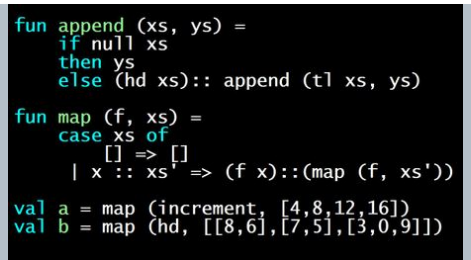
Introduction to Computer Science

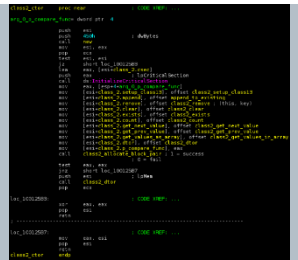
William Hsu
Department of Computer Science and Engineering
National Taiwan Ocean University

Chapter 6: Programming Languages

" If debugging is the process of removing software bugs, then programming must be the process of putting them in. "

- Edsger Dijkstra



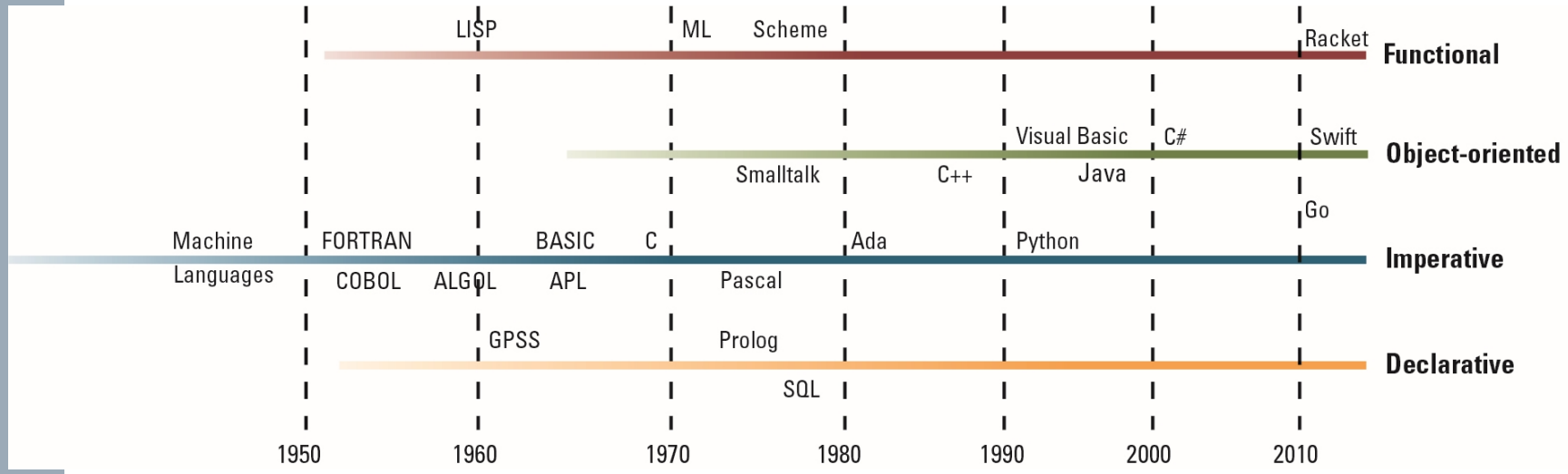


- 6.1 Historical Perspective
- 6.2 Traditional Programming Concepts
- 6.3 Procedural Units
- 6.4 Language Implementation
- 6.5 Object Oriented Programming
- 6.6 Programming Concurrent Activities
- 6.7 Declarative Programming

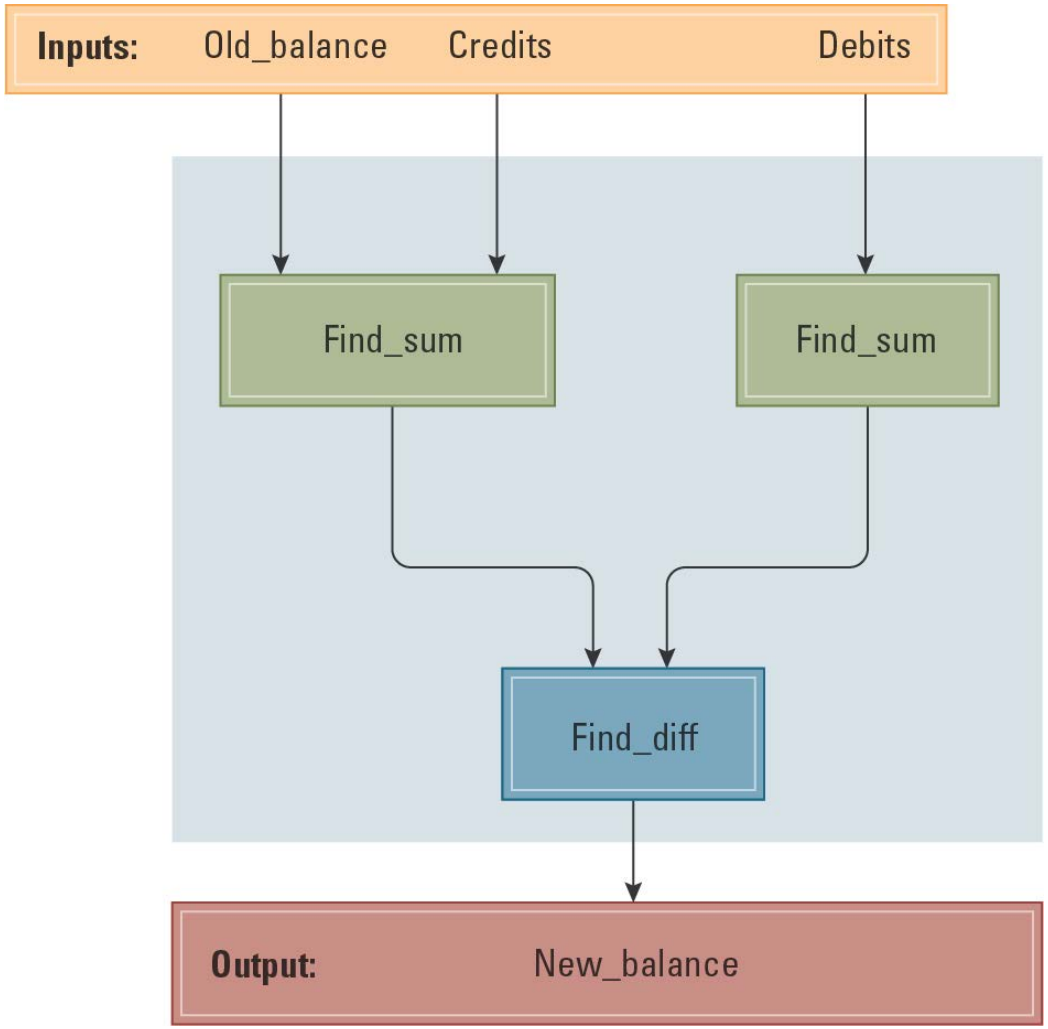
How many programming languages are there?

- 8945 ! (Since 18th century)
  - https://hopl.info/
- Languages still in use can be less than 500.

The Evolution of Programming Paradigms



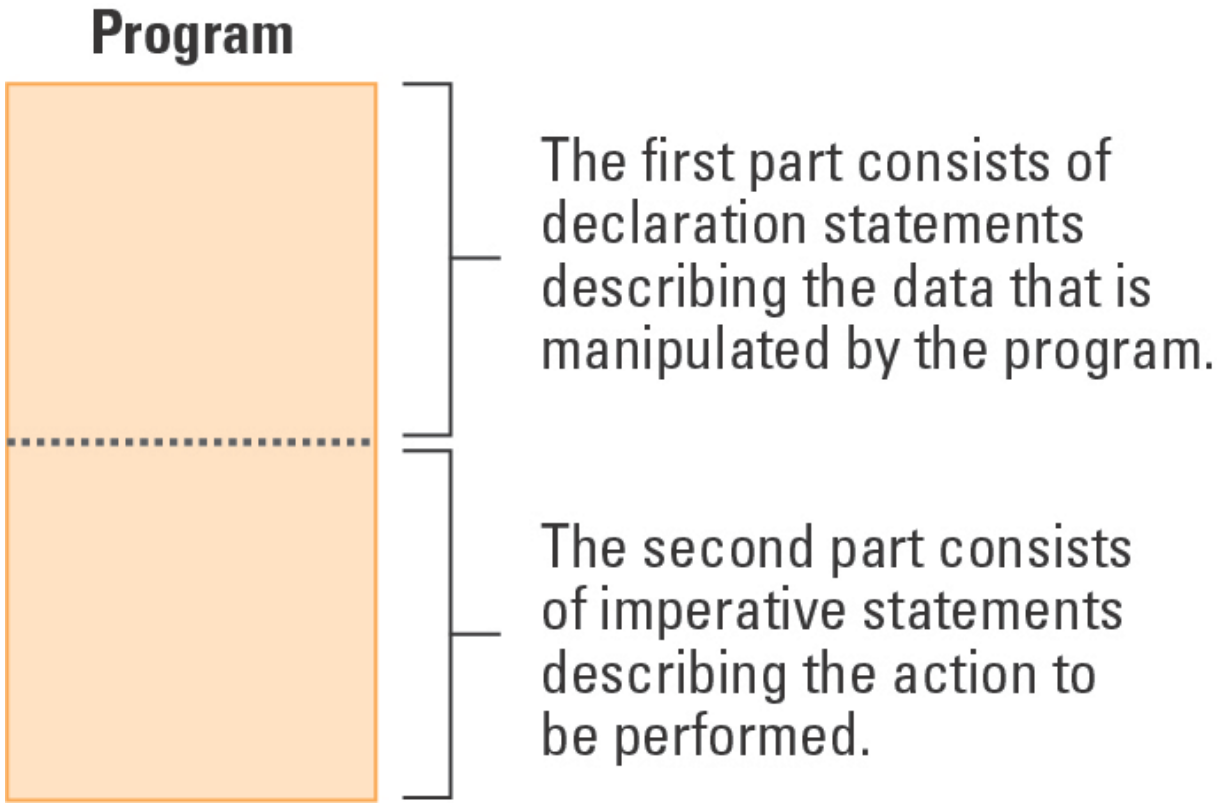
A Function for Checkbook Balancing Constructed from Simpler Functions



Traditional Programming Concepts

- High-level languages (C, C++, Java, C#, FORTRAN) include many kinds of abstractions
  - Simple: constants, literals, variables
  - Complex: statements, expressions, control
  - Esoteric: procedures, modules, libraries

The Composition of a Typical Imperative Program or Program Unit



Data Types

- `Integer`: Whole numbers
- `Real (float)`: Numbers with fractions
- `Character`: Symbols
- `Boolean`: True/false

Variables and Data types

```
float Length, Width;
int   Price, Total, Tax;
char  Symbol;

int   WeightLimit = 100;
```

Data Structure

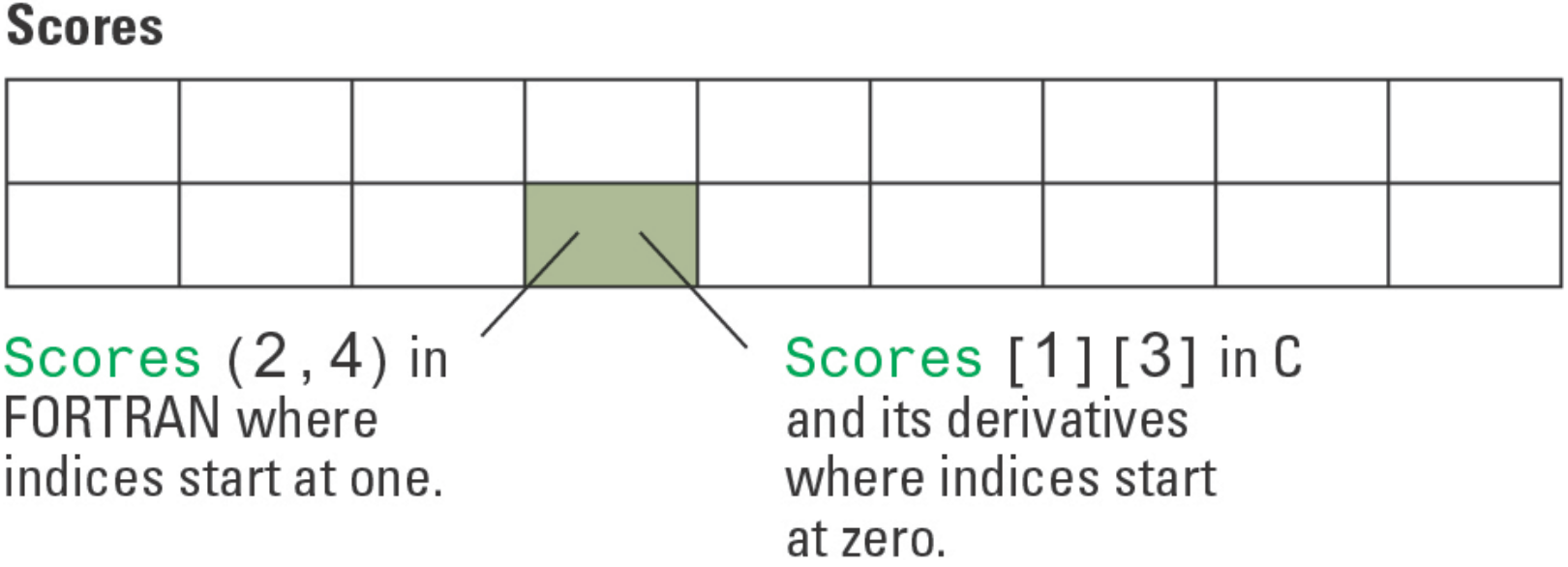
- Conceptual shape or arrangement of data
- A common data structure is the **array**
  - In C

```
int Scores[2][9];
```

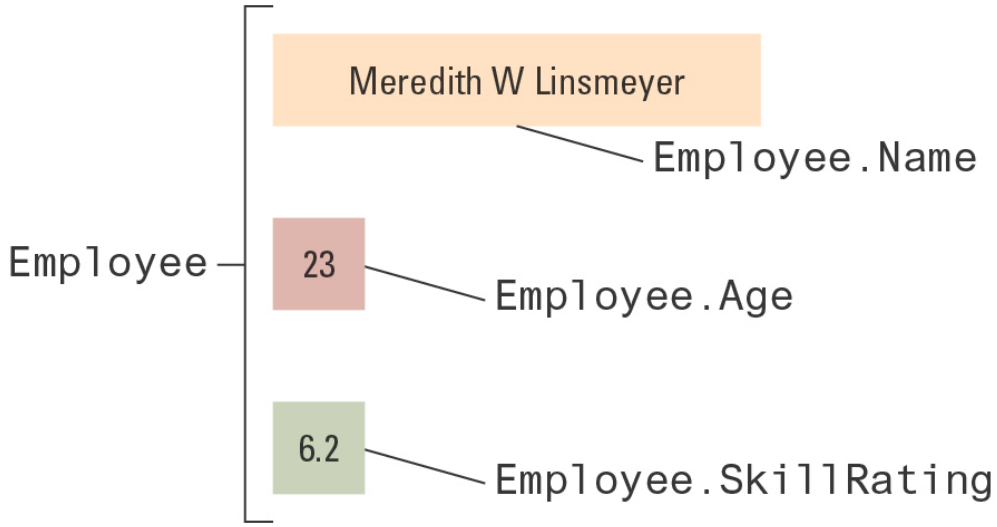
  - In FORTRAN

```
INTEGER Scores(2,9)
```

A Two-dimensional Array with Two Rows and Nine Columns



The Conceptual Structure of the Aggregate type `Employee`




```
struct {
        char  Name[25];
        int   Age;
        float SkillRating;
    } Employee;
```

Assignment Statements

- In C, C++, C#, Java

```
Z = X + y;
```

- In Ada

```
Z := X + y;
```

- In APL (A Programming Language)

```
Z ← X + y
```

Control Statements

- Go to statement

```
        goto 40
20      Evade()
        goto 70
40      if (KryptoniteLevel < LethalDose) then goto 60
        goto 20
60      RescueDamsel()
70      ...
```

- As a single statement

```
if (KryptoniteLevel < LethalDose):
    RescueDamsel()
else:

    Evade()
```

Control Statements (continued)

- If in Python

```
if (condition):
    statementA
else:
    statementB
```

- In C, C++, C#, and Java

```
if (condition) statementA; else statementB;
```

- In Ada

```
IF condition THEN
    statementA;
ELSE
    statementB;
END IF;
```

Control Statements (continued)

- While in Python

```
while (condition):
    body
```

- In C, C++, C#, and Java

```
while (condition)
{ body }
```

- In Ada

```
WHILE condition LOOP
    body
END LOOP;
```

Control Statements (continued)

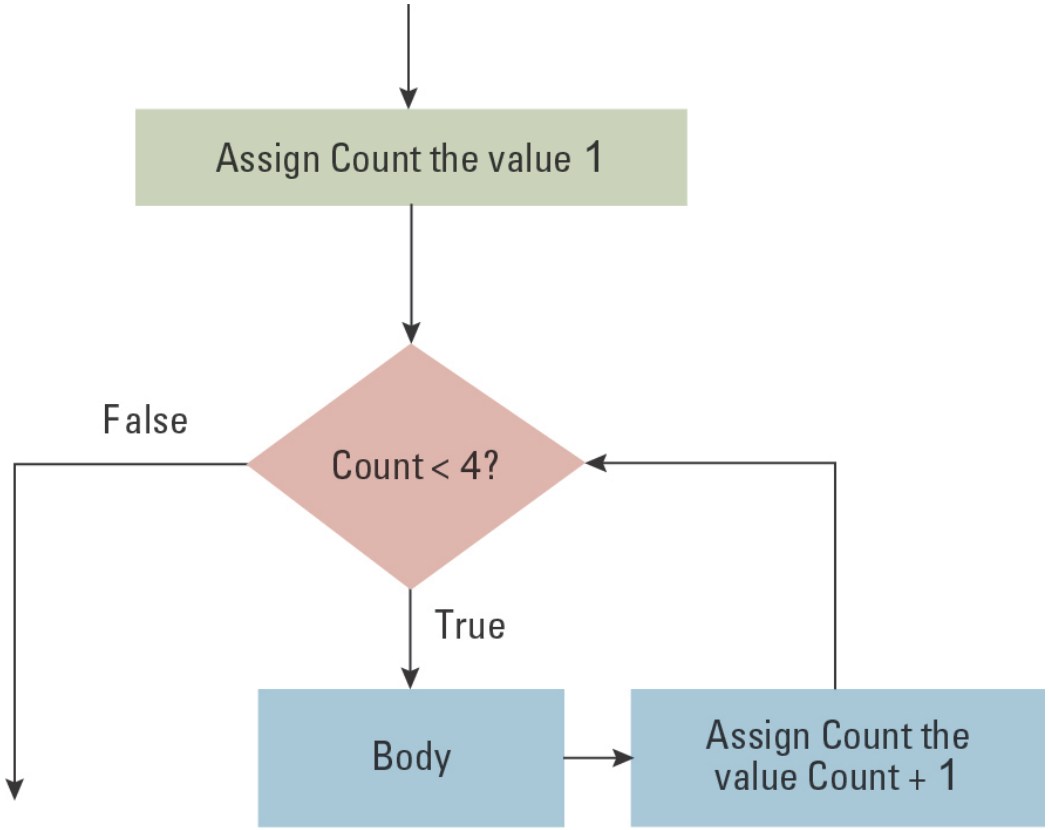
› Switch statement in C, C++, C#, and Java

```
switch (variable) {
    case 'A': statementA; break;
    case 'B': statementB; break;
    case 'C': statementC; break;
    default:  statementD; }
```

› In Ada

```
CASE variable IS
    WHEN 'A'=> statementA;
    WHEN 'B'=> statementB;
    WHEN 'C'=> statementC;
    WHEN OTHERS=> statementD;
END CASE;
```

The `for` Loop Structure and its Representation in C++, C#, and Java



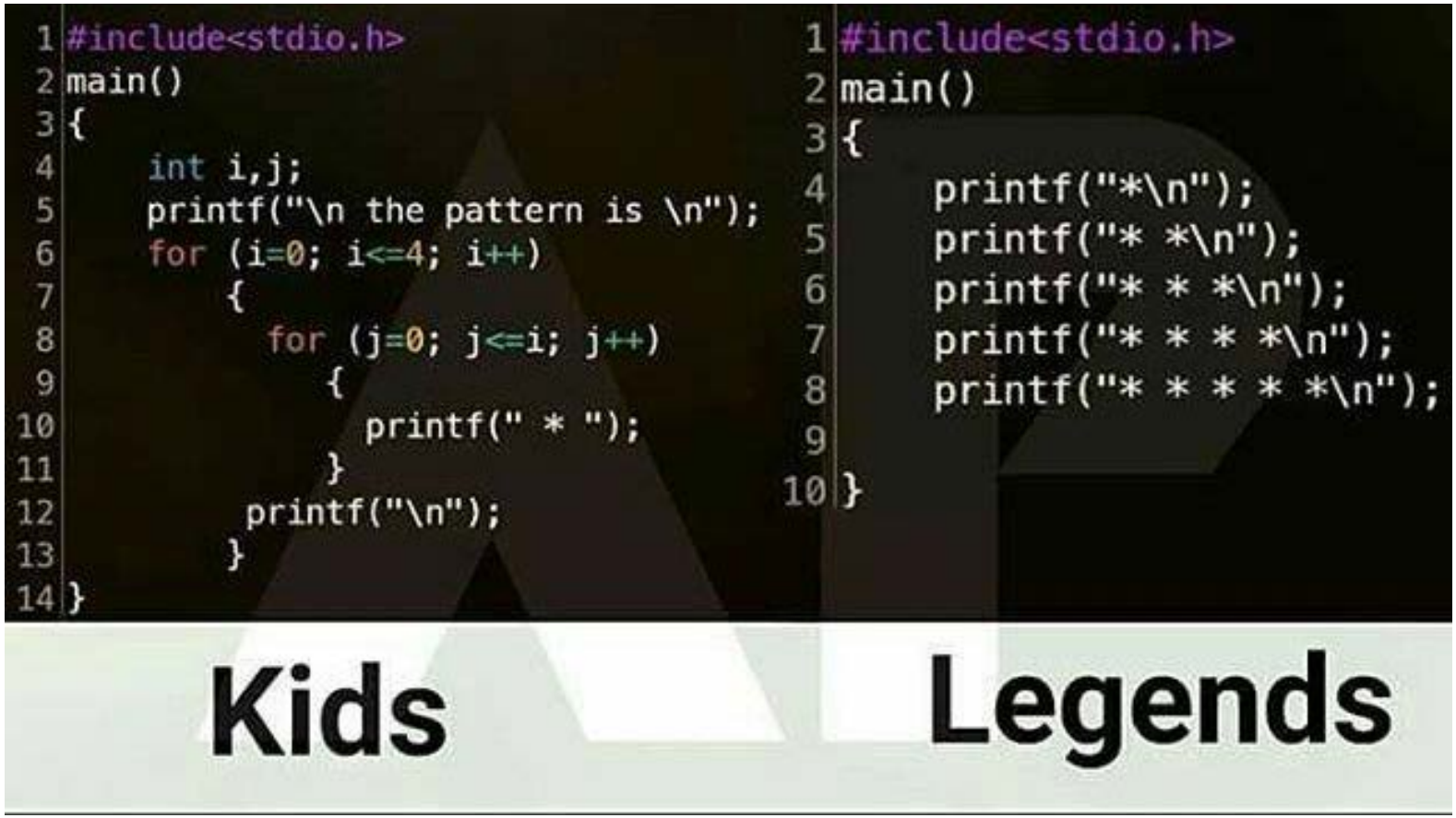

```
for (int Count = 1; Count < 4; Count++)
    body;
```

Comments

- Explanatory statements within a program
- Helpful when a human reads a program
- Ignored by the compiler

```
/* This is a comment in C/C++/Java. */

// This is a comment in C/C++/Java.

<!-- This is a comment -->

% This is a comment in LaTeX
```

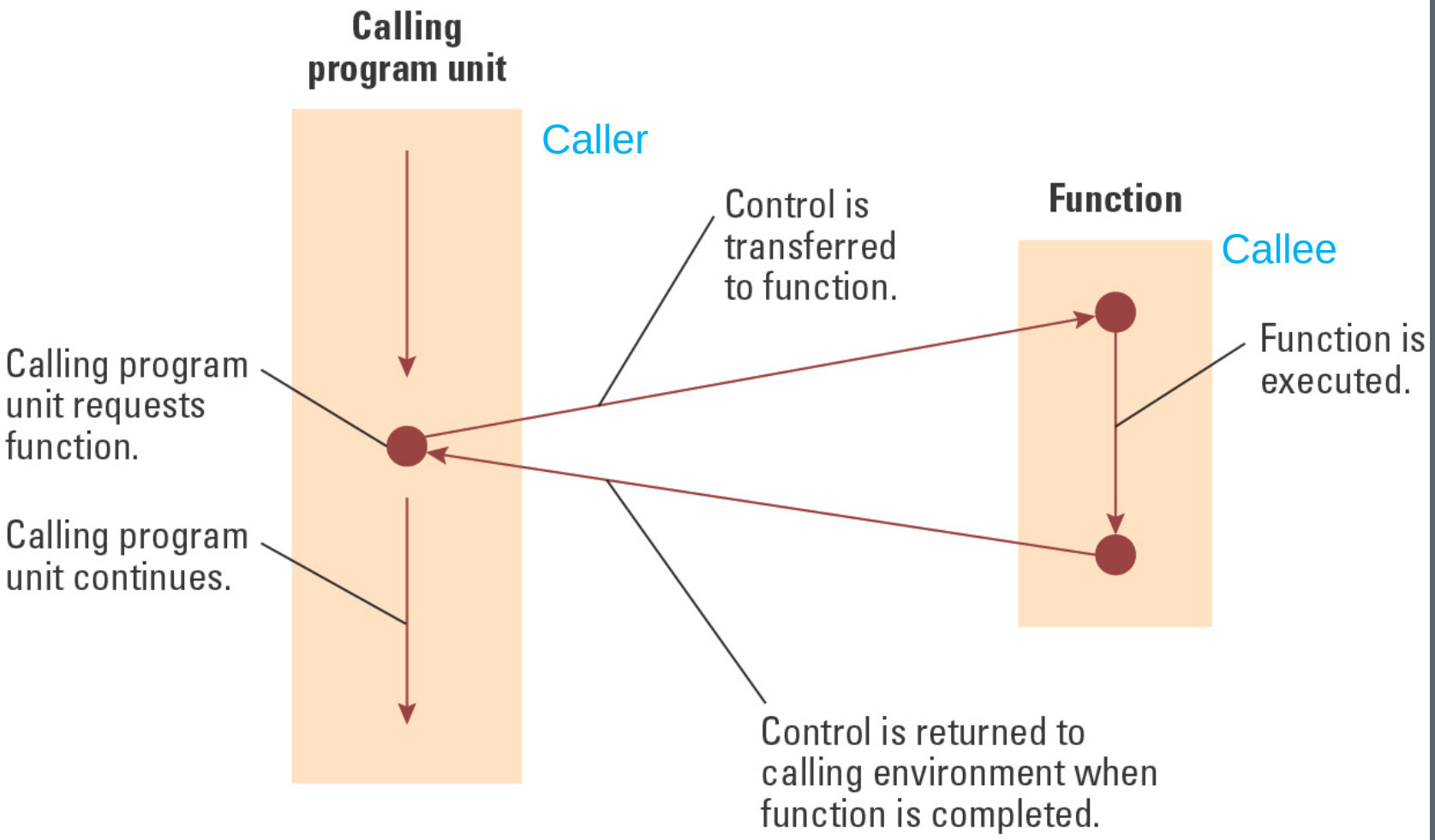
Legend never dies



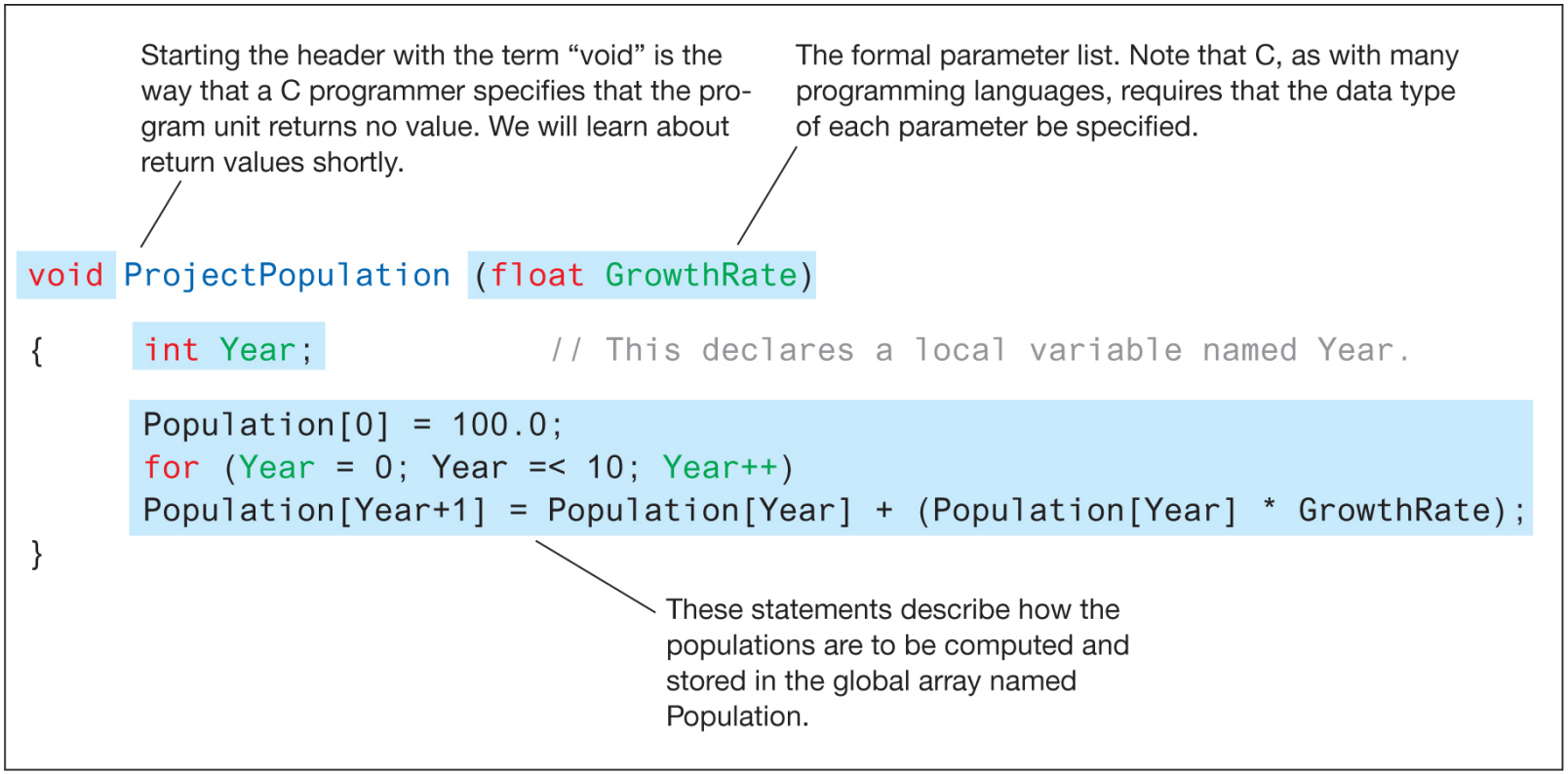
Procedural Units

- Many terms for this concept:
  - Subprogram, subroutine, procedure, method, function.
- Unit begins with the function's **header**.
- Local versus Global Variables.
- Formal versus Actual Parameters.
- Passing parameters by value versus reference.

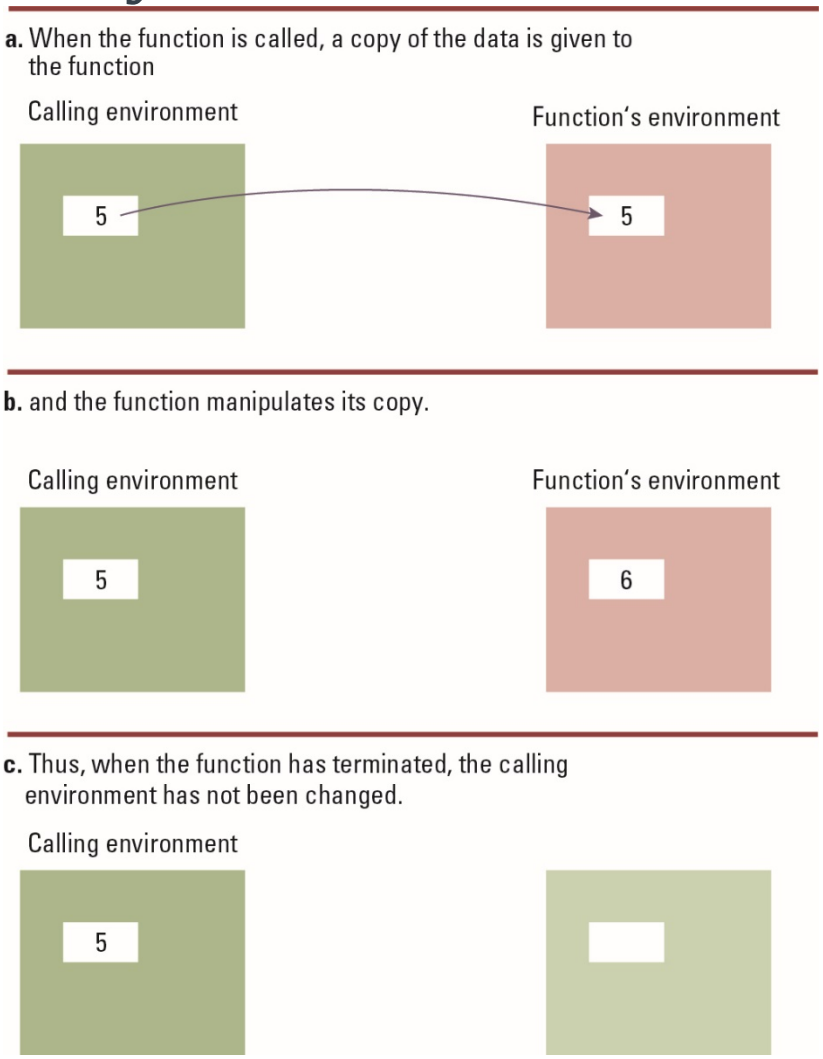
The Flow of Control Involving a Function



The Function `ProjectPopulation` Written in the Programming Language C



Executing the Function `Demo` and Passing Parameters by Value



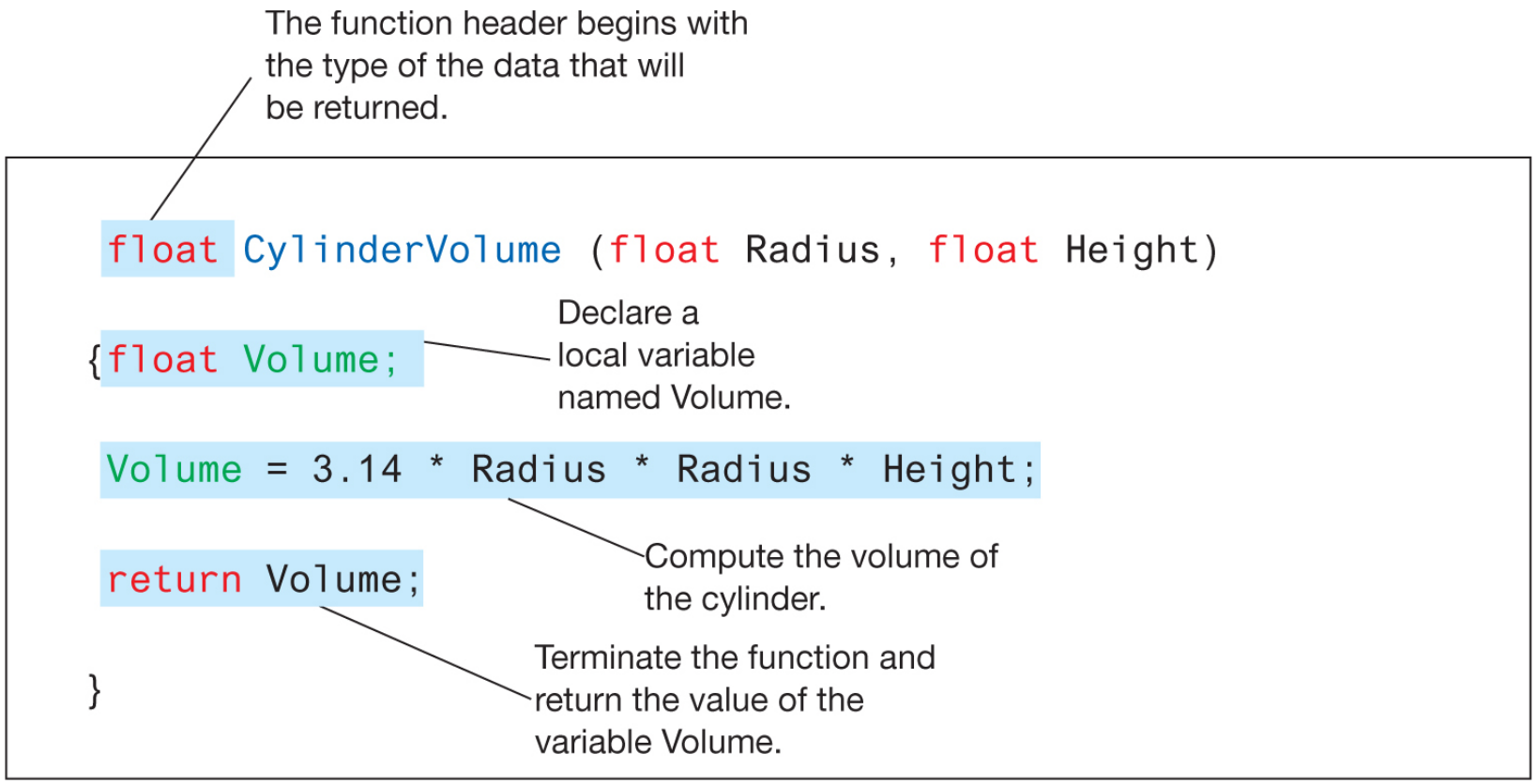
Executing the Function Demo and Passing Parameters by Reference

a. When the function is called, the formal parameter becomes a reference to the actual parameter.

b. Thus, changes directed by the function are made to the actual parameter

c. and are, therefore, preserved after the function has terminated.

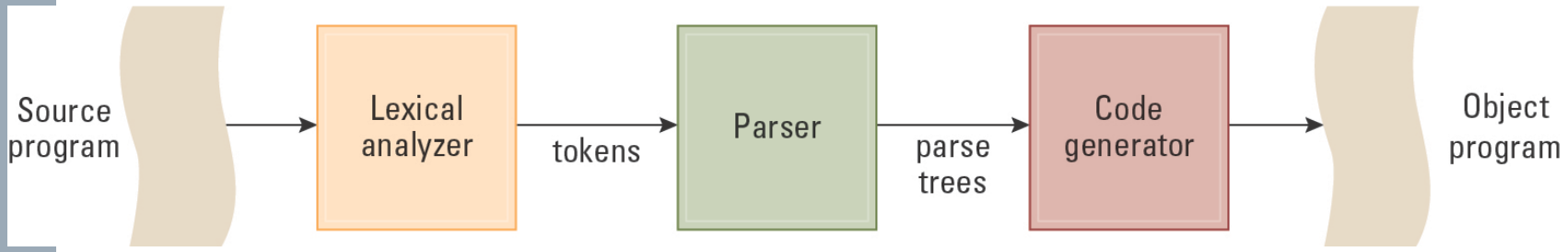
The Fruitful Function `CylinderVolume` Written in the Programming Language C



Language Implementation

- The process of converting a program written in a high-level language into a machine-executable form.
  - The Lexical Analyzer recognizes which strings of symbols represent a single entity, or token.
  - The Parser groups tokens into statements, using syntax diagrams to make parse trees.
  - The Code Generator constructs machine-language instructions to implement the statements.

The Translation Process

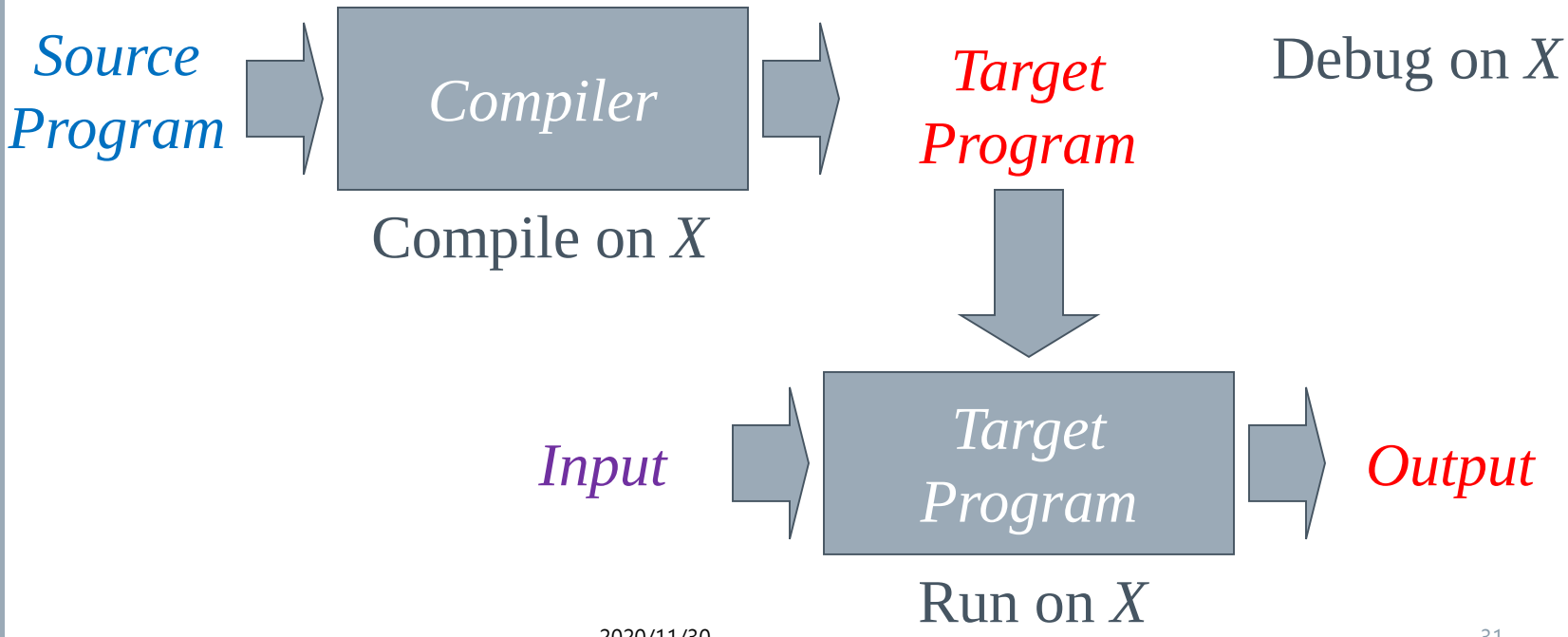


Compilation vs. Interpretation

- Compilation vs. interpretation
  - Not opposites.
  - Not a clear-cut distinction.
- Pure Compilation
  - The compiler translates the high-level source program into an equivalent target program (typically in machine language).
    - The "target program" is called the object code.
    - You translate once and run many times.

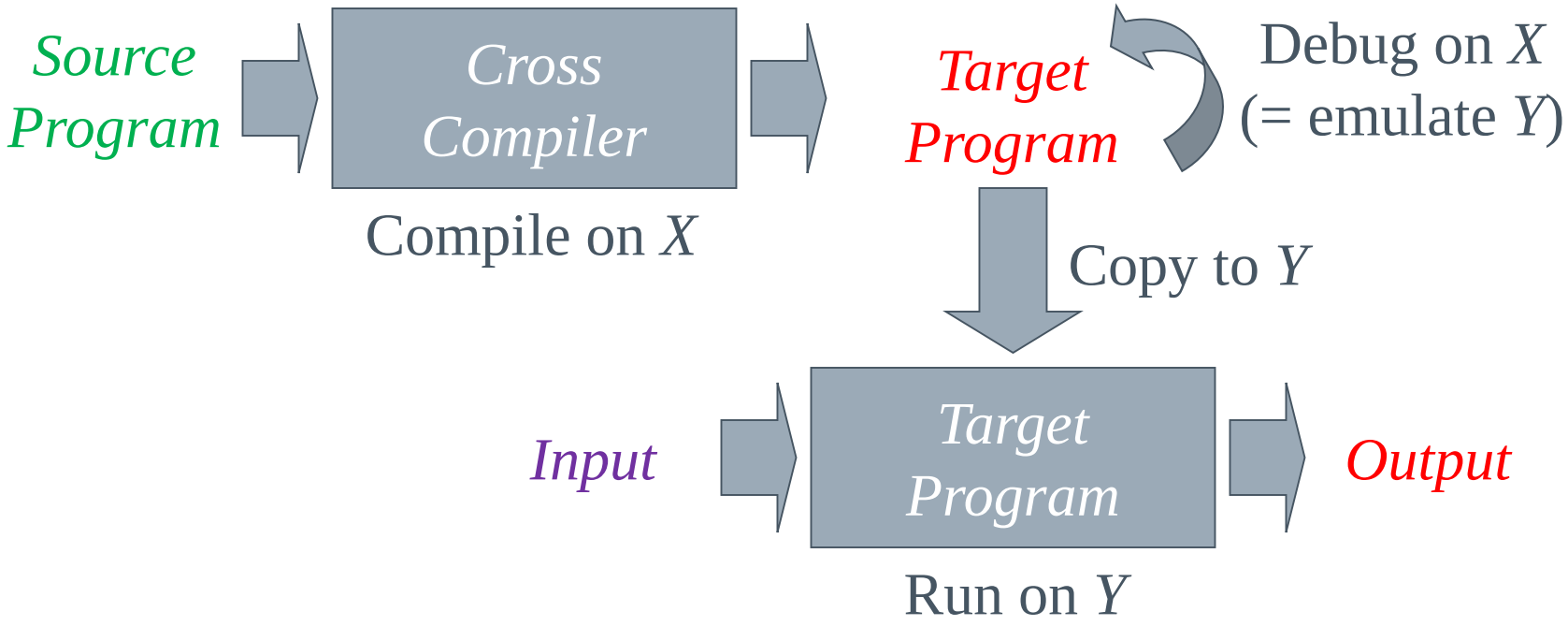
Compilation

- Compilation is the conceptual process of translating source code into a CPU-executable binary target code
- Compiler runs on the same platform *X* as the target code



Cross Compilation

› Compiler runs on platform *X*, target code runs on platform Y

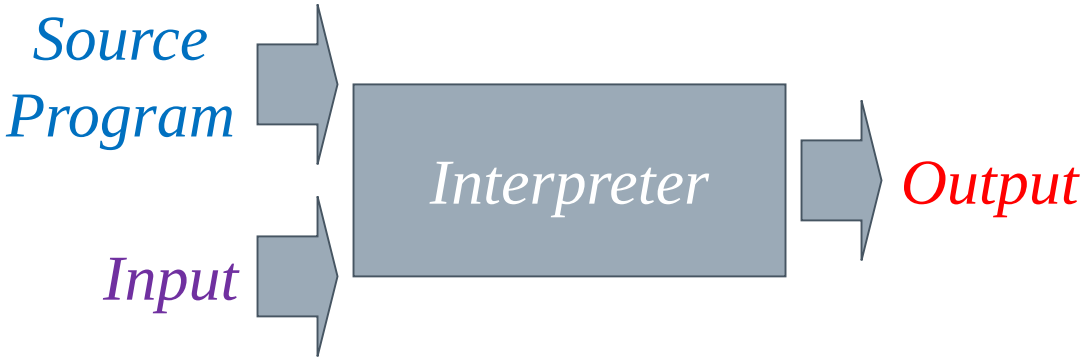


Compilation vs. Interpretation

- Pure Interpretation
  - Interpreter stays around for the execution of the program.
  - Interpreter is the locus of control during execution.
  - You translate for each run.

Interpretation

› Interpretation is the conceptual process of running high-level code by an interpreter



Compilation vs. Interpretation

- Interpretation:
  - Greater flexibility.
  - Better diagnostics (error messages), easier to debug.
- Compilation
  - Better performance.

Compilation vs. Interpretation

- Compilers “try to be as smart as possible” to fix decisions that can be taken at compile time to avoid to generate code that makes this decision at run time
  - Type checking at compile time vs. runtime
  - Static allocation
  - Static linking
  - Code optimization
- Compilation leads to better performance in general
  - Allocation of variables without variable lookup at run time
  - Aggressive code optimization to exploit hardware features

Compilation vs. Interpretation

- Benefit of interpretation?
  - Interpretation facilitates interactive debugging and testing
    - Interpretation leads to better diagnostics of a programming problem
    - Procedures can be invoked from command line by a user
    - Variable values can be inspected and modified by a user
  - Some programming languages cannot be purely compiled into machine code alone
    - Some languages allow programs to rewrite/add code to the code base dynamically
    - Some languages allow programs to translate data to code for execution (interpretation)

Compilation vs. Interpretation

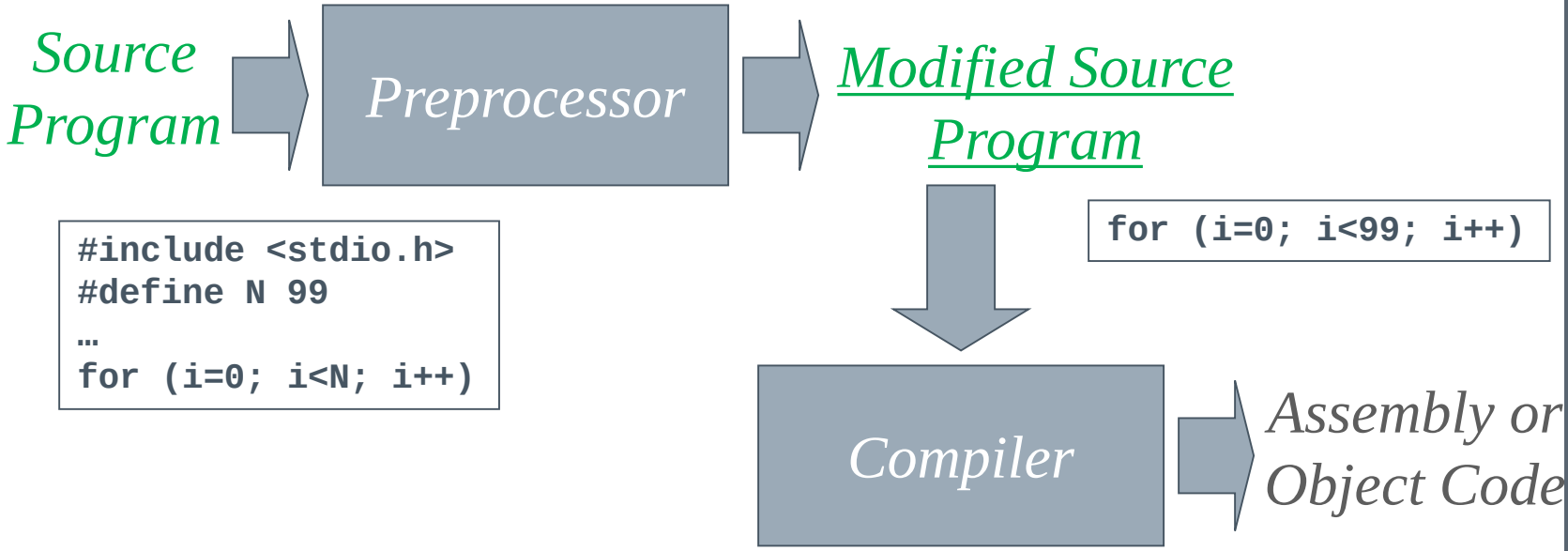
- The compiler versus interpreter implementation is often fuzzy
  - One can view an interpreter as a virtual machine that executes high-level code
  - Java is compiled to bytecode
  - Java bytecode is interpreted by the Java virtual machine (JVM) or translated to machine code by a just-in-time compiler (JIT)
  - A processor (CPU) can be viewed as an implementation in hardware of a virtual machine (e.g. bytecode can be executed in hardware)

Compilation vs. Interpretation

- Implementation strategies:
  - Preprocessor
    - Removes comments and white space.
    - Groups characters into *tokens* (keywords, identifiers, numbers, symbols).
    - Expands abbreviations in the style of a macro assembler.
    - Identifies higher-level syntactic structures (loops, subroutines).

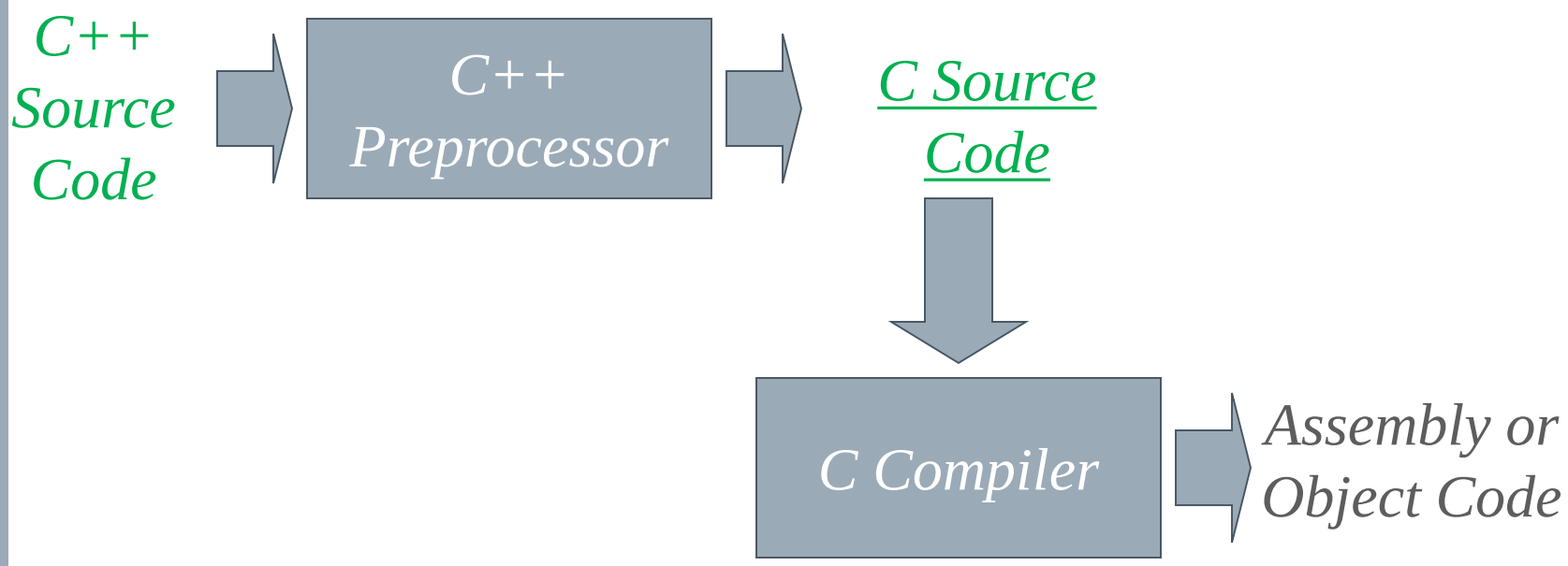
Preprocessing

› Most C and C++ compilers use a preprocessor to import header files and expand macros.



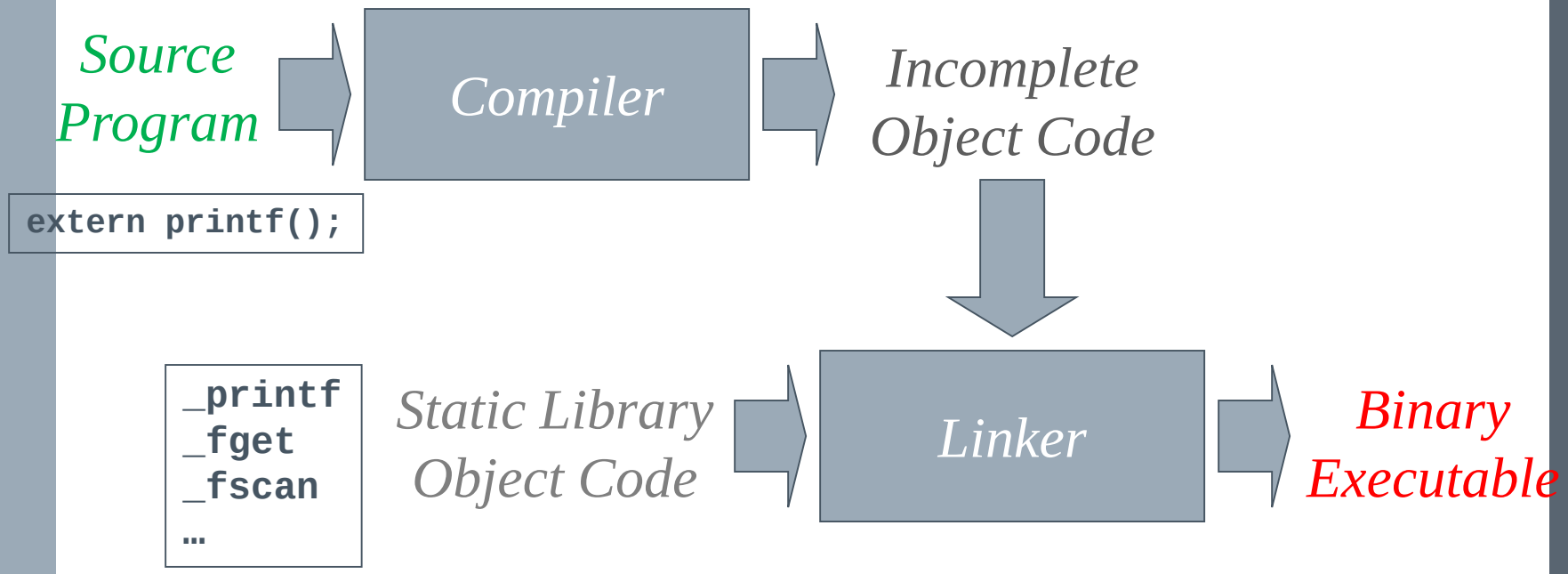
The CPP Preprocessor

› Early C++ compilers used the CPP preprocessor to generated C code for compilation



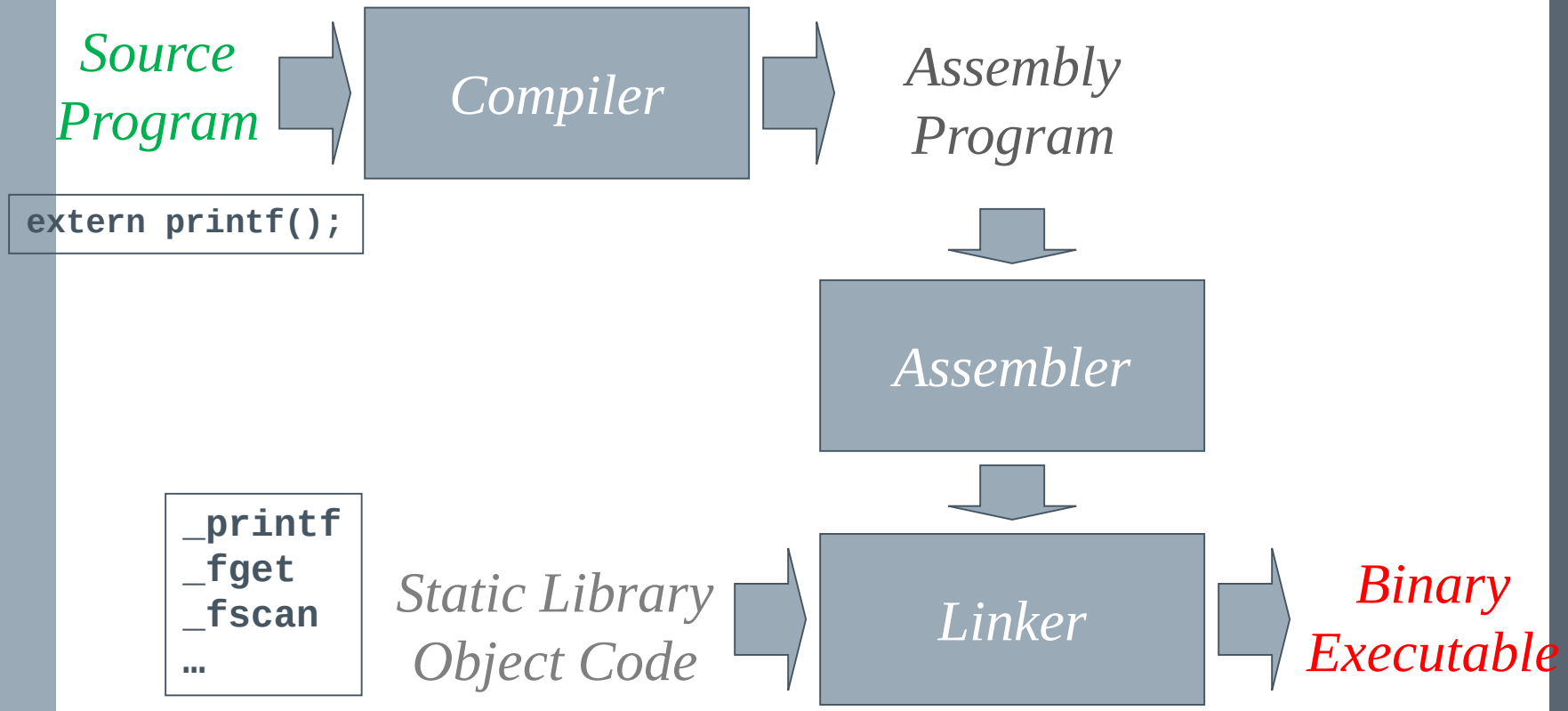
Pure Compilation and Static Linking

- Adopted by the typical Fortran systems
- Library routines are separately linked (merged) with the object code of the program



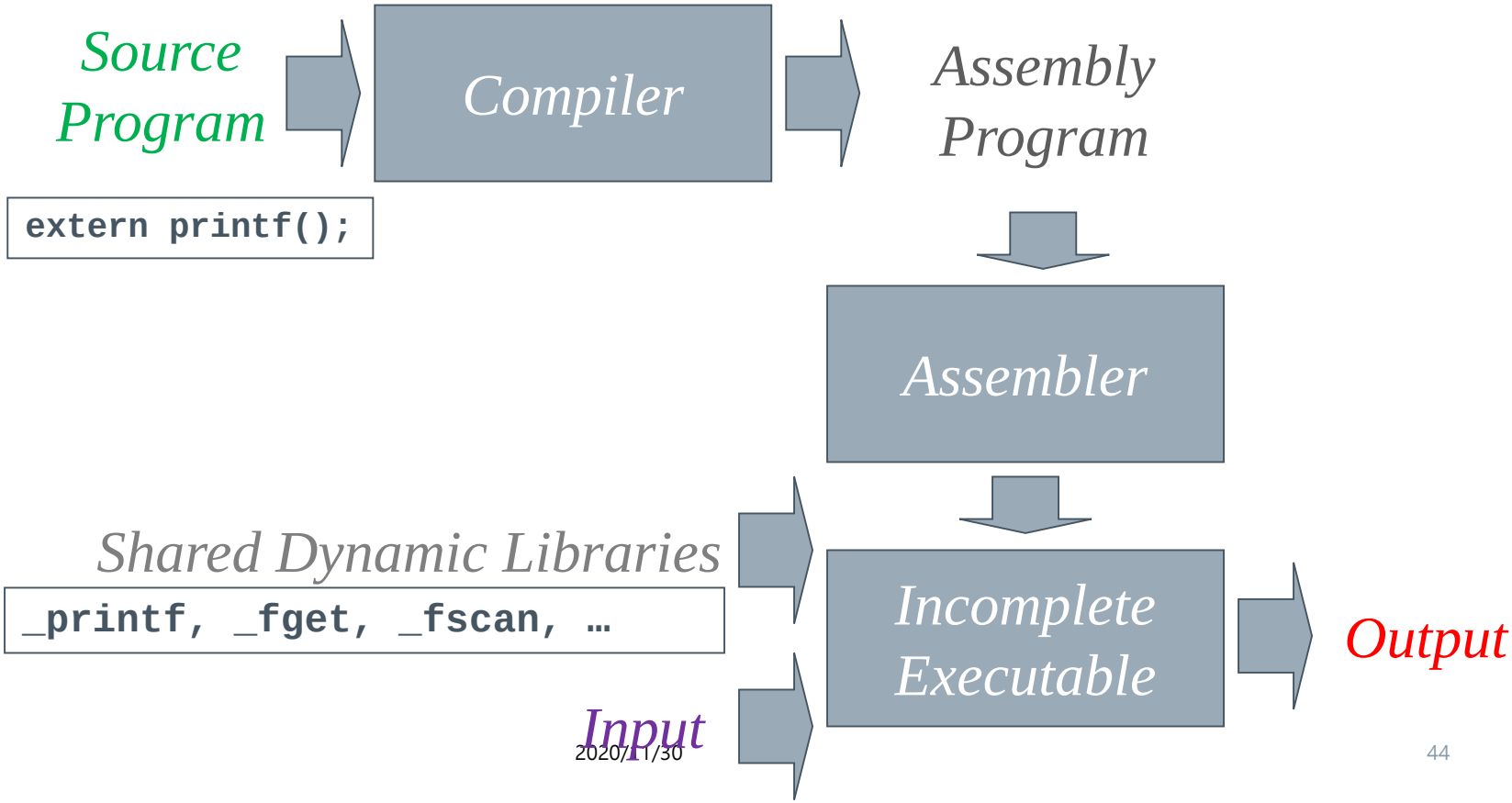
Compilation, Assembly, and Static Linking

› Facilitates debugging of the compiler



Compilation, Assembly, and Dynamic Linking

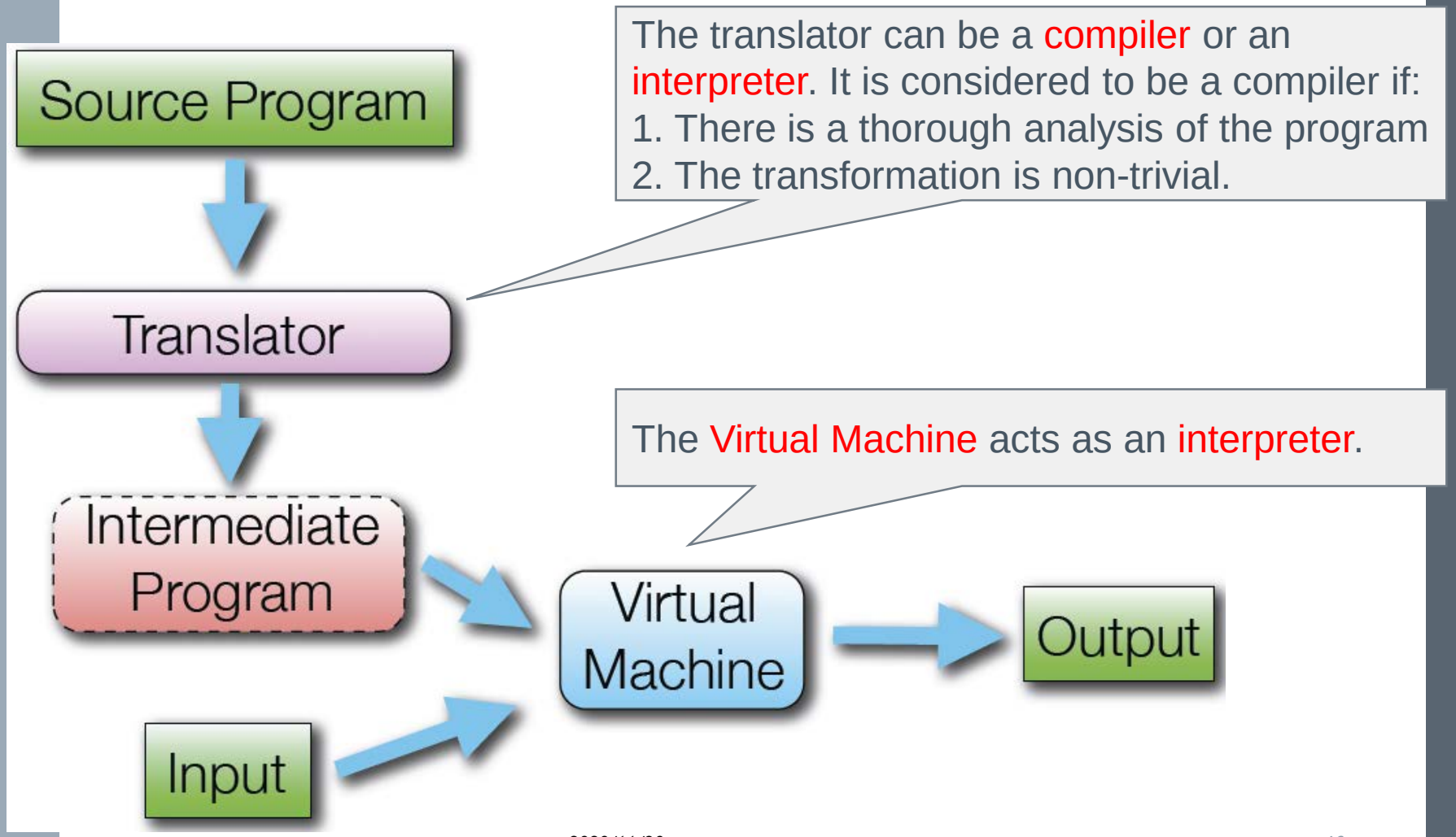
› Dynamic libraries (DLL, .so, .dylib) are linked at run-time by the OS (via stubs in the executable)



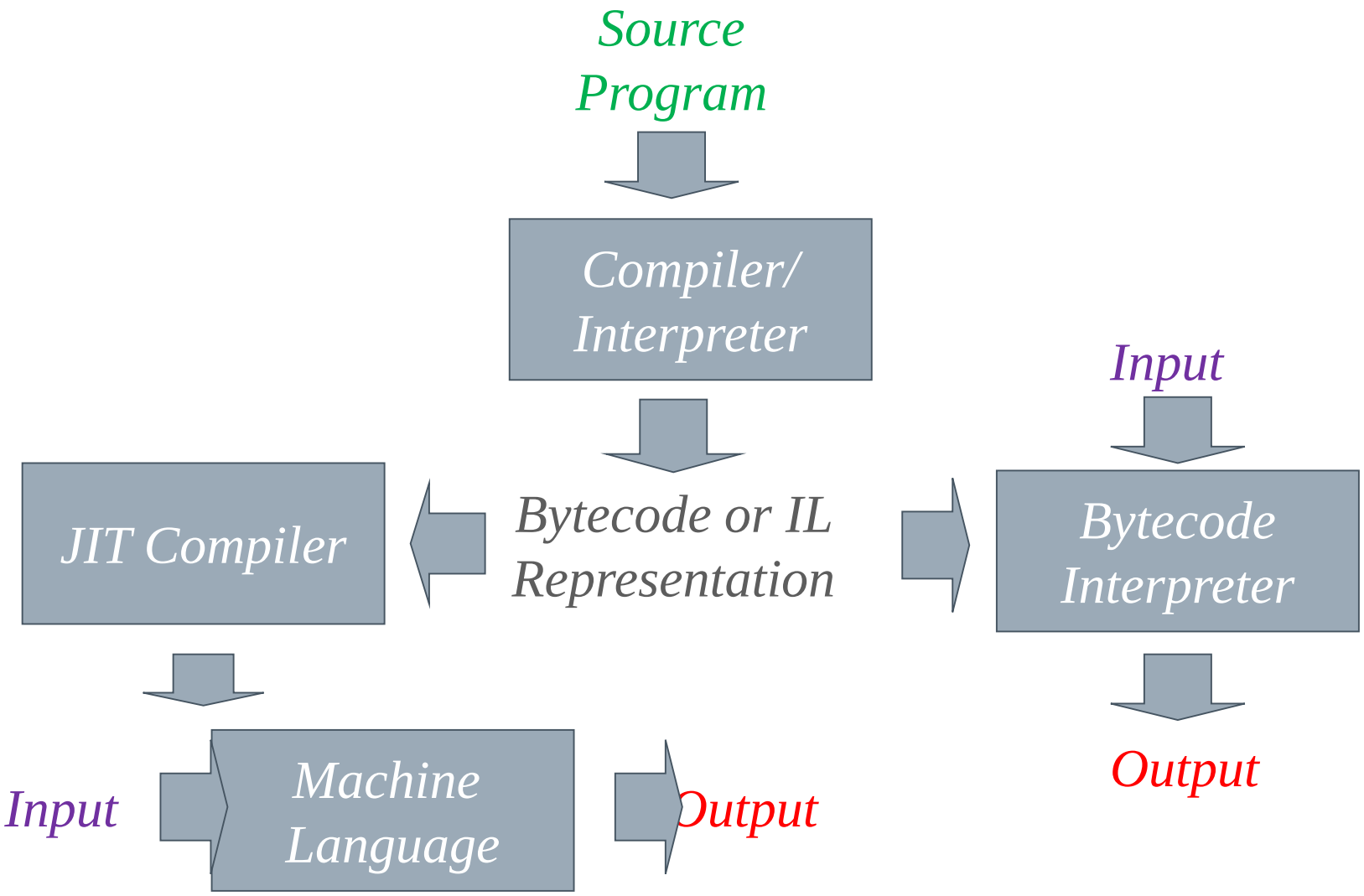
Compilation vs. Interpretation

- Implementation strategies:
  - Dynamic and Just-in-Time Compilation
    - In some cases a programming system may deliberately delay compilation until the last possible moment.
      - Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.
      - The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs.
      - The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

How Java does it



JIT Compilers

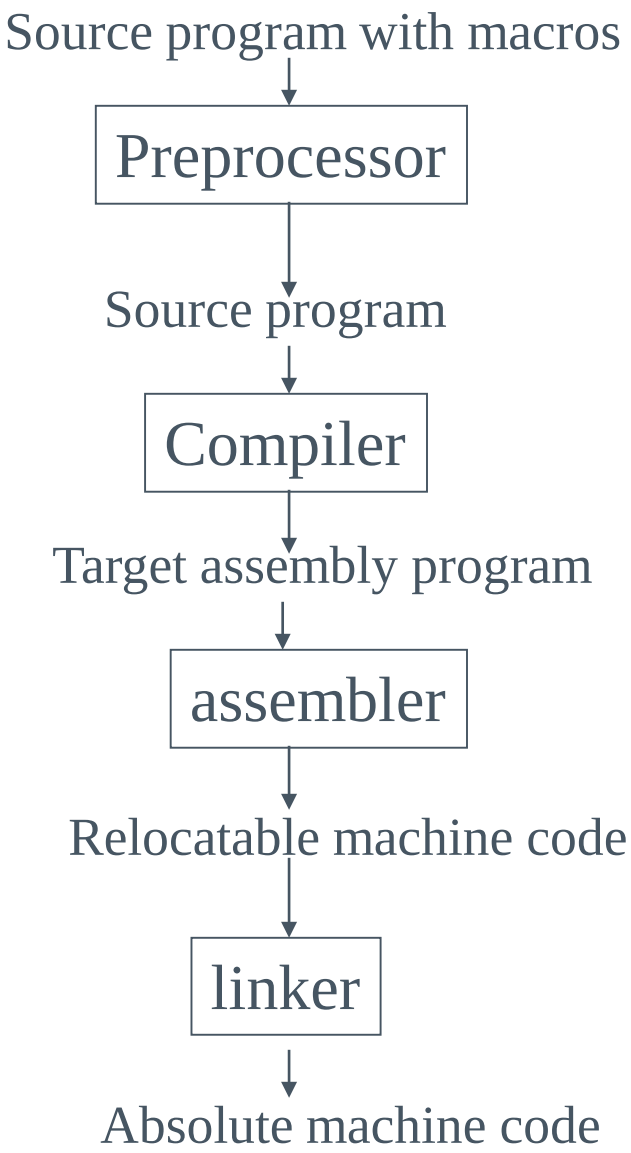


Compilation vs. Interpretation

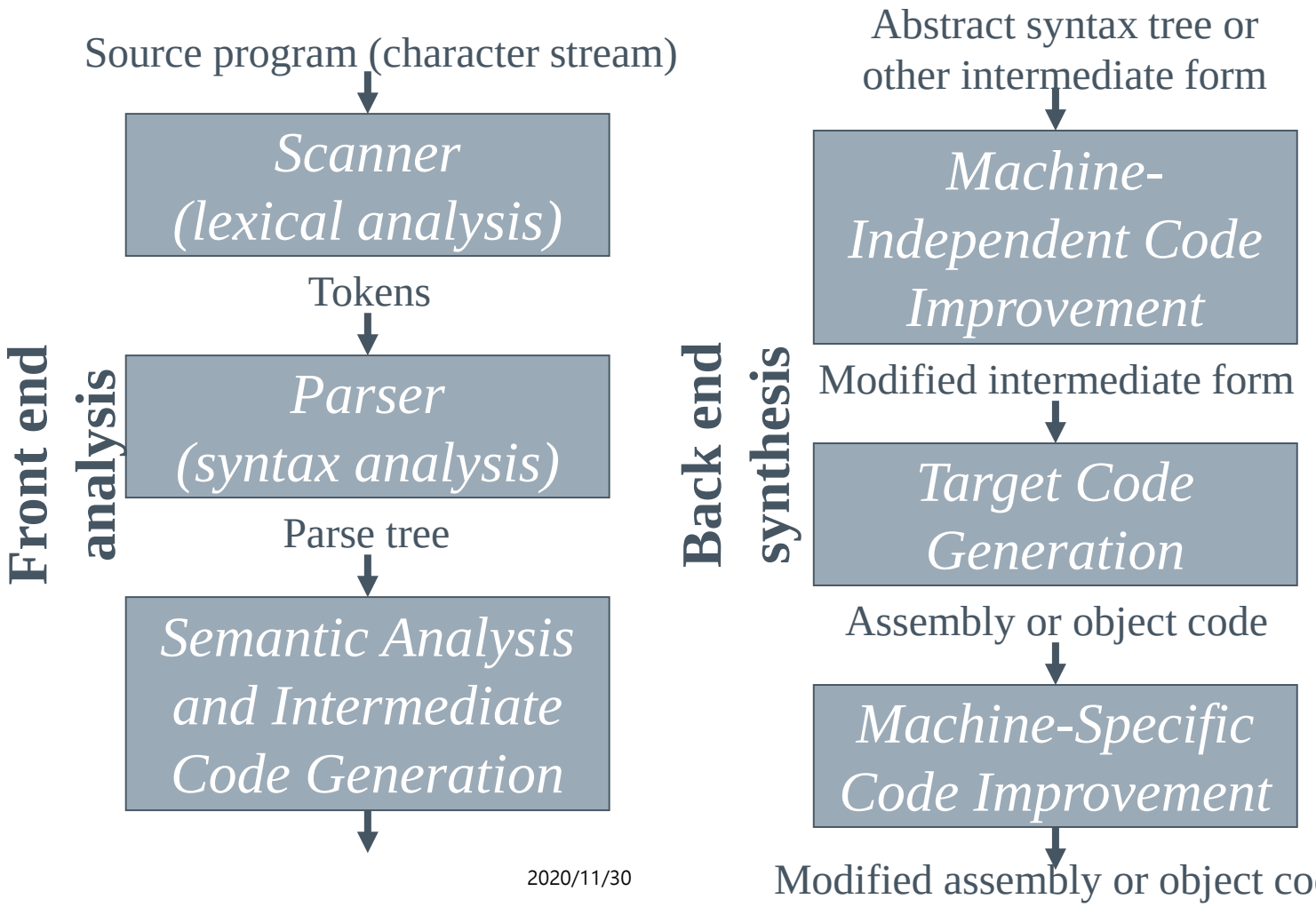
- Compilers exist for some interpreted languages, but they aren't pure:
  - Selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source.
  - Interpretation of parts of code, at least, is still necessary for reasons above.
- Unconventional compilers
  - Text formatters.
  - Silicon compilers.
  - Query language processors.

A Typical Compilation Process

- Try g++ with –v, -E, -S flags



Compiler Front- and Back-end



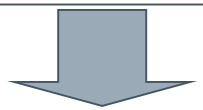
An Overview of Compilation

- ***Scanning***:
  - Divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow.
  - We can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages.
  - You can design a parser to take characters instead of tokens as input, but it isn't pretty.
  - Scanning is recognition of a *regular language*, e.g., via DFA.

Scanner: Lexical Analysis

- Lexical analysis breaks up a program into tokens
  - Grouping characters into non-separatable units (tokens)
  - Changing a stream to characters to a stream of tokens

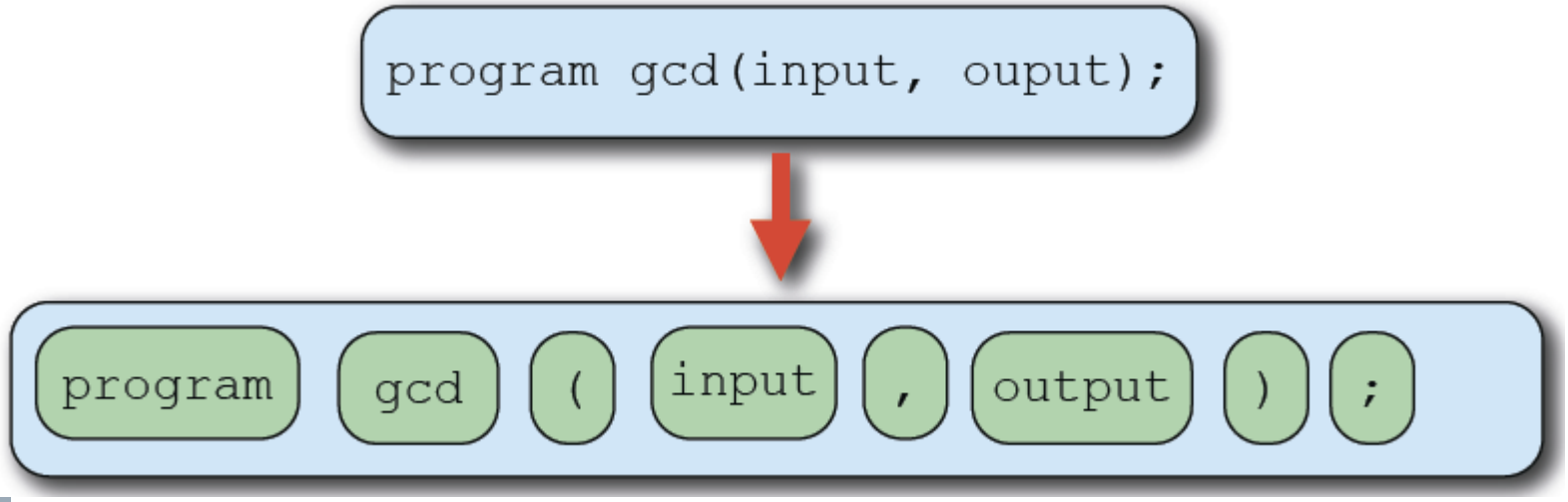
```
program gcd (input, output);
var i, j : integer;
begin
  read (i, j);
  while i <> j do
    if i > j then i := i - j else j := j - i;
  writeln (i)
end.
```



program	gcd	(	input	,	output	)	;
var	i	,	j	:	**integer**	;	**begin**
read	(	i	,	j	)	;	**while**
i	<>	j	**do**	**if**	i	>	j
then	i	:=	i	-	j	**else**	j
:=	i	-	i	;	writeln	(	i
)	**end**	.					

Scanner: Lexical Analysis

› Recognize structures without regard to meaning and groups them into tokens.



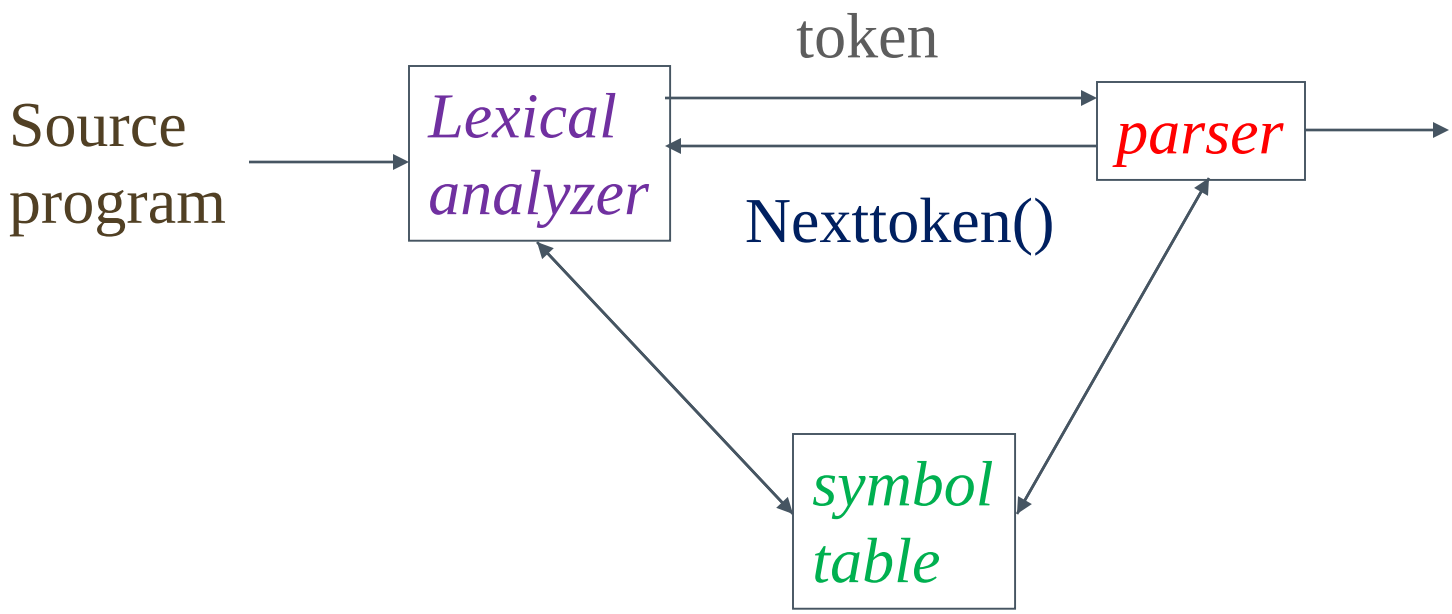
› The purpose of the scanner is to simplify the parser by reducing the size of the input.

Lexical Analysis

- Lexical analyzer: reads input characters and produces a sequence of tokens as output (nexttoken()).
  - Trying to understand each element in a program.
  - Token: a group of characters having a collective meaning.
    - const pi = 3.14159;

    - Token 1: (const, -)
    - Token 2: (identifier, ‘pi’)
    - Token 3: (=, -)
    - Token 4: (realnumber, 3.14159)
    - Token 5: (;, -)

Interaction of Lexical Analyzer with Parser



An Overview of Compilation

- Parsing is recognition of a context-free language, e.g., via push down automata (PDA).
  - Parsing discovers the "context free" structure of the program.
  - Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual).

Parser: Syntax Analysis

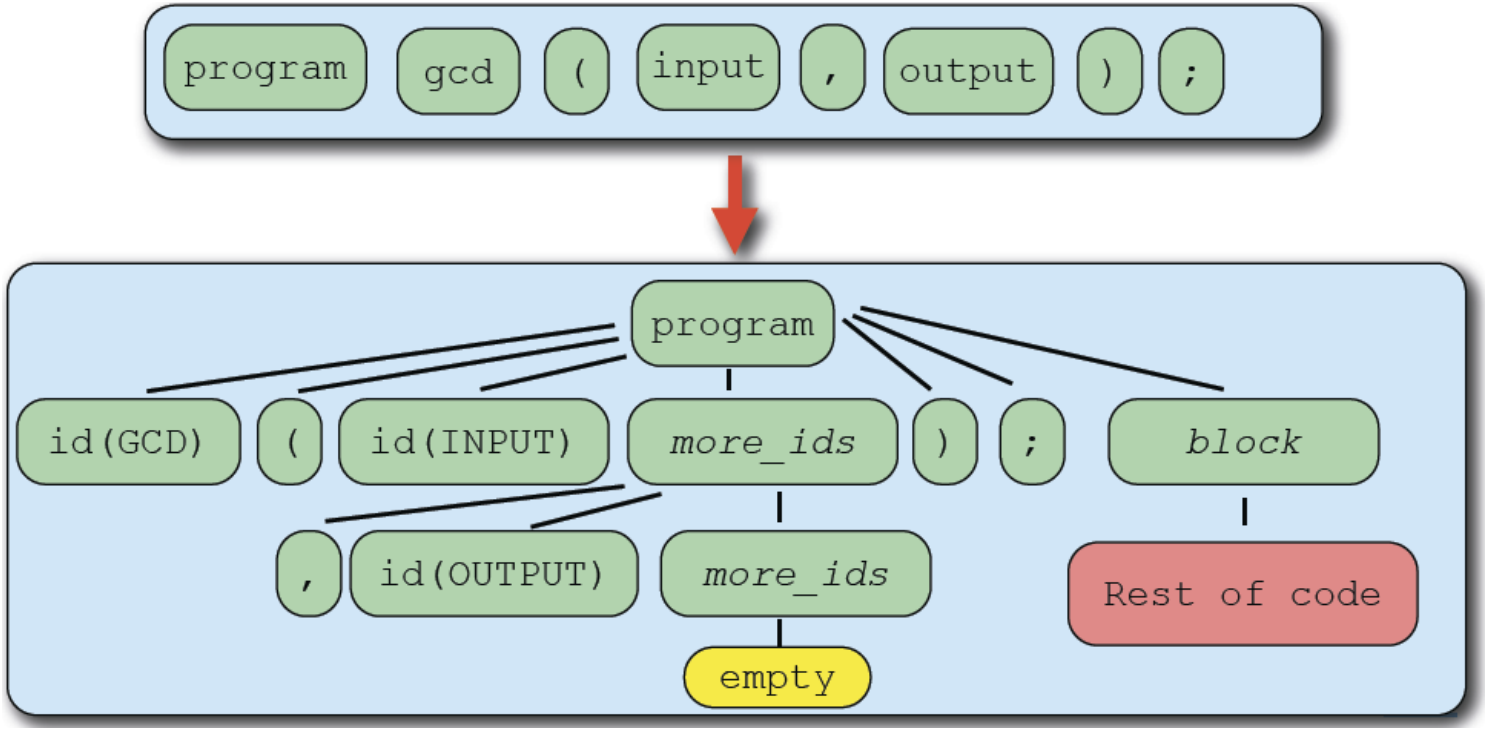
- Checks whether the token stream meets the grammatical specification of the language and generates the syntax tree.
  - A syntax error is produced by the compiler when the program does not meet the grammatical specification.
  - For grammatically correct program, this phase generates an internal representation that is easy to manipulate in later phases
    - Typically a syntax tree (also called a parse tree).
- A grammar of a programming language is typically described by a context free grammar, which also defines the structure of the parse tree.

Parser: Syntax Analysis

- The Syntax analysis catches all malformed statements.
- The parse tree is sometimes called a concrete syntax tree because it contains how all tokens are derived.
- Much of this information is extraneous for the “meaning” of the code (e.g., the only purpose of “;” is to end a statement).

Parser: Syntax Analysis

› Parsing organizes the tokens into a context-free grammar (i.e., syntax).

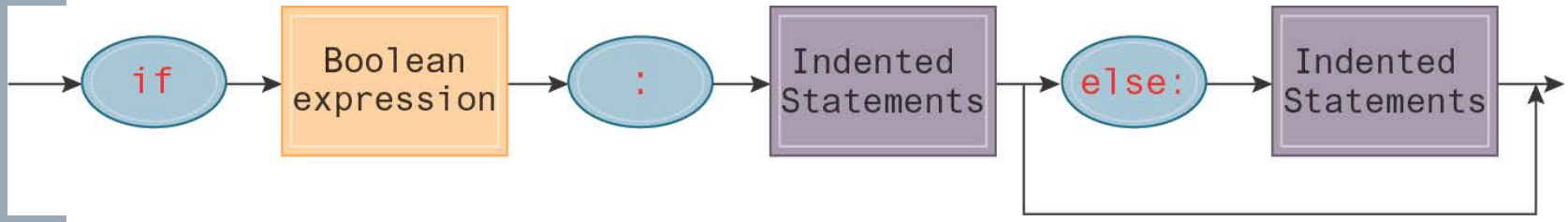


Context-Free Grammars

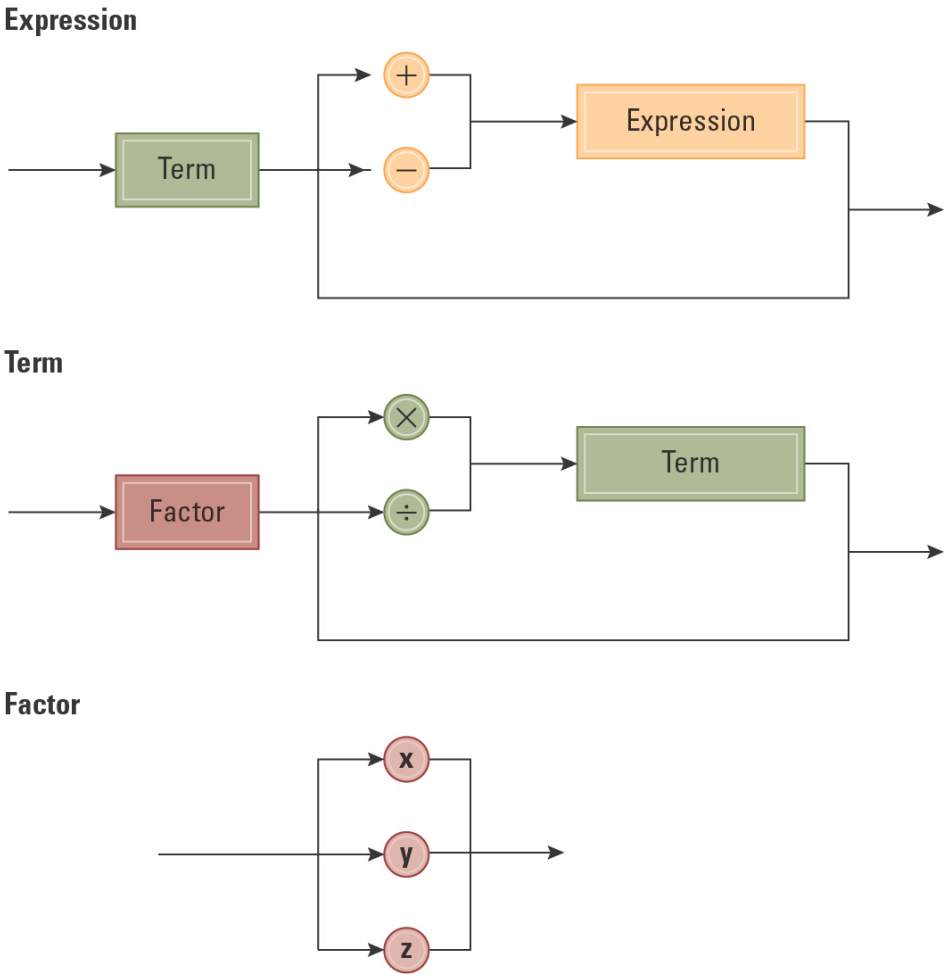
- A context-free grammar defines the syntax of a programming language
- The syntax defines the syntactic categories for language constructs
  - Statements, Expressions, Declarations
- Categories are subdivided into more detailed categories
  - A Statement is a
    - For-statement
    - If-statement
    - Assignment

```
<statement>      ::= <for-statement> | <if-statement> | <assignment>
<for-statement>  ::= for ( <expression> ; <expression> ; <expression> ) <statement>
<assignment>     ::= <identifier> := <expression>
```

A Syntax Diagram of Python's `if-then-else` Statement



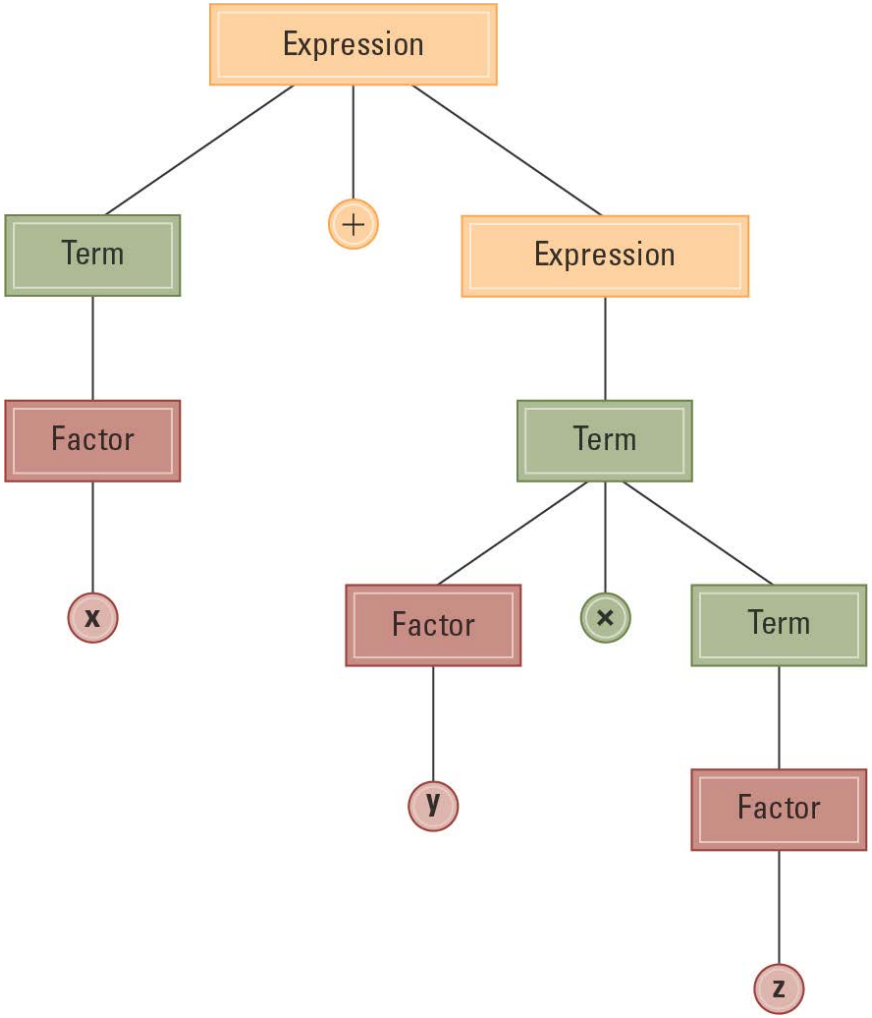
Syntax Diagrams Describing the Structure of a Simple Algebraic Expression



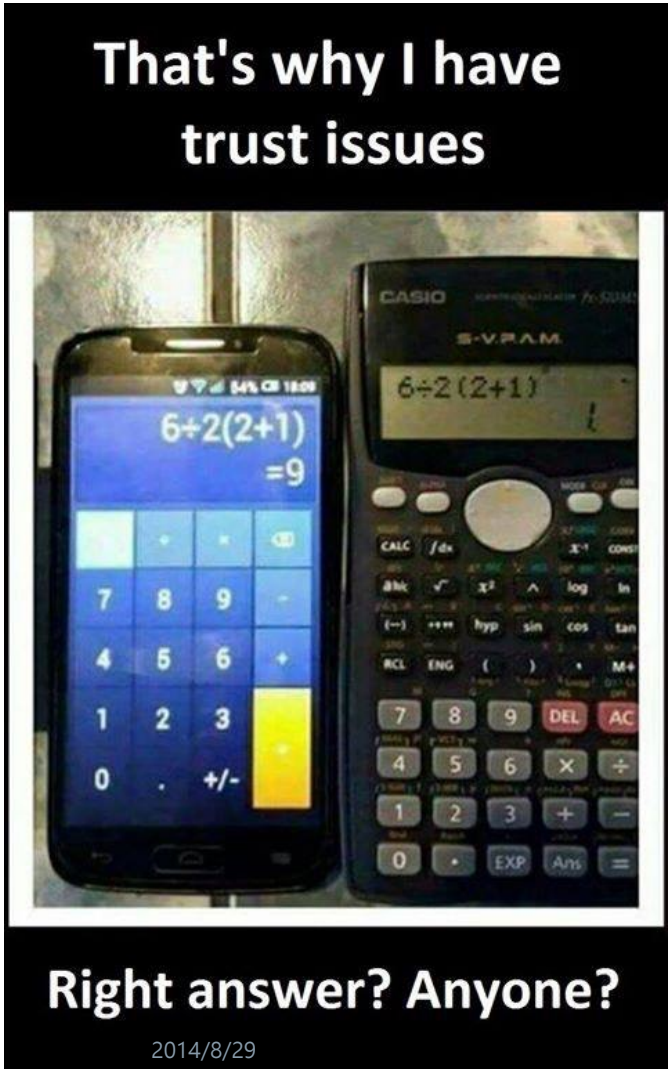
Syntax Diagrams Describing the Structure of a Simple Algebraic Expression

```
Expression -> Term Expression | Term OP1 Expression

Term -> Factor Term | Factor OP2 Term

Term -> x | y | z

OP1 -> + | -

OP2 -> * | /
```

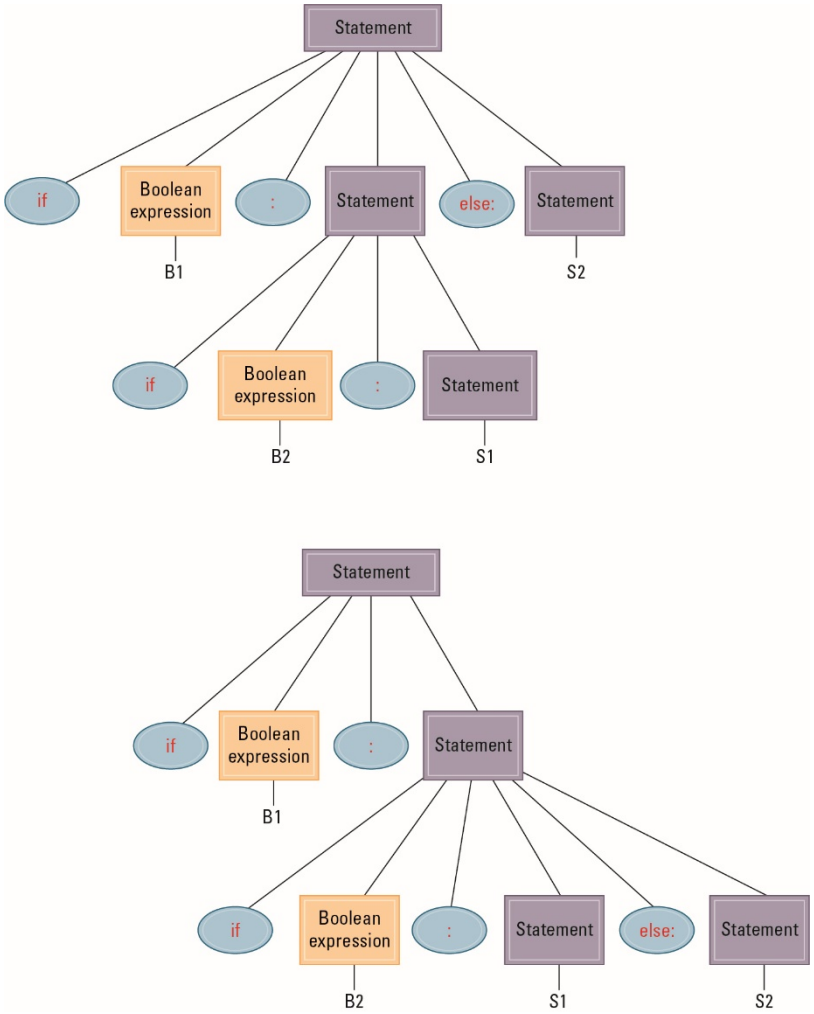
The Parse Tree for the String x + y * z Based on the Syntax Diagrams



Operator precedence



Two Distinct Parse Trees for the Statement: `if B1 then if B2 then S1 else S2`



C99 language grammar

- Total **69** production rules with **201** top alternatives and **806** symbols.
- Vocabulary: **155** = **73** nonterminals + **82** terminals + **0** labels + **0** markers.
- Total **82** terminal symbols:
  - **36** keywords
    (*"typedef"*, *"extern"*, *"static"*[5], *"auto"*, *"register"*, *"void"*, *"char"*, *"short"*, *"int"*, *"long"*, *"float"*, *"double"*, *"signed"*, *"unsigned"*, *"_Bool"*, *"_Complex"*, *"struct"*, *"union"*, *"const"*, *"restrict"*, *"volatile"*, *"sizeof"*[2], *"enum"*[3], *"inline"*, *"case"*, *"default"*, *"if"*[2], *"else"*, *"switch"*, *"while"*[2], *"do"*, *"for"*[2], *"goto"*, *"continue"*, *"break"*, *"return"*)
  - **46** signs
    ("{"[8], "}"[8], ":"[11], ","[12], ":"[5], "*"[6], "("[18], ")"[18], "["[10], "]"[10], "?", "||", "&&", "|", "^", "&"[2], "=="[2], "!=", "<", ">", "<=", ">=", "<<", ">>", "+"[2], "-"[2], "/", "%", "++"[2], "--"[2], "."[2], "->", "...", "="[4], "~", "!", "*=", "/=", "%=", "+=", "-=", "<<=", ">>=", "&=", "^=", "|=").

https://gist.github.com/codebrainz/2933703

C99 language grammar

```
declaration-specifiers ::=
        storage-class-specifier declaration-specifiers?
        type-specifier declaration-specifiers?
        type-qualifier declaration-specifiers?
        function-specifier declaration-specifiers?
```

```
storage-class-specifier ::=
        "typedef"
        "extern"
        "static"
        "auto"
        "register"
```

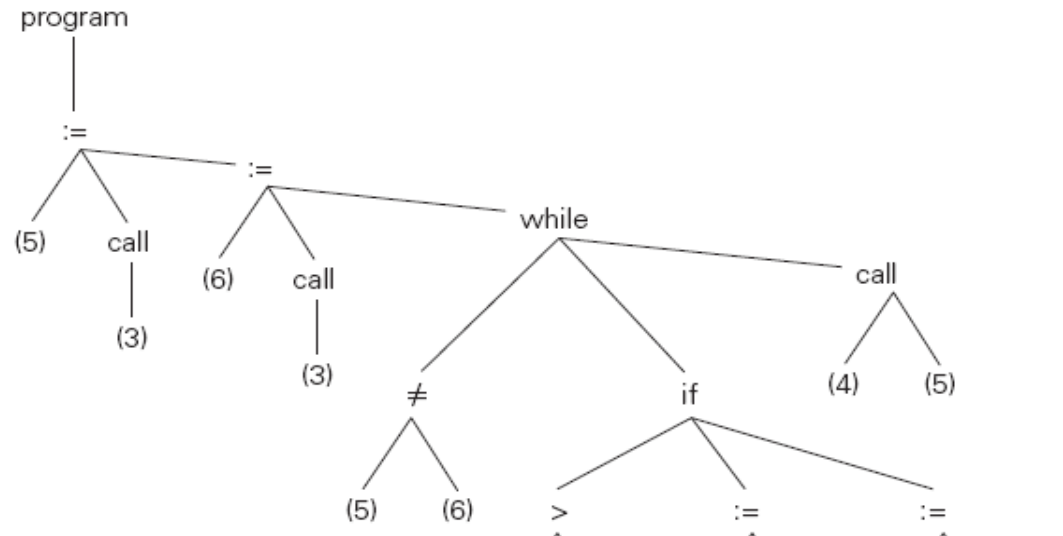
```
type-specifier ::=
        "void"
        "char"
        "short"
        "int"
        "long"
        "float"
        "double"
        "signed"
        "unsigned"
        "_Bool"
        "_Complex"
        struct-or-union-specifier
        enum-specifier
        typedef-name
```

A Pascal Example

```
program gcd(input, output);
var i, j: integer;
begin
   read(i,j); // get i & j from read
   while i<>j do
      if i>j then i := i-j
      else j := j-1;
   writeln(i)
end.
```

Parsing Examples

- Syntax Tree
  - GCD Program Parse Tree



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

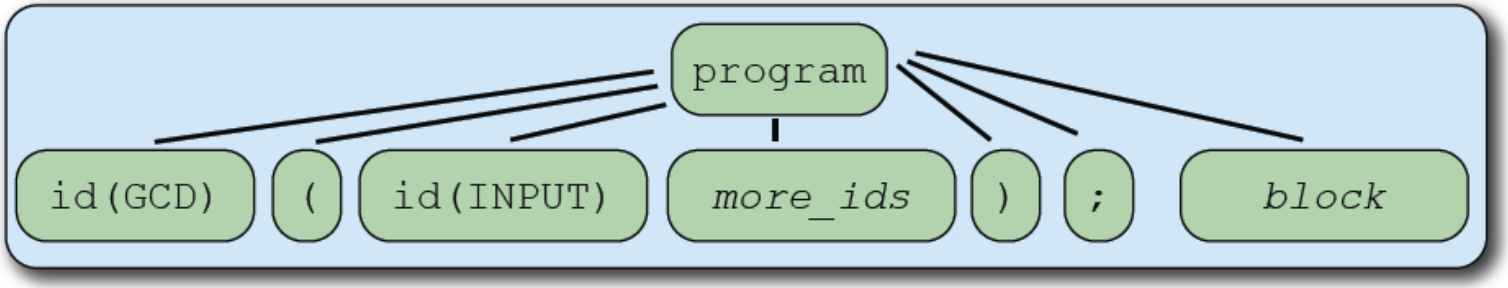
An Overview of Compilation

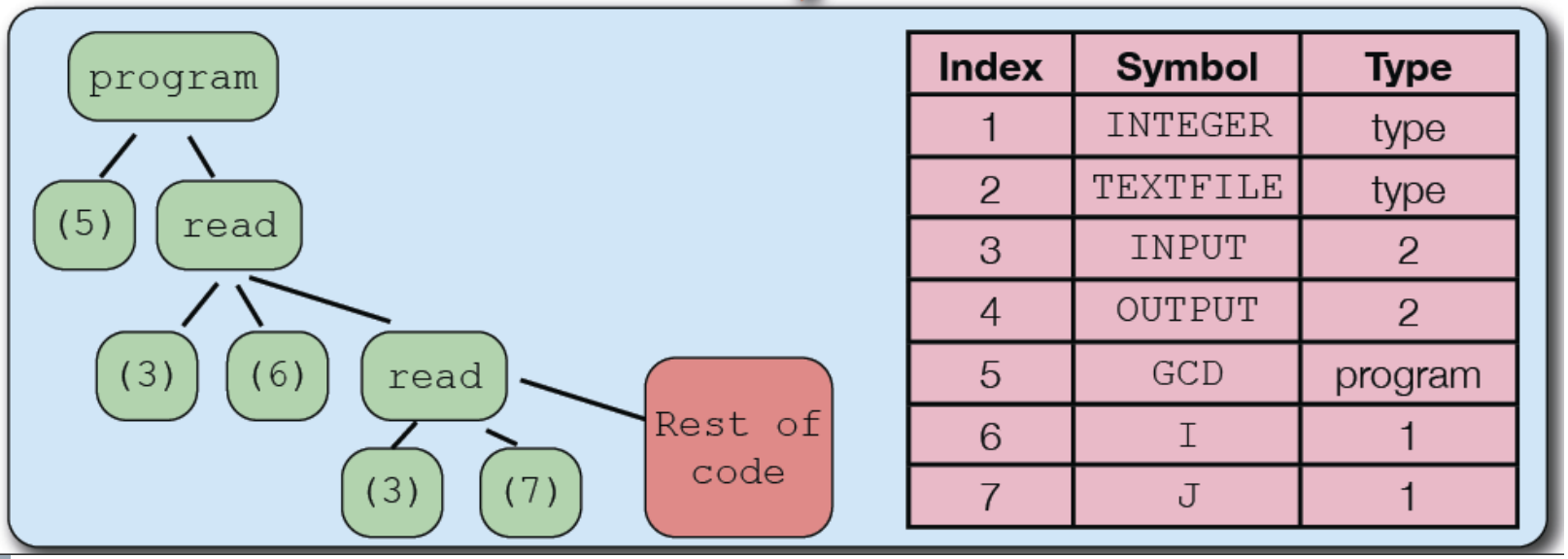
- Semantic analysis is the discovery of meaning in the program.
  - The compiler actually does what is called STATIC semantic analysis. That's the meaning that can be figured out at compile time.
  - Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's DYNAMIC semantics.

Semantic Analysis

- Semantic analysis discovers the **meaning of a program** by creating an **abstract syntax tree** that removes "extraneous" tokens.
- To do this, the analyzer builds & maintains a **symbol table** to map identifiers to information known about it. (i.e., scope, internal structure, etc...)
- By using the symbol table, the semantic analyzer can **catch problems not caught by the parser**.
  - Identifiers are declared before used
  - Subroutine calls provide correct number and type of arguments.

Semantic Analysis





Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Semantic Analysis

- Not all semantic rules can be checked at compile time.
  - Those that can are called **static** semantics of the language.
  - Those that cannot are called **dynamic** semantics of the language.
    - Arithmetic operations **do not overflow**.
    - Array subscripts expressions **lie within the bounds of the array**.

Semantic Analysis

- Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree.
- A program without grammatical errors may not always be correct program.
  - `pos = init + rate * 60`
  - What if `pos` is a class while `init` and `rate` are integers?
  - This kind of errors cannot be found by the parser
  - Semantic analysis finds this type of error and ensure that the program has a meaning.

Semantic Analysis

- Static semantic checks (done by the compiler) are performed at compile time
  - Type checking
  - Every variable is declared before used
  - Identifiers are used in appropriate contexts
  - Check subroutine call arguments
  - Check labels

Semantic Analysis

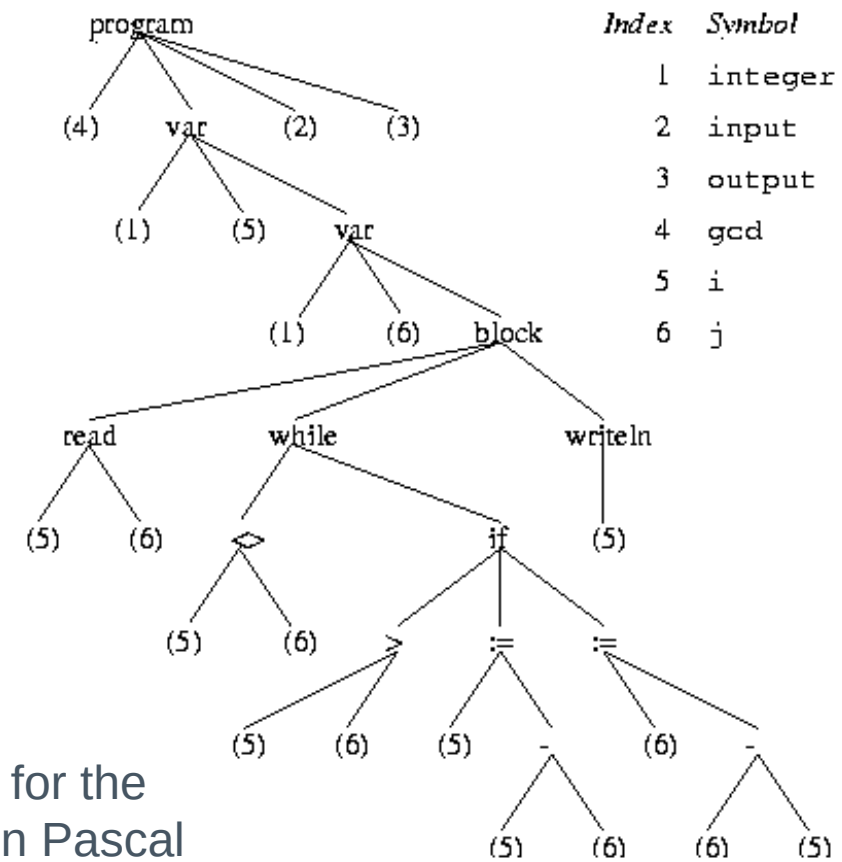
- Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks
  - Array subscript values are within bounds
  - Arithmetic errors, e.g. division by zero
  - Pointers are not dereferenced unless pointing to valid object
  - A variable is used but hasn't been initialized
  - When a check fails at run time, an exception is raised

An Overview of Compilation

- ***Intermediate form*** (IF) done after semantic analysis (*if* the program passes all checks)
  - IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory).
  - They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
  - Many compilers actually move the code through more than one IF.

Code Generation and Intermediate Code Forms

- A typical intermediate form of code produced by the semantic analyzer is an abstract syntax tree (AST)
- The AST is annotated with useful information such as pointers to the symbol table entry of identifiers



Example AST for the gcd program in Pascal

An Overview of Compilation

- Optimization takes an intermediate-code program and produces another one that does the same thing faster, or in less space.
  - The term is a misnomer; we just improve code
  - The optimization phase is optional.
- Code generation phase produces assembly language or (sometime) relocatable machine language.

Optimization

- The process so far will produce **correct code**, but it **may not be fast**.
- Optimization will adjust the code to improve performance.
  - A possible *machine-independent optimization* would be to keep the variables *i* and *j* in **registers throughout the main loop**.
  - A possible *machine-specific optimization* would be to assign the variables *i* and *j* to **specific registers**.

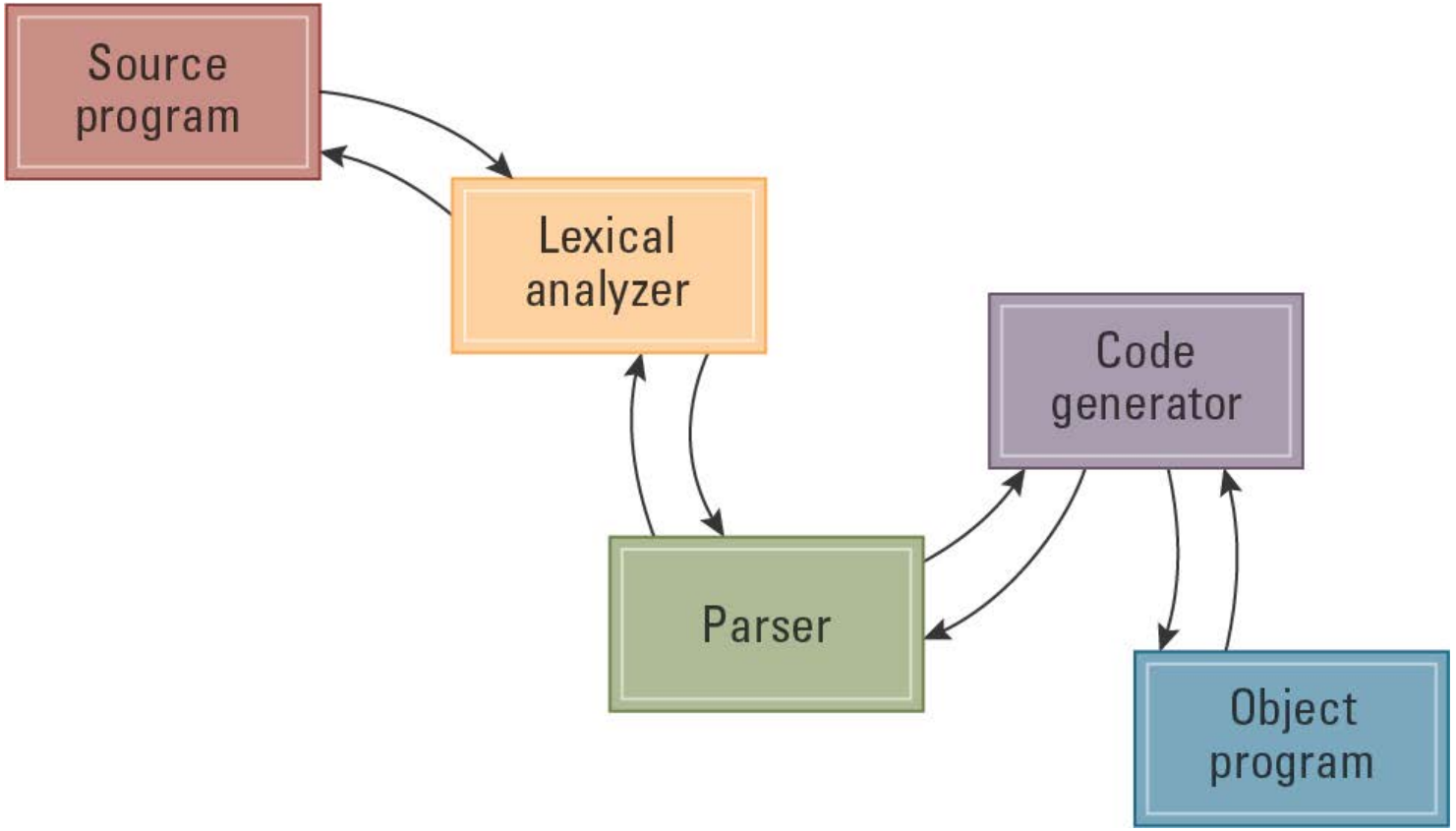
Target Code Generation and Optimization

- From the machine-independent form assembly or object code is generated by the compiler.

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

- This machine-specific code is optimized to exploit specific hardware features.

An Object-oriented Approach to the Translation Process

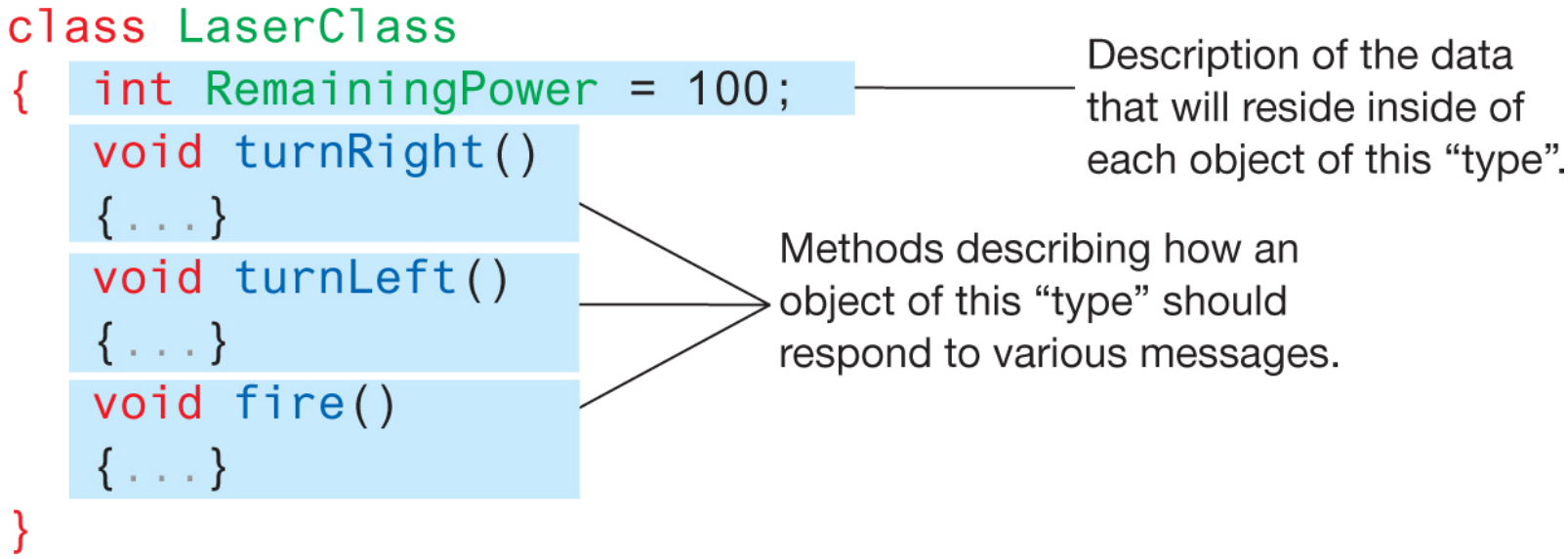


Objects and Classes

- **Object:** Active program unit containing both data and procedures.
- **Class:** A template from which objects are constructed.

- An object is called an **instance** of the class.

The Structure of a Class Describing a Laser Weapon in a Computer Game



Components of an Object

- **Instance Variable:** Variable within an object.
  - Holds information within the object.
- **Method:** Procedure within an object.
  - Describes the actions that the object can perform.
- **Constructor:** Special method used to initialize a new object when it is first constructed.
- **Destructors:** Special method used to de-initialize a existing object when it is removed.

A Class with a Constructor

```
class LaserClass
{ int RemainingPower;
  LaserClass(InitialPower)
  { RemainingPower = InitialPower;
  }
  void turnRight()
  {...}
  void turnLeft()
  {...}
  void fire()
  {...}
}
```

Constructor assigns a value to RemainingPower when an object is created.

Object Integrity

- **Encapsulation:** A way of restricting access to the internal components of an object.
  - Private
  - Public
  - (Protected)
  - (Friend)

Our `LaserClass` Definition Using Encapsulation as it Would Appear in a Java or C# Program

Components in the class are designated public or private depending on whether they should be accessible from other program units.

```
class LaserClass
{private int RemainingPower;
 public LaserClass (InitialPower)
 {RemainingPower = InitialPower;
 }
 public void turnRight (  )
 {...}
 public void turnLeft (  )
 {...}
 public void fire (  )
 {...}
}
```

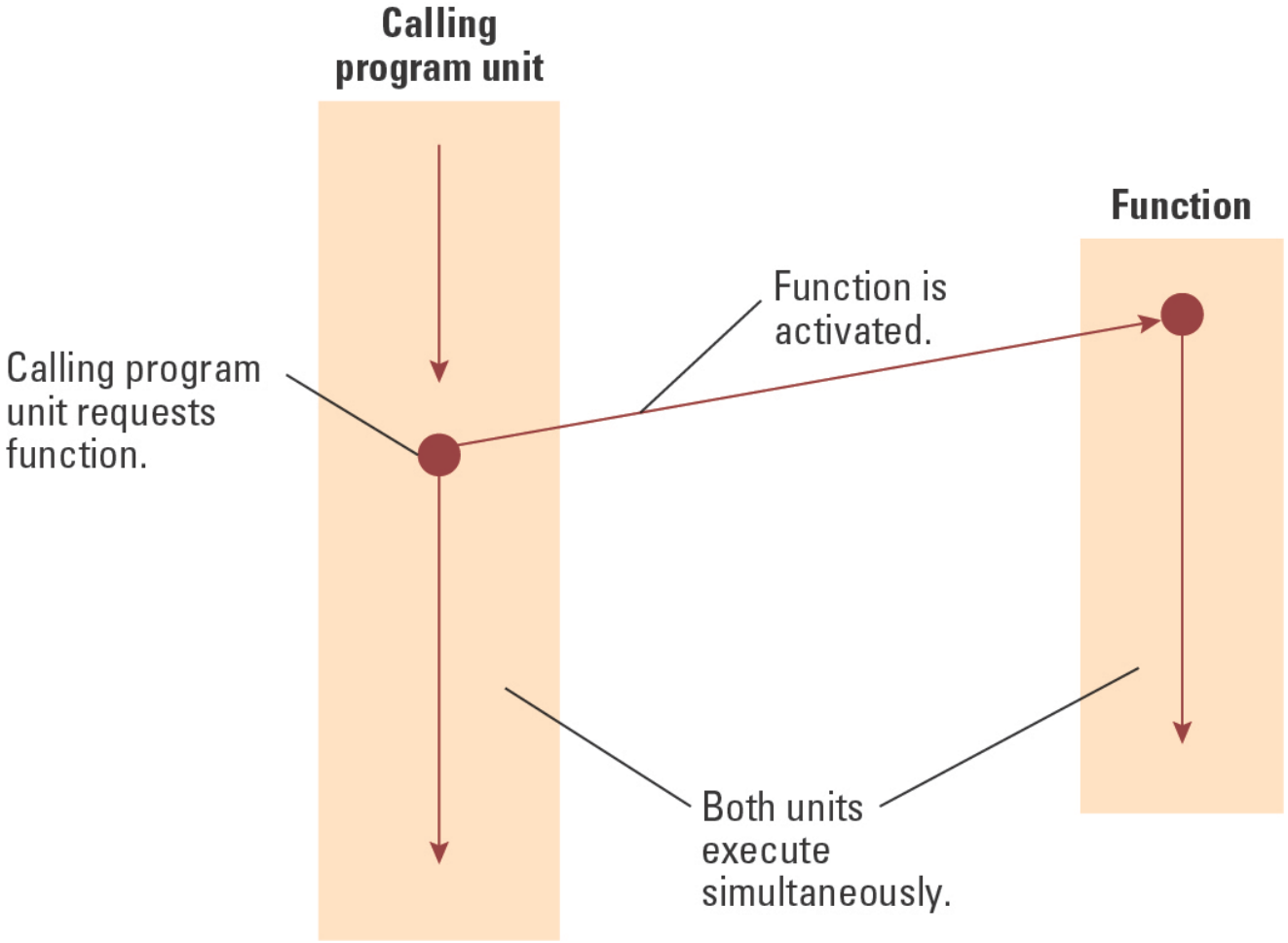
Additional Object-oriented Concepts

- **Inheritance:** Allows new classes to be defined in terms of previously defined classes.
- **Polymorphism:** Allows method calls to be interpreted by the object that receives the call.

Programming Concurrent Activities

- **Parallel** (or **concurrent**) **processing:** simultaneous execution of multiple processes.
  - True concurrent processing requires multiple CPUs.
    - Or called cores, threads.
  - Can be simulated using time-sharing with a single CPU.

Spawning Threads (Processes)



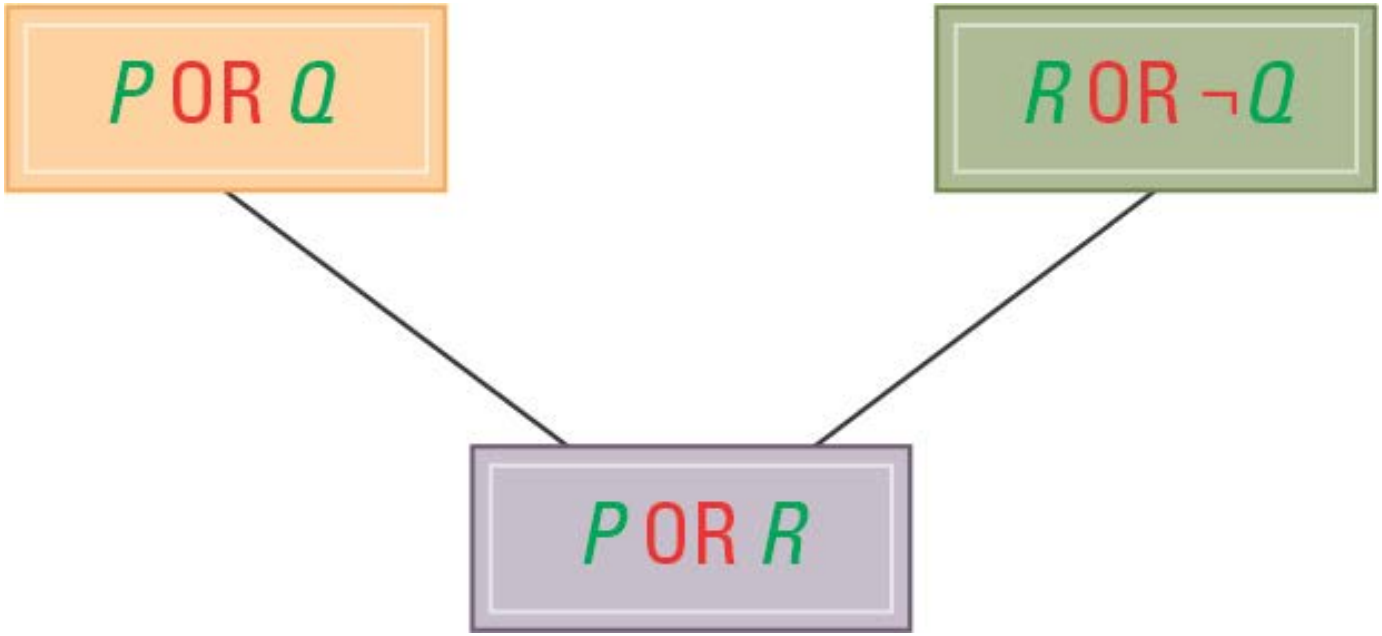
Controlling Access to Data

- **Mutual Exclusion:** A method for ensuring that data can be accessed by only one process at a time.
- **Monitor:** A data item augmented with the ability to control access to itself.

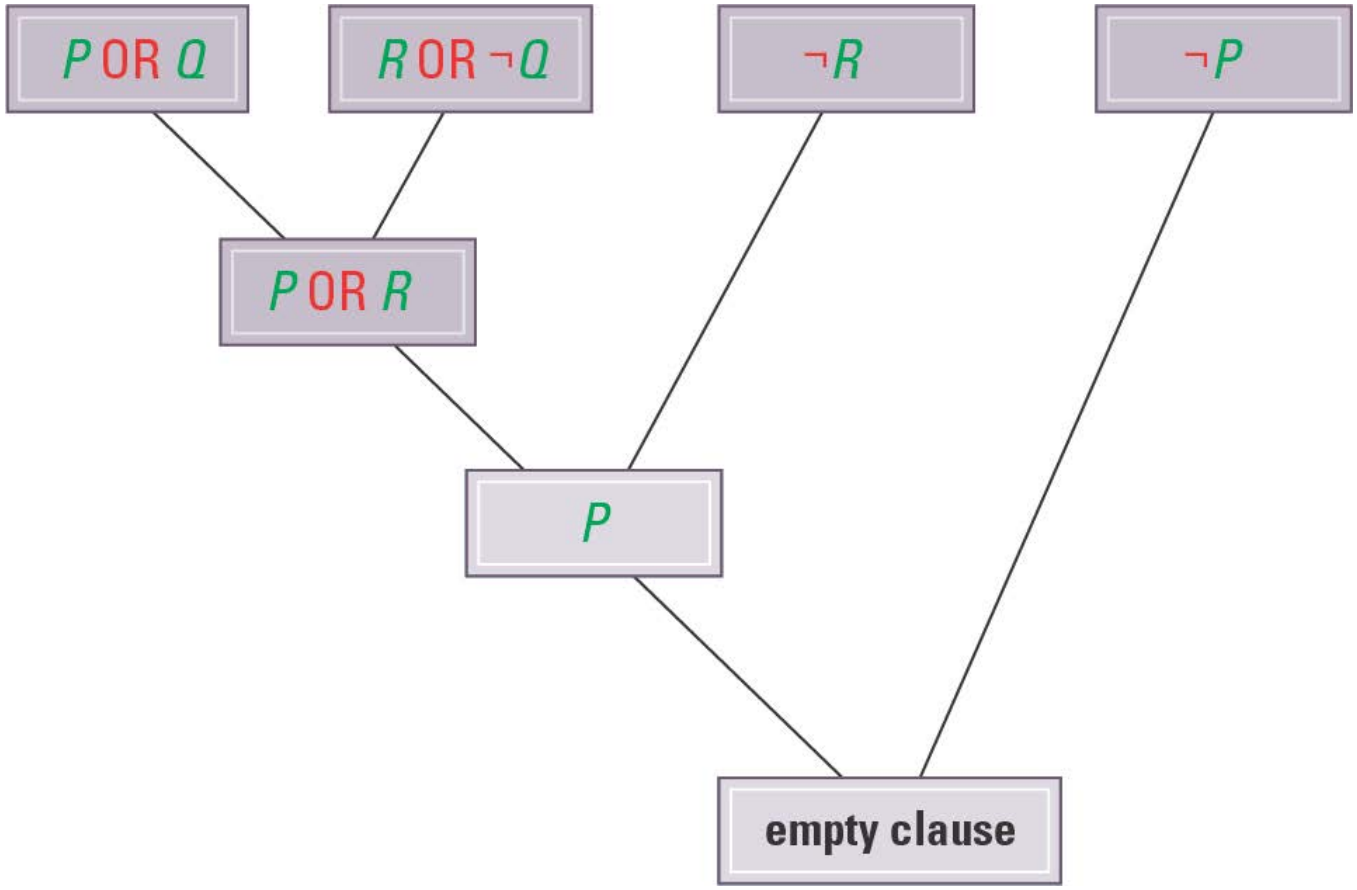
Declarative Programming

- **Resolution:** Combining two or more statements to produce a new statement (that is a logical consequence of the originals).
  - Example: `(P OR Q) AND (R OR ¬Q)` resolves to `(P OR R)`
  - **Resolvent:** A new statement deduced by resolution
  - **Clause form:** A statement whose elementary components are connected by the Boolean operation `OR`.
- **Unification:** Assigning a value to a variable so that two statements become "compatible."

Resolving the Statements (P OR Q) and (R OR $\neg Q$) to Produce (P OR R)



Resolving the Statements (P OR Q), (R OR $\neg Q$), $\neg$R, and $\neg P$



Prolog

- **Fact:** A Prolog statement establishing a fact
  - Consists of a single predicate
  - Form: *predicateName*(*arguments*).
    - Example: `parent(bill, mary).`
- **Rule:** A Prolog statement establishing a general rule
  - Form: *`conclusion :- premise`*.
    - `:-` means “if”
  - Example: `wise(X) :- old(X).`
  - Example: `faster(X,Z) :- faster(X,Y), faster(Y,Z).`