COP4020 Programming Languages

Compilers and Interpreters

Robert van Engelen & Chris Lacher

Overview

- Common compiler and interpreter configurations
- Virtual machines
- Integrated development environments
- Compiler phases
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Intermediate (machine-independent) code generation
  - Intermediate code optimization
  - Target (machine-dependent) code generation
  - Target code optimization

Compilers versus Interpreters

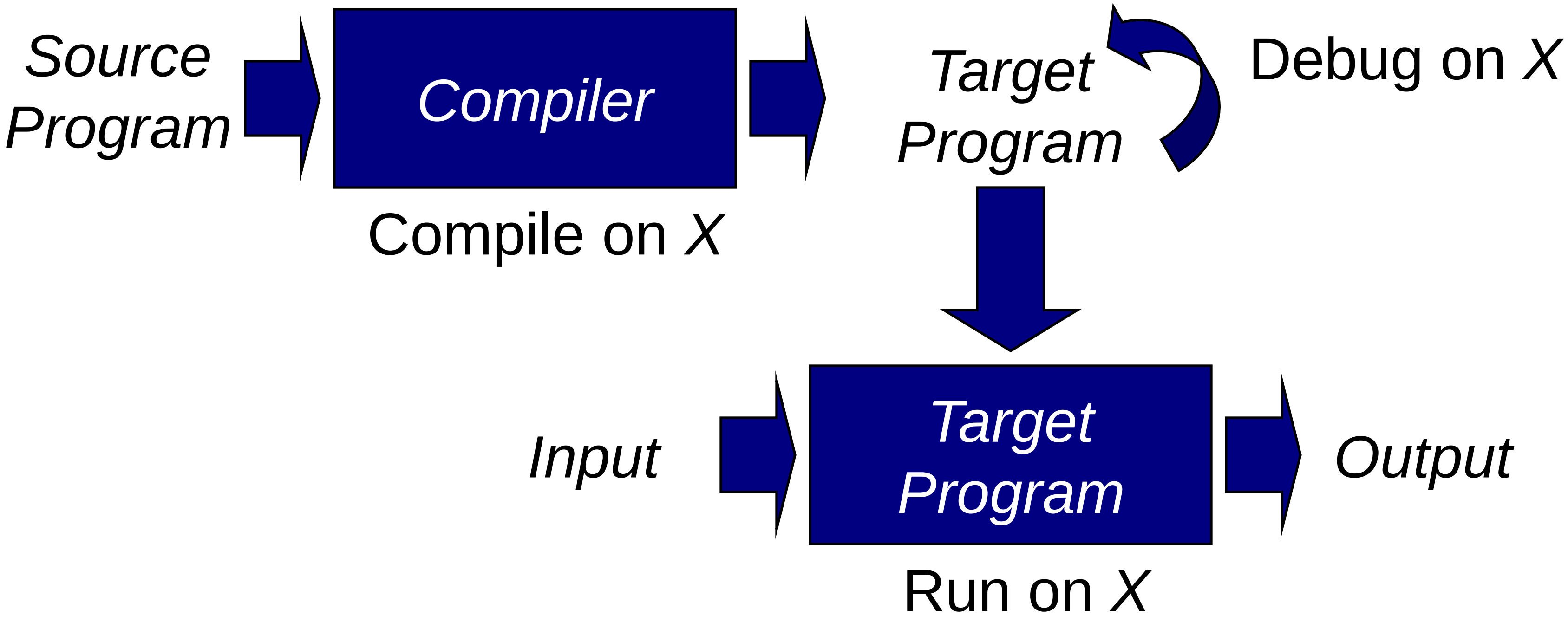
- The compiler versus interpreter implementation is often fuzzy
  - One can view an interpreter as a virtual machine that executes high-level code
  - Java is compiled to bytecode
  - Java bytecode is interpreted by the Java virtual machine (JVM) or translated to machine code by a just-in-time compiler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

Compilers versus Interpreters

- 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
- 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

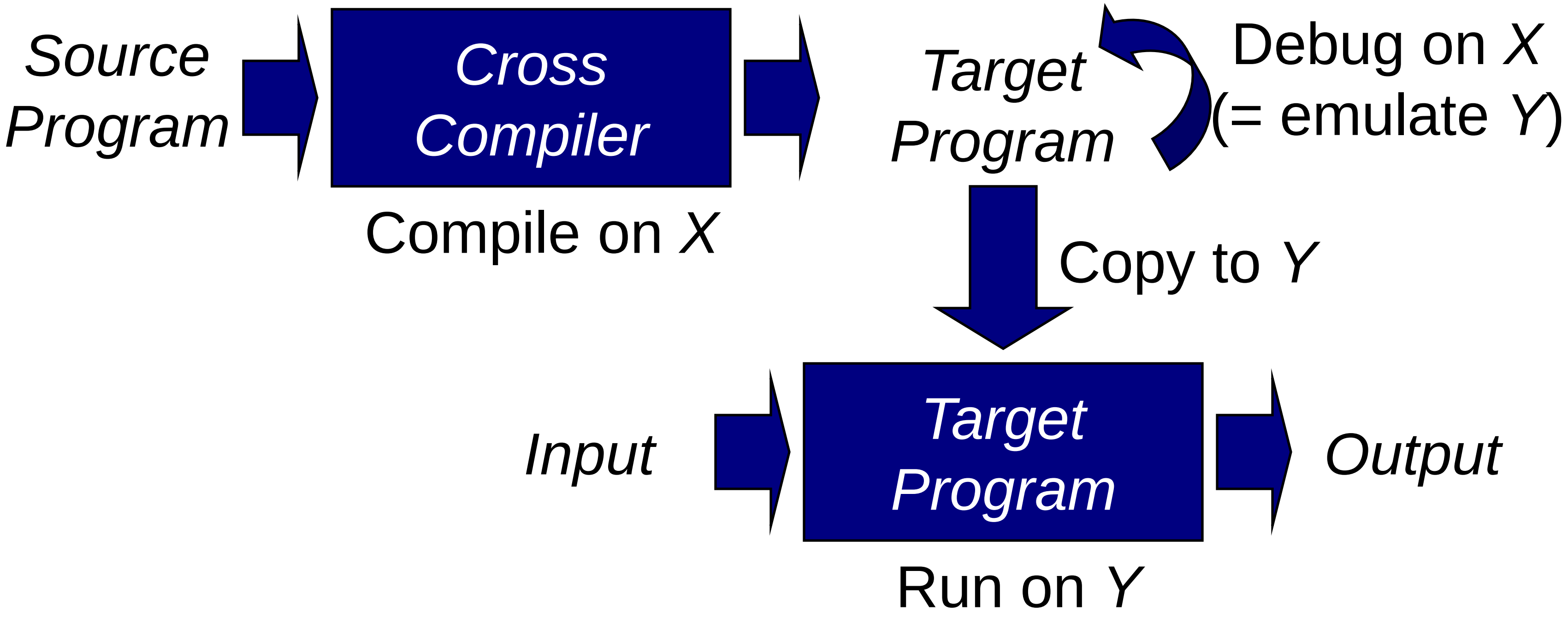
Compilation

- 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



Interpretation

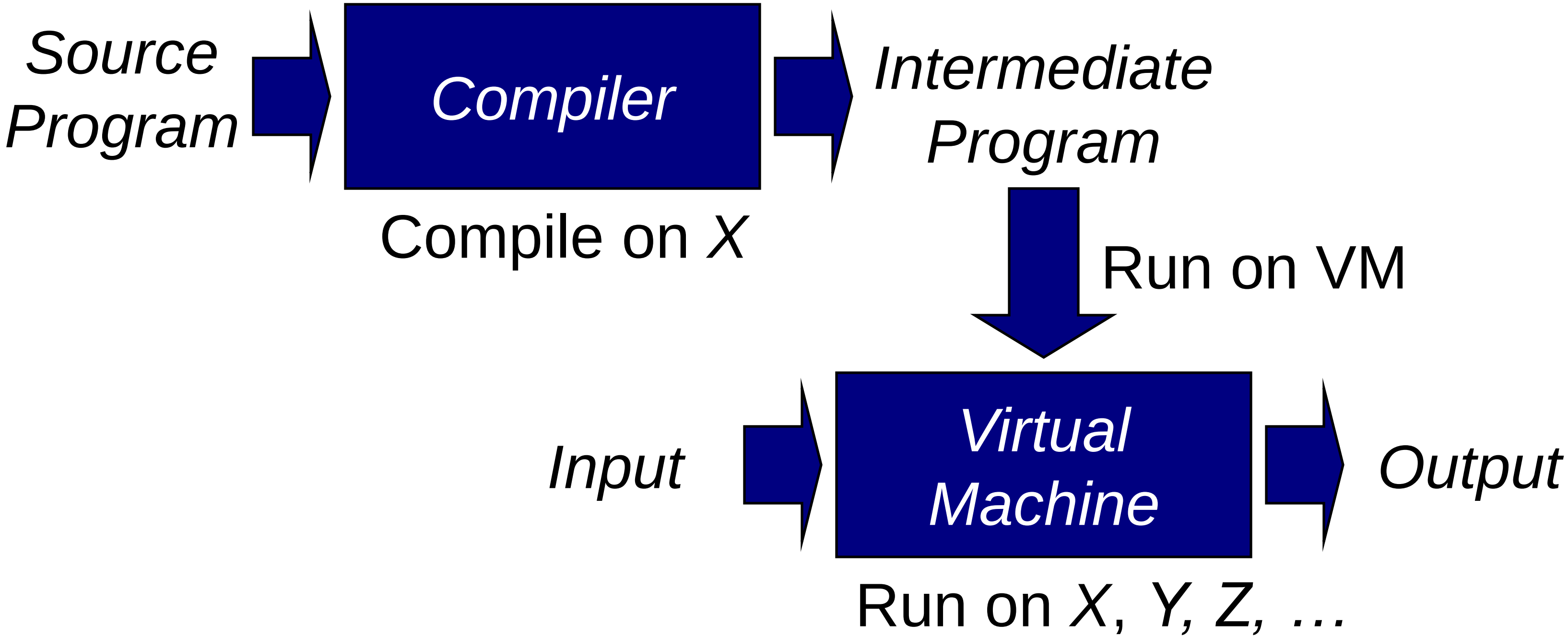
- Interpretation is the conceptual process of running high-level code by an interpreter

Virtual Machines

- A virtual machine executes an instruction stream in software
- Adopted by Pascal, Java, Smalltalk-80, C#, functional and logic languages, and some scripting languages
  - Pascal compilers generate P-code that can be interpreted or compiled into object code
  - Java compilers generate bytecode that is interpreted by the Java virtual machine (JVM)
  - The JVM may translate bytecode into machine code by just-in-time (JIT) compilation

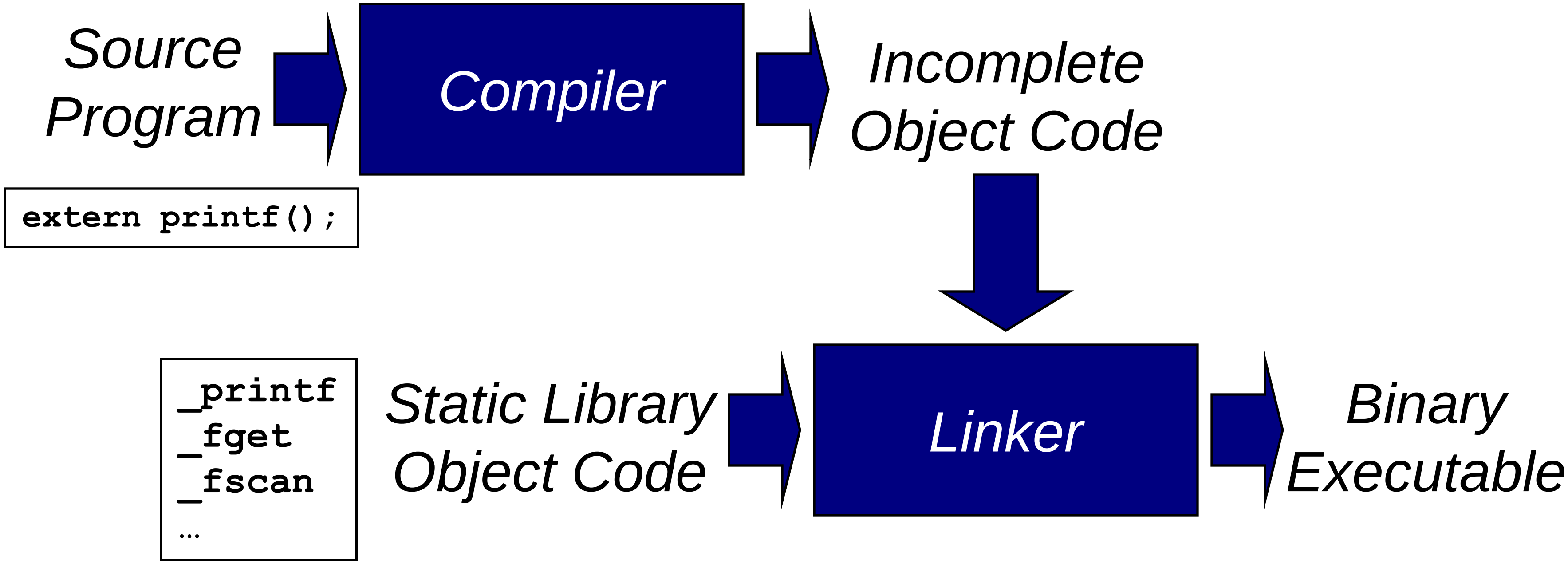
Compilation and Execution on Virtual Machines

- Compiler generates intermediate program
- Virtual machine interprets the intermediate program



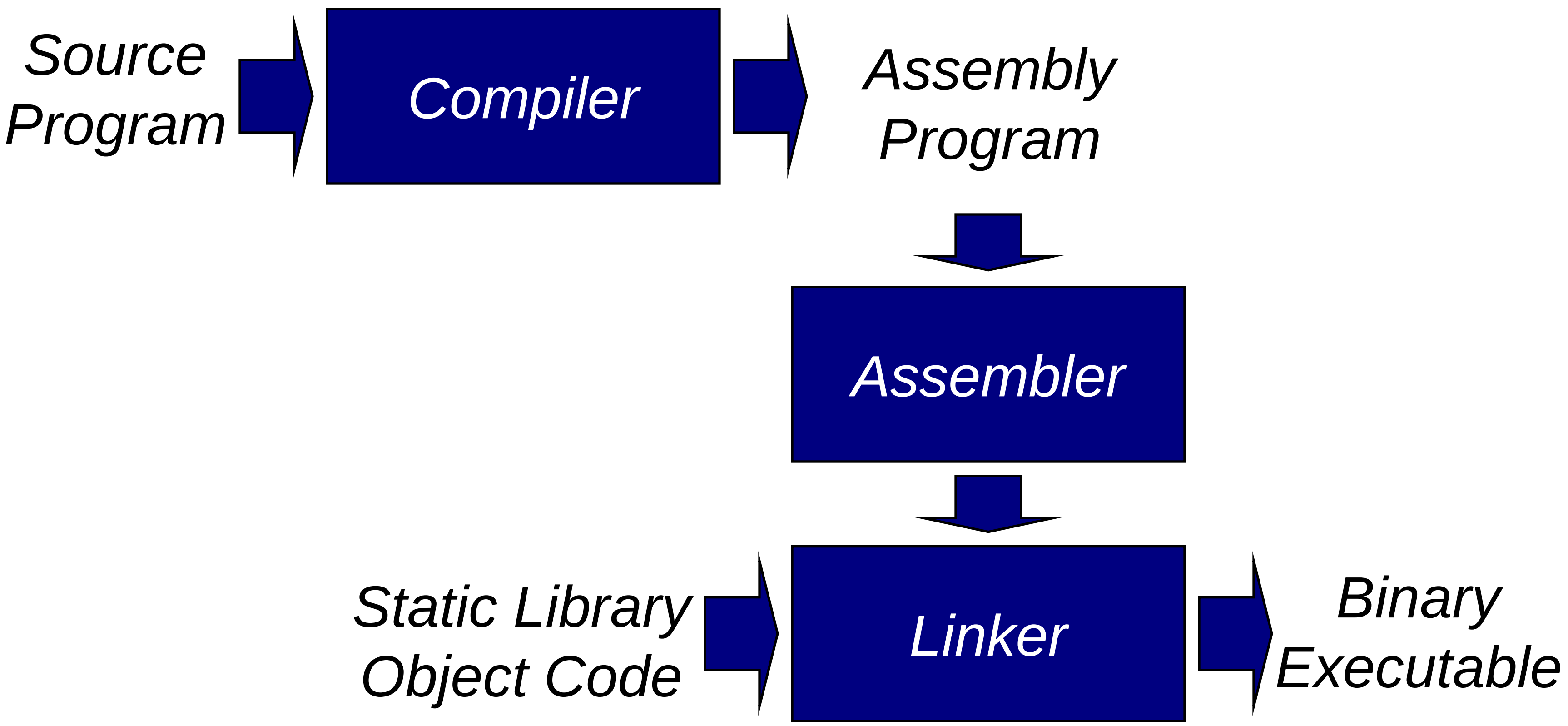
Pure Compilation and Static Linking

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



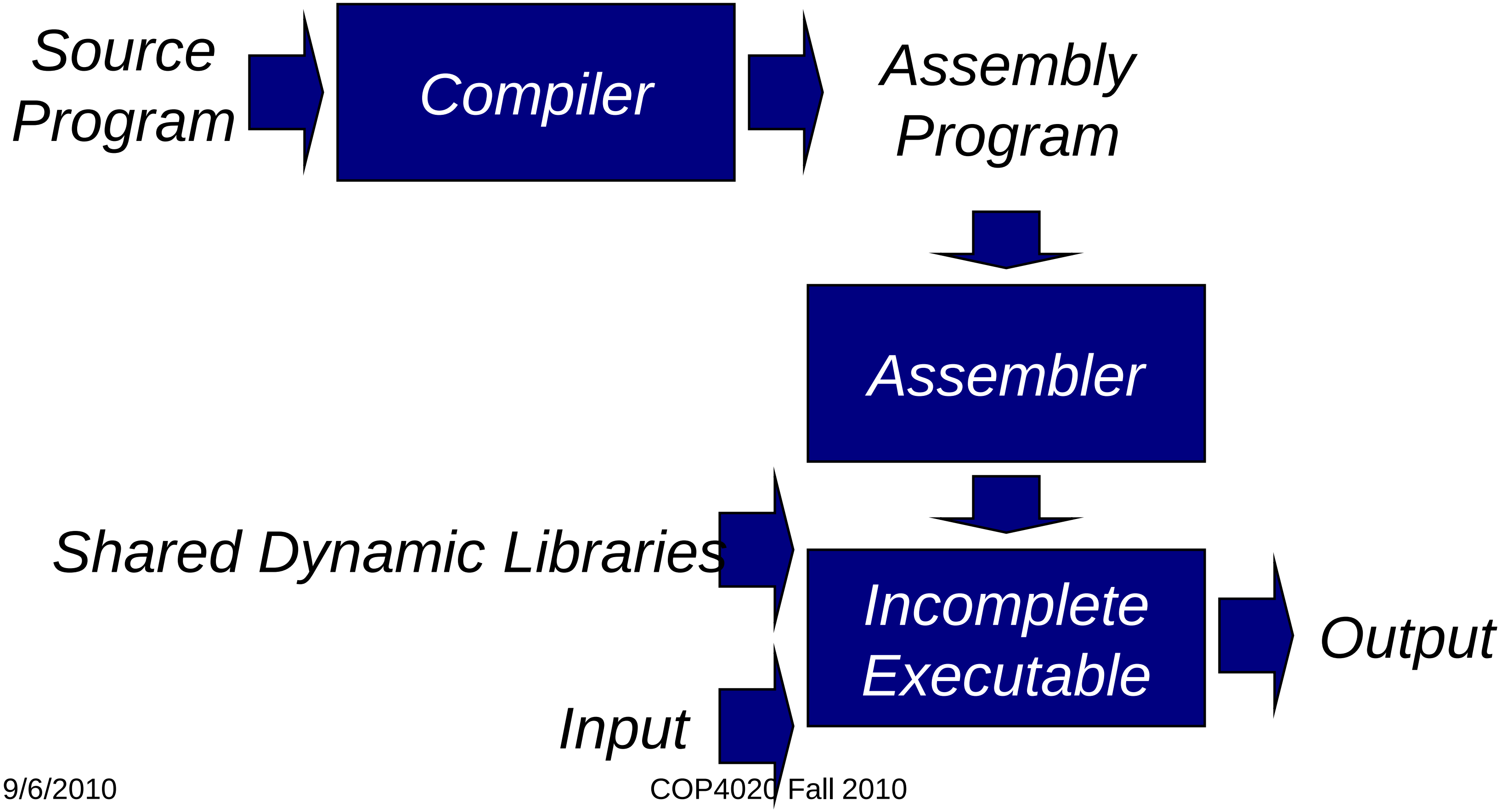
Compilation, Assembly, and Static Linking

- Facilitates debugging of the compiler



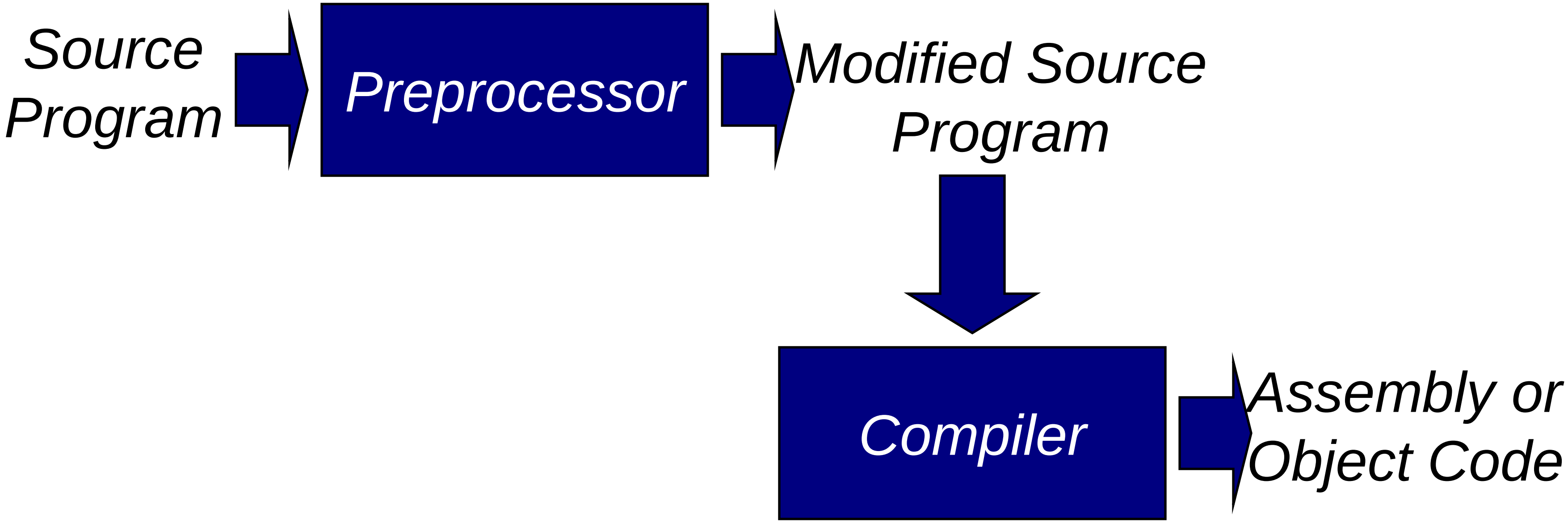
Compilation, Assembly, and Dynamic Linking

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



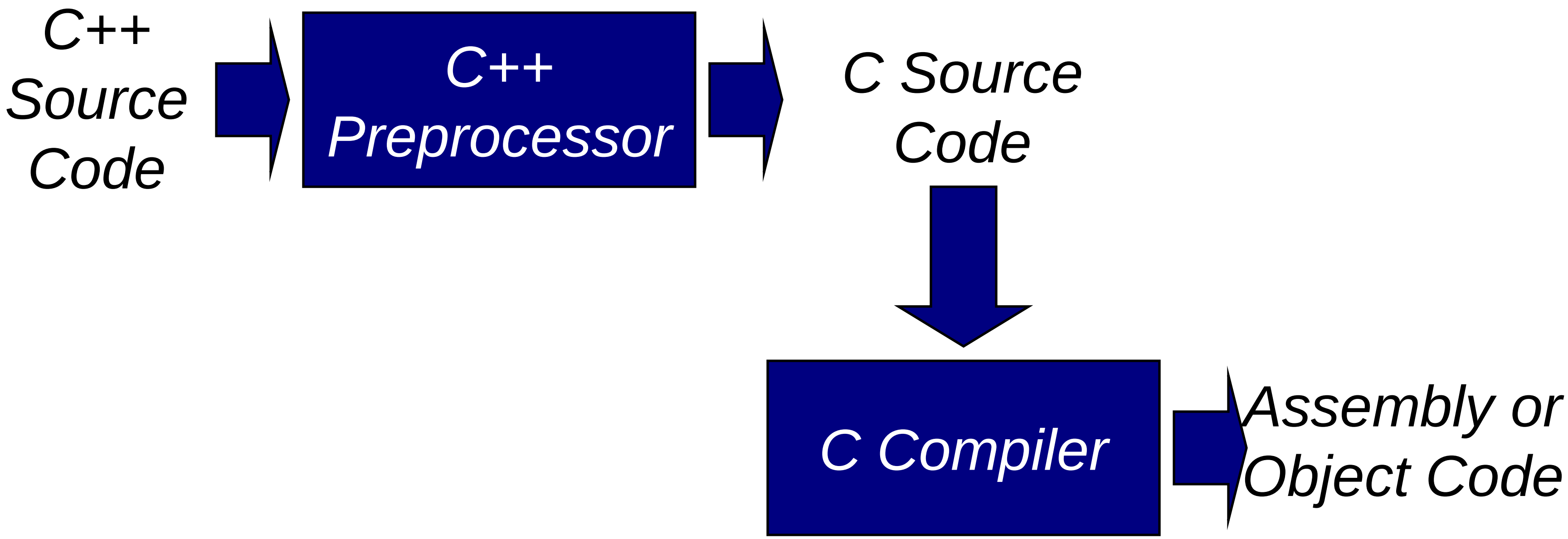
Preprocessing

- Most C and C++ compilers use a preprocessor to expand macros



The CPP Preprocessor

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



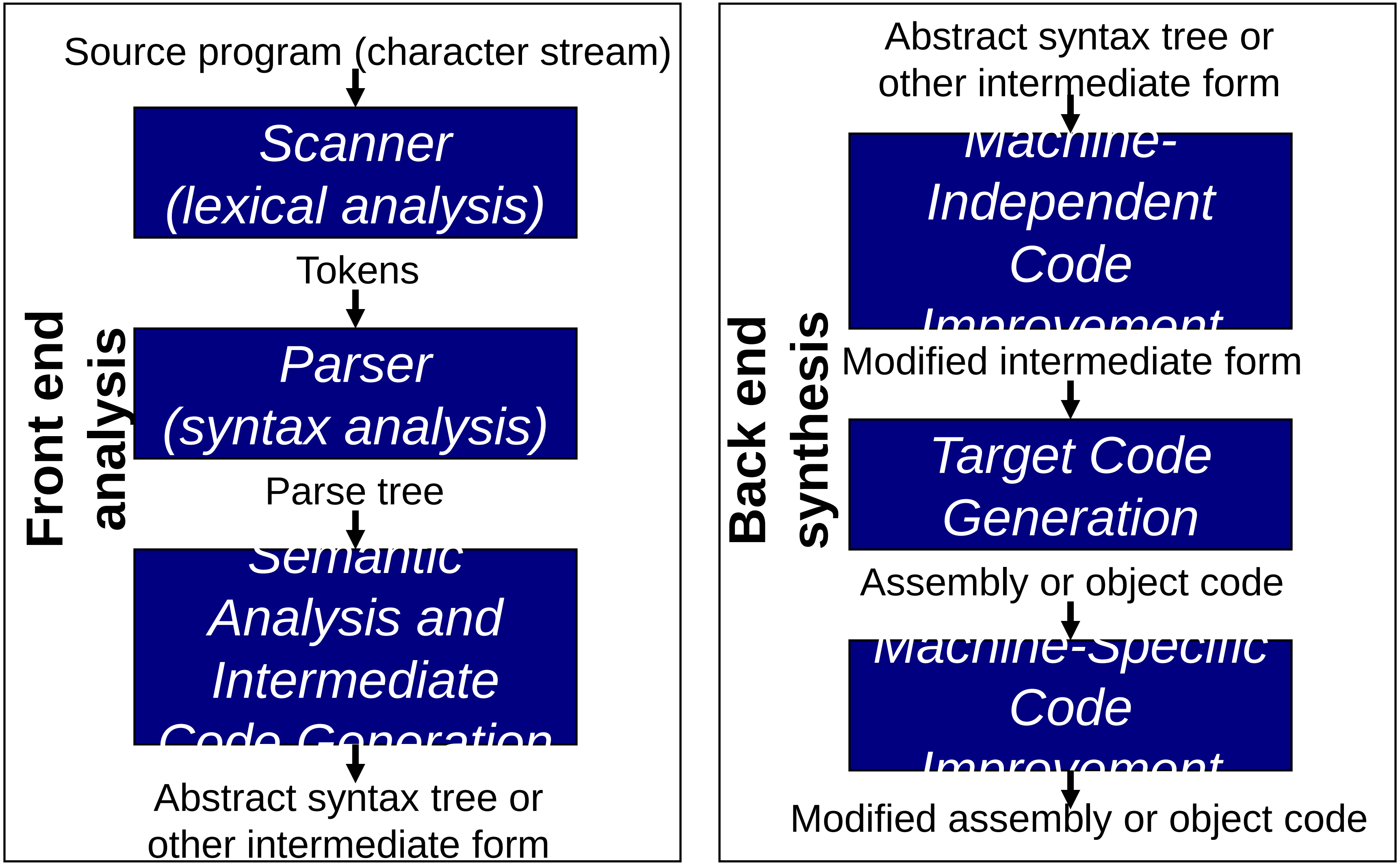
Integrated Development Environments

- Programming tools function together in concert
  - Editors
  - Compilers/preprocessors/interpreters
  - Debuggers
  - Emulators
  - Assemblers
  - Linkers
- Advantages
  - Tools and compilation stages are hidden
  - Automatic source-code dependency checking
  - Debugging made simpler
  - Editor with search facilities
- Examples
  - Smalltalk-80, Eclipse, MS VisualStudio, Borland

Compilation Phases and Passes

- Compilation of a program proceeds through a fixed series of phases
  - Each phase use an (intermediate) form of the program produced by an earlier phase
  - Subsequent phases operate on lower-level code representations
- Each phase may consist of a number of passes over the program representation
  - Pascal, FORTRAN, C languages designed for one-pass compilation, which explains the need for function prototypes
  - Single-pass compilers need less memory to operate
  - Java and ADA are multi-pass

Compiler Front- and Back-end



Scanner: Lexical Analysis

- Lexical analysis breaks up a program into 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
:=	j	-	i	;	writeln	(	i
)	**end**	.					

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>*

Example: Micro Pascal

<Program> ::= **program** *<id>* **(** *<id>* *<More_ids>* **)** **;** *<Block>* **.**
<Block> ::= *<Variables>* **begin** *<Stmt>* *<More_Stmts>* **end**
<More_ids> ::= **,** *<id>* *<More_ids>*
| ε
<Variables> ::= **var** *<id>* *<More_ids>* **:** *<Type>* **;** *<More_Variables>*
| ε
<More_Variables> ::= *<id>* *<More_ids>* **:** *<Type>* **;** *<More_Variables>*
| ε
<Stmt> ::= *<id>* **:=** *<Exp>*
| **if** *<Exp>* **then** *<Stmt>* **else** *<Stmt>*
| **while** *<Exp>* **do** *<Stmt>*
| **begin** *<Stmt>* *<More_Stmts>* **end**
<Exp> ::= *<num>*
| *<id>*
| *<Exp>* + *<Exp>*
| *<Exp>* **-** *<Exp>*

Parser: Syntax Analysis

- Parsing organizes tokens into a hierarchy called a parse tree (more about this later)
- Essentially, a grammar of a language defines the structure of the parse tree, which in turn describes the program structure
- A syntax error is produced by a compiler when the parse tree cannot be constructed for a program

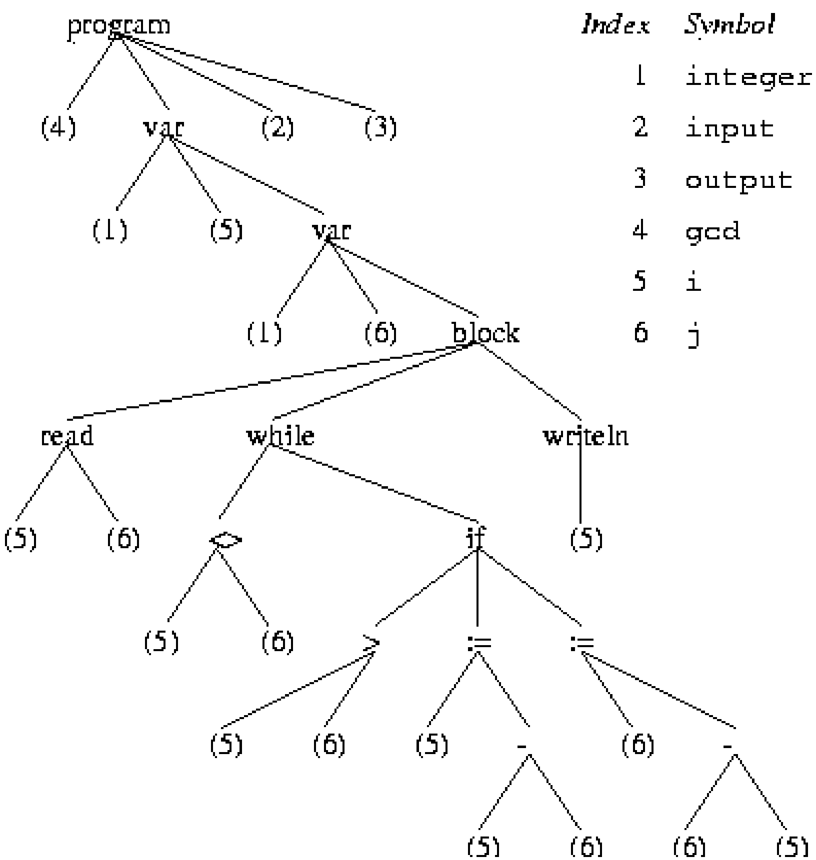
Semantic Analysis

- Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree
- Static semantic checks 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
- 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

Semantic Analysis and Strong Typing

- A language is strongly typed "if (type) errors are always detected"
  - Errors are either detected at compile time or at run time
  - Examples of such errors are listed on previous slide
  - Languages that are strongly typed are Ada, Java, ML, Haskell
  - Languages that are not strongly typed are Fortran, Pascal, C/C++, Lisp
- Strong typing makes language safe and easier to use, but potentially slower because of dynamic semantic checks
- In some languages, most (type) errors are detected late at run time which is detrimental to reliability e.g. early Basic, Lisp, Prolog, some script languages

Code Generation and Intermediate Code Forms



Index	*Symbol*
1	integer
2	input
3	output
4	gcd
5	i
6	j

- 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

Target Code Generation and Optimization

- The AST with the annotated information is traversed by the compiler to generate a low-level intermediate form of code, close to assembly
- This machine-independent intermediate form is optimized
- From the machine-independent form assembly or object code is generated by the compiler
- This machine-specific code is optimized to exploit specific hardware features