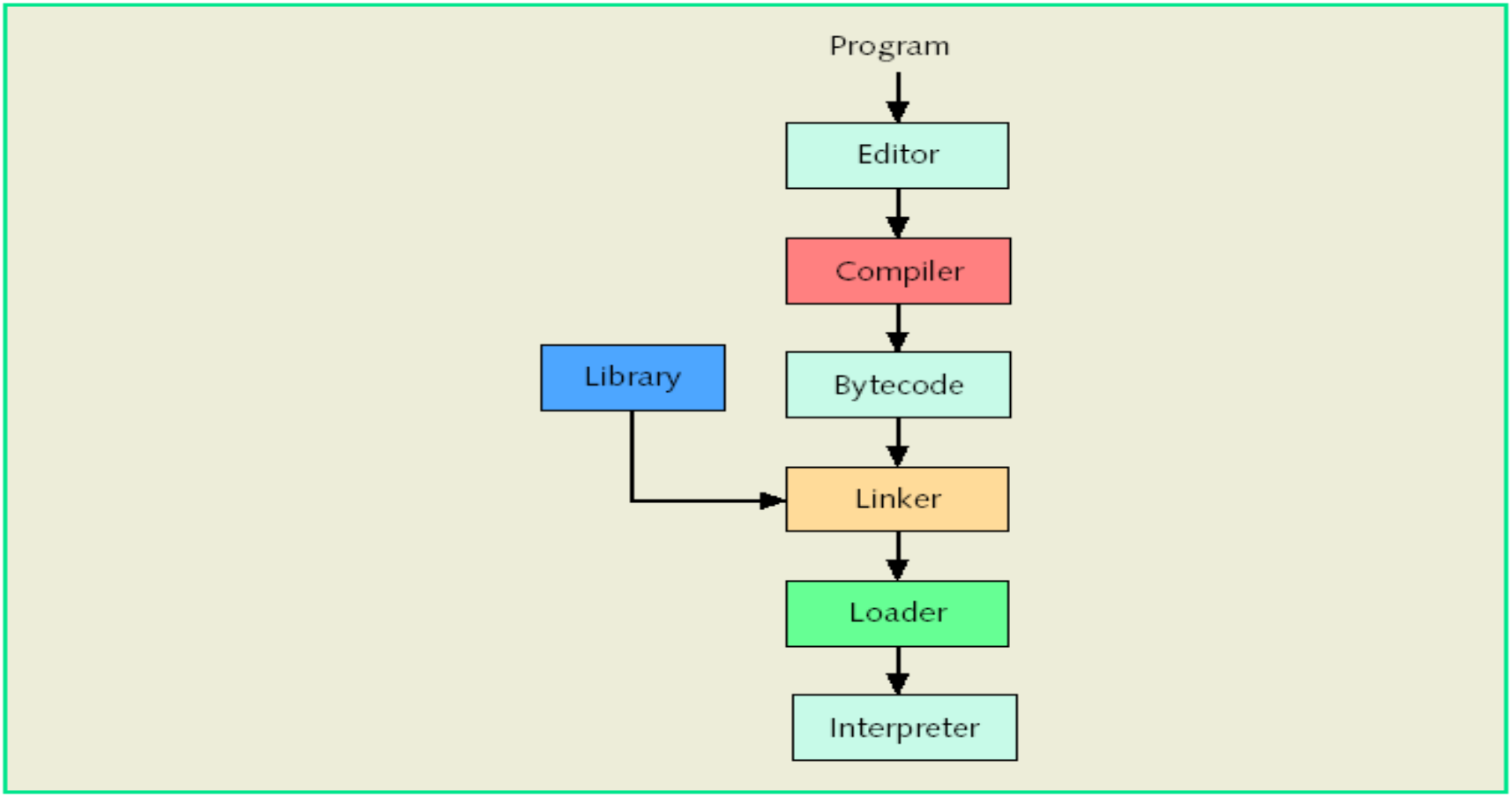
- **Compiler:** A program that translates a program written in a high-level language into the equivalent machine language. (In the case of Java, this machine language is the bytecode.)

- **Java Virtual Machine (JVM)** - hypothetical computer developed to make Java programs machine independent

Processing a Java Program

- Two types of Java programs: applications and applets
- **Source program:** Written in a high-level language
- **Linker:** Combines bytecode with other programs provided by the SDK and creates executable code
- **Loader:** transfers executable code into main memory
- **Interpreter:** reads and translates each bytecode instruction into machine language and then executes it

Processing a Java Program



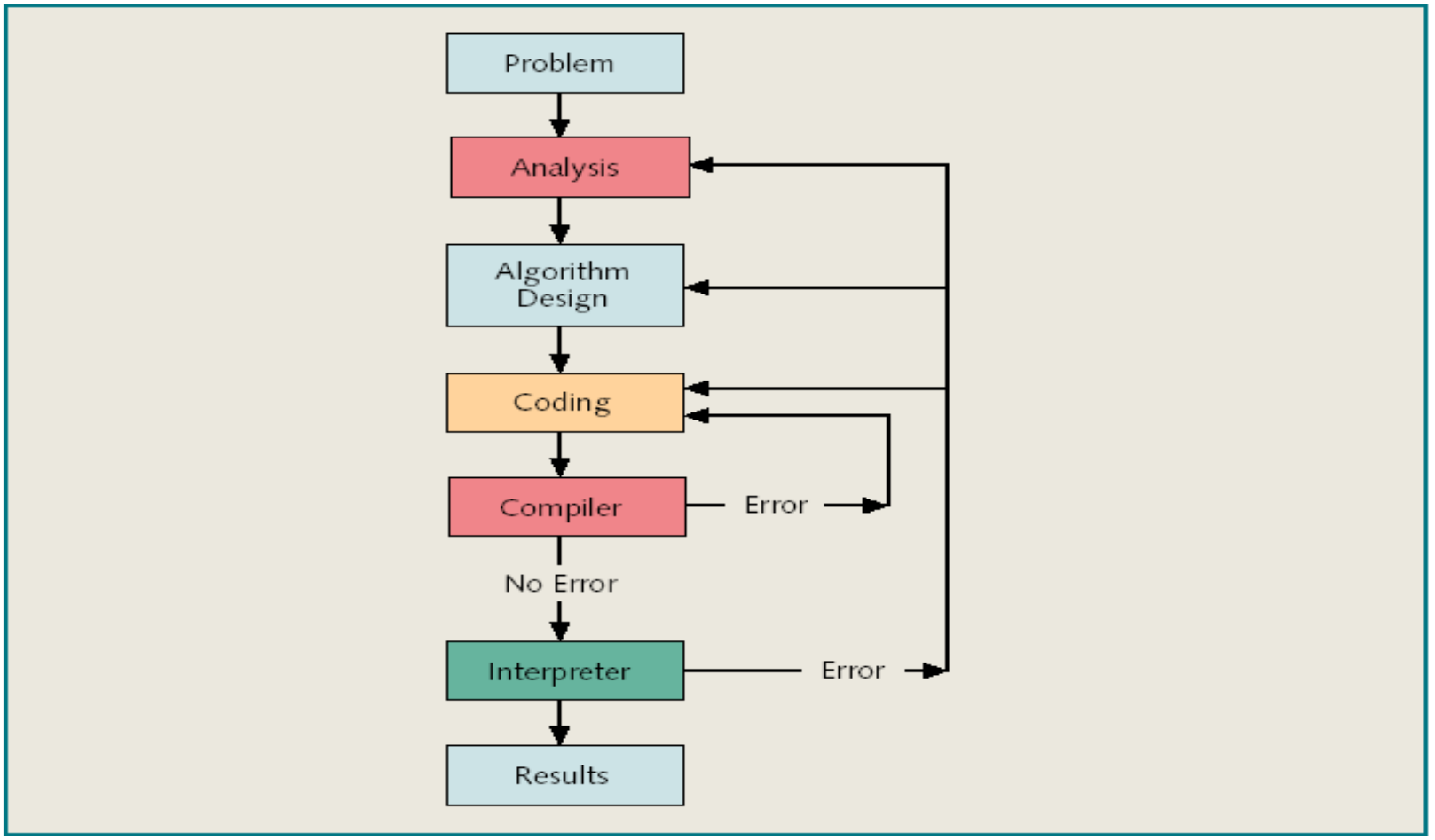
Problem-Analysis-Coding-Execution Cycle

- **Algorithm:** A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time

Problem-Solving Process

1. **Analyze** the problem: outline solution requirements and design an algorithm
2. **Implement** the algorithm in a programming language (Java) and verify that the algorithm works
3. **Maintain** the program: use and modify if the problem domain changes

Problem-Analysis-Coding-Execution Cycle



Programming Methodologies

- Two basic approaches to programming design:
  - Structured design
  - Object-oriented design

Structured Design

1. A problem is divided into smaller subproblems
2. Each subproblem is solved
3. The solutions of all subproblems are then combined to solve the problem

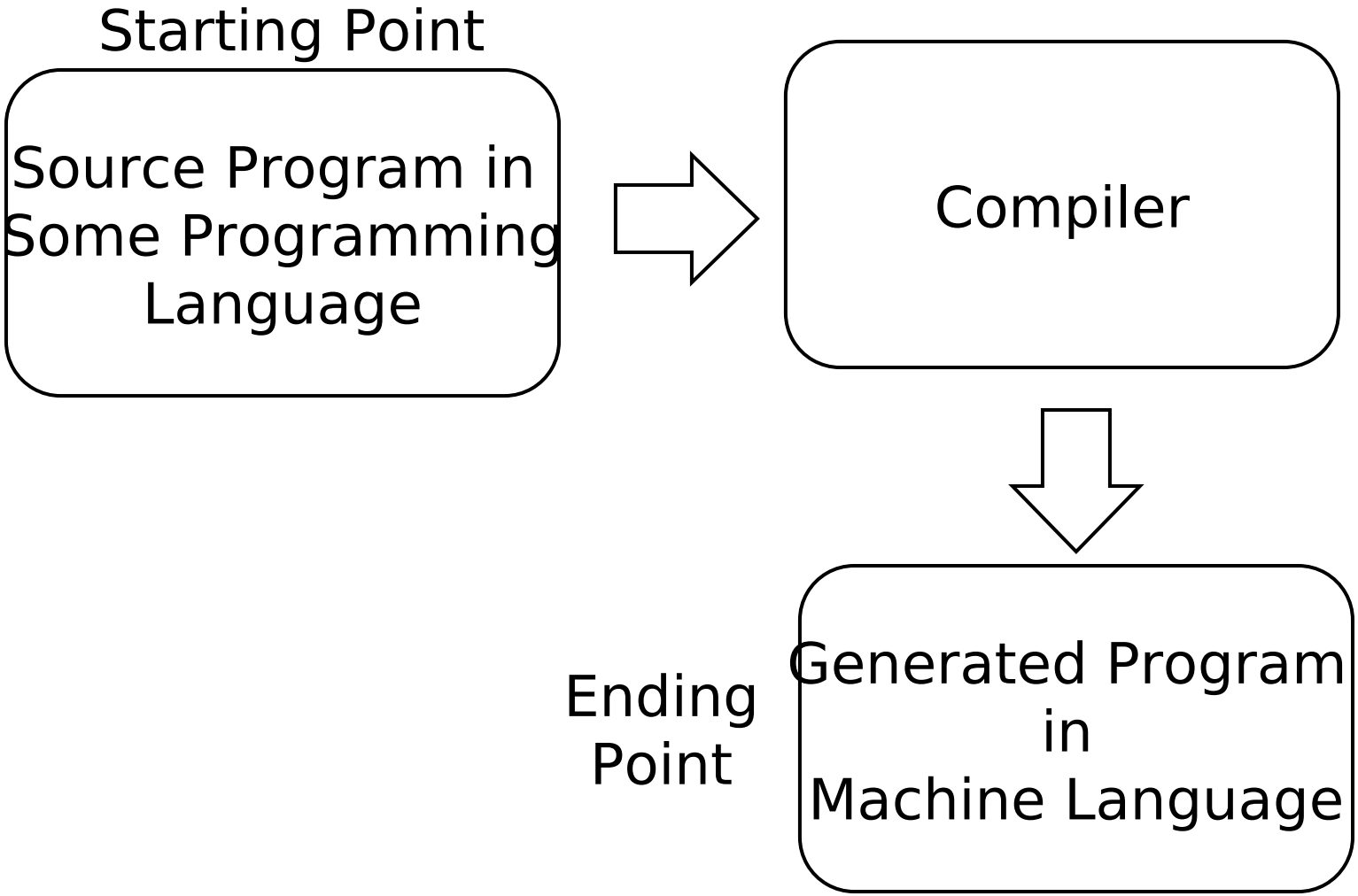
Object-Oriented Design (OOD)

- In OOD, a program is a collection of interacting objects
- An object consists of data and operations
- Steps in OOD:
  1. Identify objects
  2. Form the basis of the solution
  3. Determine how these objects interact

Programming Language Dilemma

- How to instruct computer what to do?
  - Need a program that computer can execute
  - Must be written in machine language
- Unproductive to code in machine language
- Design a higher level language
- Implement higher level language
  - Alternative 1: Interpreter
  - Alternative 2: Compiler

Compilation As Translation



Starting Point

Programming language (Java, C, C++)

- State
  - Variables,
  - Structures,
  - Arrays
- Computation
  - Expressions (arithmetic, logical, etc.)
  - Assignment statements
  - Control flow (conditionals, loops)
  - Procedures

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

- Batch compilation systems dominate
  - E.g., gcc
- Some languages are primarily interpreted
  - E.g., Java bytecode
- Some environments (Lisp) provide both
  - Interpreter for development
  - Compiler for production

The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

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

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 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:

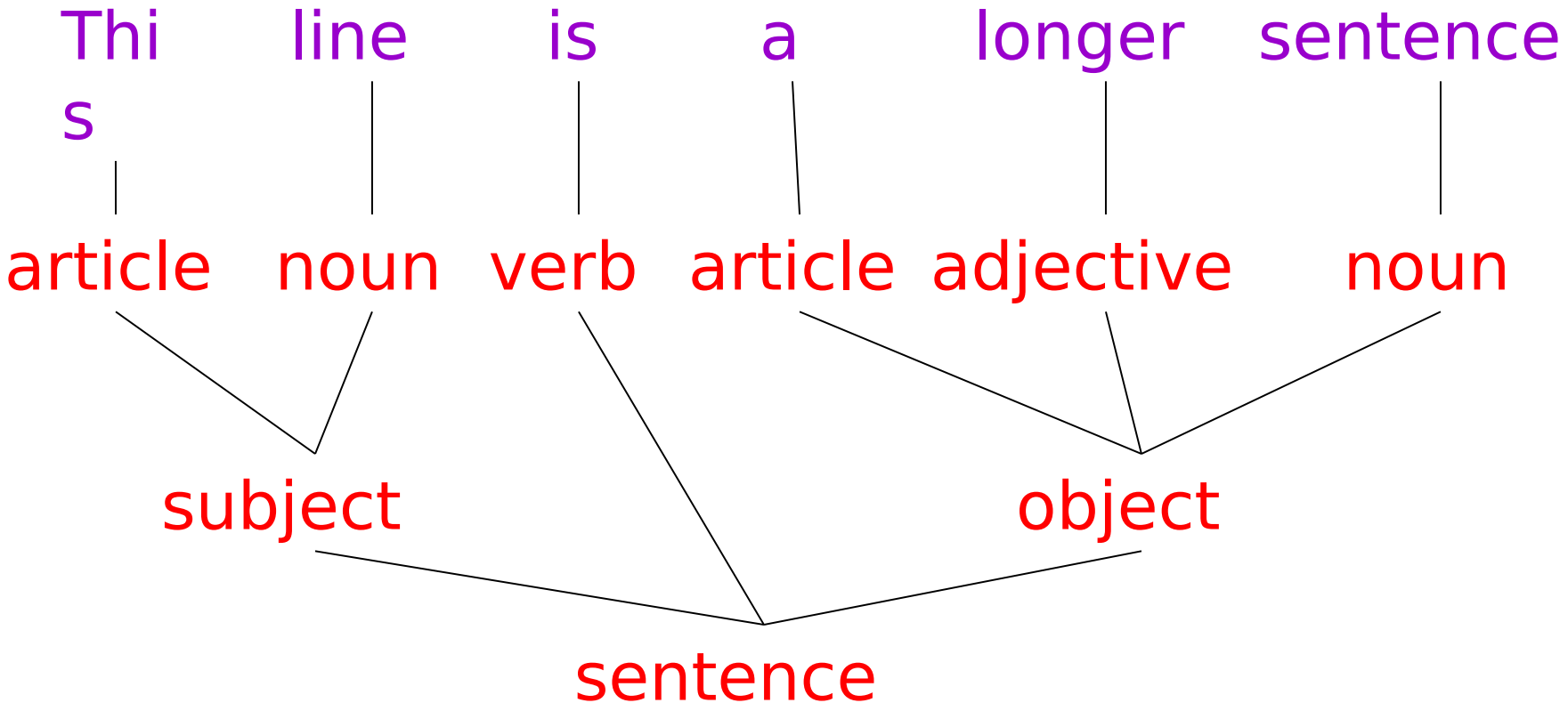
  if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;

Parsing

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

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

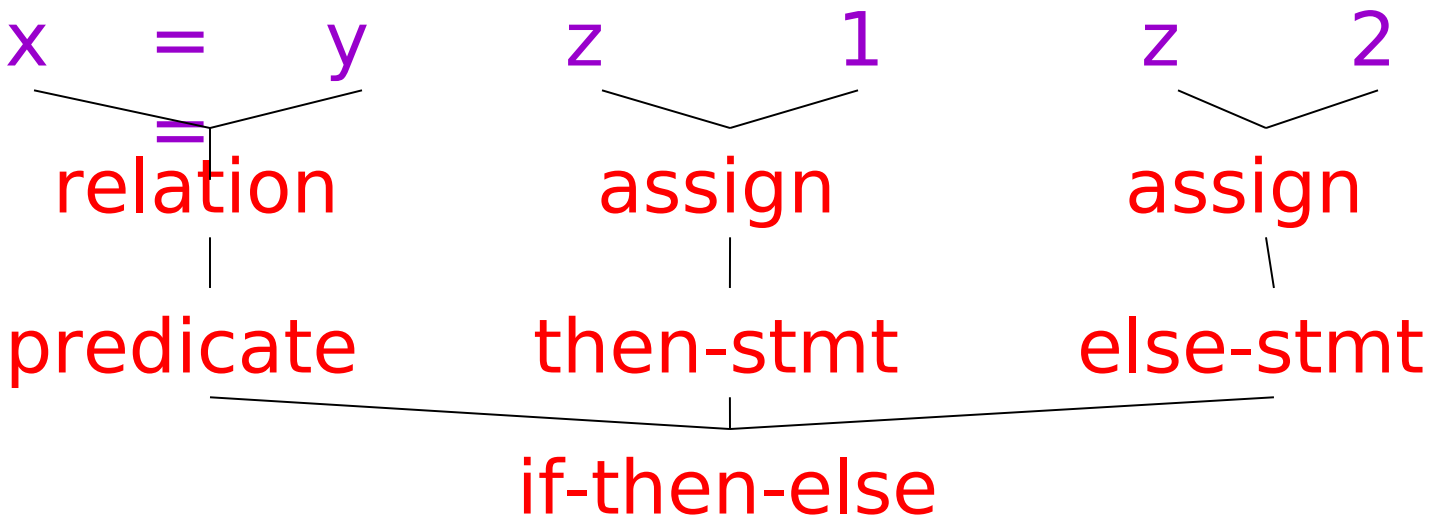


Parsing Programs

- 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 "meaning"
  - But meaning is too hard for compilers
- Compilers perform limited analysis to catch inconsistencies
- Some 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

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints "4"; the inner definition is used

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

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings

- Example:

  Jack left her homework at home.

- A "type mismatch" between her and Jack; we know they are different people
  - Presumably Jack is male

Examples of Semantic Checks

- Variables defined before used
- Variables defined once
- Type compatibility
- Correct arguments to functions
- Constants are not modified
- Inheritance hierarchy has no cycles

Optimization

- No strong counterpart in English

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

Optimization Example

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

NO!

Valid for integers, but not for floating point numbers

Examples of common optimizations

- Dead code elimination
- Evaluating repeated expressions only once
- Replace expressions by simpler equivalent expressions
- Evaluate expressions at compile time
- Move constant expressions out of loops

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 1 Intermediate Language

- IL's 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 layout

- 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: lexing, parsing most complex, expensive

  - Today: optimization dominates all other phases, lexing and parsing are cheap

Trends in Compilation

- Compilation for speed is less interesting. But:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures)

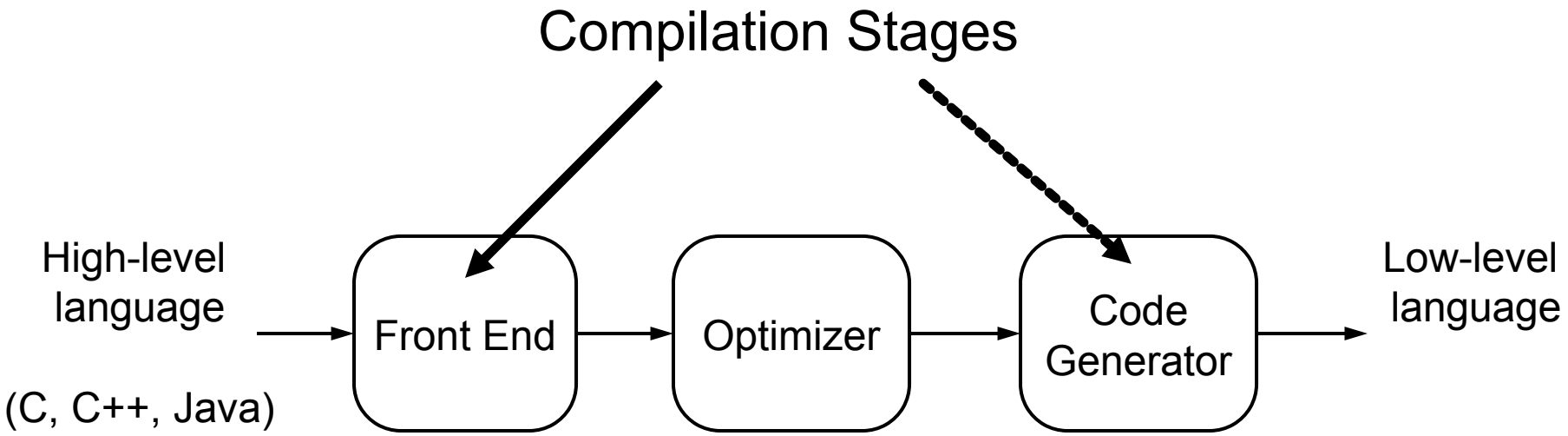
- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting concurrency errors (data races)

Compilers and Optimization

What makes a compiler good?

What makes a compiler different?

What's in an optimizing compiler?



What are compiler optimizations?

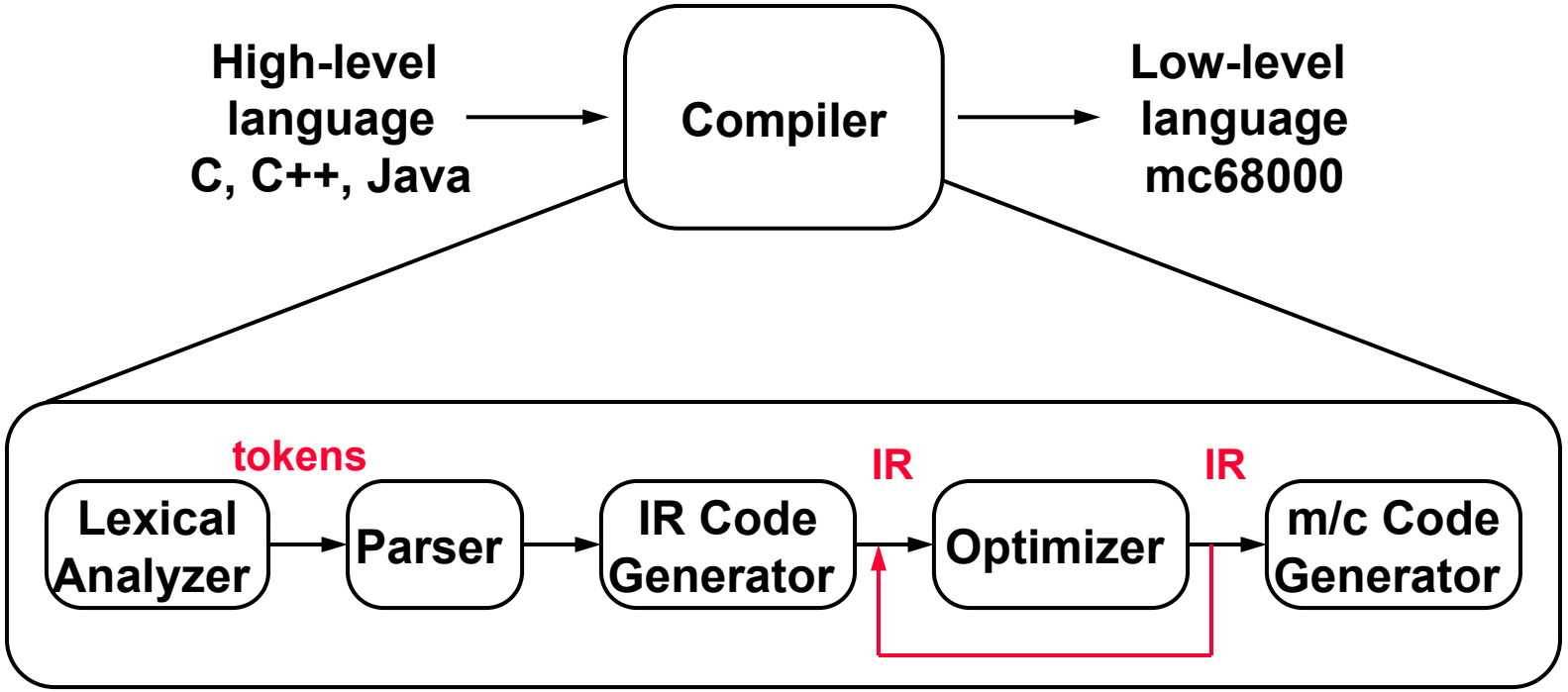
Optimization: the transformation of a program P into a program P', that has the same input/output behavior, but is somehow "better".

- "better" means:
  - faster
  - or smaller
  - or uses less power
  - or whatever you care about
- P' is not optimal, may even be worse than P

An optimizations must:

- Preserve correctness
  - the speed of an incorrect program is irrelevant
- On average improve performance
  - P' is not optimal, but it should usually be better
- Be "worth the effort"
  - 1 person-year of work, 2x increase in compilation time, a 0.1% improvement in speed?
  - Find the bottlenecks
  - 90/10 rule: 90% of the gain for 10% of the work

Compiler Phases (Passes)



IR: Intermediate Representation

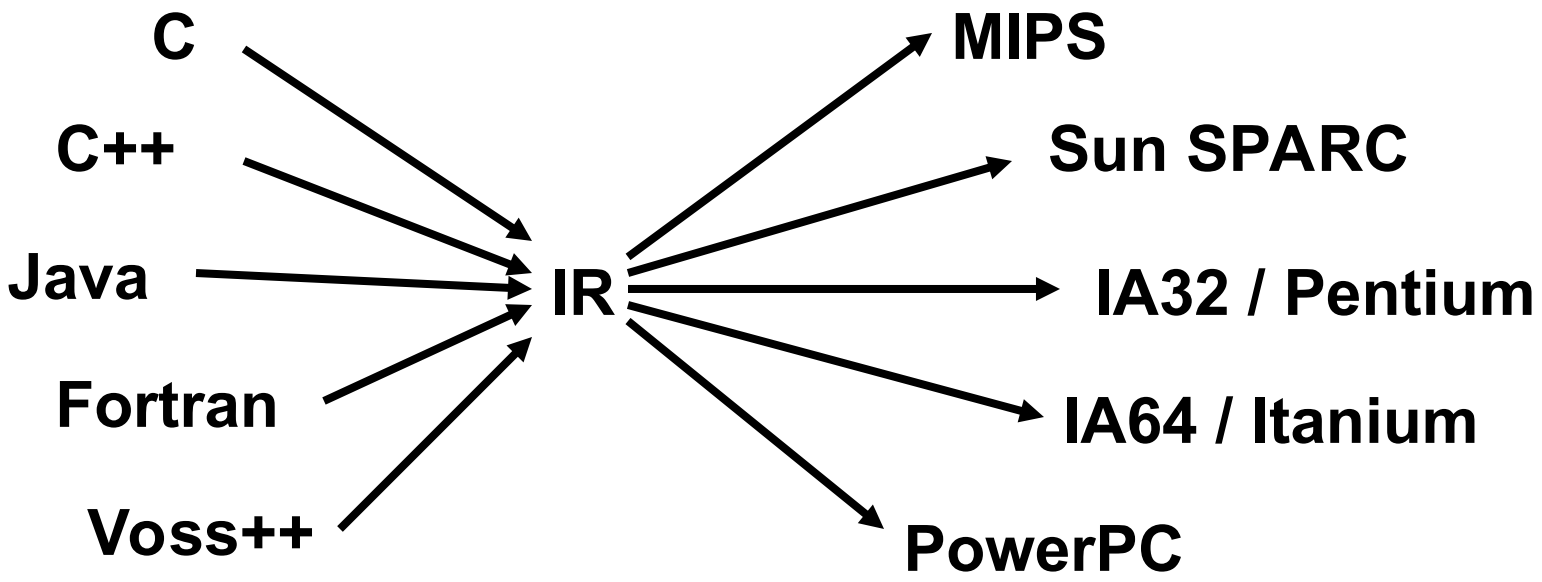
Lexers & Parsers

- ***The lexer*** identifies tokens in a program
- ***The parser*** identifies grammatical phrases, or constructs in the program
- There are freely available lexer and parser generators…
- The parser usually constructs some intermediate form of the program as output

Intermediate Representation (IR)

- The representation or language on which the compiler performs it's optimizations
- There are as many IRs as compiler suites
- Some IRs are better for some optimizations
  - different information is maintained
  - easier to find certain types of information

Why Use an IR?



- Good Software Engineering
  - Portability
  - Reuse

A Simple Compilation Example

Writing C Programs

- A programmer uses a **text editor** to create or modify files containing C code.
- Code is also known as **source code**.
- A file containing source code is called a **source file**.
- After a C source file has been created, the programmer must **invoke the C compiler** before the program can be **executed** (**run**).

Invoking the gcc Compiler

At the prompt, type

gcc -ansi -Wall *pgm.c*

where *pgm.c* is the C program source file.

- **-ansi** is a **compiler option** that tells the compiler to adhere to the **ANSI C standard**.
- **-Wall** is an option to turn on all compiler **warnings** (best for new programmers).

The Result : a.out

- If there are no errors in pgm.c, this command produces an **executable file**, which is one that can be executed (run).
- The gcc compiler names the executable file **a.out**
- To execute the program, at the prompt, type

  a.out
- Although we call this process "compiling a program," what actually happens is more complicated.

Stages of Compilation

Stage 1: **Preprocessing**

- Performed by a program called the **preprocessor**
- Modifies the source code (in RAM) according to **preprocessor directives (preprocessor commands**) embedded in the source code
- Strips comments and whitespace from the code
- The source code as stored on disk is <u>not</u> modified.

Stages of Compilation (con't)

Stage 2: **Compilation**

- Performed by a program called the **compiler**
- Translates the preprocessor-modified source code into **object code (machine code)**
- Checks for **syntax errors** and **warnings**
- Saves the object code to a disk file.
  - If any compiler errors are received, no object code file will be generated.
  - An object code file will be generated if only warnings, not errors, are received.

Stages of Compilation (con't)

The Compiler consists of the following sub-systems:

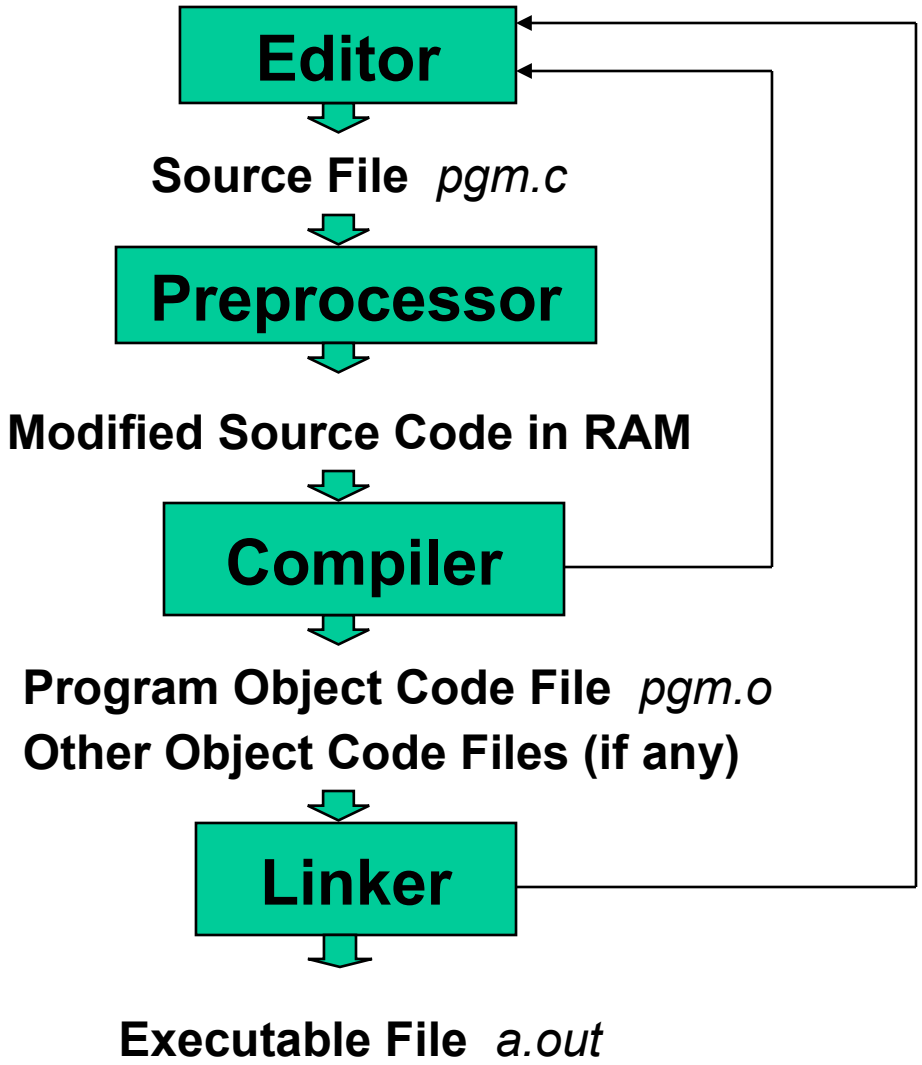
- **Parser** - checks for errors
- **Code Generator** - makes the object code
- **Optimizer** - may change the code to be more efficient

Stages of Compilation (con't)

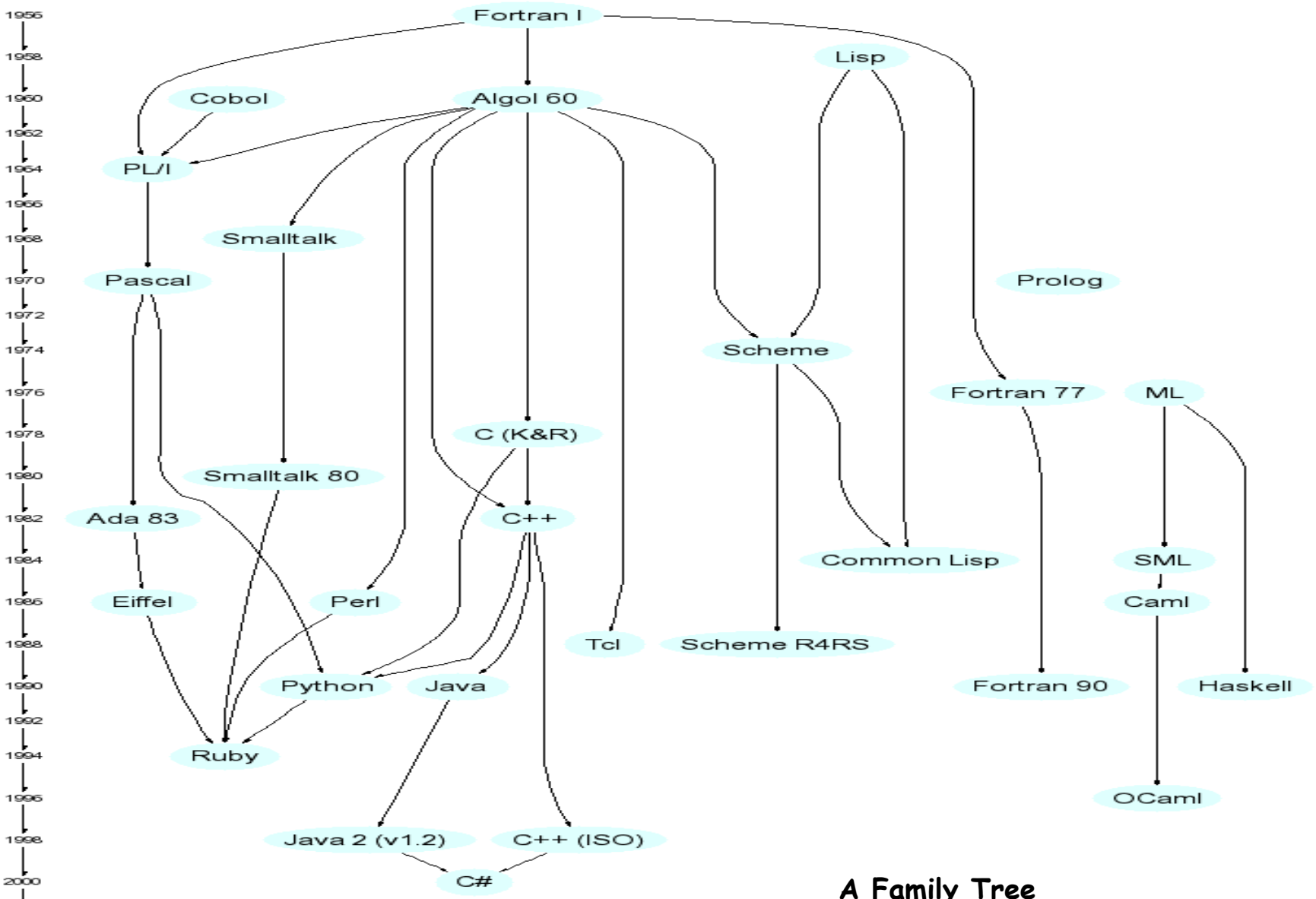
Stage 3: **Linking**

- Combines the program object code with other object code to produce the executable file.
- The other object code can come from the **Run-Time Library** (a collection of object code with an index so that the linker can find the appropriate code), other libraries, or object files that you have created.
- Saves the executable code to a disk file. On a Linux/Unix system, that file is called **a.out**.
  - If any linker errors are received, no executable file will be generated.

Program Development Using gcc



The Programming Language Family Tree



A Family Tree

http://merd.net/pixel/language-study/diagram.html

Plankalkul

- Developed by Konrad Zuse (1945)
- Never implemented
- Advanced data structures (floating point, arrays, records)
- Odd notation:
  - To code `A[7] = 5 * B[6]`

```
        | 5 * B => A
V       |     6    7     (subscripts)
S       |   1.n  1.n     (data types)
```

Pseudocodes

- Machine code had many negatives:
  - Poor readability, writability, modifiability
  - Low-level machine operations had deficiencies, including no indexing or floating point
- Example: Short Code
  - Developed by Mauchly in 1949 for the BINAC
  - Expressions coded from left to right
  - Operations included:
    - 1n => (n+2)nd power (e.g., 12 represented the 4^{th} power)
    - 2n => (n+2)nd root
    - 07 => addition

FORTRAN I (1957)

- Designed for IBM 704, which had
  - Index registers
  - Floating point hardware
- Development environment
  - Computers small and unreliable
  - Applications were scientific
  - There was no programming methodology or tools
  - Machine efficiency was of primary importance

FORTRAN I

- Impact of development environment
  - Needed good array handling and counting loops
  - No need for
    - Dynamic storage
    - String handling
    - Decimal arithmetic
    - Powerful input/output operations

FORTRAN I

- First implemented version of FORTRAN
  - Names could have up to 6 characters
  - Posttest counting loop (DO statement)
  - Formatted input/output
  - User-defined subprograms
  - 3-way selection statement (arithmetic IF)
  - No data typing statements
  - Released in 1957 after 18 worker/years effort
  - Programs larger than 400 lines rarely compiled correctly (704 was unreliable)
  - Code was fast;
  - FORTRAN I quickly became widely used

FORTRAN IV (1960-62)

- Added features
  - Explicit type declarations
  - Logical IF statement
  - Subprogram names could be parameters

FORTRAN 77 (1978)

- Character string handling
- Logical loop control statement
- IF-THEN-ELSE statement

FORTRAN 90 (1990)

- Modules
- Dynamic arrays
- Pointers
- Recursion
- CASE statement
- Parameter type checking

LISP (1959)

- LISt Processing language
- Designed by John McCarthy at MIT
- Designed to meet AI research needs
  - Process data in lists, not arrays
  - Symbolic computation, rather than numeric
- Two data types: atoms and lists
- Syntax based on lambda calculus
- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- COMMON LISP and Scheme are dialects

ALGOL 58 (1958)

- Development Environment
  - FORTRAN just developed for IBM 704
  - Many other machine-specific languages were being developed
  - There were no portable languages
  - There was no universal language for communicating algorithms

ALGOL 58

- Goals for the language
  - Close to mathematical notation
  - Good for describing algorithms
  - Must be translatable to machine code
- Notes:
  - Not meant to be implemented, but variations were (MAD, JOVIAL)
  - IBM was initially enthusiastic, but dropped support by mid 1959

ALGOL 58 Features

- Type concept was formalized
- Names could have any length
- Arrays could have any number of subscripts
- Parameters had mode (**`in/out`**)
- Subscripts were placed in brackets
- Compound statements (**`begin..end`**)
- Semicolon used as statement separator
- Assignment operator was `:=`
- **`if`** statement had an **`else-if`** clause

ALGOL 60 (1960)

- New features included
  - Block structure (local scope)
  - Two parameter passing methods
  - Subprogram recursion
  - Stack-dynamic arrays
- Still lacked
  - Input/output
  - String handling

ALGOL 60

- Successes
  - Standard for publishing algorithms for more than 20 years
  - Basis for all subsequent imperative languages
  - First machine-independent language
  - First language whose syntax was formally defined (BNF was used)

ALGOL 60

- Failures
  - Never widely used, especially in U.S.
  - No I/o
  - Too flexible, making it hard to implement
  - FORTRAN was firmly entrenched
  - Formal syntax description (BNF considered strange at the time)
  - Lack of support by IBM

COBOL (1960)

- Based on UNIVAC FLOW-MATIC
  - Names up to 12 characters, with embedded hyphens
  - English names for arithmetic operators
  - Data and code were completely separate
  - Verbs were the first word in every statement
- Design Goals
  - Must look like simple English
  - Ease of use more important than powerful features
  - Broaden base of computer users
  - Must not be biased by current compiler problems

COBOL

- Contributions
  - First macro facility in HLL
  - Hierarchical data structures (records)
  - Nested selection statements
  - Long names (up to 30 characters)
  - Data Division
- Notes
  - First language required by DoD
  - Still the most widely used business application language

BASIC (1964)

- Kemeny & Kurtz (Dartmouth)
- Design goals
  - Easy to learn and use (non-science students)
  - Fast turnaround for homework
  - Free and private access
  - User time more important than computer time
- Current dialects
  - QuickBASIC, Visual BASIC

PL/I (1965)

- Designed by IBM and SHARE for scientific and business computing
- Contributions
  - First unit-level concurrency
  - First exception handling
  - Switch-selectable recursion
  - Pointer data type
  - Array cross sections
- Notes:
  - Many features were poorly designed
  - To large and complex
  - Still used for scientific and business applications

Early Dynamic Languages

- Characterized by dynamic typing and dynamic storage allocation
  - APL (A Programming Language) 1962
    - Designed a hardware description language
    - Many operators for scalars and arrays
    - Programs are difficult to read
  - SNOBOL (1964)
    - String manipulation language (Bell Labs)
    - Powerful pattern matching operators

ALGOL 68 (1968)

- Not really a superset of ALGOL 60
- Design based on orthogonality
- Contributions
  - User-defined data structures
  - Reference types
  - Dynamic (flex) arrays
- Notes
  - Even less usage than ALGOL 60
  - Strong influence on Pascal, C, and Ada

Pascal (1971)

- Designed by Wirth (quit ALGOL 68 committee)
- Intended to be used for teaching structured programming
- Small, simple, introduced nothing new
- Widely used for teaching programming in college, but is being supplanted by C++ and Java

C (1972)

- Designed for systems programming (Bell Labs, Dennis Richie)
- Evolved from B and ALGOL 68
- Powerful set of operators
- Poor type checking
- Initially spread via UNIX

Other Descendants of ALGOL

- Modula-2 (Wirth, mid-1970s)
  - Pascal plus modules and low-level features for systems programming
- Modula-3 (late 1980s)
  - Modula-2 plus classes, exception handling, garbage collection, and concurrency
- Oberon
  - Adds OOP support to Modula-2
- Delphi (Borland)
  - Pascal plus OOP features
  - More elegant, safer than C++

Prolog (1972)

- Developed by Comerauer, Roussel, Kowalski
- Based on formal logic
- Non-procedural
- Can be described as being an intelligent database system that uses an inferencing process to infer the truth of given queries

Ada (1983)

- Huge design effort
- Contributions
  - Packages support data abstraction
  - Elaborate exception handling
  - Generic program units
  - Concurrency through the tasking model
- Ada 95
  - OOP support through type derivation
  - New concurrency features for shared data

Smalltalk (1972-1980)

- Developed at Xerox PARC (Alan Kay)
- First full implementation of an object-oriented language
  - Data abstraction
  - Inheritance
  - Dynamic type binding
- Pioneered the graphical user interface

C++ (1985)

- Bell Labs (Stroustrup)
- Evolved from C and SIMULA 67 (class)
- Facilities for OOP were added to C
- Also has exception handling
- Supports both procedural and OO programming
- Large and complex
- Eiffel (1992), a related language that supports OOP
  - Not directly derived from any other language
  - Smaller and simpler than C++

Java (1995)

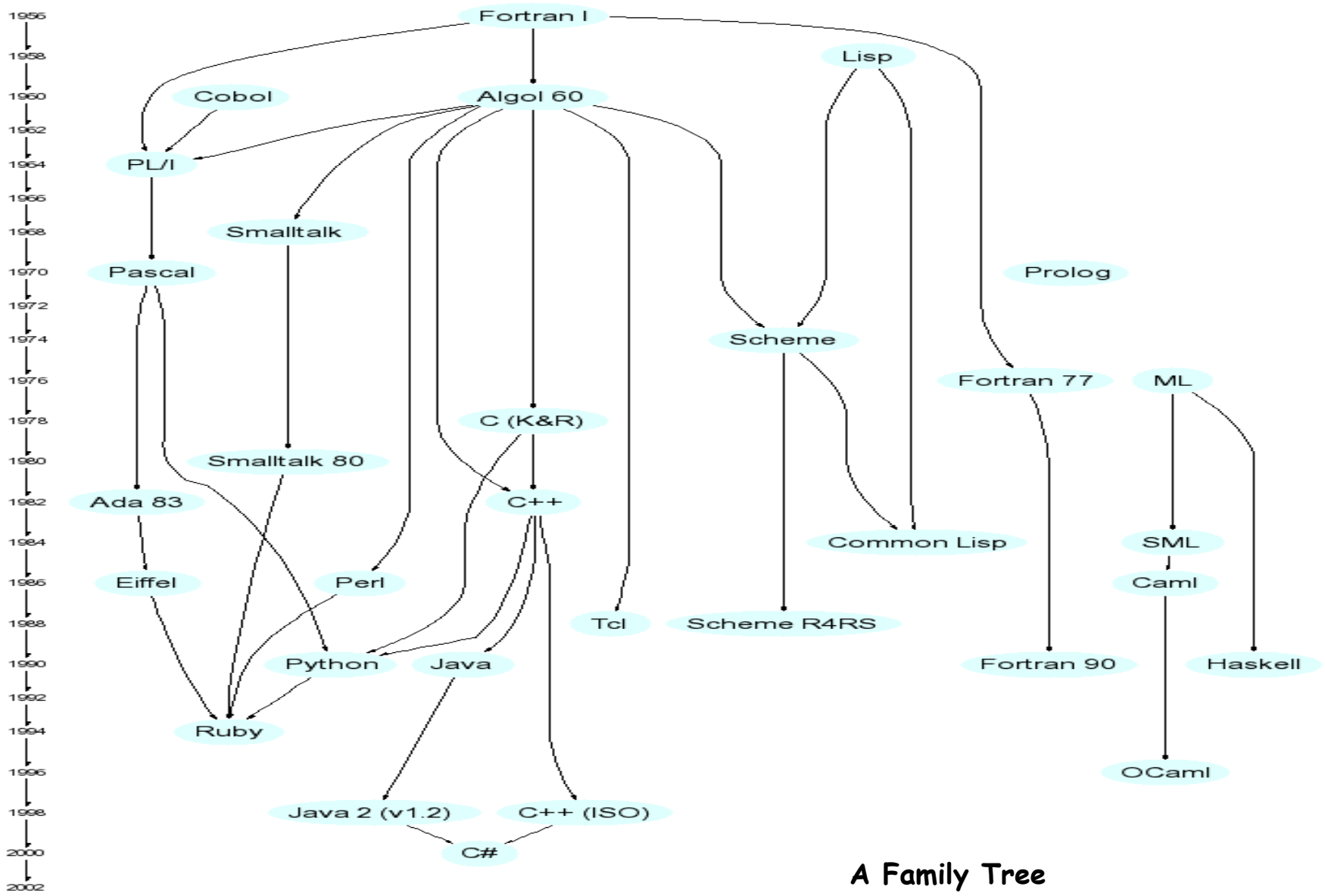
- Developed at Sun in early 1990s
- Based on C++
  - Significantly simplified
  - Supports only OOP
  - Has references, but not pointers
  - Includes support for applets and a form of concurrency (threads)

Scripting Languages

- JavaScript (1995)
  - Originally LiveScript, developed jointly by Netscape and Sun Microsystems
  - Used as an HTML-embedded client-side scripting language, primarily for creating dynamic HTML documents
- PHP (1995)
  - Developed by Rasmus Lerdorf
  - An HTML-embedded server side scripting language
  - Provides support for many DBMS

C# (2002)

- Based on C++ and Java, with ideas from Delphi and Visual Basic
- A language for component-based development in the .NET Framework
- Likely to become very popular because of Microsoft's huge effort to push .NET



A Family Tree

http://merd.net/pixel/language-study/diagram.html

The End

Read Chapters 1 and 2 in the Textbook